Proof Carrying Code

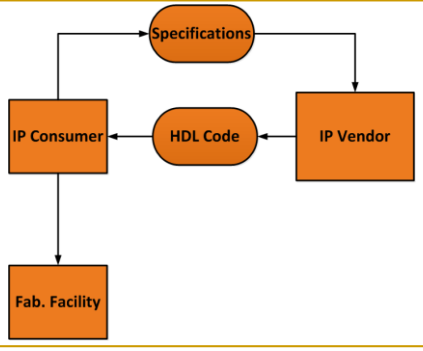
Mohammad Tehranipoor

ECE6095: Hardware Security & Trust
University of Connecticut
ECE Department

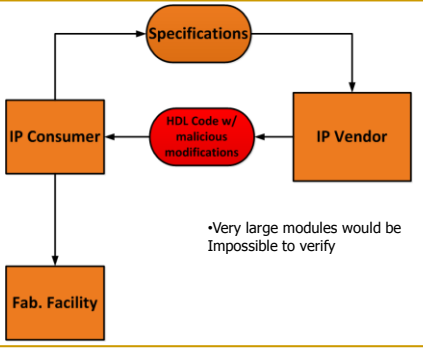
Outline

- Hardware IP Verification
- Hardware PCC Background
- Software PCC Description
- Software PCC Example
- Hardware PCC Description
- Hardware PCC Example

Background: Module Acquisition

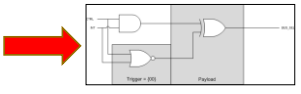


Background: Module Acquisition



- Very large modules would be Impossible to verify

Hardware Trojan Threat



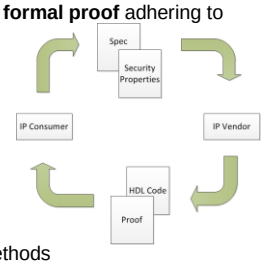
- Adversary can make malicious modifications to a module's HDL code.
  - Trojan could degrade performance, leak information, render chip useless, etc.
  - In very large circuits this would be very difficult to detect.

Why Compromise HDL Code?

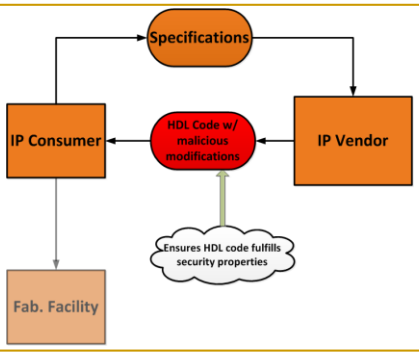
- Criminal activity
  - Ruin a company's reputation
- Personal gain
  - Money
  - Extract information
- People who just want to cause havoc.

Solution: Hardware Proof Carrying Code (HPCC)

- IP consumer provides vendor with specifications as well as a **list of specific security related properties**
- Vendor must then construct a **formal proof** adhering to these security properties
  - Create a set of definitions that models behavior of any Verilog statement.
  - Use these proof frameworks to model modules.
- Consumer can easily and automatically check these proofs to make sure the code is valid.
- Better than many previous methods



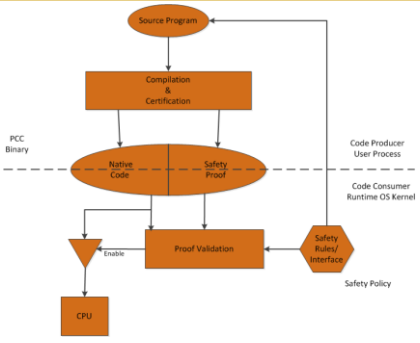
Module Acquisition with HPCC



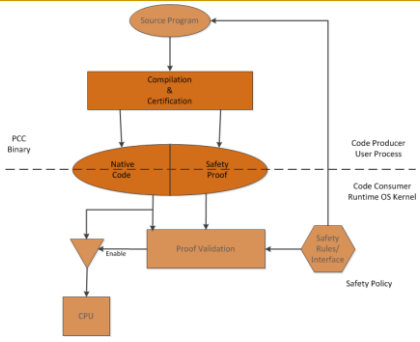
PCC Background- Software

- PCC for hardware module acquisition is based off of software PCC.
  - Makes sure it is safe for a code consumer to run foreign code
- In type-specialized PCC a verification condition, or logical formula, must be proven as true in order for the code to be trusted.
- The code producer must prove, and the code consumer must check that the safety conditions are satisfied for a given program.

