# 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**

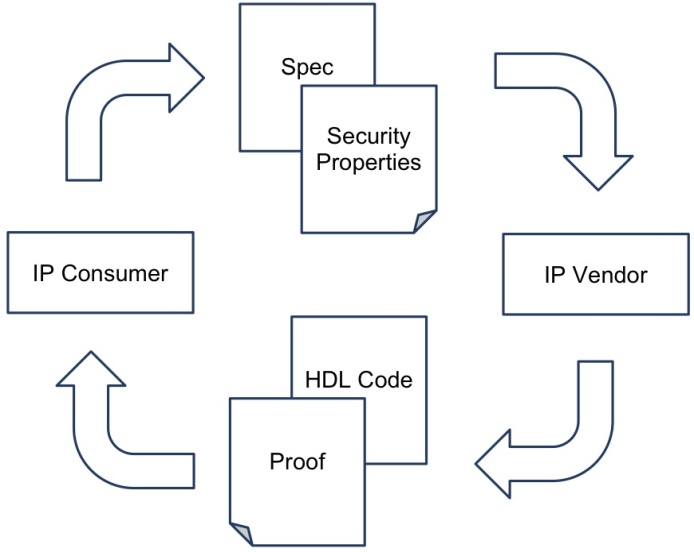
**II. PCHIP**

Recently, research on how to combat the threats of hardware Trojan on third party IP has emerged. Pertinent efforts attempt to exploit a well-developed body of work in the software domain, known as proof-carrying code (PCC). Originally developed by Necula et al. [16] in 1996, PCC provided a new way of determining whether code from a potentially untrusted source is safe to execute. This was accomplished by establishing a formal, automatically verifiable proof that the questionable code obeys a set of formalized properties.

The idea of expanding this methodology to the hardware trust domain first appeared 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.

This work represented a first step towards proof-based security. It is clear that many types of Trojans could be prevented under such a system because modifications to the combinational behavior of an FPGA bitstream’s logic functions would be immediately detectable. Nevertheless, the expressiveness of this approach is limited by the need to specify exact Boolean functionality. In software PCC, security policies have generally specified a broader definition of “safe” behavior without necessarily stipulating precisely what a program must compute.

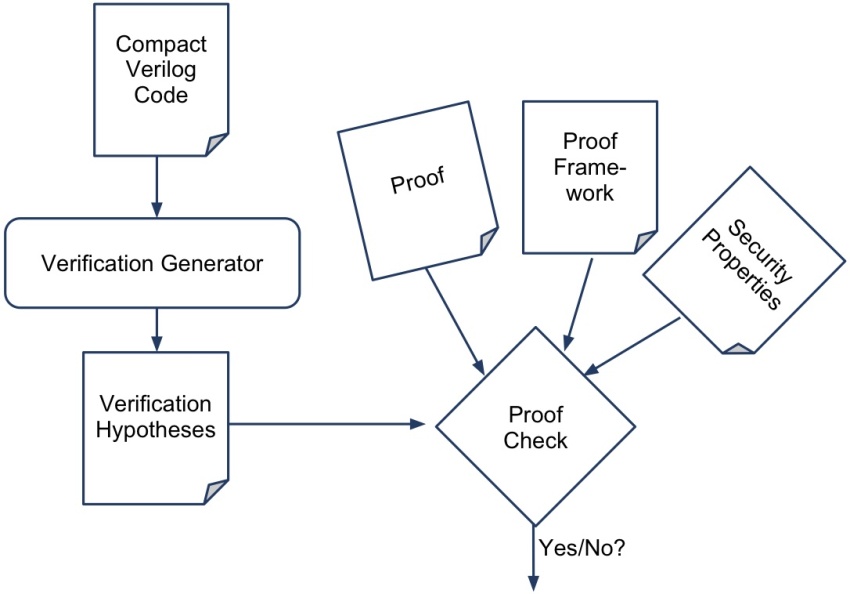
In response to this limitation, other researchers have sought ways of expanding PCH to encompass a more abstract notion of security-related properties. The authors in [PCHIP\_Eric] present Proof-Carrying Hardware Intellectual Property (PCHIP) to guarantee proofs about a circuit’s HDL representation, rather than the FPGA bitstream.



*Figure xx. IP core design and aciqusition protocol [ ]*

PCHIP introduces a new protocol, shown in Figure xx, for the design and acquisition of hardware intellectual property IP cores. Under this system, a hardware IP consumer commissions a vendor to construct a module according to both a standard functional specification and a set of formal security properties stated in a temporal logic. These properties are markedly different from the specification in that they do not necessarily describe the functional behavior of the module; rather, they delimit the acceptable boundaries of this behavior so that the consumer can rest assured no undesired functionality exists. It is then the IP vendor’s task to construct a formal proof that his module complies with these properties. Just as in PCC and PCH, the resulting proof is then transmitted back to the consumer along with the IP cores.

