Formal Methods in Use at Galois, Inc.

Joe Hurd

Galois, Inc.
joe@galois.com

A survey of work by many contributors

Guest Lecture, Specification & Verification I
Computer Laboratory, University of Cambridge
Wednesday 10 February 2010

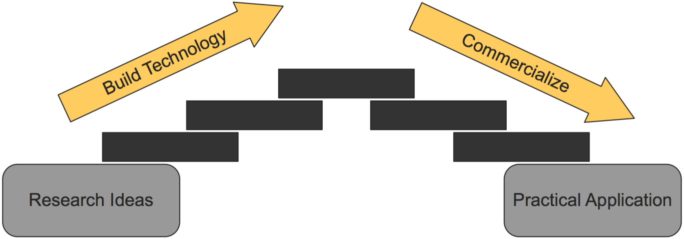
Talk Plan

Galois Background

- Spun out of university research in functional programming and formal methods.
  - Started in 1999.
  - 40 employees, primarily technical.
  - Based in Portland, Oregon, USA.
- Technology transition company.
  - Outsourced R&D, product incubation, technology licensing.
  - High assurance software engineering.
  - Mostly DoD customers.

galois

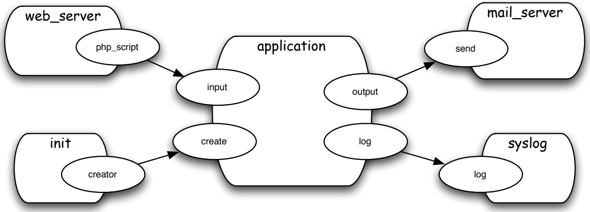
Technology Transition Business



- Galois uses formal methods to build high assurance technology.
- This lecture will look at two examples.

Security Domains

- Security domains contain sensitive information.
- A security policy specifies how information may be shared between domains:

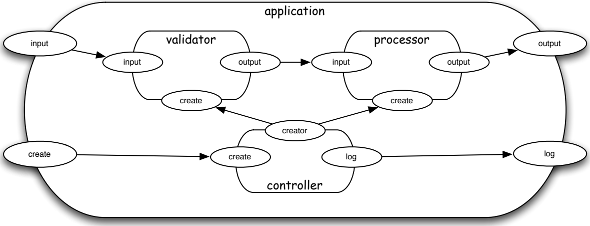


- The arrows show the permissable information flows.

galois

Hierarchical Security Domains

- Security domains can be nested.
- Example: Components of an application, which is running on a system with other applications:



Cross Domain Solutions

- A cross domain solution is a device for transferring information between security domains.
- Automating such transfers creates new risks, but enables timely distribution of information to where it is needed.
- Example: A coalition needing to know where each others' troops are.
- Critical Property: The implementation of the device ensures that transfers are consistent with the security policy.
- Formal methods can be applied to verify the critical property.

galois

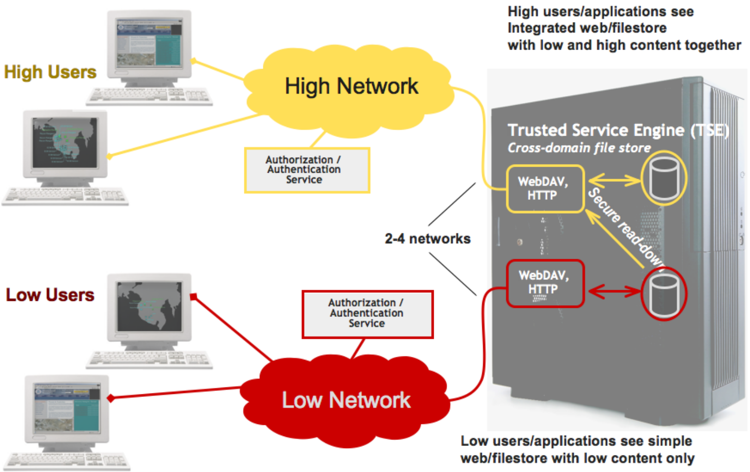
Cross Domain Solutions: Example

- The KG-255 Inline Network Encryptor:

- The security policy might require that any data that is transferred from the secret network onto the unclassified network is encrypted.
- Verification Goal: Prove that the implementation can never violate the security policy.