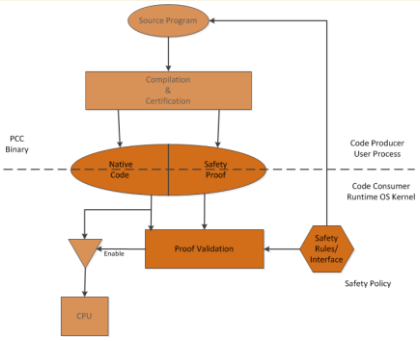
PCC Process



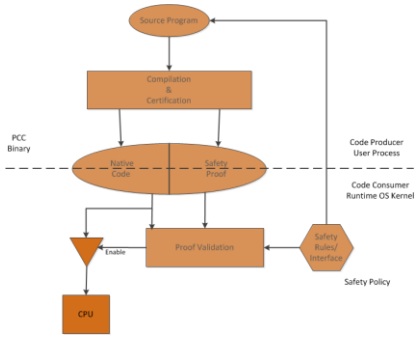
PCC Process: Certification



PCC Process: Verification



PCC Process: Execution



Defining The Safety Policy

- Three different parts to the Safety Policy:
  - **Verification-condition generator-** Procedure that computes a predicate (safety predicate) in first order logic based on the code to be certified.
  - **Set of axioms** that can be used to prove the safety predicate.
  - **Precondition**-defines the calling convention of the PCC binaries. Postcondition establishes the state of the system must be in at the end of execution.

Verification-Condition Generator

- Using a subset of Assembly language we generate an abstract machine. These derived statements are also known as **Safety Predicates**

$$(\rho, pc) \rightarrow \begin{cases} (\rho[\mathbf{r}_d \leftarrow \rho(\mathbf{r}_s) \oplus \rho(op)], pc+1), & \text{if } \Pi_{pc} = \text{ADD } \mathbf{r}_s, op, \mathbf{r}_d \\ (\rho[\mathbf{r}_d \leftarrow \mathbf{sel}(\rho(\mathbf{r_m}), \rho(\mathbf{r}_s) \oplus n)], pc+1), & \text{if } \Pi_{pc} = \text{LD } \mathbf{r}_d, n(\mathbf{r}_s) \text{ and } \boxed{\rho \models \mathbf{r}_s \oplus n : \mathbf{ro\_addr}} \\ (\rho[\mathbf{r_m} \leftarrow \mathbf{upd}(\rho(\mathbf{r_m}), \rho(\mathbf{r}_d) \oplus n, \rho(\mathbf{r}_s))], pc+1), & \text{if } \Pi_{pc} = \text{ST } \mathbf{r}_s, n(\mathbf{r}_d) \text{ and } \boxed{\rho \models \mathbf{r}_d \oplus n : \mathbf{addr}} \\ (\rho, pc+n+1), & \text{if } \Pi_{pc} = \text{BEQ } \mathbf{r}_s, n \quad \text{and} \quad \mathbf{r}_s = 0 \\ (\rho, pc+1), & \text{if } \Pi_{pc} = \text{BEQ } \mathbf{r}_s, n \quad \text{and} \quad \mathbf{r}_s \neq 0 \\ (\rho, pc+1), & \text{if } \Pi_{pc} = \text{INV } p \end{cases}$$

Precondition

- Specifies which properties can be assumed to hold when the PCC is invoked.
- Expressed in a language of predicates:

$$\begin{array}{lll} P & ::= & \mathbf{true} \mid P_1 \wedge P_2 \mid P_1 \supset P_2 \mid \forall \mathbf{r}_i.P \mid e_1 = e_2 \mid e_1 \neq e_2 \mid e : \tau \\ \tau & ::= & \mathbf{addr} \mid \mathbf{ro\_addr} \end{array}$$

- Where tao is the typing predicate.

