# Proof Carrying-Based Information Flow Tracking for Data Secrecy Protection and Hardware Trust

Yier Jin & Yiorgos Makris

## **Abstract.**

We discuss a new approach for protecting the secrecy of internal information in an Integrated Circuit (IC) from malicious hardware Trojan threats and, thereby, enhancing hardware trust. The proposed approach is based on Register Transfer Level (RTL) code certification and is built as an extension of a previously developed framework for trusted hardware acquisition through Proof-Carrying Hardware Intellectual Property (PCHIP). The key novelty lies in the introduction of a new semantic model for the Verilog Hardware Description Language (HDL) using the Coq theorem-proving environment, which facilitates tracking and proving secrecy labels of internal sensitive data and, by extension, security properties of the design. Additional framework enhancements include the ability to encapsulate sub-module properties in the top module proof environment, thereby strengthening the ability of Coq representation to reason on hierarchically organized RTL code. We demonstrate the proposed framework on a DES encryption core, wherein we employ it to prevent secret information (i.e. encryption key) leaking by hardware Trojans inserted at the RTL description of the circuit.

**I. Introduction**

Due to increasingly globalized IC supply chain, circuit designers and system integrators are highly liberated from the tedious work to design functional modules but to only focus on the development of high efficiency circuit architectures. However, the vast choice of intellectual property (IP) core supply breeds the security problem because it is impossible to have complete control on all IP core suppliers. Before arriving at the hand of system integrator, an IP core has travelled through many stages and is modified by various design houses. There are plenty of opportunities for attackers to insert malicious logic in the IP core along the IP transaction process. Such modifications, known as hardware Trojan, are purportedly done without the knowledge of the IP consumer. The additional functionality can be exploited by a perpetrator to cause catastrophic results if the IP core is used in mission-critical device.

Although researchers have already started working on the field of hardware trust and Trojan detection, most of previous work tries to detect hardware Trojan at the post-silicon stage and relies on ``golden models’’ to set the trusted boundary of chip’s side channel information [][][][][]. Far less is known or has been researched regarding this problem in pre-silicon stages to detect malicious modifications in IP cores [].

Until very recently, research on how to combat the threats of hardware Trojan on third party IP has emerged with the efforts to exploit software-based security approaches to the hardware trust domain, among which proof-carrying code (PCC) attracts most of the attention. PCC was first proposed by Necula et al. [ ] in 1996, which relies on a set of formalized properties to determine whether software code is safe to execute. The proof which is delivered with the original code acts as a certificate of the original code irrelevant to the (untrusted) source who provides the code.

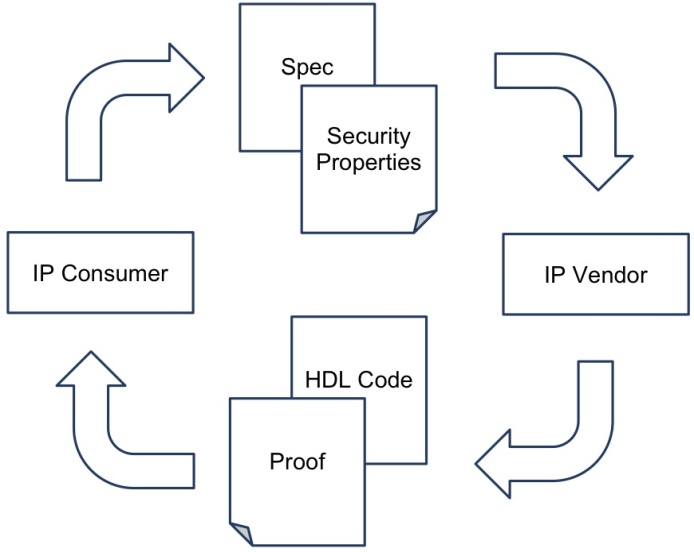
The first work to implement this methodology in hardware trust domain is proposed in [PCH]. The authors made a case for the necessity of Proof-Carrying Hardware (PCH) based on the increasing prominence of FPGAs and reconfigurable devices; if hardware itself becomes just as instantaneously reprogrammable as software, then the capability to establish the trustworthiness of unknown circuitry is inherently desirable. However, the method to only focus on the logic functions of the FPGA bitstream is limited by the need to specify exact Boolean functionality. And this work does not touch the security property checking at the RT-level.

