Toward Automatic Proof Generation for Information Flow Policies in Third-Party Hardware IP

Mohammad-Mahdi Bidmeshki and Yiorgos Makris
Department of Electrical Engineering, The University of Texas at Dallas
Email: {bidmeshki, yiorgos.makris}@utdallas.edu

***Abstract*—The proof carrying hardware intellectual property (PCHIP) framework ensures trustworthiness by developing proofs for security properties designed to prevent introduction of malicious behaviors via third-party hardware IP. However, converting a design to a formal representation and developing proofs for the desired security properties is a cumbersome task for IP developers and requires extra knowledge of formal reasoning methods, proof development and proof checking. While security properties are generally specific to each design, information flow policies are a set of policies which ensure that no secret information is leaked through untrusted channels, and are mainly applicable to the designs which manipulate secret and sensitive data. In this work, we introduce the *VeriCoq-IFT* framework which aims to (i) automate the process of converting designs from HDL to the Coq formal language, (ii) generate security property theorems ensuring information flow policies, (iii) construct proofs for such theorems, and (iv) check their validity for the design, with minimal user intervention. We take advantage of Coq proof automation facilities in proving the generated theorems for enforcing these policies and we demonstrate the applicability of our automated framework on two DES encryption circuits. By providing essential information, the trustworthiness of these circuits in terms of information flow policies is verified automatically. Any alteration of the circuit description against information flow policies causes proofs to fail. Our methodology is the first but essential step in the adoption of PCHIP as a valuable method to authenticate the trustworthiness of third party hardware IP with minimal extra effort.**

I. Introduction

Financial and other market realities have made integrated circuit (IC) design and manufacturing a highly geographically dispersed endeavor. As such, contemporary ICs have also become more vulnerable to inconspicuous modifications resulting in the proliferation of malicious capabilities known as hardware Trojans. While IC designers and users are unaware of the existence of such capabilities, knowledgeable adversaries can take advantage of them in order to stage attacks. The presence of technology in every aspect of our everyday life creates more opportunities for the adversaries and can make the outcome of such attacks, utilizing hardware Trojans, disastrous. Therefore, many research efforts have been devoted to preventing and/or detecting hardware Trojan inclusion in various phases of IC design and fabrication [1], [2].

Utilization of previously developed designs in the form of hardware IPs (in-house or third-party) is commonly practiced in the hardware design flow, in order to enhance time-to-market of the final product. Soft IPs, delivered in the form of HDL code, are more susceptible to malicious modifications and hardware Trojan insertion due to their flexibility and the fact that functional testing can by no means reveal the design capabilities exhaustively. Moreover, soft IPs are also widely used in FPGA-based designs. Consequently, hardware Trojans concealed in soft IPs have a significantly wider domain of action, as compared to hardware Trojans which are implanted during the later fabrication stages. Considering this intensified threat, prevention and/or detection of hardware Trojans in soft IPs has become extremely important.

Various methodologies have been introduced for identifying hardware Trojans in soft IPs at the design stage. Two examples are FANCI [3] and VeriTrust [4]. Although such methods are systematic, smart Trojan designs can still evade their checking mechanisms [5]. Along a different direction and utilizing formal methods and mathematical theorems, a proof-carrying hardware intellectual property (PCHIP) [6], [7], [8], [9] framework was proposed for trusted 3rd party IP acquisition. To prevent the insertion of hardware Trojans in a design, in the PCHIP framework, which is based on the Proof-Carrying Code (PCC) principles [10], formal proofs that a given IP abides by a set of security properties are developed. IP consumers receive a bundle containing not only the HDL code but also the proofs for these security properties, and can then automatically check that the provided proofs are actually valid for the acquired HDL code.

Despite its tremendous potential, PCHIP faces a few challenges for its broad deployment. First, developing security properties is not straightforward. While a few have been proposed for cryptographic hardware [8], [9] and microprocessors [7], security properties are usually specific to each design, and cannot be reused for other types of designs. Second, converting HDL code to a formal representation, such as the Coq [11] language used in PCHIP, and developing proofs for security properties, requires additional knowledge of formal methods, theorem proving environments, and proof writing. Even for someone that has this expertise, the process is tedious and time-consuming, which makes IP developers hesitant to adopt PCHIP.