Set of Axioms

- Also known as the **proof system**, these axioms serve to prove the predicates given in the logic.

$$\frac{}{\triangleright \mathbf{true}}\mathbf{true\_i} \qquad \frac{\triangleright P_1 \quad \triangleright P_2}{\triangleright P_1 \wedge P_2}\mathbf{and\_i} \qquad \frac{\triangleright P_1 \wedge P_2}{\triangleright P_1}\mathbf{and\_el} \qquad \frac{\triangleright P_1 \wedge P_2}{\triangleright P_2}\mathbf{and\_er}$$

$$\frac{\begin{array}{c}\overline{\triangleright P_1}^{\,u} \\ \vdots u \\ \triangleright P_2\end{array}}{\triangleright P_1 \supset P_2}\mathbf{impl\_i}^{u} \qquad \frac{\triangleright P_1 \supset P_2 \quad \triangleright P_1}{\triangleright P_2}\mathbf{impl\_e} \qquad \frac{\begin{array}{c}\\ v \\ \vdots v \\ \triangleright [v/x]P\end{array}}{\triangleright \forall x\,.\,P}\mathbf{all\_i}^{v} \qquad \frac{\triangleright \forall x\,.\,P}{\triangleright [e/x]P}\mathbf{all\_e}$$

Software PCC Example

- Suppose the kernel has an internal table consisting of two memory words per user process- tag and data word.
  - Tag describes whether the word is user-writeable.
- User processes can access their table entry by installing native code on the kernel.
- The kernel invokes this user installed code with the address of the table entry corresponding to the parent process in machine register r_0.

```
                        %Address of tag in r0
1    ADD  r0, 8, r1     %Address of data in r1
2    LD   r0, 8(r0)     %Data in r0
3    LD   r2, -8(r1)    %Tag in r2
4    ADD  r0, 1, r0     %Increment Data in r0
5    BEQ  r2, L1        %Skip if tag == 0
     ST   r0, 0(r1)     %Write back data
L1   RET                %Done
```

Software PCC Example Cont.

- We would like to provide user-installed code with full access to the corresponding table entries which still maintaining integrity of kernel.
- Safety Policy:
  - The user code cannot access other table entries besides the one pointed to by r_0.
  - The tag is read only.
  - The data word is read only unless the tag is non-zero.
  - The code does not modify reserved and callee-saves registers.

Software PCC Example Cont.

- In formal notation the precondition for this safety policy is written:

$$Pre_r = \mathbf{r}_0 : \mathbf{ro\_addr} \ \wedge \ \mathbf{r}_0 \oplus 8 : \mathbf{ro\_addr} \ \wedge \ \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0) \neq 0 \supset \mathbf{r}_0 \oplus 8 : \mathbf{addr}$$

- This ensure that it is safe to read from r_0 and r_0 with an offset of 8 as long as the safety tag is non-zero.
- Postcondition is “true”.

Software PCC Example Cont.

- Certification involves generating safety predicates (VC Generator):

$$VC_i = \begin{cases} [\mathbf{r}_s \oplus op/\mathbf{r}_d]VC_{i+1}, & \text{if } \Pi_i = \text{ADD } \mathbf{r}_s, op, \mathbf{r}_d \\ \mathbf{r}_s \oplus n : \mathbf{ro\_addr} \wedge [\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s \oplus n)/\mathbf{r}_d]VC_{i+1}, & \text{if } \Pi_i = \text{LD } \mathbf{r}_d, n(\mathbf{r}_s) \\ \mathbf{r}_d \oplus n : \mathbf{addr} \wedge [\mathbf{upd}(\mathbf{r_m}, \mathbf{r}_d \oplus n, \mathbf{r}_s)/\mathbf{r_m}]VC_{i+1}, & \text{if } \Pi_i = \text{ST } \mathbf{r}_s, n(\mathbf{r}_d) \\ (\mathbf{r}_s = 0 \supset VC_{i+n+1}) \wedge (\mathbf{r}_s \neq 0 \supset VC_{i+1}), & \text{if } \Pi_i = \text{BEQ } \mathbf{r}_s, n \\ Post, & \text{if } \Pi_i = \text{RET} \\ \mathcal{P}, & \text{if } \Pi_i = \text{INV } \mathcal{P} \end{cases}$$

- Which yeilds the final safety predicate:

$$SP(\Pi, Inv, Post) = \forall \mathbf{r_k}. \bigwedge_{i \in Inv} Inv_i \supset VC_{i+1}$$

Software PCC Example Cont.