In order to overcome the limitation and to fully leverage achievements in software PCC domain, other researchers try to expand PCH to encompass a more abstract notion of security-related properties. The authors in [PCHIP] present Proof-Carrying Hardware Intellectual Property (PCHIP) to guarantee proofs about a circuit’s HDL representation, rather than the FPGA bitstream. They also proposed a genuine IP design and acquisition protocol.

In this paper, we relied on the IP design and acquisition protocol in [PCHIP] and proposed a new semantic model in Coq proof assistant platform to better model RTL code structure in Coq. Besides the value, extra property was attached to all signals indicating whether the underlying signal contains sensitive data or not. This sensitivity property facilitates tracking and proving secrecy labels of internal sensitive data and, by extension, security properties of the design.

The remainder of this paper is organized as follows: in section \ref{sec:pchip}, we briefly introduce the IP design and acquisition protocol which our semantic model relies on. In section \ref{sec:model}, we provide details of the proposed new semantic model for Verilog code. Sensitivity property for information flow tracking and property transferring along module instantiation are described in section \ref{flow}. The implementation of our Coq framework on DES core is presented in section \ref{sec:des}. Conclusions are drawn in section \ref{sec:conclusion}.

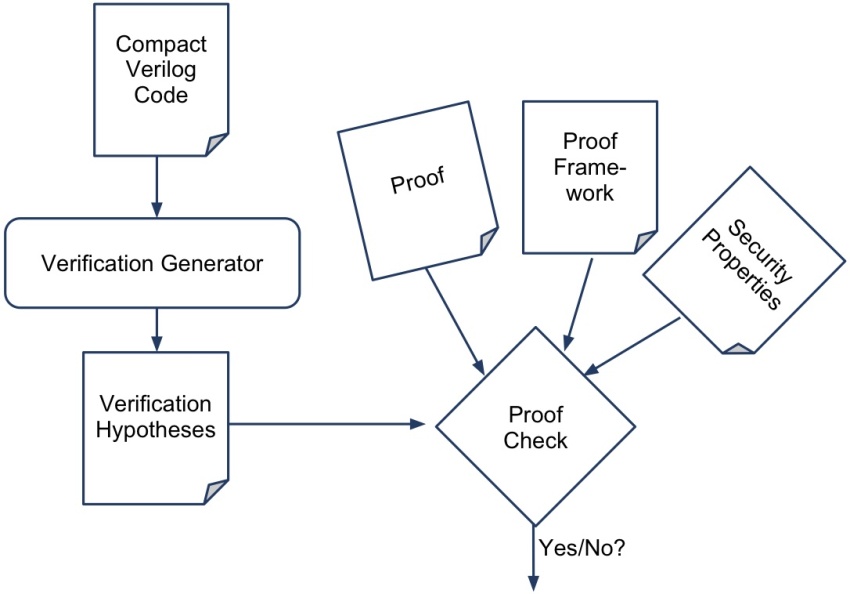
**II. IP Design and Acquisition Protocol**



*Figure 1. IP core design and acquisition protocol [ ]*

Figure 1 shows the outline of the IP core design and acquisition protocol. Different from normal IP transactions, under this protocol, hardware IP consumers and IP vendors should agree upon a set of formal security properties stated in formal logic besides the standard functional specification. These properties are not designed to depict the functionality of the design, but rather restrict the structure of the designed circuit so that IP consumers can be convinced that no malicious logics exist. It is then the IP vendor’s task to construct a formal proof that his module complies with these properties. The resulting proof is then transmitted back to the consumer along with the IP cores.

A formal semantics for a carefully codified syntax of an HDL (Hardware Description Language) using the Coq proof assistant is also proposed along this protocol. In the context of this semantic model, the authors in [] were able to then craft a temporal logic describing the behavior of signals in synchronous circuits. Security-related properties are then specified in this logic as trees of complex predicates and quantifiers.



*Figure 2. Automated design verification [PCHIP]*

Similar to the software PCC, the computation burden solely falls on IP vendors to write the proof and leaves IP consumer trivial workload to verify the proof. Figure 2 shows the automated design verification procedure based on the protocol at the IP consumer’s side. Upon receiving untrusted RTL code, IP consumers need to regenerate the Coq formal logic of the original circuit based on the Verilog-Coq conversion rules. The regenerated semantic description is then recombined with proofs and security properties, and is checked by the Coq interpreter. If the proofs are found to be valid, then the module is accepted as trustworthy.