Cryptographic hardware plays an important role in many applications such as data transmission and storage. For such hardware designs, the PCHIP framework has been employed to enforce information flow policies [8], [9], where instead of focusing on functional details, it assigns tags to signals in order to track them throughout the design, and ensures that no sensitive information is leaked through the primary outputs. However, this approach still requires tremendous manual work, including conversion to Coq representation, generation of security theorems and development of proofs of such theorems. Evidently, automating the PCHIP framework to the extent possible could make it far more appealing and could help in its broader utilization, leading to lower risk in hardware IP acquisition. Toward this goal, in this paper, we

978-1-4673-7421-7/15/$31.00

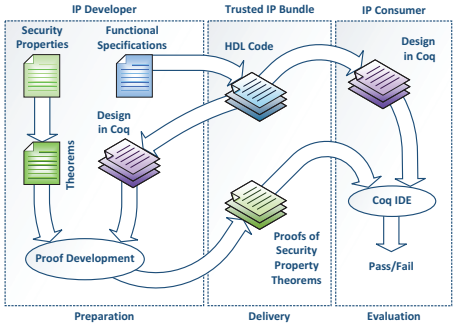

Fig. 1. PCHIP framework

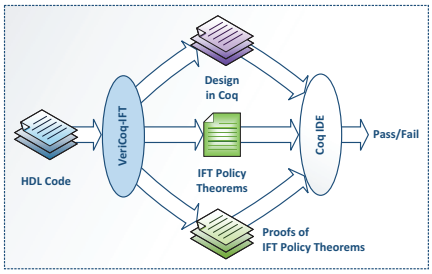

Fig. 2. Automated PCHIP framework for information flow policies

introduce *VeriCoq-IFT*, an automated PCHIP framework to enforce information flow policies which includes converting the design from HDL to the Coq formal language, generating security property theorems ensuring information flow policies, constructing proofs for such theorems, and checking their validity for the design, with minimal user intervention. We demonstrate the applicability of *VeriCoq-IFT* on two different implementations of the data encryption standard (DES) [12]. By providing essential information, the trustworthiness of these circuits in terms of information flow policies is checked automatically. *VeriCoq-IFT* gathers needed information through special comments (pragmas) inserted in the HDL code by the IP designer. IP consumers, on the other hand, only need to check the validity of these special comments and utilize *VeriCoq-IFT* to assess the trustworthiness of the acquired design.

The rest of this paper is organized as follows. Section II reviews the PCHIP framework and introduces its fundamental principles. In Section III, we illustrate the details of the *VeriCoq-IFT* framework including conversion to Coq, generation of security property theorems and construction of proofs. Section IV demonstrates the capabilities of *VeriCoq-IFT* on two different implementations of the DES encryption algorithm and Section V concludes the paper.

II. Poof-Carrying Hardware IP (PCHIP) Overview

In this section, we briefly review the PCHIP framework, which is depicted in Fig. 1. In this framework, along with the HDL code for a design, IP developers are required to develop and deliver another essential piece: formal proofs that the code abides by a set of security properties that are agreed upon by both the IP developer and the IP consumer. These properties do not necessarily impose restrictions on the details of implementation. Rather, they institute a high level boundary of trusted functionality, which prevents misbehavior or unsolicited actions. For example, a security property for a microprocessor IP could be defined as follows: Each instruction is only allowed to access memory locations which are specified in the corresponding fields of its op-code [7]. This property prevents stealthy information leakage. However, it does not restrict the details of instruction implementation.

Mechanized proof development and checking requires a theorem proving language and proof checking environment, such as Coq and CoqIDE, respectively. Therefore, in order to be applicable and leverage the rich collection of hardware IPs developed in HDLs such as Verilog and VHDL, PCHIP defines conversion rules from HDLs to a Coq representation. Consequently, PCHIP does not intervene in the current hardware IP design and test methodology, as is the case when introducing a new formal HDL [13]. Rather, it adds extra steps in parallel to the current design methodology, namely conversion to Coq, stating security properties as theorems in Coq, constructing proofs for such theorems based on the hardware design and delivering those proofs along with the HDL code to the IP consumer. PCHIP does not inflict IP consumers with much extra burden. Along with the IP developers, they need to agree on the desired security properties. The onerous task of proof development is, then, the responsibility of the IP developers.

PCHIP can be employed in various types of hardware designs and can be adaptively modified to fit the requirements of the design and IP consumer. For example, a microprocessor IP may have different requirements than a cryptographic core. Yet PCHIP has been shown to be effective in both hardware types [7], [8], [9]. In cryptographic hardware, the emphasis goes on the flow of information in the design, not the actual functionality. For this purpose, PCHIP introduces a framework with information flow tracking capabilities [8], [9], which assigns a sensitivity (secrecy) level tag to the signals in the design. By tracking those signals through the design, this methodology ensures that no sensitive information reaches the outputs, without going through proper sensitivity reducing (declassifying) operations.

