Data Secrecy Protection Through Information Flow Tracking in Proof-Carrying Hardware IP—Part II: Framework Automation

Mohammad-Mahdi Bidmeshki, *Student Member, IEEE*, Xiaolong Guo, *Student Member, IEEE*, Raj Gautam Dutta, *Student Member, IEEE*, Yier Jin, *Member, IEEE*, and Yiorgos Makris, *Senior Member, IEEE*

***Abstract*— Part II of this paper series focuses on automation of the extended proof-carrying hardware intellectual property (PCHIP) framework for data secrecy protection in third-party IPs, which was presented in part I. Specifically, we introduce: 1) *VeriCoq-IFT*, an automated PCHIP framework for information flow policies and 2) *VeriCoq-H*, a hierarchy-preserving Verilog-to-Coq converter. *VeriCoq-IFT* aims to: 1) automate the process of converting designs from an HDL to the Coq formal language; 2) generate security property theorems ensuring compliance with information flow policies; 3) construct proofs for such theorems; and 4) check their validity in a design, with minimal user intervention. *VeriCoq-H*, on the other hand, seeks to convert the entire functionality of a Verilog design to its Coq representation while preserving design hierarchy. It facilitates the development of hierarchical proofs and enables the construction of hybrid module libraries containing the HDL code and the corresponding reusable lemmas for each module. Applicability of our automated *VeriCoq-IFT* framework is demonstrated by evaluating trustworthiness of two DES encryption circuits and several genuine and Trojan-infested advanced encryption standard (AES) designs, along with the utility of *VeriCoq-H* in preventing malicious modification of sensitive data, such as the secret key of an encryption circuit.**

***Index Terms*— Hardware trust, proof-carrying code, proof-carrying hardware IP, information flow tracking.**

I. Introduction

IN PART I of this paper series, we reviewed the fundamentals of the proof-carrying hardware IP (PCHIP) methodology and we developed frameworks for information flow tracking (IFT) to ensure that a hardware design is secure in terms of information flow policies. We also presented a new hierarchy-preserving methodology for the PCHIP framework, on which our solution for protecting the hardware design against malicious modification of data is based. Despite their tremendous potential, broad deployment of PCHIP-based solutions faces certain challenges. Converting HDL code to a formal representation, such as the Coq [1] 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 who has this expertise, the process is tedious and time-consuming, making IP developers hesitant to adopt PCHIP. In this paper, we present our efforts toward automation of the PCHIP framework and we demonstrate the applicability of our methods in detecting design flaws and hardware Trojans in cryptographic cores.

The dynamic information flow tracking method presented in part I of this paper series offers the opportunity to develop a fully automated PCHIP framework, specifically geared towards enforcing information flow policies. An initial attempt in this direction was described in our earlier work [2]. *VeriCoq-IFT*, shown in Fig. 1, is an automated PCHIP framework for enforcing information flow policies, covering various tasks such as conversion of the design from HDL to the Coq formal language, generating security property theorems that ensure information flow policies, constructing proofs for such theorems, and checking their validity for the design, with

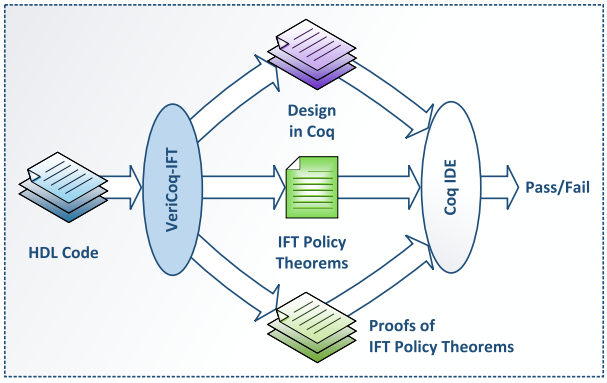

Fig. 1. Automated PCHIP framework for information flow policies.