- After applying the VC generator to our sample program (with simplifications), we obtain the following safety predicate:

$$\begin{aligned} SP_r = \forall \mathbf{r}_0. \forall \mathbf{r}_1. \forall \mathbf{r_m}. \quad (Pre_r \quad &\supset \quad ((\mathbf{r}_0 \oplus 8) \ominus 8 : \mathbf{ro\_addr} \wedge \mathbf{r}_0 \oplus 8 : \mathbf{ro\_addr}) \wedge \\ &\quad (\mathbf{sel}(\mathbf{r_m}, (\mathbf{r}_0 \oplus 8) \ominus 8) = 0 \supset \mathbf{true}) \wedge \\ &\quad (\mathbf{sel}(\mathbf{r_m}, (\mathbf{r}_0 \oplus 8) \ominus 8) \neq 0 \supset \mathbf{r}_0 \oplus 8 : \mathbf{addr})) \\ \wedge (\mathbf{r}_1 : \mathbf{addr} \quad &\supset \quad \mathbf{r}_1 : \mathbf{addr}) \end{aligned}$$

- In order to generate the final safety predicate, the consumer must use the safety rules (set of axioms) which are part of the safety policy.
  - This last predicate is also known as the formal **safety proof**.
- The safety proof may now be proven.

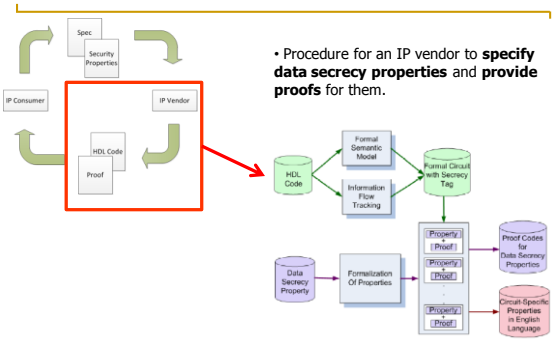
Software PCC Difficulties

- In order for SPCC to work we must have trust in all steps in the process.
  - Bugs in the VC-Gen, Logical Axioms, or proof checker.
- There may be “loopholes” in the precondition and postconditions.
- Very large PCC binaries
  - Possibility of proof growing exponentially.

Hardware PCC

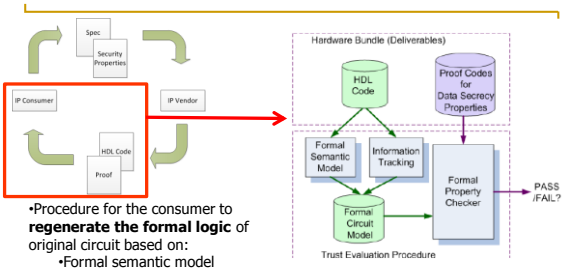
- Instead of untrusted code, we are verifying untrusted **third party Hardware IPs**.
- VC generator generates safety predicates from HDL statements.
- Safety policy may have multiple parts that are decided upon by the IP consumer.
  - Both consumer and vendor must agree upon a fixed translation of these policies into formal mathematical codification.
- Process must take into account the concurrent and sequential nature of HDL statements.

HPCC: Vendor Side



• Procedure for an IP vendor to **specify data secrecy properties** and **provide proofs** for them.

HPCC: Consumer Side



•Procedure for the consumer to **regenerate the formal logic** of original circuit based on:
- •Formal semantic model
- •Information flow tracking rules

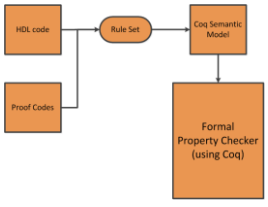
•Formal logic and the provided proof codes then pass through the **formal property checker** to generate a PASS/FAIL

HPCC Property Checker

- Formal Property Checker (FPC) is given the proof codes and formal circuit model to generate a pass/fail for the IP core.
- Verilog, VHDL, etc. are poorly developed for property checking
  - Creating FPCs in native HDL would be very complicated.
- The proof assistant platform Coq is used for property checking (same as SPCC).
  - Therefore, circuit and proof codes must be expressed using Coq theorem language