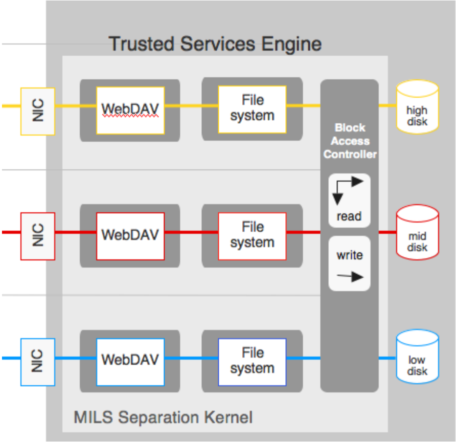
galois

TSE: A Cross Domain Filestore with Read-Down



TSE Architectural Principles

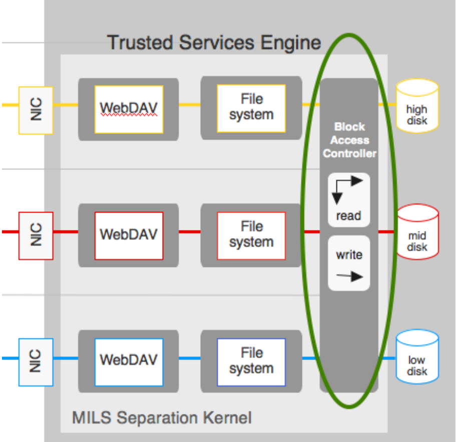
1. Factor the security architecture.
2. Minimize the number of components requiring high assurance.
3. Keep each as simple as possible.
4. Use formal methods in critical places.



Block Access Controller (BAC)

The BAC directs accesses across disk drives at multiple levels.

- High assurance component.
- Must eliminate data channels between levels.
- Must control timing channels between levels.



BAC Verification

- The overall security goal is non-interference: a partition's state is not dependent on the actions of any higher level partition.
- Formalize von Oheimb's theory of non-interference in the Isabelle theorem prover.
- Implement the BAC as an Isabelle function.
  - Build with functions of type state $\rightarrow$ error $+ \alpha \times$ state.
- Non-Interference Verification: Complete an Isabelle proof that the BAC function satisfies non-interference.
- Model-to-Code Correspondence: Pretty-print the 800 line C implementation of the BAC from the Isabelle implementation.

Non-Interference Verification

- Safety proof (Hoare logic):
  1. Guess invariants for triply nested loops.
  2. Try to prove the resulting verification conditions.
  3. On all times except the last, fail and go back to step 1.
- Non-Interference proof:
  - Assumes safety.
  - Unwinding lemmas (adjusted for state and error).

Model-to-Code Correspondence

Definition

```
responseSet (level::level) (resp::response) (i::nat) =
let respN = responseOvec level;
    x0 = bytesPerResponseRep * i in
do oldToggle <- ovecRef respN x0;
   ovecSet respN (1 + x0) (respOk resp);
   ovecSet respN x0 (toggleNat oldToggle)
```

Code

```
void responseSet (level level, response resp, nat i) {
  nat respN = responseOvec(level);
  nat x0 = bytesPerResponseRep * i;
  nat oldToggle = ovecRef(respN, x0);
  ovecSet(respN, 1 + x0, respOk(resp));
  ovecSet(respN, x0, toggleNat(oldToggle));
}
```

Domain Specific Languages

- Domain Specific Languages (DSLs) are high level languages for design capture in particular problem areas.
- A program in the DSL is an unambiguous specification that:
  - guides and documents implementations;
  - can be executed to generate test vectors;
  - can be compiled directly to an implementation.
- Ideal Situation: Reason at a high level about a program in the DSL, and the properties also apply to the low-level implementation.

Cryptol: A Domain Specific Language for Crypto

- Cryptol is a declarative language for describing crypto algorithms.
    - Primitives for common operations on blocks and streams of bits.
    - No assumptions are made about the implementation platform.
- An expressive type system ensures consistency of crypto algorithms.
- Download the Cryptol interpreter and give it a try:

`http://www.cryptol.net`

Cryptol Type System

- The Cryptol type system captures important details of interfaces.
- The type system is Hindley-Milner plus arithmetic constraints.
- Numeric literals are one source of constraints:

```
13 : {a} (a >= 4) => [a]
```

"The literal 13 is represented by a bit vector that requires at least 4 bits to represent"

Cryptol Type System: Example

- From the Advanced Encryption Standard:

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.*

- In 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 Type Checking

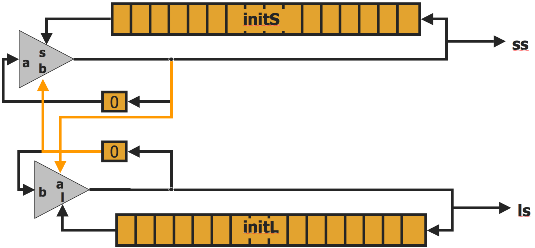
- Constraints in Cryptol types can be arbitrary arithmetic expressions:

```
split : {a b c} [a*b]c -> [a][b]c
```

- Problem: In general, type checking a fully type-annotated Cryptol program is an undecidable problem.
- Domain Specific Solution: Implement a custom type checker with knowledge of the constraints that result from typical cryptographic operations.
- The type checker uses narrowing and term rewriting.
- Simplification is used to present type inference results that are easier on the eyes (and brain!).