**III. New Semantic Model for Verilog**

As mentioned in [PCHIP], a translation from RTL code to Coq representative is a necessity before defining and proving any secure properties of the underlying IP core. Different from the previously proposed translation rules which convert RTL code into its Coq counterpart with strict limitations, a new user-defined structural semantic model is proposed with much higher flexibility to model RTL circuit. Under this model, the architecture of the RTL circuit is accurately converted to Coq semantic description but designers (or proof writers) can adjust the way on how to translate the functionality of the original circuit. This structural semantic model clearly defines the role of hardware proof-carrying code and, for the first time, explicitly declares the differences between security property proving and functional testing. The success on implementing the new model also demonstrates that a non-accurate functional level translation from RTL code to Coq formal logic will not harm the process of proof-checking and property proving.

We choose Verilog as the HDL code and develop the new semantic model in Coq to represent the original circuit. Hereafter we’ll introduce the new semantic model step by step, from the preliminary definition for signals and their operation rules to the higher level syntax for the whole circuit.

**III.A Signal Definition**

Values of signals are defined in an inductive set with two constructors, *hi* and *lo* meaning the high voltage level and low voltage level in hardware circuit \footnote{As we only support synthesizable Verilog subset, we do not consider unknown and high resistance status.}. Instead of defining one-bit signals and multi-but buses separately, we unified both definitions under the bus scope, i.e., a one-bit signal will be treated as a one-bit width bus. The bus is then defined as a mapping of time, specified in clock cycles and given as a natural number, onto a *bus\_value* which is composed by a list of values. The natural number *t* defines an important property to be used in temporal logic, i.e., values of buses are vary according to system clock cycles. We also define a function to get the width of the bus, *bus\_length*.

*Inductive value := lo | hi.*

*Definition bus\_value := list value.*

*Definition bus := nat -> bus\_value.*

*Definition bus\_length (b : bus) :=*

*Fun t : nat => length (b t).*

**III.B Signals Operation**

Since any bus operations will finally be decomposed into the computation on values, we introduce definitions of basic operations on values first. Note that although Coq itself provides us a handful set of operations, in order to purify our Coq semantic model for RTL code, we use a bottom-up methodology to construct all necessary operations within Coq platform ourselves but rely little on the available operators. These basic operations include *not*, *and*, *nand*, *or*, *nor*, *xor*, *nxor,* etc with their definition showed below.

*Definition not (a : value) : value :=*

*Match a with*

*| lo => hi*

*| hi => lo*

*end.*

*Definition and (a b : value) : value :=*

*match a with*

*| lo => lo*

*| hi => match b with*

*| lo => lo*

*| hi => hi*

*end*

*end.*

*%Definition xor (a b : value) := match a with*

*% | lo => match b with*

*% | lo => lo*

*% | hi => hi end*

*% | hi => match b with*

*% | lo => hi*

*% | hi => lo end*

*% end.*

*……*

On top of these basic operations, we construct bus handling methods in the Coq model. These methods include logic operations such as *and*, *or*, *xor*, etc as well as bus comparisons like the checking of bus equality, *bus\_eq*, less-than comparison, *bus\_lt*, etc. Considering the frequent case in RTL code that signals are often compared with 0 to switch branches of if…else… statement, we add a special function to compare the bus value with 0, *bus\_eq\_0*.