Unlike PCH, however, the properties allowed in PCHIP are much more abstract. In [ ], the authors described a formal semantics for a carefully codified syntax of an HDL (Hardware Description Language) using the Coq proof assistant [ ]. In the context of this semantic model they were able to then craft a temporal logic describing the behavior of signals in synchronous circuits. Security-related properties could be specified in this logic as trees of complex predicates and quantifiers, as opposed to the simpler Boolean function specifications allowed in PCH. PCHIP provides a set of rules to be applied to HDL code in order to generate a set of propositions in Coq that represent that code’s behavior in the already established semantic model. These propositions are then referred to in the proofs that must be constructed for each security-related property.



*Figure xx. Automated design verification [ ]*

Figure xx shows how proof-checking proceeds when the consumer receives a circuit’s IP code and its corresponding Coq proofs. The code is first passed through a “verification generator” to regenerate all the propositions describing the circuit’s behavior, since it is not known whether those included in the received proof actually match the vendor’s untrusted code. However, because they are generated according to the same set of rules on both the vendor’s side and the client’s, it is guaranteed that the resulting propositions will also be the same. Were this not so, then proofs would not be able to refer to them consistently. 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. Revised Semantic Model for Verilog**

As mentioned in [], 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 framework which will convert RTL code into its Coq counterpart in a functional level, the revised semantic model is a user-defined structural model. Under this model, the architecture of the RTL circuit will be precisely converted to Coq code but designers (or proof writers) can decide the way to translate the functionality of the original circuit. The key point here is that the proof code is used to prove pre-defined property of the RTL circuit but not to replace functional testing. A non-accurate functional level translation from RTL code to Coq code won’t harm the process of proof-checking.

Again, we choose Verilog as an example of the HDL code and develop the new semantic model in Coq to present the original circuit. Hereafter we’ll introduce the new semantic model in three steps, from the basic definition of signal and its operation to the highest level, circuit definition.

**III.A Signal Definition**

We first define a signal value as an inductive set with two constructors, hi and lo to represent the high voltage and low voltage of the hardware circuit. Instead of defining one-bit signals and multi-but buses, we unified both definitions under the bus domain, i.e., a one-bit signal will be treated as a one-bit width bus. In order to deal with synchronized sequential logic, the bus is 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 width of a specified bus is the same as the width of the value list through the bus\_length function we defined.

*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**

We start this section with definitions of basic operations on values. Although Coq provides us a handful set of operations, in order to purify our Coq representative of RTL code, we use a bottom-up methodology to construct all necessary operations within Coq platform. These basic operations include not, and, nand, or, nor, xor, nxor, etc.

*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 the basic operations, we construct bus handling methods at RT-level. These methods include logic operations such as and, or, xor, etc as well as bus comparison like bus\_eq, bus\_lt, etc. Considering the frequently happened case in RTL code that signals are often need to compare with 0 as a condition for if…else… instruction, we only define a special function to compare the bus value with 0, called 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**

Different from software signal handling, in hardware domain, we often need to run operation on certain bits of the bus but not the whole bus bits. Even worse, Verilog does not have strict rule on how to define the ending of a bus, either big ending or little ending. In order to deal with both cases and ensure the structure of the converted circuit is the same as original circuit in RTL code, we developed two bus-slicing operations to locate data bits from big end and little end buses. To simplify the code writing, bus bits notations are also proposed. The notations can also make it easy for users to understand the circuit architecture in Coq platform.

*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 signals and their operation rules, we build expressions to represent general operations in a hierarchical way. These expressions are also defined as constructors of an inductive set, similar but not equivalent to the parse tree generated by a Verilog compiler because only the circuit structure will be precisely represented as a network of symbols. It’s solely on the designer’s discretion whether the denoted expressions should be the same as RTL code at the functional level.

*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 example of Coq representative for block encryption IP cores. A constant bus value and a bus are converted to expressions with econv and econb, respectively. The eand, eor and exor constructors take two expressions as inputs and return one expression as an output and perform logical AND, OR and XOR operations (which will be explicitly defined by the semantic model). While the enot construct only takes one expression and return one expression, performing the logical NOT operation. The perm and sbox constructors are used to present permutation and S-box mapping. As we will also find in the semantic model below, there are various ways to perform both permutation and S-box mapping, so we do not try to cover all possible types of bus permutation or S-box mapping, but only use structural models, perm and sbox, to indicate that permutation operation and S-box mapping will be implemented. The structural constructors liberate us from tedious functional conversion which is unnecessary for properties proving and leave the testing of function to functional testing [].