Although PCHIP is a powerful methodology, the heavy burden of conversion to formal representation and proof development hinders its wide deployment. While automating the entire PCHIP framework is highly desirable, it may not be completely feasible, in general, given the strenuous details of proof construction. The PCHIP framework for information flow tracking [8], [9] is still geared toward manual conversion and proof development and does not consider the detailed requirements involved in automating these tasks. *VeriCoq-IFT* redefines this base framework toward automation of the entire process, and is a step toward an automated PCHIP framework, focusing on information flow policies. In the next section, we depict the details of *VeriCoq-IFT*, as shown in Fig. 2.

III. *VeriCoq-IFT* in Detail

VeriCoq-IFT is a comprehensive solution for enforcing information flow policies on hardware designs, as depicted in Fig. 2. It automates the three intricate tasks of: (i) converting the HDL code to Coq representation, (ii) generating theorems enforcing information flow policies, and (iii) providing proofs of such theorems. Then, these three essential pieces are delivered to Coq which evaluates the validity of the proofs for the design. Although information flow policies seem to be simple in principle (e.g. no sensitive data should leak through untrusted channels), enforcing them in a design is complicated. In the following sections, we explain how *VeriCoq-IFT* automates these three main tasks in the PCHIP framework to ensure trustworthiness in terms of information flow policies.

```
1  module des (desOut, desIn, key, decrypt, roundSel, clk);
2
3  output [63:0] desOut;
4  /* vericoq init_sensitivity_level_1 */
5  input [63:0] desIn;
6  /* vericoq init_sensitivity_level_2 */
7  input [55:0] key;
8  input decrypt;
9  input [3:0] roundSel;
10 input clk;
11
12 wire [1:48] K_sub;
13 wire [1:64] IP, FP;
14 reg  [1:32] L, R;
15 wire [1:32] Xin;
16 wire [1:32] Lout, Rout;
17 wire [1:32] out;
18
19 assign Lout = (roundSel == 0) ? IP[33:64] : R;
20 assign Xin  = (roundSel == 0) ? IP[01:32] : L;
21
22 /* vericoq sensitivity_reducer */
23 assign Rout = Xin ^ out;
24 assign FP = { Rout, Lout};
25
26 crp u0(.P(out), .R(Lout), .K_sub(K_sub));
27
28 always @(posedge clk)
29   L <= #1 Lout;
30
31 always @(posedge clk)
32   R <= #1 Rout;
33
34 // Select a subkey from key.
35 key_sel u1 (.K_sub (K_sub), .K (key),
36  .roundSel (roundSel), .decrypt(decrypt));
37
38 // Perform initial and final permutation
39 assign IP[1:64] = { desIn[06], desIn[14] /* ... */};
40 assign desOut = { FP[40], FP[08] /* ... */};
41
42 endmodule
43
44 module  key_sel (K_sub, K, roundSel, decrypt);
45 // key_sel module body ...
46 endmodule
47
48 module crp(P, R, K_sub);
49 // crp module body ...
50 endmodule
```

Fig. 3. Partial Verilog source code of a DES core [14]

A. Conversion To Coq Representation

Adopting the rules defined in the PCHIP framework to convert HDL code to Coq representation [6], [8], [9], [7], we devise new rules for this task in *VeriCoq-IFT*, considering precise tracking of signals in the design as well as automation requirements. In this section, we illustrate the conversion of Verilog constructs to their Coq representation, as performed by *VeriCoq-IFT*. For this purpose, we utilize the partial Verilog source code of a DES encryption core [14] shown in Fig. 3 and its Coq representation generated by *VeriCoq-IFT* as it appears in Fig. 4.

Sensitivity (Secrecy) Tags: To enforce information flow policies, functionality and result of operations are not important. However, a mechanism is required in which each signal is accompanied by an attribute or tag which demonstrates its secrecy or sensitivity level and the flow of the signal through the design can be tracked precisely. To this end, *VeriCoq-IFT* defines the following types in the Coq representation.

```
Definition sensitivity := (nat * (option nat))%type.
Definition bus_sensitivity := list sensitivity.
Definition sen_list := list bus_sensitivity.
```