Converting HDL to Coq

- We must create:
  - A **structural semantic model** within the Coq platform to represent hardware circuitry.
  - **Set of rules** to convert HDL to Coq semantic model



Coq Semantic Model

- New HDL represented in Coq formal logic.
- Architecture of the circuit is accurately described in Coq Semantic model, but designers have flexibility in defining functionality.
- Model is effective in information tracking.
- Conversion Rules are used to automate the process and include:
  - Signal Definitions
  - Signal Operations
  - Expressions

Coq Semantic Model

- On top of signal definitions we create **expressions**, consisting of combinational logic and control operations.
- Expressions are essentially equivalent to a parse tree generated by Verilog compiler.
- These in turn are interpreted by the *eval* function.
  - This recursively maps the expression tree onto the value of signals, at a specified time.
- This effectively models the *assign* statement.

Signal Definitions

- Value of a signal:

```
Inductive value := lo | hi.
```

- One bit signal is treated as a 1-bit wide bus:

```
Definition bus_value := list value.
Definition bus := nat -> bus_value.
Definition bus_length (b : bus) :=
  Fun t : nat => length (b t).
```


Coq Signal Operations

- Bus handling methods include *and*, *or*, *xor*, equality operations, etc.
- Example of *and* operation:

```
Fixpoint bv_bit_and  (a b : bus_value)
{struct a} : bus_value :=
  match a with
  | nil => nil
  | la :: a' => match b with
                | nil => nil
                | lb :: b' => (and la lb) ::
                  (bv_bit_and a' b')
                end
  end.

Definition bus_bit_and (a b : bus) : bus :=
  fun t:nat => bv_bit_and (a t) (b t).
```


Coq Expressions

- On top of signals definitions and operations expressions are built.
- Excerpt from the complete expression definition:

```
Inductive expr :=
  | econv : bus_value -> expr
  | econb : bus -> expr
  | eand : expr -> expr -> expr
  | eor : expr -> expr -> expr
  | exor : expr -> expr -> expr
  | enot : expr -> expr
  | cond : expr -> expr -> expr -> expr
  | perm : expr -> expr
  | sbox : bus -> expr
...
```

• The *eand* constructor, for example, Connects two expressions to form A new expression.

Evaluation of Coq Expressions

- Evaluation of expressions is recursively defined to calculate the value of an expression at a specified time *t*.

```
Fixpoint eval (e : expr) (t : nat)
{struct e) : bus_value :=
  match e with
  | econv v => v
  | econb b => b t
  | eand ex1 ex2 =>
    bv_bit_and (eval ex1 t) (eval ex2 t)
  | eor ex1 ex2 =>
    bv_bit_or (eval ex1 t) (eval ex2 t)
  | enot ex =>
    bv_bit_not (eval ex t)
  | cond cex ex1 ex2 =>
    match (bv_eq_0 (eval cex t)) with
      | hi => eval ex1 t
      | lo => eval ex2 t end
  | perm ex => eval ex
  | sbox b => b t
...
```


Final Coq Semantic Model

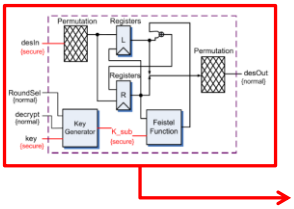
- Signals, expression, and their semantic models are now defined.
- Constructors are chosen to improve user-friendlyness.

Notation is added to pile the code through the ; symbol.

```
Inductive code :=
  | outb : bus -> code
  | inb : bus -> code
  | wireb : bus -> code
  | regb : bus -> code
  | assign_ex : bus -> expr -> code
  | assign_b : bus -> bus -> code
  | assign_case3 : bus -> expr -> code
  | nonblock_assign_ex : bus -> expr -> code
  | nonblock_assign_b : bus -> bus -> code
  | codepile : code -> code -> code.

Notation " c1 ; c2 " := (codepile c1 c2)
  (at level 50, left associativity).
```