Manuscript received June 26, 2016; revised January 20, 2017; accepted May 10, 2017. Date of publication May 23, 2017; date of current version July 20, 2017. This work was supported in part by the National Science Foundation under Grant NSF-1318860 and Grant NSF-1319105 and in part by the Army Research Office under Grant ARO W911NF-12-1-0091 and Grant ARO W911NF-16-1-0124. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Jean-Luc Danger. *(Corresponding author: Yiorgos Makris.)*

M.-M. Bidmeshki and Y. Makris are with the Department of Electrical and Computer Engineering, The University of Texas at Dallas, Richardson, TX 75080 USA (e-mail: bidmeshki@utdallas.edu; yiorgos.makris@utdallas.edu).

X. Guo, R. G. Dutta, and Y. Jin are with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816 USA (e-mail: guoxiaolong@knights.ucf.edu; rajgautamdutta@knights.ucf.edu; yier.jin@eecs.ucf.edu).

This paper has supplementary downloadable material available at http://ieeexplore.ieee.org., provided by the authors. The material includes all RTL source code, the Coq representation, the data secrecy theorems and their proofs. Contact the authors for further questions about this work.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2017.2707327

minimal user intervention. Herein, we extend the capabilities of *VeriCoq-IFT* so that it is able to protect not only outputs but also intermediate signals in the design. Additionally, we provide a procedure for automatically establishing the initial sensitivity levels of all signals.

Applicability of *VeriCoq-IFT* is demonstrated on two different implementations of the data encryption standard (DES) [3], as well as on the genuine and several Trojan-infested versions of the advanced encryption standard (AES) [4]. Based on minimal input provided by the user, trustworthiness of these circuits with respect to information flow policies is automatically checked. *VeriCoq-IFT* gathers the required 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. Any alteration of circuit description against information flow policies causes proofs to fail. This methodology is a first essential step towards establishing PCHIP as a valuable method for assessing credibility of third party IPs with minimal extra effort. We note that, while both a static and a dynamic IFT method were introduced in part I, *VeriCoq-IFT* is established atop the dynamic one, which supersedes its static counterpart.

An alternative way through which hardware Trojans may introduce harmful actions, and which may not always be detectable through the enforcement of information flow policies using *VeriCoq-IFT*, is the malicious modification of data, such as the key in a cryptographic core. To prevent such attacks, in part I of this paper series we proposed a solution based on the general PCHIP framework, which involves manual development of security theorems and their proofs. To assist in the process, in our earlier work we introduced *VeriCoq* [5], which automates conversion of the design from Verilog to its Coq representation. However, *VeriCoq* performs the flattened functional conversion presented in part I, removing design hierarchy and, therefore, making manual proof development very challenging. Therefore, herein we introduce *VeriCoq-H*, an enhanced version of *VeriCoq*, which automatically converts the Verilog design to its Coq representation following the hierarchy-preserving methodology presented in part I. *VeriCoq-H*, whose role in the general PCHIP framework is shown in Fig. 2, enables hierarchical proof development. Additionally, hybrid libraries of hardware modules including proofs of various reusable lemmas can be developed to expedite the proof development for higher-level designs. Consequently, the hierarchy-preserving methodology reduces the proof-writing burden in the PCHIP framework.

Application of *VeriCoq-H* towards facilitating proof development for security properties which prevent malicious modification of data in cryptographic cores is detailed through an example. We note that this approach is different from the methodology for detecting malicious modification of data in third-party hardware IPs which was introduced in [6]. The latter employs a formal state-space exploration approach for this purpose, which is limited in verifiable clock cycles and cannot guarantee the trustworthiness of the design beyond a given time-stamp. In contrast, the PCHIP-based solution does not have such a restriction.