`sensitivity` is the type of the tag maintained for each signal. The first `nat` in this tuple represents the time stamp in which the secrecy level has been updated and we describe its usage later. The second one, defined as `option nat`, represents the actual secrecy level of the signal which evolves in time. Type `bus_sensitivity` represents the secrecy level of buses and is defined as **`list`** `sensitivity`. Note that *VeriCoq-IFT* considers all signals in the design, even single-bit ones, as buses. This allows treating single-bit and multi-bit signals in the same manner, without the need for additional definitions. To be able to track the sensitivity levels of buses through the design and their transitions, *VeriCoq-IFT* defines `sen_list` which is a list storing sensitivities of all buses. Each bus has a position in this list, maintaining its current secrecy level.

```
1  Require Import Vericoq_ift.
2
3  Inductive module :=
4   | module_des : bus->bus->bus->bus->bus->bus->bus->bus->bus->bus->bus->bus
5                  ->bus->bus->bus->module->module->module
6   | module_key_sel : bus->bus->bus->bus->bus->bus->bus->bus->bus->bus->bus->bus
7                      ->bus->bus->bus->bus->module
8   | module_crp : bus->bus->bus->bus->bus->bus->bus->module->module->module
9                 ->module->module->module->module->module->module->module
10  (* Other module defs ... *)
11 .
12
13 Fixpoint module_inst (m:module)  :=
14  match m with
15  | (module_des desOut desIn key decrypt roundSel clk Lout R IP Xin L
16                Rout out FP module_key_sel_u1 module_crp_u0) =>
17      (assign Lout (cond (ebinop (econb roundSel) (econb (Const None)))
18                         (econb (IP [(64 - 33), (64 - 64)])) (econb R)));
19      (assign Xin (cond (ebinop (econb roundSel) (econb (Const None)))
20                        (econb (IP [(64 - 1), (64 - 32)])) (econb L)));
21      (assign Rout (ereduce (ebinop (econb Xin) (econb out))));
22      (assign FP (ecat (econb Rout) (econb Lout)));
23      (assign L (econb Lout));
24      (assign R (econb Rout));
25      (* module_des body ... *)
26      (module_inst module_key_sel_u1) ;
27      (module_inst module_crp_u0)
28  | (module_key_sel K_sub K roundSel decrypt roundSelH decryptH K8 K7
29                    K6 K5 K4 K3 K2 K1) =>
30    (* module_key_sel body ... *)
31  | (module_crp P R K_sub E X Ss module_sbox8_u7 module_sbox7_u6
32                module_sbox6_u5 module_sbox5_u4 module_sbox4_u3
33                module_sbox3_u2 module_sbox2_u1 module_sbox1_u0) =>
34    (* ... *)
35 end.
36
37 Definition desOut : bus := Id 0.
38 Definition desIn : bus := Id 1.
39 Definition key : bus := Id 2.
40 Definition decrypt : bus := Id 3.
41 (* ... *)
42
43 Definition des:=
44  module_inst (module_des desOut desIn key decrypt roundSel clk
45                          Lout R IP Xin L Rout out FP
46     (module_key_sel K_sub key roundSel decrypt u1_roundSelH
47                     u1_decryptH u1_K8 u1_K7 u1_K6 u1_K5 u1_K4
48                     u1_K3 u1_K2 u1_K1)
49     (module_crp out Lout K_sub u0_E u0_X u0_S
50       (* module instantiations required by module_crp ... *)
51     )
52  ).
53
54 Definition init_state : sen_list :=
55   ((0, None)::(0, None)::(0, None)(* ... *)::nil) :: (* desOut *)
56   ((0, Some 1)::(0, Some 1)::(0, Some 1)(* ... *)::nil) :: (* desIn *)
57   ((0, Some 2)::(0, Some 2)::(0, Some 2)(* ... *)::nil) :: (* key *)
58   ((0, None)::nil) :: (* decrypt *)
59   (* ... *)
60  nil
61 .
```

Fig. 4. Partial *VeriCoq-IFT* generated Coq code for the DES core

Basic Circuit Elements in Coq Representation: *VeriCoq-IFT* considers every signal in the design as a bus. Selecting part of buses is common in HDLs like Verilog. Also, constants have wide usage in expressions and module instantiations. To facilitate the manipulation of buses, partial bus selection and constants, and to allow treating them uniformly, *VeriCoq-IFT* defines inductive type `bus` as follows.

```
Inductive bus : Type :=
| Id : nat -> bus
| Part : bus -> nat -> nat -> bus
| Const : option nat -> bus.
```

Constructor **`Id`** is the main constructor for type `bus`, which gets a `nat` representing the corresponding position in the sensitivity list. **`Part`** is used for partial selection of a bus and *VeriCoq-IFT* uses `[ , ]` notation for it. Note that the current definition of **`Part`** restricts indices to constants. The extension to non-constant indices will be considered in our future research. **`Const`** creates a `bus` from constants. These do not have a position in the sensitivity list and their sensitivity level is fixed. *VeriCoq-IFT* defines `read` and `update` functions to *read* or *update* the current sensitivity level of a `bus`, whole or partial, respectively. *VeriCoq-IFT* considers all Verilog signal definitions such as **`input`**, **`output`**, **`wire`** and **`reg`** as `bus` in Coq representation. As an example, note the definition of `desOut`, `desIn`, `key`, and `decrypt` in lines 3, 5, 7 and 8

of the Verilog source code in Fig. 3. These signals are defined as `bus` in the Coq representation of Fig. 4.