*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).*

*Fixpoint bv\_bit\_or (a b : bus\_value) {struct a} : bus\_value :=*

*match a with*

*| nil => nil*

*| la :: a' => match b with*

*| nil => nil*

*| lb :: b' => (or la lb) :: (bv\_bit\_or a' b')*

*end*

*end.*

*Definition bus\_bit\_or (a b : bus) : bus :=*

*fun t:nat => bv\_bit\_or (a t) (b t).*

*……*

*Fixpoint bv\_eq (a b : bus\_value) {struct a} : value :=*

*match a with*

*| nil => hi*

*| la :: a' => match b with*

*| nil => hi*

*| lb :: b' => match (la, lb) with*

*| (lo, lo) => bv\_eq a' b'*

*| (lo, hi) => lo*

*| (hi, lo) => lo*

*| (hi, hi) => bv\_eq a' b'*

*end*

*end*

*end.*

*Definition bus\_eq (a b : bus) (t : nat) : value :=*

*bv\_eq (a t) (b t).*

*Fixpoint bv\_eq\_0 (a : bus\_value) {struct a} : value :=*

*match a with*

*| hi :: lt => lo*

*| lo :: lt => bv\_eq\_0 lt*

*| nil => hi*

*end.*

*……*

*Definition bus\_eq\_0 (a : bus) (t : nat) : value :=*

*bv\_eq\_0 (fst (a t)).*

……

**III.C Bus Slicing**

In hardware domain, operations are often performed on certain bits of the bus but not the whole bus. Similar to other hardware description languages, Verilog provides quite flexible syntax to define bus length and bus sequence, known as big ending and little ending. In order to convert all kinds of bus definitions into Coq formal logic, we developed two bus-slicing operations to locate data bits from big end and/or little end types of buses. To simplify the code writing and make the final converted logic easy to read, bits selection notations are also proposed. With the help of the simplification notation, it becomes much easier for IP consumers to understand the architecture of the converted Coq circuit model.

*Definition sliceA (b : bus) (p1 p2 : nat) : bus :=*

*fun t : nat => firstn (p2-p1+1) (skipn (p1-1) bv).*

*Definition sliceD (b : bus) (p1 p2 : nat) : bus :=*

*fun t : nat => rev (firstn (p1-p2+1) (skipn p2 (rev bv)).*

*Notation " b [ m , n ] " := (sliceD b m n ) (at level 50, left associativity).*

*Notation " b @ [ m , n ] " := (sliceA b m n ) (at level 50, left associativity).*

**III.D Expressions**

On top of the signal definitions and their operation rules, we build expressions to represent more complicated design logic. The expression is defined as an inductive set with constructors to construct new expressions or combine expression together. The procedure to convert Verilog expressions into Coq expression is similar but not equivalent to that of a Verilog compiler because only the structure of the expression is precisely modeled. It’s solely on the designer’s discretion whether the Coq expressions should be functionally the same as RTL code.

*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*

*| eq : expr -> expr -> expr*

*……*

The expression definition showed above is an exempt from the full expression definition suitable for encryption IP cores. A constant value list and a bus can be directly converted to expressions under *econv* and *econb* constructors, respectively. The *eand*, *eor* and *exor* constructors connect two expressions to form a new expression. We will find shortly in the definition of the semantic model that these three constructors perform logical AND, OR and XOR operations (we can also guess their meanings from titles of constructors). The *enot* constructor only converts one expression to another performing logical NOT. The conversions of these operations retain both structure and function of the Verilog code.

On the other hand, the *perm* and *sbox* constructors are used to indicate permutation and S-box mapping operations. But since there are various ways to perform permutation and S-box mapping, it becomes impossible to list all of the operation types in the semantic model (without knowing the bus length and S-box sizes). So the perm and sbox constructors do not specify what permutation and/or S-box mapping are actually performed. These structural constructors liberate us from tedious functional conversion which is unnecessary for properties proving and leave the testing of function to functional testing.

The semantic model for expression is recursively defined to calculate the value of expressions at a specified time (denoted by *t* parameter) and return data of type *bus\_value*, a list of values whose length depends on the width of the underlying bus. Details of the *eval* function confirm our claim that although expressions like *eand* performs logical AND on two sub-expressions, a case that Coq representative is both functional and structural, other expressions like perm is only a structural mapping from RTL code to Coq representative.

*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)*

*| exor ex1 ex2 => bv\_bit\_xor (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*

*| eq ex1 ex2 => match (bv\_eq (eval ex1 t) (eval ex2 t)) with*

*| hi => hi :: nil*

*| lo => lo :: nil end*

*……*

**III.E Coq Circuit Model**

The definition of signals, expressions and their semantic model paves the way to finally represent Verilog code in Coq theorem-proving language. Since there is no restrictions on how to convert the Verilog code into Coq representative, any notations can be used in Coq mapping to RTL code. For example, authors in [PCHIP] mimic a Verilog compiler to construct the parse tree of the original code. However, the converted Coq code with a parse tree-alike architecture is too complex to follow such that it is difficult to verify the correctness of the Coq circuit model.