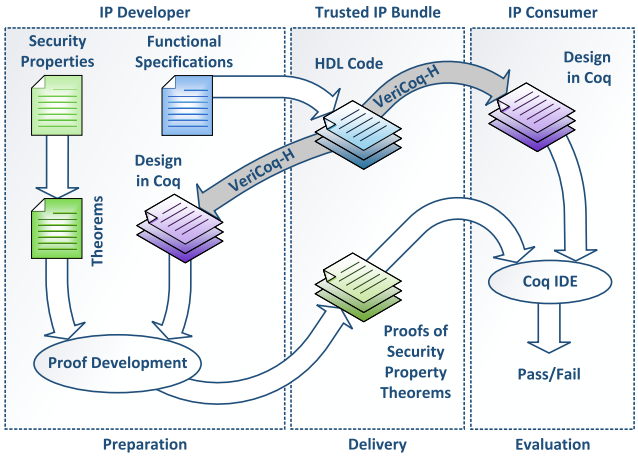

Fig. 2. General PCHIP framework and *VeriCoq-H*.

The rest of this paper is organized as follows. In Section II, we illustrate the details of the *VeriCoq-IFT* framework including conversion to Coq, generation of security property theorems and construction of proofs. In Section III, we depict the hierarchical conversion of the entire functionality of a Verilog design to Coq representation within the general PCHIP framework, as performed by *VeriCoq-H*. Section IV provides a case study of the static IFT method presented in part I of this paper series and demonstrates the capabilities of *VeriCoq-IFT* on two implementations of the DES encryption algorithm, and several AES designs including a genuine and various Trojan-infested versions. Additionally, it details the application of *VeriCoq-H* in preventing malicious modification of the secret key in a DES core. A discussion is provided in Section V and conclusion are drawn in Section VI.

II. *VeriCoq-IFT* IN DETAIL

VeriCoq-IFT is a comprehensive solution for enforcing information flow policies on hardware designs, as depicted in Fig. 1. 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-IDE, which evaluates the validity of the proofs for the design. While information flow policies appear 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 with respect to information flow policies.

A. Conversion to Coq Representation

VeriCoq-IFT builds upon the HDL-to-Coq conversion rules defined in the original PCHIP framework [7]–[10] and in part I of this paper series and enhances them with additional rules which consider not only precise tracking of signals in the design but also automation requirements. In this section,

```
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  /* vericoq init_sensitivity_level_1 */
10 input [3:0] roundSel;
11 input clk;
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 [11].

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 [11], shown in Fig. 3, and its Coq representation generated by *VeriCoq-IFT*, as it appears in Fig. 4.

1) Sensitivity (Secrecy) Tags: To enforce information flow policies, functionality and result of operations are not important. However, a mechanism is required whereby each signal is accompanied by an attribute, or tag, which demonstrates its secrecy or sensitivity level, so that 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, the usage of which is described 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, as well as their transitions, *VeriCoq-IFT* defines `sen_list` which is a list storing the 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
5                  ->bus->bus->bus->module->module->module
6   | module_key_sel : 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->module->module->module
9                 ->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.

2) Basic Circuit Elements in Coq Representation: VeriCoq-IFT considers every signal in the design as a bus. Selecting part of a bus is common in HDLs such as Verilog. Also, constants have wide usage in expressions and module instantiations. To facilitate manipulation of buses, partial bus selection and use of 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. Extension to non-constant indices is feasible but is deferred to future extensions of this framework. `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.

3) Module Definitions: VeriCoq-IFT converts module definitions in the Verilog source code to an inductive type in the Coq representation, similar to what we described for the flattened functional model in part I of this paper series. 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.

4) Local Signals: 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 the local signals defined in lines 12-17 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.

5) 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.

6) 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.

7) 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
operator - 1 bit  result*)
| ebinop : expr -> expr -> expr (*binary
 operator*)
| ebinop_bit : expr -> expr -> expr (*binary
operator - 1 bit result*)
| 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 built 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 tracking presented in part I, certain operations of each design are considered as sensitivity level reducers (or declassifying operations). 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 a 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. We note that not all eXclusive-OR operations are considered sensitivity reducers. Sensitivity reducing operations should be individually marked and these decisions should be verified by the IP consumers using a clean high-level architecture or block diagram of the design.

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.

8) 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`. Nevertheless, this definition does not impose actual restrictions on the 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. `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; 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, when evaluating statements inside conditional statements *VeriCoq-IFT* considers the sensitivity level produced by the condition expression together with each branch.

9) 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. Thus, to make gathering of 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. 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-IFT* 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 an initial sensitivity of 1 for `desIn` and 2 for `key` in the `init_state` created in Coq representation.

Finding the exact sensitivity level of sensitive signals requires an understanding of the overall flow of information in the design, and as mentioned before, determining how many sensitivity reducing operations act on those signals before reaching the outputs. To facilitate this task, we can take advantage of *VeriCoq-IFT* auxiliary capabilities to help the users. For this purpose, designers need to mark the sensitivity reducing operations in the design as explained before. They can use the following procedure to get recommendations on the initial sensitivity levels.

For each sensitive signal bit b:

1) Set the initial sensitivity level of b to a high value, e.g., 100, in the initial sensitivity list.
2) Set the sensitivity level of all other signal bits in the initial sensitivity list to zero.
3) Evaluate the code using the updated initial sensitivity list until a stable list is reached.
4) Check all output bits in the resulting stable sensitivity list and consider the minimum reduction in the sensitivity of the signal seen in the output as the initial sensitivity level of the signal bit b.

We note that this procedure should be performed on a clean architecture of the design which only comprises high level blocks and operations, without detailed implementation information. This ensures that any design flaws or malicious design modifications will not interfere with the the task of finding the initial sensitivity levels.

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. Similarly, 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 has a function which is used to evaluate the statements in the code and to 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

