High Assurance Programming with Cryptol

CSIIRW 2009, Oak Ridge

Levent Erkök and John Matthews
{lerkok,matthews}@galois.com

The Cryptol team, past and present:
Sally Browning, Magnus Carlsson, Levent Erkök, Sighjorn 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

Galois Vision for Software

- Let the software itself be trustworthy
  - Software artifacts to speak for themselves
  - Rather than hoping to rely 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
    - E.g. rule of signs

- Follow open standards
  - Build components with high internal integrity
  - Maximize interoperability

We want to see software built with the same diligence and analysis as other engineers build bridges
DOE cyber security R&D planning document

Galois invited to participate in October 2008 DOE grassroots cyber security workshop

We were encouraged by the thrust of the final document, e.g:

- “...the Department should develop ... science and mathematics to develop information system architectures and protective measures that go beyond stopping traditional threats to *rendering both traditional and new threats harmless.*” (p. 3)

The Problem

Daniel G. Wolf, Director
Information Assurance Directorate
National Security Agency
January, 2006

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

Cryptol

A Domain Specific Specification Language
- Precise, declarative semantics
- High level design exploration

Automated Synthesis down to FPGA (and working on C)
- Algebraic rewrite-based compilation
- Traceability back to specification

Validation and Verification Tools
- Directed testing
- Quickcheck random testing
- AIG-based equivalence checking
- SAT and SMT-based property checking
- Isabelle theorem prover

Cryptol language design goals

- High-level domain-specific language (DSL) for cryptographic algorithms
  - Specify algorithms precisely and unambiguously
  - But also be executable
- Use Cryptol specifications to guide and document crypto implementations
  - And even generate them
- Be neutral as to implementation platform
  - Don’t bake in Von Neumann assumptions
  - Same Cryptol specification can be compiled to multiple architectures
- Have a clean, unambiguous semantics
  - Essential for any specification language
  - Simplifies design of Cryptol compiler and verification tools
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

blockEncrypt : {k} (k >= 2, 4 >= k) => ([128], [64*k]) -> [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

Cryptol

encrypt128 : ([4][32],[4][4][8]) -> [4][4][8];

encrypt128 (initialKey, plainText) = cipherText where {
  roundKeys = [ initialKey ] # [] nextKey (round, prev_key)
  || round <- [1..10]
  || prev_key <- roundKeys
  ||

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

  cipherText = last(rounds);
};
Cryptol: Equivalence checking

- Given two Cryptol functions $f$, $g$
  - Either prove they agree on all inputs: $\forall x. f\ x == g\ x$
  - Or, provide a counter example $x$ such that $f\ x \neq g\ x$

- Typically:
  - $f$: Specification, written for clarity
  - $g$: Implementation, optimized for speed/space/FPGA, etc.

Simple equivalence checking examples

$$f, g, h : [64] \rightarrow [64];$$
$$f\ x = 2\*x;$$
$$g\ x = x << 1;$$
$$h\ x = x >> 1;$$

Cryptol> :eq f g
True

Cryptol> :eq f h
False
  g 2 = 4
  h 2 = 1
Other equivalence checking use-cases

- Actually, \( f \) and \( g \) don’t have to be written in Cryptol. Our equivalence checker also supports:
  - Xilinx FPGA netlists
  - C code (in progress)
  - Anything else that can be translated into And-Inverter Graphs (our formal model notation)

Cryptol use-case: Verify VHDL crypto cores
Cryptol use-case: Stepwise refinement to implementation

Other Cryptol Assurance tools

- “Quickcheck” property-based testing
  - User gives a property, Cryptol automatically tests it on random inputs.

- Translators to SAT- and SMT-based property checkers
  - User can give more general properties to these tools
  - SAT: Checks for satisfiability of large Boolean formulas
  - SMT extends SAT with higher-level constraint solvers (linear arithmetic, arrays, functions, etc.)

- Safety checking
  - Automatically checks that a Cryptol function will never raise an exception
  - Some possible exceptions: Divide-by-zero, Out-of-bounds array access, assertion failures

- Semi-automatic theorem proving
  - Translator from Cryptol to Isabelle theorem prover
  - User can specify arbitrary Cryptol properties, but proof may need human guidance
The Problem with Equivalence Checking...

- Equivalence checking is hard!
  - To be specific - it’s NP-Complete!
  - For every possible input vector \(2^{\text{#input-bits}}\), the outputs must be equivalent
  - Fortunately, equivalence checking two “similar” Boolean functions is often easier

- We’ll demonstrate our equivalence checking algorithm in the following slides...

Example: Are These Circuits Equivalent?

![Circuit #1](image1)

![Circuit #2](image2)
Model Generation

Key to Circuit Symbols
- AND function, output numbered
- NAND function, output numbered
- OR function, output numbered
- Inverter (NOT function)

Circuit #1

Recall that \( \overline{c} \) is equivalent to \( \overline{c} \)

Circuit #2

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

AIG for Circuit #1

AIG for Circuit #2

Identifying shared structure

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

- Combine the $c$ inputs

Random Simulation

- Looks for nodes that always seem to behave the same:
SAT Sweeping

- Use the AIG forms of the circuits and the candidate equivalent nodes identified by random simulation

SAT Sweeping: Is $3 \equiv 5$?

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

SAT Solver says YES!
**SAT Sweeping: Merge 3 and 5**

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

**SAT Sweeping: Is 2 \equiv 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: $x \equiv y \therefore \text{Circuit } \#1 \equiv \text{Circuit } \#2$

Case Study at Rockwell-Collins
High Speed Encryptor Project
Equivalence Checking: Some Lessons Learned

- Technique is very useful in practice
  - Originally tried BDD-based approach
    - But crypto is engineered to resist useful analysis of the decision tree!
  - Equivalence Checking various versions of AES and DES against reference models take < 5 minutes (most take < 30 seconds)
    - Models typically in range of half a million nodes
  - Checking some 3DES VHDL implementations took about an hour

- However, it has its limitations
  - Other 3DES VHDL implementations timed out
  - The block cipher RC6 also timed out, due to 32-bit multiplier
  - Only works on core ciphers - not crypto modes (finite input/output)

- Next steps
  - We’ve had some initial success using term rewriting in Isabelle to verify harder crypto primitives
  - We plan to automate this by customizing an SMT solver with Cryptol-specific rewrite rules
  - Joint proposal with LLNL to build a parallel SMT solver for analyzing network traffic.

Thank you

www.cryptol.net

Evaluation licenses freely available
Contact: cryptol@galois.com