In order to make the Coq circuit model easy to read, we set two goals when choosing notation marks: 1) the conversion process should be a one-to-one mapping which facilitates the implementation of automation tools; 2) the Coq circuit model should be concise and easy to understand (at least as easy as reading Verilog code). Guided by these two goals, a new syntactic set for Coq model is then proposed to convert Verilog code into Coq circuit models which is showed below. An extra notation is added to pile the code through the ‘;’ symbol. The selection of ‘;’ mark is consistent with Verilog syntax.

*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).*

The constructor *outb* is used to denote output signals of the module. Similarly, *inb* means input signals; *wireb* represents internal wire signals; and *regb* denotes the internal registers (note that in Verilog code, the reg type is not necessary synthesized to be register in netlist so the regb here is only used to mapping the reg declaration of the original code). Two assignment constructors are defined where the *assign\_\** works for combinational logic and *nonblock\_assign\_\** is for the non-blocking assignment in sequential logic.

**IV. Module Instantiation and Information Flow**

**IV.A Information Flow**

Although the Coq semantic model introduced in Section III can help to convert Verilog code into Coq representatives and prove certain properties similar to those in [PCHIP], this model can do little to prevent information leaking if hardware Trojans are inserted because all properties within the domain of this semantic model can only check the integrity of the circuit architecture. Hardware Trojans targeting information leaking often use signal bypassing strategy [] trying not to touch original circuit but only propagating internal sensitive data to primary output [] or through Trojan side channels []. The construction of property tracking internal information flow will be rather difficult, if not impossible, based on this basic semantic model. In order to better serve our purpose that the new semantic model should facilitate tracking and proving secrecy labels of internal sensitive data to test the signal integrity of the whole design, we then modified the basic model by granting another property, sensitivity, to internal signals besides the already granted value property. The bus is then redefined as a *value\*sensitivity* pair at a specified time *t*. The sensitivity is defined as an inductive set with two constructors, *secure* and *normal*, indicating whether the signals are sensitive and need protection or not. More precisely, a bus with the secure tag is not allowed to be propagated to primary output or Trojan side channels.

*Inductive sensitivity := secure | normal.*

*Definition bus := nat -> (bus\_value \* sensitivity).*

Now that each bus has a sensitivity tag, we need to define the propagation rules for those tags through buses operation, similar to the computational rules we defined for bus values. Three operation rules are defined.

*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.*

The *uoptag* function deal with the case when the bus is an operand of a unary operator and, as the definition indicated, the unary operator reserves the sensitivity tag of the bus. The definition for binary operator, *boptag*, is similar to OR logic where the output signal is of *secure* tag as long as one of the input signal is *secure*. The only way to change the *secure* tag back to *normal* tag is through *rmtag* function. As we will find shortly, permutation is the only legal operation calling *rmtag* function to remove *secure* tag from buses in the example of DES core. To grant very few operations the ability of removing secure tag matches our requirement to prevent any leakage of internal sensitive data.

The redefinition of bus, as a natural consequence, alters all other functions and operations with bus parameters and/or with output in bus type but, fortunately, the updating of the semantic model is straightforward and easy to achieve. For example, bus slicing operation is modified to only consider two sensitivity status cases of the original bus signals.

*Definition sliceA (b : bus) (p1 p2 : nat) : bus :=*

*fun t : nat => match (b t) with*

*| (bv, secure) => (firstn (p2-p1+1) (skipn (p1-1) bv), secure)*

*| (bv, normal) => (firstn (p2-p1+1) (skipn (p1-1) bv), normal)*

*end.*

*Definition sliceD (b : bus) (p1 p2 : nat) : bus :=*

*fun t : nat => match (b t) with*

*| (bv, secure) => (rev (firstn (p1-p2+1) (skipn p2 (rev bv))), secure)*

*| (bv, normal) => (rev (firstn (p1-p2+1) (skipn p2 (rev bv))), normal)*

*end.*

More importantly, the *eval* function which calculates the value of expressions at a specified time will not just return the calculated bus value but its sensitivity.