```
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.

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 at all times. This theorem covers all 4 requirements presented as theorem generation functions in part I. 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.

Some Trojans may not leak sensitive information to an output of the hardware IP design. Instead, they establish a side-channel inside the design, e.g., by manipulating the power consumption or the leakage current, to facilitate leaking sensitive information indirectly [12]. While *VeriCoq-IFT* cannot automatically detect such side-channels, it provides a special comment as `/* vericoq force_nonsensitive */` for the designers or, more importantly, the IP consumers, to mark specific signals which they might be suspicious of leaking sensitive information. *VeriCoq-IFT* also generates security theorems for these specific internal signals, in order to ensure that no sensitive information reaches them.

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. This function evaluates the code cycle-by-cycle and returns the clock cycle at which the resulting sensitivity list evaluated in two consecutive clock cycles is equal. 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 proven. Since the lemma is proven 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.

III. *VeriCoq-H*: Hierarchical Verilog to Coq Converter

Although information flow polices, enforced automatically by *VeriCoq-IFT*, are able to capture sensitive information leakage through the outputs or suspicious signals, they cannot thoroughly prevent or capture malicious modifications of sensitive data in the design. As an example of such malicious modifications, consider a hardware Trojan which, upon activation, sets the key to a constant or a value created from the plaintext or the original key through a transformation known to the attacker. Such action might not be detected by functional tests due to its rare activation, yet can disrupt the core functionality and leak sensitive information because the attacker obtains the key.

In order to protect the hardware design against such attacks, in part I of this paper series we presented a solution based on the general PCHIP framework. As opposed to the PCHIP framework for information flow policies implemented by *VeriCoq-IFT*, which does not consider the functionality of operations and only tracks the flow of information, the general PCHIP methodology converts the exact functionality of the circuit to the Coq representation. Using this general framework, security properties which prevent malicious modifications of data can be developed and proven. We note that, besides conversion of the design to its corresponding Coq representation, generating these security properties and proving them is design-dependent, still requires the developer's effort, and is not yet automated.

