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

- 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

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. Consumer can easily and automatically check these proofs to make sure the code is valid. Better than many previous methods.

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

- Certification
- 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: 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.

- Verification-condition generator: Procedure that computes a predicate (safety predicate) in first-order logic based on the code to be certified.

Set of Axioms

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

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 r0.

Precondition

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

\[ P \rightarrow \text{true} \mid P_1 \land P_2 \mid P_1 \lor P_2 \mid \forall v_i \cdot P_i \mid e_1 = e_2 \mid e_1 \neq e_2 \mid v \in \tau \]

Where type is the typing predicate.
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.

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

\[
Pr_{\phi} = \neg \text{ro.addr} \land \neg \text{ro.addr} \land \text{sel}(\text{ro.addr}, r_0) \neq 0 \lor r_0 = \text{addr}
\]

This ensures 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”.

Certification involves generating safety predicates (VC Generator):

\[
\text{VC} = \begin{cases} 
0 & \text{if } r_0 = \text{addr} \land \text{sel}(\text{ro.addr}, r_0) \lor r_0 \text{ or } (r_0 = \text{addr} \land \text{wrd}(\text{ro.addr}, r_0)) \lor (r_0 < \text{addr} \land \text{wrd}(\text{ro.addr}, r_0)) \\
1 & \text{otherwise}
\end{cases}
\]

Which yeilds the final safety predicate:

\[
SP(\text{inc}, \text{Por}) = \text{VC} \lor \text{Por}
\]

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.

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:
  \[ \text{Inductive value} : = 0 | 1. \]
- One bit signal is treated as a 1-bit wide bus:
  \[
  \text{Definition bus_value} : = \text{list value}.
  \]
  \[
  \text{Definition bus} : = \text{nat} \rightarrow \text{bus_value}.
  \]
  \[
  \text{Definition bus_length} (b: \text{bus}) : = \text{Fun} \ t : \text{nat} \rightarrow \text{length} (b \ t).
  \]

Coq Signal Operations

- Bus handling methods include `and`, `or`, `xor`, equality operations, etc.
- Example of `and` operation:
  ```coq
  Fixpoint bit_and (a : \text{bit}; b : \text{bit}) :=
  match a with
  | \text{true} => \text{true}
  | \text{false} => b
  end.
  
  Definition bus_bit_and (a : \text{bus}; b : \text{bus}) :=
  \text{fun} \ t \rightarrow \text{bit_and} (a \ t) (b \ t).
  ```

Coq Expressions

- On top of signals definitions and operations expressions are built.
- Excerpt from the complete expression definition:
  ```coq
  Inductive expr :=
  | error : \text{expr} \rightarrow \text{expr}
  | cond : \text{expr} \rightarrow \text{expr} \rightarrow \text{expr}
  | expr : \text{expr} \rightarrow \text{expr} \rightarrow \text{expr}
  | and : \text{expr} \rightarrow \text{expr} \rightarrow \text{expr}
  | or : \text{expr} \rightarrow \text{expr} \rightarrow \text{expr}
  | xor : \text{expr} \rightarrow \text{expr} \rightarrow \text{expr}
  | permix : \text{expr} \rightarrow \text{expr}
  | sub : \text{bus} \rightarrow \text{bus} 
  ```

- The `and` 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 \( t \).

Final Coq Semantic Model

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

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

Coq Semantic Model Example

- Definition `dec` :=
  ```coq
  \text{Code} :=
  \text{dec}; \text{dec}
  ```
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.

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.

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

Bus definition extension:

- Inductive sensitivity := secure | normal.
- Definition bus := nat -> (bus value & sensitivity).
- Three propagation rules are then defined for tags:
  - Definition upTag (a : sensitivity) := a.
  - Definition bUpTag (a b : sensitivity) := match a with
  - | secure => secure
  - | normal => match b with
  - | secure => secure
  - | normal => normal
  - end.
- Definition rmTag (a : sensitivity) := normal

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.

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

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.

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

$S$ is the successor function (primitive recursive function)
- Say, for example $x \leftarrow x+1$
- 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.

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 $transfer : \forall (t \ n \ f : \text{nat}) \rightarrow$
  $\forall a : \text{nat}, \ a < \text{regs} \rightarrow \exists \text{nnat}, \ n > 0 \ \land \ n < \text{nf} \ \land \ \exists \text{Xnat},$
  $(\text{read} \ a \ n \ t \ X) \ \land \ \exists \text{nnat}, \ \text{nw} > 0 \ \land \ \text{nw} < \text{nf} \ \land \ \text{write} \ x \ \text{nw} \ t \ \text{RF1}.$

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

```verilog
assign we2 = (oprev & c) ? l'b1 : (c ? l'b0 : we2_in);
```

Hypothesis assign_we2 : (assign we2
  (cond (and (oprev oprev) (eqwe c))
    (eqwe NN))
  (cond (eqwe c) (eqwe (not (eqwe l'))))).
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 by 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