Coq Semantic Model Example




```
Definition des : code :=
  outb desOut;
  inb desIn;
  inb key;
  inb decrypt;
  inb roundSel;
  inb clk;
  wireb K_sub;
..
  assign_ex Lout (cond (eq (econb roundSel)
    (econv (lo::lo::lo::lo::nil)))
    (econb (IP @ [33, 64])) (econb R));
  assign_ex Xin (cond (eq (econb roundSel)
    (econv (lo::lo::lo::lo::nil)))
    (econb (IP @ [1, 32])) (econb L));
  assign_ex Rout (econb (bus_bit_xor Xin out));
  assign_ex FP (econb (bus_app Rout Lout));
  module_inst2in out Lout K_sub;
  nonblock_assign_ex L (econb Lout);
..
```

Information Tracking and Flow

- At this point, our circuit is simply defined in another form of HDL and does not have any additional security properties.
  - Signal bypassing strategies may be used
- Tags are used to track and protect sensitive internal signals by using an additional property:
  - *Sensitivity*
- We therefore extend the bus definition to return a *value*sensitivity* pair at a specified time *t*.
  - Sensitivity is defined as an inductive value with two constructors: *Secure* and *normal.*
  - These indicate whether the signal needs protection or not.
- A signal with a *secure* tag is not allowed to propagate to a primary output or Trojan side-channel.

Information Tracking and Flow

- Bus definition extension:

```
Inductive sensitivity := secure | normal.
Definition bus := nat ->
  (bus_value * sensitivity).
```

- Three propagation rules are then defined for tags:

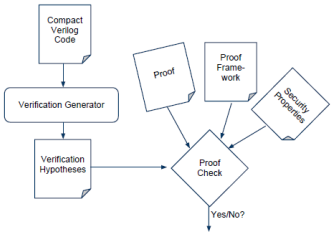
```
Definition uoptag
  (a : sensitivity) : sensitivity := a.
Definition boptag
(a b : sensitivity) : sensitivity :=
  match a with
  | secure => secure
  | normal => match b with
              | secure => secure
              | normal => normal
              end
  end.
Definition rmtag
  (a : sensitivity) : sensitivity := normal.
```

• Note: only permutation and module instantiation operations are allowed to call *rmtag*

Automatic Proof Validation

- Basic Coq language proof solving procedure:
  - Proof begins with one goal (statement to be proven) and no hypotheses.
  - A goal can be solved and eliminated when it exactly matches one hypotheses in the context.
  - The proof is completed when all goals are solved.
- In our application the hypotheses are created by the Verification Generator
- Hypothesis basically admit a proposition as true so that it can be used as a precondition for proof.

Automatic Proof Validation



HPCC Example

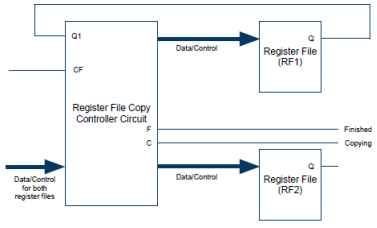
This example will show the following steps:

1. Vendor and Consumer agree on security related properties preventing malicious behavior.
2. Security Properties are converted into formal Coq logic.
3. IP Vendor constructs a correctness proof.
4. Consumer checks this proof against the code

HPCC Example

- Consumer needs a circuit which controls access to two register files.
- Controller must have a “copy mode” which when activated by a flag signal, *CF*, causes the contents of register 1 (RF1) to transfer to register 2 (RF2).
- Only requirement is that the file in the first register must be copied and transferred, unchanged, to some location in the second

HPCC Example



•Sequence of reads/writes, addresses at which values are stored, etc are not taken into account.

HPCC Example: Choosing Security Related Properties

1. **Stability:** do not enter copy mode unless the copy flag is raised.
2. **Transparency:** When not in copy mode, simply pass control signals through to both RFs.
3. **Termination Transfer:** When copy flag is raised, enter copy mode, transfer all values unmodified from RF1 to RF2, and then exit copy mode within a predefined number of clock cycles.

HPCC Example: Translate Properties to Logic- Stability

- We must remain outside of copy mode for all cycles in which the controller is not already in copy mode and *CF* is low.

```
Theorem stable_c : forall t:nat, t > 0 ->
  c t = lo -> cf t = lo -> c (S t) = lo.
```

- S is the successor function (primitive recursive function)
  - Say, for example *x <= x+1*

    if *(x t)* represents the value of x at time t, then

    (x (S t)) = (x t)+1
  - *In General, S(t)=t+1*
- In this way, copy mode for all values of time are checked recursively.