Earlier, we introduced *VeriCoq* [5] for automated conversion of the exact functionality of a hardware design in Verilog to its corresponding Coq representation based on the hierarchy flattening methodology presented in Part I. A key limitation in *VeriCoq* is, that similar to *VeriCoq-IFT*, it flattens the design hierarchy. For information flow policies, flattening the hierarchy helps to automate the entire framework including the proofs. In case of the general PCHIP, however, it makes the manual development of proofs for security property theorems complicated. In this section, we introduce an enhanced version of *VeriCoq*, namely *VeriCoq-H*, which converts the design to its Coq representation according to the hierarchy-preserving methodology of part I. As we mentioned earlier, this conversion approach enables the development of hierarchical proofs

```
1  Require Import Vericoq.
2
3  Module Type module_sbox1.
4
5  Definition instantiate (addr dout : bus) (t:nat) :=
6  (* ... *)
7  .
8  End module_sbox1.
9
10 (* Other sub-modules *)
11
12 Module Type module_crp.
13
14 Parameters  E X Ss : wire.
15
16 Declare Module sbox1_u0 : module_sbox1.
17 Declare Module sbox2_u1 : module_sbox2.
18 (* ... *)
19
20 Definition instantiate (P R K_sub : bus) (t:nat) :=
21       (adoif (
22       (* ... *)
23       ) t)
24       /\
25         (* ... *)
26         (sbox2_u1.instantiate (X [(48 - 7), (48 - 12)])
27           (Ss [(32 - 5), (32 - 8)]) t) /\
28         (sbox1_u0.instantiate (X [(48 - 1), (48 - 6)])
29           (Ss [(32 - 1), (32 - 4)]) t)
30 .
31 End module_crp.
32 (* ... *)
33
34 Module Type module_des.
35
36 Parameters K_sub IP FP L R Xin Lout Rout out : wire.
37
38 Declare Module crp_u0 : module_crp.
39 Declare Module key_sel_uk : module_key_sel.
40
41 Definition instantiate (desOut desIn key decrypt roundSel clk
42   : bus) (t:nat) :=
43       (adoif (
44         (anoif (expr_assign Lout (cond (eeq (econb roundSel)
45           (econv (lo::nil)))
46           (econb (IP [(64 - 33), (64 - 64)]))
47           (econb R))));
48         (anoif (expr_assign Xin (cond (eeq (econb roundSel)
49           (econv (lo::nil)))
50           (econb (IP [(64 - 1), (64 - 32)]))
51           (econb L))));
52         (* ... *)
53       ) t)
54       /\
55       (doif (
56         (noif (upd_expr L (econb Lout)))
57         (noif (upd_expr R (econb Rout)))
58       ) t)
59       /\
60         (crp_u0.instantiate out Lout K_sub t) /\
61         (key_sel_uk.instantiate K_sub key roundSel decrypt t)
62 .
63 End module_des.
64
65 Module Type module_des_top.
66
67 Parameters desOut desIn key decrypt roundSel clk : bus.
68
69 Declare Module des_top : module_des.
70
71 Axiom des: forall (t:nat),
72   des_top.instantiate desOut desIn key decrypt roundSel clk t.
73
74 End module_des_top.
```

Fig. 6. Partial *VeriCoq-H* generated Coq code for the DES core.

and proof reuse. Lemmas proven in sub-modules can be applied to prove theorems in higher level modules. A similar hierarchical security theorem proving process has also been demonstrated on a large-scale SoC design [13]. Moreover, a library of modules can be created, including the lemmas and proofs for each module, and utilized for the development of new designs as well as the proof of their security properties. The evolution of such a module library will extensively reduce the burden of proof development for IP designers who choose to utilize the PCHIP framework in production.

Fig. 6 shows the partial hierarchical Coq representation of the DES core of Fig. 3 generated by *VeriCoq-H*, following the hierarchical formal model presented in Part I. To assist with understanding it, a few details or additional capabilities of *VeriCoq-H* are described below.

A. Parameters

VeriCoq-H handles Verilog numeric parameters similar to *VeriCoq-IFT* and considers them as additional arguments when defining the module through the `instantiate` function 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.

B. Top Module Instantiation

VeriCoq-H automatically creates an axiom for the top module of the Verilog source code, representing the top module instantiation. For this purpose, *VeriCoq-H* creates the appropriate variables, parameters and module instantiation. As lines 65-74 in Fig. 6 show, a `Module Type` is defined for the top module and an axiom is created to instantiate `des`, corresponding to `des` module as the top module in the Verilog source code of Fig. 3.