The evaluation function eval is then defined to calculate the value of recursively defined expr types at a specified time (restricted by t parameter) and return the data of type bus\_value, a list of values whose length depends on the width of the bus. The eval function is also known as the operational semantics of expressions. Although expressions like eand performs logical AND on two sub-expressions, a case that Coq representative is both functional and structural the same as RTL code, 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 Representatives (Code)**

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 [] mimic a Verilog compiler to construct the parse tree of the original code. However, the converted Coq code is so complex that it is hard to follow the code and to understand its behavior. In this paper, we set two goals before developing the Coq representative: 1) the conversion process is an easy one-to-one mapping which facilitates the implementation of automation tools; 2) the Coq representative should be concise and easy to understand (at least as easy as reading Verilog code). A new syntactic set for Coq code is then proposed to represent all Verilog code in a Verilog-similar way, as showed below. A further 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).*

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). Since signal assignment can be divided into combinational logic and sequential logic, two assign constructors are proposed. The assign\_\* works for combinational logic while nonblock\_assign\_\* is for the sequential cases.

**IV. Module Instantiation and Information Flow**

**IV.A Information Flow**

Although the semantic model for Verilog introduced in Section III can help to convert Verilog code into Coq representatives and prove certain properties similar to those in [], this model can do little to prevent information leaking through added hardware Trojans because all properties within the domain of this basic model will 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 channel []. Furthermore, to construct the property tracking internal information flow will be rather difficult, if not impossible, based on the 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 on top of their value property. The bus is then redefined as a value\*sensitivity pair at a specified time. The sensitivity is defined as a inductive set with two constructors, secure and normal, indicating whether the signals are sensitive or not. More precisely, a bus with the secure tag is not allowed to be propagated to primary output or Trojan side channel.

*Inductive sensitivity := secure | normal.*

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

Since the tag will flow inside the chip along bus values, we also need to define the propagation rules for sensitivity tags, similar to the computational rules we defined for bus values. Three kind of operations 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, unary operator reserves the sensitivity tag of the bus. The definition for binary operator is similar to OR operation where the output signal is of secure tag as far as one of the input signal is secure. The only way to change the secure tag back to normal tag is through rmtag operation. As we will find shortly, permutation is the only legal operation using rmtag to remove secure tag on internal buses in the example of DES core. To grant very few operations the ability to remove secure tag fulfills our requirement to prevent any leakage of internal sensitive data.

As a natural consequence, the redefinition of bus alters all other functions, operations with bus parameters and/or with output in bus type but, fortunately, the updating is quite straightforward. For example, below is the bus updated bus slicing operation where we only need to 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 to calculate 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 is proposed to evaluate the sensitivity of the Coq representative (no matter the length of the code) and return a sensitivity constructor, secure or normal, indicating the security of the whole code. If the return value is secure, the code under test has security problem by either leaking internal sensitive information to primary output or to Trojan side channel. If the return value is normal, the code passes the information leaking security testing. Details of the chk\_code\_sen function are easy to understand since it only scrutinizes the structure of the target code and uses sensitivity computation rules to calculate the status.

*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**

Almost all the modern circuit designs implement hierarchical architecture to better manage the integration of IP cores into large systems. The hierarchical architecture raises a challenge in the domain of proof-carrying hardware IP, i.e., how to switch security property at the module instantiation interface at the higher level. We give one solution here in a bottom-up way which will be valid only in cases where the security property we need to prove for modules at different levels are the same, e.g., preventing information leaking.

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). Go to the second to the last level and redefine the sensitivity status of internal signals transferring information between current module and modules at lower level. Check the security of the module.

(3). Repeat Step (2) until the top module.

The example on 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 [] 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 formal logic, how the proof is written on the security property chosen to prevent information leaking and, more importantly, 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 a new formal logic based on Coq (or any other proof assistant languages) may be proposed to replace Verilog, or VHDL on designing IP cores. An automatic properties extraction will be a universal solution to protect IP core transactions. We also admitted that more work needs to be done on this direction.

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

The architecture of the DES core written in Verilog [] is showed in Figure xx, from which we can find 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 its Coq representative. From which we can easily find that the new syntax of formal logic is quite close 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, especially PCC.

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

Among the hardware Trojans 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 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 in 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 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 whole key\_selh block 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))*. Non-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 sensitive signal integrity.

**VI. Conclusion**