HPCC Example: Translate Properties to Logic- Termination Transfer

- Exhibits much greater complexity than the previous two properties and takes two parts.
- In the first sub-property we define the operation of reading from address *a.*

```
Definition read := fun (a n t X : nat) =>
  (a1 (t+n)) = a /\ (q1 (t + n)) = X.
```

  - Asserts that the value sent on address line to RF1 during the stated clock cycle is equal to *a.*
  - *X* is sent to capture the value returned from RF1, allowing us to refer to this value later when it is written.

HPCC Example: Translate Properties to Logic- Termination Transfer

- For the write operation to RF2, it is assumed that the write enable is high during clock cycle *t+n* and that the value sent to RF2 is equal to X.
- Write definition is completed by specifying write uniqueness:
  - Given as a unique property, this asserts that a value once stored will not be overwritten.
- Finally, a non-recursive function *transfer* is defined

```
Definition transfer := fun (t nf : nat) =>
  forall a:nat, a <= regs -> exists n:nat,
   n > 0 /\ n < nf /\ exists X:nat,
   (read a n t X) /\ exists nw:nat, nw > 0
   /\ nw < nf /\ (write X nw t nf).
```


HPCC Example: Proving Security Compliance

- Once the circuit has been coded, the first step of the proof construction is to generate the Verification Hypotheses using compact Verilog code:

```
assign we2 = (cprev & c) ? 1'b1 : (c ? 1'b0 : we2_in);
```

```
. . .
Hypothesis assign_we2 : (assign we2
  (cond (and (econs cprev) (econs c))
    (econs Vdd)
    (cond (econs c)
      (econs Gnd)
      (econs we2_in)))).
```

HPCC Example: Proving Security Compliance

- The next step in proving security compliance is to construct a proof.
- First two proofs are trivial, below is a high level description of the termination transfer property:
- The method of induction on clock cycles is used for the termination transfer property:
  - Lemmas rely on a "transition cycle" t, marking the transition into copy mode, and an index n which counts a certain amount of cycles after this transition.
  - For example, if the transition occurs at time 15, then time 18 could be represented as t=15 and n=3.
- A lemma is written called *read_eq* to inductively establish that the current read address remains one less than the write address for the duration of the copy.
  - Other lemmas are constructed on top of this
  - Example: Lemma is written to prove the uniqueness of sub-property holds on all writes and that the sequence of operations performed in copy mode is a complete transfer.
- The entire proof takes up almost 10 pages.

HPCC Summary

- While most Trojan detection techniques rely on post-fab actions, this technique is entirely pre-fab.
- Hardware PCC provides a definitive guarantee that the HDL code obeys a set of security-related properties.
  - This removes the difficulty of placing blame on one of the third parties involved if there is a problem with the circuit.
- The Vendor is assigned the task of constructing compliance proofs for the IP.
- Consumer is only responsible for developing security properties which will be proven be the Vendor.

HPCC Weaknesses

- Perhaps the largest weakness is the necessity for many security properties which are not automatically generated.
  - There may be loopholes
  - Very large programs
- Huge proofs associated with even small modules.
- Extra cost is involved because more work is placed on the Vendor in constructing the proof.

Questions?

References

[1] George C. Necula and Peter Lee, "Proof-Carrying Code" in CMU-CS-96-165, November 1996

[2] Andrew W. Appel, "Foundational Proof Carrying Code", Princeton University

[3] Yier Jin and Yiorgos Makris, "Proof Carrying-Based Information Flow Tracking for Data Secrecy Protection and Hardware Trust" in IEEE 30th VLSI Test Symposium (VTS), 2012

[4] Eric Love and Yier Jin and Yiorgos Makris, "Enhancing Security via Provable Trustworthy Hardware Intellectual Property", Yale University, 2011

[5] Yier Jin and Yiorgos Makris, "Hardware Trojan Detection Using Path Delay Fingerprint", Yale University, 2011

[6] Aaron Bohannon, "Library Coq Intro", http://www.cis.upenn.edu/~plclub/popl08-tutorial/code/coqdoc/CoqIntro.html

[7] Christine Paulin-Mohring, "Introduction to the Coq Proof Assistant for Practical Softwate Verification", Univ. Paris-Sud