C. Part Selection

VeriCoq-H handles part selection similar to *VeriCoq-IFT* and uses the same `[ , ]` notation. Likewise, it normalizes indices in part selection of buses such that the least significant bit (LSB) of a bus is always referred to by index 0. As an example, note the part selection of signal `IP` in lines 46 and 50 of the Coq representation in Fig. 6, corresponding to lines 19-20 of the Verilog source code of Fig. 3.

The code generated by *VeriCoq-H*, as partially shown in Fig. 6, is directly usable in the Coq environment to develop proofs for the desired security properties, such as theorems which prevent malicious modification of data in cryptographic cores, as we show in Section IV-C. The first line of Fig. 6 imports a Coq library containing the general PCHIP definitions.

IV. Demonstration on Cryptographic Hardware

In this section, we demonstrate the capabilities of the extended PCHIP frameworks by evaluating trustworthiness of two DES cores provided in [11] and several genuine and Trojan-infested AES cores offered in [14]. We first demonstrate the static IFT approach on an area efficient DES implementation. We then show how the dynamic IFT methodology implemented by *VeriCoq-IFT* can be used to evaluate trustworthiness of various genuine and Trojan-infested cryptographic cores. Finally, using two Trojan-infested DES cores, we showcase the effectiveness of the PCHIP framework, as enhanced through the hierarchy-preserving capabilities of *VeriCoq-H*, in detecting malicious data modifications. Details of the Verilog source codes, the corresponding Coq representations and proofs can be found in the supplementary material.

A. Evaluation of DES Cores

A block diagram of the DES [3] algorithm is shown in Fig. 7. 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.

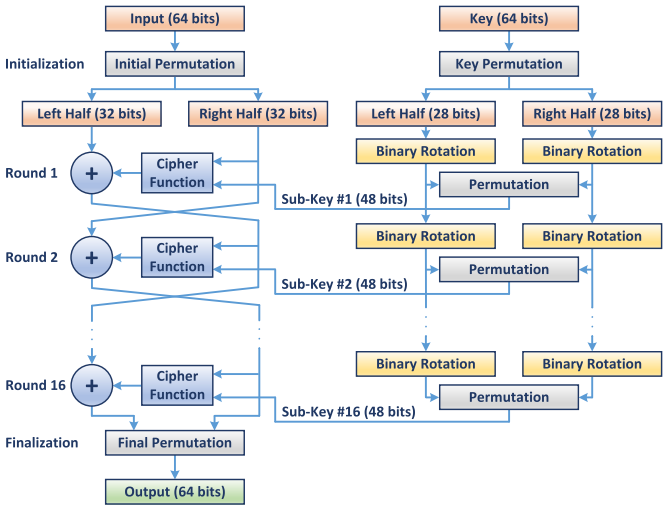

Fig. 7. DES block diagram [3], [15].

```
Definition des : code :=
  outb desOut;
  inb desIn;
  inb key;
  inb decrypt;
  inb roundSel;
  inb clk;
  wireb K_sub;
  wireb IP;
  wireb FP;
  regb L;
  regb R;
  wireb Xin;
  wireb Lout;
  wireb Rout;
  wireb out;

  assign_ex Lout (cond (eq (econb roundSel)
    (econv (lo::lo::lo::lo::nil))) (econb (IP @ [33, 64])) (econb R));
  assign_ex Xin (cond (eq (econb roundSel)
    (econv (lo::lo::lo::lo::nil))) (econb (IP @ [1, 32])) (econb L));
  assign_ex Rout (exor_key (econb Xin) (econb out));
  assign_ex FP (econb (bus_app Rout Lout));

  module_inst2in out Lout K_sub;

  nonblock_assign_ex L (econb Lout);
  nonblock_assign_ex R (econb Rout);

  module_inst3in K_sub key roundSel decrypt;

  assign_ex IP (perm (econb desIn));
  assign_ex desOut (perm (econb FP)).
```

Fig. 8. Coq representation of area efficient DES core for static information flow tracking.