*Fixpoint eval (e : expr) (t : nat) {struct e} : bus\_value\*sensitivity :=*

*match e with*

*| econv v => (v, normal)*

*| econb b => b t*

*| eand ex1 ex2 => (bv\_bit\_and (fst (eval ex1 t)) (fst (eval ex2 t)), boptag (snd (eval ex1 t)) (snd (eval ex2 t)))*

*| eor ex1 ex2 => (bv\_bit\_or (fst (eval ex1 t)) (fst (eval ex2 t)), boptag (snd (eval ex1 t)) (snd (eval ex2 t)))*

*| exor ex1 ex2 => (bv\_bit\_xor (fst (eval ex1 t)) (fst (eval ex2 t)), boptag (snd (eval ex1 t)) (snd (eval ex2 t)))*

*| enot ex => (bv\_bit\_not (fst (eval ex t)), uoptag (snd (eval ex t)))*

*| cond cex ex1 ex2 => match (bv\_eq\_0 (fst (eval cex t))) with*

*| hi => eval ex1 t*

*| lo => eval ex2 t end*

*| perm ex => (fst (eval ex t), normal)*

*| sbox b => (b t)*

*| eq ex1 ex2 => match (bv\_eq (fst (eval ex1 t)) (fst (eval ex2 t))) with*

*| hi => (hi :: nil, normal)*

*| lo => (lo :: nil, normal) end*

*……*

Finally, a code sensitivity checking function, *chk\_code\_sen*, is proposed to evaluate the sensitivity of the Coq circuit model and return a sensitivity tag, *secure* or *normal*, indicating the sensitivity of all input, output and internal signals. A return value of *secure* means the Coq circuit model under test has security problem by either leaking internal sensitive information to primary output or to Trojan side channel. More work will be done to locate the malicious logic if the return of the sensitivity checking function is *secure*. A return value is *normal* tells the IP consumer that the code passes the information leaking security testing.

*Fixpoint chk\_code\_sen (c : code) (t : nat) : sensitivity :=*

*match c with*

*| assign\_ex b ex => expr\_sen ex t*

*| assign\_b b1 b2 => bus\_sen b2 t*

*| assign\_case3 b ex => expr\_sen ex t*

*| nonblock\_assign\_ex b ex => expr\_sen ex t*

*| nonblock\_assign\_b b1 b2 => bus\_sen b2 t*

*| module\_inst2in bout b1 b2 => normal*

*| module\_inst3in bout b1 b2 b3 => normal*

*| codepile c1 c2 => boptag (chk\_code\_sen c1 t) (chk\_code\_sen c2 t)*

*end.*

**IV.B Module Instantiation**

Modern circuit designs often implement hierarchical architecture to better manage the integration of IP cores into large systems and SoC design. The hierarchical architecture simplifies the testing of circuit designs but raises a challenge in the domain of proof-carrying hardware IP, i.e., the way of transferring security property from lower level modules to higher level module at the interface of module instantiation. In cases that the security property we need to prove for modules at different levels are the same, e.g., preventing information leaking, we proposed a bottom-up property transferring solution to solve this problem. The procedure to perform this bottom-up security checking on hierarchical system is listed below:

(1). Check the security of modules at the bottom level.

(2). Move to modules in higher level. Redefine the sensitivity tags of internal signals transferring information between current module and modules at lower level. Check the security of the module.

(3). Repeat Step (2) until we move to the top level module.

The example of DES core showed below demonstrates the effectiveness of this method to ensure the security property integrity on hierarchical designs.

**V. DES Example**

In order to demonstrate the capability of the proposed semantic model on protecting internal sensitive information and prevent secret information (i.e., encryption key, plaintext) leaking by hardware Trojans inserted at the RTL description of the circuit, we employ DES core [DES] as our sample circuit. We argue that this is the first time the concept of proof-carrying code being used on a cryptography designs. In this example we will show how the Verilog code is converted into Coq formal logic, how the proof is written on the security property chosen to prevent information leaking and how the proof checker detect the inserted bypassing hardware Trojans. Of particular noteworthiness in this example is the similarity between the formal logic and its Verilog counterpart. It leads us to the possibility that we can design a new hardware description language (HDL) on formal logic in Coq (or any other proof assistant languages) to be an alternative to Verilog, or VHDL on designing IP cores. An automatic properties extraction will be a universal solution to protect IP core transactions. But we should admit that more work needs to be done on this direction.

**V.A Formal Logic of DES Core**

The architecture of the DES core is showed in Figure xx, which shows that the top module, des.v, instantiates two sub-modules, crp.v and key\_sel.v, to perform round encryption/decryption and round keys generation, respectively.