Formalizing "Circuit Diagrams"

Informal circuit diagrams are often used by cryptographers:



Code (Cryptol implementation)

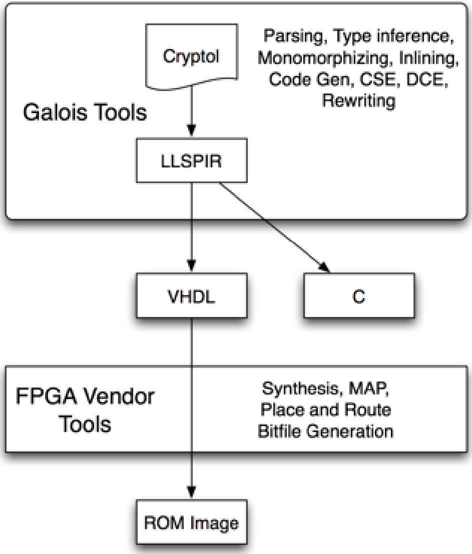
```
ss = [| (s+a+b) <<< 3 || s <- initS # ss
                      || a <- [0] # ss
                      || b <- [0] # ls |];

ls = [| (l+a+b) <<< (a+b) || l <- initL # ls
                          || a <- ss
                          || b <- [0] # ls |];
```

Compiling Cryptol to FPGAs

- For high-grade crypto, hardware-only solutions are the norm.
  - Commodity hardware is not trusted.
  - There is trust when the evaluators can see as much of the solution as possible.
- Natural match between crypto and FPGAs:
  - manipulation of arbitrary length bit sequences;
  - highly parallel encryption/decryption;
  - highly parallel cryptanalysis.
- Goal: Verifying compilation of a Cryptol specification to an FPGA implementation.

galois

Cryptol to FPGA Toolchain



Key Observations

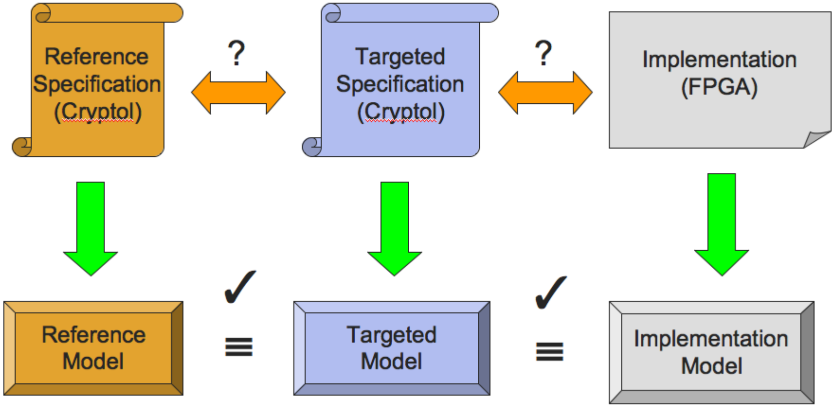
- Sequentialization in Cryptol comes only from data dependency.
  - Just like hardware—no Von Neumann bottleneck.
- Sequences are descriptions only.
- Implementation of sequences can be:
  - laid out in time (loops and/or state machines);
  - laid out in space (parallel and/or pipeline);
  - or a mixture of both.
- The mathematical specification is the same.

"Henceforth space by itself, and time by itself, are doomed to fade away into mere shadows, and only a kind of union of the two will preserve an independent reality."
– Minkowski, Space and Time, Sept. 21, 1908

Cryptol to FPGA Verification

- Reference Cryptol program serves as the specification of the crypto algorithm.
  - Optimized for simplicity and human understanding.
- Targeted Cryptol program resolves design choices such as the time/space layout of sequences.
  - Optimized for compiling to efficient FPGA implementations.
- Verification Goal: Compiled FPGA implementation is equivalent to the reference Cryptol program.

Cryptol to FPGA Verification



Cryptol-FPGA Verification

- Subgoal: Verify an FPGA implementation is equivalent to a targeted Cryptol program:
- Tactic: Verify a VHDL circuit is equivalent to a Cryptol program.
  1. Symbolically execute the Cryptol program to generate an And-Inverter Graph (AIG).
  2. Symbolically execute the VHDL to get another AIG.
  3. Use a SAT solver to verify that the two AIGs are equal.
- Bonus: Has found several Cryptol-FPGA compiler bugs!

galois

Cryptol-Cryptol Verification

- Subgoal: Verifying a targeted Cryptol program is equivalent to a reference Cryptol specification.
- Tactic: The Cryptol toolset provides an equivalence checker for Cryptol programs.

Summary

- Formal methods is being actively used at Galois as part of our mission to *"create trustworthiness in critical systems"*.
- Please get in touch if you are interested in finding out more or working for us.

`joe@galois.com`
`http://www.galois.com`