1) 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. Several iterations are then invoked to perform the entire encryption. The iteration number, which is determined by the `roundSel` input, is also considered sensitive.

a) Static methodology: First, we show how the static IFT approach can be used for evaluating this core. The `des` module of Fig. 3, instantiates two sub-modules, the Feistel function (`crp.v`) and the key generator (`key_sel.v`) to perform round encryption/decryption and round key generation, respectively. 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. 7, 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. Analysis of the partial Verilog source code of Fig. 3, focusing on `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`. In essence, this area-efficient design only implements one round of the algorithm and has a potential risk in leaking information, which can be caught by PCHIP-based static IFT, as we show below.

Based on the Verilog-to-Coq conversion rules and the *formal semantic model* of part I, this design is converted to Coq. The complete version of the converted Coq representation for `des.v` is shown in Fig. 8.

Static Property Proofs: For each module, we need to denote the signal secrecy tags and prove the data secrecy properties. For the `des.v` module, among all input, output and internal signals, the input key (`key`), input plaintext (`desIn`), input round count (`RoundSel`), and internal generated round keys (`K_sub`) require protection. Reflected in Coq formal logic, three axioms are added to indicate the semantics that the `key`, `desIn`, `RoundSel`, and `K_sub` have `sensitive` tags in *all* clock cycles. The separation of signal property denotation axioms and circuit code constitutes a key characteristic of the Coq platform. These axioms act as *preconditions* for all security properties extracted from the Coq circuit.

```
Axiom secret_key : forall (t : nat),
bus_sen key t = sensitive.
Axiom secret_desIn : forall (t : nat),
bus_sen desIn t = sensitive.
Axiom secret_K_sub : forall (t : nat),
bus_sen K_sub t = sensitive.
Axiom secret_RoundSel : forall (t : nat),
bus_sen roundSel t = sensitive.
```

With both the preconditions and the DES circuit itself available in Coq representation, the next step is to construct the data secrecy property that we wish to prove and express it also in Coq. Our requirement that "no internal sensitive information is leaked through a primary output" is formalized into the following `no_leaking_des` theorem in Coq.

```
Theorem no_leaking_des : forall (t : nat),
chk_code_sen des t = non_sensitive.
```

If the proof can be successfully constructed for the `no_leaking_des` theorem given the Coq circuit and the preconditions, we can then declare that the delivered HDL code is trusted *with respect to data secrecy protection policy*. However, the theorem cannot be proven. In fact, we can prove the opposite conclusion, i.e. the DES output is sensitive. This vulnerability stems from the single-round implementation of DES, which is repeatedly used to perform the entire encryption. As illustrated in partial Verilog source code of Fig. 3, in this implementation the intermediate results are visible at the primary outputs; hence, the right half of the plaintext, which does not go through a sensitivity reducing operation in the first round, becomes exposed and causes the security theorem proof to fail. From an IP customer perspective, this implies that an IP vendor cannot provide a trusted bundle for this DES core since a proof cannot be constructed. While this static IFT approach is effective in successfully revealing design flaws, such as the one mentioned above, it involves a tedious manual process and is more suitable for single-stage or small designs.

TABLE I
SUMMARY OF TROJAN INFECTED AES DESIGNS

Group	Trojan	Trigger	Leaking mechanism	Detected by *VeriCoq-IFT*?
1	T100	Always on	CDMA like modulation, LFSR initialized by a constant	Yes
	T200	Always on	CDMA like modulation, LFSR initialized by plaintext	Yes
	T700	Specific plaintext value	CDMA like modulation, LFSR initialized by a constant	Yes
	T800	Specific plaintext sequence	CDMA like modulation, LFSR initialized by a constant	Yes
	T1000	Specific plaintext value	CDMA like modulation, LFSR initialized by plaintext	Yes
	T1100	Specific plaintext sequence	CDMA like modulation, LFSR initialized by plaintext	Yes
	T900	Specific number of encryptions	CDMA like modulation, LFSR initialized by a constant	Yes
	T1200	Specific number of encryptions	CDMA like modulation, LFSR initialized by plaintext	Yes
2	T400	Specific plaintext value	AM modulation	Yes
	T1600	Specific plaintext sequence	AM modulation	Yes
	T1700	Specific number of encryptions	AM modulation	Yes
3	T600	Specific plaintext value	Internal load for side-channel	Yes (with guidance)
	T300	Always on	Enabling internal shift registers for side-channel	Yes (with guidance)
	T1300	Specific plaintext value	Enabling internal shift registers for side-channel	Yes (with guidance)
	T1400	Specific plaintext sequence	Enabling internal shift registers for side-channel	Yes (with guidance)
	T1500	Specific number of encryptions	Enabling internal shift registers for side-channel	Yes (with guidance)
4	T1800	Specific plaintext value	No leak - deplete battery	Not evaluated
	T500	Specific plaintext sequence	No leak - deplete battery	Not evaluated
	T1900	Specific number of encryptions	No leak - deplete battery	Not evaluated