A challenge for partial selection is that Verilog does not restrict the range of buses to ascending/descending order or to start/end by index 0. Therefore, to prevent complexities in the Coq representation, *VeriCoq-IFT* normalizes indices in partial selection of buses such that the least significant bit (LSB) of a bus is always referred to by index 0. The Verilog source of Fig. 3 shows two methods of defining and using ranges. Specifically, the LSB of the `IP` bus has index 64 while it has index 0 in `desOut`. *VeriCoq-IFT* normalized the LSB of `IP` to index 0, as seen in lines 18 and 20 of the converted code in Fig. 4.

Module Definitions: *VeriCoq-IFT* converts module definitions in the Verilog source code to an inductive type in the Coq representation. It creates a constructor for the module and considers module inputs and outputs as parameters of this constructor. The body of the module is created in a function named `module_inst`. For example, lines 3-11 in Fig. 4 show the created `module` type definition for the modules defined in the Verilog source code of Fig. 3. Then, as shown in lines 13-35 of Fig. 4, *VeriCoq-IFT* also creates the `module_inst` function which constitutes the body of the modules. More details on the structure of this function are provided below.

Local Signals: Coq does not provide a flexible way for defining local variables inside functions. However, Verilog modules make extensive use of local signals. In order to resolve this Coq restriction, *VeriCoq-IFT* traces all the signals in a module and, whenever a local signal is needed, it adds it to the parameter list of the module in the Coq representation, even though such signals are not present in the port list of the module in Verilog. For example, consider local signals defined in lines 13-17 of the `des` module in Fig. 3. As lines 15-16 of Fig. 4 show, these signals are considered as parameters for this module in its Coq representation. However, if a local signal is used only to connect module instantiations and is not assigned or read directly inside a module (*e.g.* `K_sub` in `des` module of Fig. 3), there is no need to treat it as a module parameter. *VeriCoq-IFT* can correctly identify such local signals and accurately create their equivalent Coq description.

Parameters: *VeriCoq-IFT* supports Verilog numeric parameters which are often defined within modules. Since such parameters can be modified by each module instance, *VeriCoq-IFT* considers them as additional parameters when defining the module in its Coq representation. It also tracks the parameter definitions in the Verilog source code and passes the correct values for these parameters when creating module instances.

Module Instantiations: To support hierarchy, *VeriCoq-IFT* tracks module instantiations inside a module and defines them as parameters of the `module` definition in the Coq representation. For example, the `des` module in Fig. 3 instantiates two modules named `crp` and `key_sel`. As lines 5 and 16 in Fig. 4 show, these modules are added to the definition of the `des` module in its Coq representation.

VeriCoq-IFT automatically creates a definition for the top module of the Verilog source code, representing the top module instantiation. For this purpose, *VeriCoq-IFT* creates the appropriate variables, parameters and module instantiations. As lines 37-52 in Fig. 4 show, to instantiate `des`, *VeriCoq-IFT* defined the required buses with their corresponding position in the sensitivity list. It also created two module instances required for the `des` module, namely `module_key_sel` and `module_crp`, in order to instantiate `module_des`. Each of these two module instances require other module instantiations which *VeriCoq-IFT* creates recursively and are not shown in Fig. 4 due to space limitations.

Verilog Operations and Expressions: *VeriCoq-IFT* defines an inductive type `expr` in order to build expressions based on basic mathematical and logical operations of Verilog as follows.

```
Inductive expr :=
| econb : bus -> expr (*bus to expr*)
| euop : expr -> expr (*unary operator*)
| euop_bit : expr -> expr (*unary op.-1 bit result*)
| ebinop : expr -> expr -> expr (*binary operator*)
| ebinop_bit : expr -> expr -> expr(*bin op.1-b res*)
| ereduce : expr -> expr (*sensitivity reduction*)
| ecat : expr -> expr -> expr (*concatenation*)
| cond : expr -> expr -> expr -> expr.(*query(? : )*)
```

Note that this definition of `expr` is meant to work on the sensitivity tags in Coq representation, not the binary values of the signals, which we omitted in the conversion to Coq for information flow tracking purposes, and covers almost all basic Verilog operations. For example, addition, subtraction, or logical AND operation are all considered as binary operations build by the `ebinop` constructor. Such definition for `expr` reduces the effort required for proof development yet accurately tracks the flow of information in the hardware design. *VeriCoq-IFT* identifies the type of operators and creates their corresponding Coq representation accordingly, using the appropriate constructors. For example, we point out the eXclusive-OR operation in line 23 of Fig. 3, which is converted as line 21 in Fig. 4 using `ebinop`.