Figure 1. TBD.

In Appendix A, we list the Verilog code of des (top) module and the converted Coq representative. We can easily make a conclusion that the new syntax of formal logic is quite similar to the Verilog syntax. Considering another fact that the semantic model of the formal logic is also similar to that in Verilog, we argue that the proposed framework is suitable for IP designers with hardware background but of little knowledge of software foundation.

**V.B Security Property and Proof Writing**

Among the hardware Trojans detection/prevention methods proposed recently in the field of trusted IC, many researchers expressed their concerns on information leaking through primary output or Trojan side channels on cryptography chips. So the security property we are interested in the DES core is the prevention of leaking information and we achieve this goal through a proof-carrying-based information flow tracking. From Figure xx, we know that the top des module instantiates a round encryption/decryption module and a key generation module. Using the methodology proposed in Section VI.B, we prove the security property for these two modules first \footnote{In Verilog code, the round encryption/decryption module instantiates another eight S-box mapping modules to finish the S-box operations. But since we don’t need to convert the functional S-box operation in formal logic, all we need in our model is the S-box structure. So we add a sbox constructor when defining code type. As a result, all eight S-box mappings are replaced by sbox structures.}

For the key generation module, after defining all input, output, wire, and reg type signals, we propose an axiom to explicitly declare that the sensitivity tag of input K is *secure* at any time.

*Variables K\_sub K roundSel K1 K2 K3 K4 K5 K6 K7 K8 roundSelH : bus.*

*Variables decrypt decryptH : bus.*

*Axiom secret\_K : forall (t : nat), bus\_sen K t = secure.*

Part of the Coq formal logic of key generation module is presented below:

*Definition key\_selh : code :=*

*outb K\_sub;*

*inb K;*

*inb roundSel;*

*inb decrypt;*

*wireb K1;*

*wireb K2;*

*……*

We then prove the property theorem that the sensitivity status of the key\_selh module is normal even though the input signal K is secure.

*Theorem no\_leaking\_key\_selh : forall (t : nat), chk\_code\_sen key\_selh t = normal.*

Similarly, for the round encryption/decryption module, the round keys are declared to be of secure tag in any time.

*Variables P R K\_sub : bus.*

*Variables E X S : bus.*

*Axiom secret\_K\_sub : forall (t : nat), bus\_sen K\_sub t = secure.*

The secure property we need to prove is of similar style.

*Theorem no\_leaking\_crp : forall (t : nat), chk\_code\_sen crp t = normal.*

After we prove the security property of two sub-modules, we move one level higher at the hierarchical ladder to the top module. According to the methodology in Section IV.B, we re-declare the sensitivity status of internal signals to include all sensitive data into protection, no matter these signals are from primary inputs or sub-modules (even though the sub-module itself has been proved to fulfill security property). For example, the output of key generation module, *K\_sub*, is declared as a secure signal at the top level. Other secure signals include plaintext, key.

*Axiom secret\_key : forall (t : nat), bus\_sen key t = secure.*

*Axiom secret\_desIn : forall (t : nat), bus\_sen desIn t = secure.*

*Axiom secret\_K\_sub : forall (t : nat), bus\_sen K\_sub t = secure.*

With these signals being secure, we demonstrate that the DES core top module does not allow any information leaking.

*Theorem no\_leaking\_crp : forall (t : nat), chk\_code\_sen des t = normal.*

**V.C Hardware Trojan Detection**

In order to demonstrate the effective of the proposed framework in preventing information leakage, we inserted a malicious key-bypassing logic to leak the internal key to primary output when the Trojan is triggered. Reflected in the formal logic, statement

*assign\_ex desOut (econb key)*

is replaced by the malicious one

*assign\_ex desOut (cond (eq (econb roundSel) (econv (lo::lo::lo::lo::nil))) (econb FP) (econb key))*. Not surprisingly, the modified code cannot pass the proof check with error message

*“Error: Impossible to unify "normal" with*

*"match snd (FP t) with*

*| secure => secure*

*| normal => normal*

*end".”*

Thus we can declare that the implementation of the proposed information leaking prevention framework in Coq platform can protect cryptographic circuits from Trojan attacks and ensure the integrity of sensitive signals.

**VI. Conclusion**