In the PCHIP framework for information flow policies, certain operations are considered as sensitivity level reducers (or declassifying operations), which vary by design. In order to identify such sensitivity reducing operations, *VeriCoq-IFT* defines a special comment (pragma) as `/* vericoq sensitivity_reducer */`, and wherever it finds this comment in the Verilog source code, it considers the next operation a sensitivity reducer. For cryptographic hardware, we consider the eXclusive-OR operation on secret input and sub-keys as sensitivity reducer. Consequently, notice the eXclusive-OR operation of line 23 in Fig. 3 which is preceded by this special comment. This operation is contained in the `ereduce` constructor, as shown in line 21 of its Coq representation in Fig. 4. *VeriCoq-IFT* also defines the `eval` function to evaluate expressions and produce the corresponding sensitivity levels. Evaluation of the query operator requires special consideration, which we illustrate next, along with the conditional statements.

Assignments and Conditional Statements: To build the code block in Coq representation, *VeriCoq-IFT* defines inductive type `code` as follows.

```
Inductive code :=
| assign : bus -> expr -> code
| ifsimp : expr -> code -> code
| ifelse : expr -> code -> code -> code
| code_cons : code -> code -> code.
```

`assign`, `ifsimp`, and `ifelse` are constructors used for Verilog assignments, if, and if-else statements, respectively. The current definition of `assign` restricts the left hand side to `bus`. This definition does not impose many restrictions on Verilog code, since assignments which do not follow this rule (*e.g.* concatenation on the left hand side) can be

broken into several assignments by the designer. We will address such restrictions in our future research. `code_cons` connects code together and *VeriCoq-IFT* uses **;** notation for its representation. For further details, notice the body of the `des` module in lines 19-36 of Fig. 3, which are converted to lines 17-27 of Fig. 4 as their Coq representation. One point to elaborate on in this conversion is that *VeriCoq-IFT* treats sequential and combinational blocks in the same way. Actually, when evaluating the code to update the sensitivity levels of the signals, all statements are treated sequentially. This does not create any problems for information flow tracking purposes. It may only delay the evolution of the sensitivity levels by a few clock cycles.

The importance of maintaining a time stamp together with the sensitivity levels of signals reveals itself in the evaluation of conditional statements. Since we omit the functionality of the operations in the conversion to Coq representation, there is no way to find out which branch of the condition is taken upon evaluation. Therefore, *VeriCoq-IFT* evaluates all branches in Coq representation, and when updating the sensitivity levels of buses in the assign statements, it uses the maximum value of the sensitivity level produced for a signal on each (virtual) clock cycle. To prevent implicit information leakage through conditions, *VeriCoq-IFT* considers the sensitivity level produced by the condition expression together with each branch, when evaluating statements inside conditional statements.

Initial Sensitivity List: To create the initial sensitivity list for the hardware design, *VeriCoq-IFT* requires knowing which signals carry sensitive information and their sensitivity levels. Therefore, to make gathering such information easier, *VeriCoq-IFT* defines a special Verilog comment (pragma) in the form of `/* vericoq init_sensitivity_level_2 */`. This special comment should appear before signal definition in the Verilog source code, and its numeric value, which appears last, is considered as the initial sensitivity level of the signal. Designers determine this value by considering the number of sensitivity reducing operations that the signal experiences through the design. For signals that *VeriCoq-IFT* finds no such information, it assumes the initial sensitivity to be **None**. Note that the initial sensitivity list is comprehensive, containing information for all signals in the design, including inputs, outputs and local signals. When instantiating the top module, *VeriCoq* also creates the initial sensitivity list using the information gathered through these comments. To elaborate further, note that the definition of `desIn` and `key` in lines 5 and 7 of the `des` module in Fig. 3 is preceded by such special comments. As seen in lines 54-61 of Fig. 4, *VeriCoq-IFT* assumes the initial sensitivity of 1 for `desIn` and 2 for `key` in the `init_state` created in Coq representation.

As we explained in this section, the conversion procedure from Verilog code to its Coq representation is completely automated in the *VeriCoq-IFT* framework, without any user intervention. IP developers are only required to provide necessary information by inserting special comments defined in the *VeriCoq-IFT* framework into the HDL code. Simlarly, IP consumers only need to check the validity of those comments for the corresponding signals and operations.

B. Security Property Theorems

VeriCoq-IFT generates functions and theorems which are required to ensure the trustworthiness of the design in terms of information flow policies. For this purpose, it generates a

```
1  Definition check_sensitivity t := check_code_sen des init_state t.
2  Definition stable := find_stable_list des init_state 20.
3
4  Definition is_safe_bef_stable_desOut :=
5    is_safe_bef_stable desOut des init_state (fst stable).
6
7  Theorem desOut_secrecy : forall (t : nat),
8    ((fst stable) < t) ->
9    is_safe_op_bus_sensitivity (read desOut (check_sensitivity t))
10   /\ is_safe_bef_stable_desOut.
11 Proof.
12   intros. split.
13   assert (get_sen_val_sen_list (check_sensitivity t) =
14           get_sen_val_sen_list (check_sensitivity (fst stable))).
15   apply check_code_sen_eq_st.
16   vm_compute. reflexivity.
17   apply H. simpl. omega.
18   assert (get_sen_val_op_bus (read desOut (check_sensitivity t)) =
19           get_sen_val_op_bus (read desOut
20             (check_sensitivity (fst stable)))).
21   apply read_sen_eq_st. apply H0.
22   assert (is_safe_op_bus_sensitivity (read desOut
23             (check_sensitivity t)) =
24           is_safe_op_bus_sensitivity (read desOut
25             (check_sensitivity (fst stable)))).
26   apply op_bus_same_sen_val_is_safe. apply H1. rewrite H2.
27   vm_compute. tauto. vm_compute. tauto.
28 Qed.
```

Fig. 5. *VeriCoq-IFT* generated theorem and proof for the DES core

function which is used to evaluate the statements in the code and update the sensitivity level based on a time stamp parameter, named `check_code_sen`. *VeriCoq-IFT* generates theorems for all the outputs of the top module to ensure that their sensitivity level remains safe at all times. To elaborate further, notice the theorem starting at line 7 of Fig. 5, which is generated as part of the conversion from Verilog to Coq representation in the *VeriCoq-IFT* framework. Since the `des` module in Fig. 3 has only one output, namely `desOut`, *VeriCoq-IFT* generates a theorem stating that the sensitivity level of `desOut` remains safe all the time. Assuming that the initial sensitivity level information and sensitivity reducing operations are provided accurately, proving this theorem ensures that no sensitive information is leaked through this primary output of the design. We describe the details of these theorems in the next section, which illustrates the proof.

C. Proofs of Security Theorems

Proofs of security theorems generated by *VeriCoq-IFT* are constituted in two parts. Since the sensitivity level of the primary inputs does not change and we do not have any sensitivity enhancing operator, the sensitivity list should reach a stable condition in which further code evaluations do not change the sensitivity values in the list. To find the clock cycle at which the sensitivity list becomes stable, *VeriCoq-IFT* defines a function named `find_stable_list`. Using this function, *VeriCoq-IFT* can automatically find the stable clock cycle, and its usage is found in line 2 of Fig. 5. Before the sensitivity list stabilizes, *VeriCoq-IFT* evaluates the code cycle-by-cycle and checks whether the sensitivity of the target signals is safe. This is performed using a function `is_safe_bef_stable` called specifically for `desOut` in lines 4-5 of Fig. 5. To prove this part, *VeriCoq-IFT* evaluates this function by computation.

After the sensitivity list becomes stable, *VeriCoq-IFT* uses proof-by-induction. *VeriCoq-IFT* proves a lemma named `check_code_sen_eq_st`, which states that after stabilization of the sensitivity list, further evaluation of the code does not change the sensitivity values, and its application is shown in lines 13-17 of the proof in Fig. 5. Using this lemma, the theorem for not leaking sensitive information is proved. Since the lemma is proved generally for all codes, proofs of all generated security property theorems can be constructed similarly and automatically, which is a big advantage of the *VeriCoq-IFT* framework.

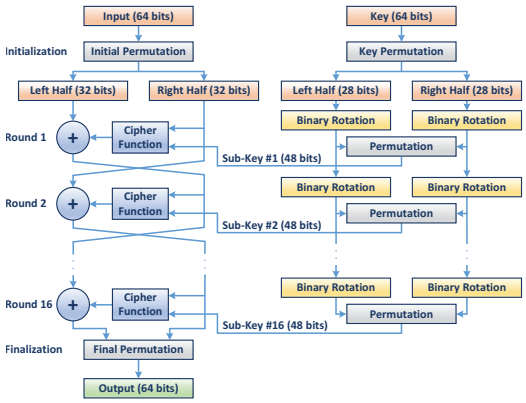

Fig. 6. DES block diagram [12], [15]

IV. Demonstration on Cryptographic Hardware

In this section, we demonstrate the capabilities of the *VeriCoq-IFT* framework by evaluating trustworthiness of two DES cores provided in [14]. A block diagram of the DES [12] algorithm is shown in Fig. 6; it includes 16 rounds, preceded and succeeded by initial and final permutations. We mark the eXclusive-OR operations of each round and the one inside the cipher function as sensitivity reducing operations. Since permutations and rotations are deterministic, we do not consider them as sensitivity reducing operations. Input text and key are considered sensitive and their initial sensitivity levels are assigned accordingly.

A. Area Efficient DES core

Since encryption rounds in the DES algorithm are similar, the first DES core, of which part of the Verilog code is shown in Fig. 3, implements a single round of DES algorithm. Several iterations are then invoked to perform the entire encryption. The iteration number is determined by the `roundSel` input.

We applied the *VeriCoq-IFT* framework to evaluate the trustworthiness of this DES core. This automatically converted the design to Coq representation and generated the security theorems and proofs. When provided to Coq for checking, however, we observed that the proofs fail. Detailed analysis of the code reveals a potential sensitive information leakage for this DES core, which might occur in the first round. As is evident in Fig. 6, the right half of the input text after the initial permutation does not go through a sensitivity reduction operation (eXclusive-OR); rather, it goes directly to the second round. Analyzing the partial Verilog source code of Fig. 6, following `desIn`, `IP`, `Lout`, `FP` and `desOut` reveals the same observation. Since the permutation operations are deterministic, at least part of the sensitive input `desIn` can leak to `desOut`. This area efficient design only implements one round of the algorithm and has a potential risk in leaking information. *VeriCoq-IFT* exposes such inherent potential problems as well as deliberate violation of information flow policies by malicious modifications, automatically and accurately.

B. Performance Optimized DES core

This DES core is implemented in a 16-stage pipeline, optimized for high performance requirements. Again, we automatically converted the design to Coq representation utilizing *VeriCoq-IFT* and used its automatically generated theorems and proofs to evaluate its trustworthiness in terms of information flow policies. Coq successfully passes the proofs of the security property theorems for this design, thereby guaranteeing that it does not violate information flow policies.

V. Conclusion

In this paper, we introduced *VeriCoq-IFT*, an automated PCHIP framework for enforcing information flow policies in hardware designs. *VeriCoq-IFT* aims at automating all critical tasks involved in PCHIP, namely conversion of the design in HDL code to Coq representation, generation of security property theorems and construction of proofs for such theorems.

To support enforcement of information flow policies, *VeriCoq-IFT* assigns sensitivity tags to each signal, maintaining their values in a sensitivity list. To facilitate proof construction, *VeriCoq-IFT* omits the functionality of operations in the conversion to Coq representation and provides lemmas which help to automatically construct proofs for the generated security properties. We successfully demonstrated the capabilities of *VeriCoq-IFT* on two DES core implementations. *VeriCoq-IFT* is an essential step toward an automated PCHIP framework, and can unlock the tremendous benefits of PCHIP in evaluating hardware trustworthiness. In our ongoing research, we plan to expand the capabilities of *VeriCoq-IFT* by adding support for a few other Verilog constructs such as **`task`** and **`generate`**, extend it to other types of security properties, and experiment with larger designs such as AES.

Acknowledgment

This work was partially supported by the National Science Foundation (NSF 1318860) and the Army Research Office (ARO W911NF-12-1-0091).

References

[1] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 10–25, Jan 2010.

[2] Y. Jin and Y. Makris, "Hardware trojans in wireless cryptographic integrated circuits," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 26–35, 2010.

[3] A. Waksman, M. Suozzo *et al.*, "FANCI: identification of stealthy malicious logic using boolean functional analysis," in *Proc. ACM Conf. Computer & Communications Security*, 2013, pp. 697–708.

[4] J. Zhang, F. Yuan *et al.*, "Veritrust: verification for hardware trust," in *Proc. Design Automation Conference*. ACM, 2013, p. 61.

[5] J. Zhang, F. Yuan *et al.*, "Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans," in *Proc. ACM Conf. Computer and Communications Security*, 2014.

[6] E. Love, Y. Jin *et al.*, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Trans. Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, 2012.

[7] Y. Jin and Y. Makris, "A proof-carrying based framework for trusted microprocessor IP," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 2013, pp. 824–829.

[8] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *Proc. IEEE VLSI Test Symposium*, 2012, pp. 252–257.

[9] Y. Jin, B. Yang *et al.*, "Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing," in *Int. Symp. Hardware-Oriented Security and Trust*. IEEE, 2013, pp. 99–106.

[10] G. C. Necula, "Proof-carrying code," in *Proc. Symp. Principles of Programming Languages*. ACM, 1997, pp. 106–119.

[11] INRIA. (2014, Oct.) The coq proof assistant. [Online]. Available: http://coq.inria.fr/

[12] NIST, "Data encryption standard (DES)," *Federal Information Processing Standards Publication*, 1999.

[13] T. Braibant and A. Chlipala, "Formal verification of hardware synthesis," in *Computer Aided Verification*. Springer, 2013, pp. 213–228.

[14] OpenCores. [Online]. Available: http://opencores.org/

[15] D. Rudolf. Development and analysis of block ciphers and the DES system. [Online]. Available: http://homepage.usask.ca/ dtr467/400/