b) Dynamic methodology implemented by VeriCoq-IFT: We also 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-IDE for checking, however, we observed that the checking of proofs fails. This is expected, since as we mentioned in the evaluation by the static approach, this area efficient DES core has the potential of leaking sensitive information. *VeriCoq-IFT* exposes such inherent potential problems as well as deliberate violation of information flow policies by malicious modifications, automatically and accurately, and is therefore capable of handling larger and more complex designs, as we discuss next.

2) Performance Optimized DES Core: This DES core is implemented in a 16-stage pipeline, optimized for high performance requirements. Evaluation of this pipelined core using the static methodology is difficult, since we need to verify every round of encryption implemented by each pipeline stage separately. For each round, we need to consider its eXclusive-OR operations as sensitivity reducers while considering these operations in other stages as normal and verify the data secrecy property. Using the dynamic IFT method, however, which can automatically propagate the sensitivity levels, is much easier and accurate for this purpose. Therefore, we evaluate this core using only the dynamic methodology.

We converted the design to Coq representation utilizing *VeriCoq-IFT* and used its automatically generated theorems and proofs to evaluate the trustworthiness of this DES core with respect to information flow policies. Coq-IDE successfully passes the proofs of the security property theorems for this design, thereby guaranteeing that, indeed, it does not violate any information flow policies.

B. Evaluation of AES Cores

AES [4] is a more sophisticated encryption algorithm as compared to DES, and can be performed using 128-, 192-, and 256-bit keys. Fig. 9 shows the block diagram of AES-128 which requires 10 rounds. Along with the genuine AES-128,

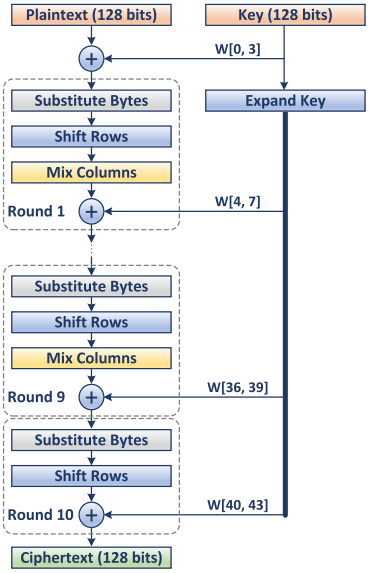

Fig. 9. AES-128 block diagram [4].

the Trust-Hub website [14] provides various Trojan-infested AES cores. Given the complexity and pipelined structure of these designs, we only use the dynamic methodology implemented by *VeriCoq-IFT* to evaluate these AES cores. Also, since the Trojan-infested AES designs combine several leaking mechanisms with various trigger conditions, we group them by their leaking mechanism for a clearer presentation. Table I summarizes the Trojan infected AES designs which we evaluated using *VeriCoq-IFT*. For each of these designs, on a Windows 7 computer with 8 GB of RAM and an Intel Core i7 processor, the conversion process takes less than a second, while proof verification completes in about 30 minutes.

1) Genuine AES Core: The genuine AES-128 core is a 10-stage pipelined design based on the block diagram shown in Fig. 9. We marked adding round key operations (+ operations) as sensitivity reducers and we defined appropriate sensitivity levels for the inputs. Then, we used *VeriCoq-IFT* to convert this design to its Coq representation and we evaluated the

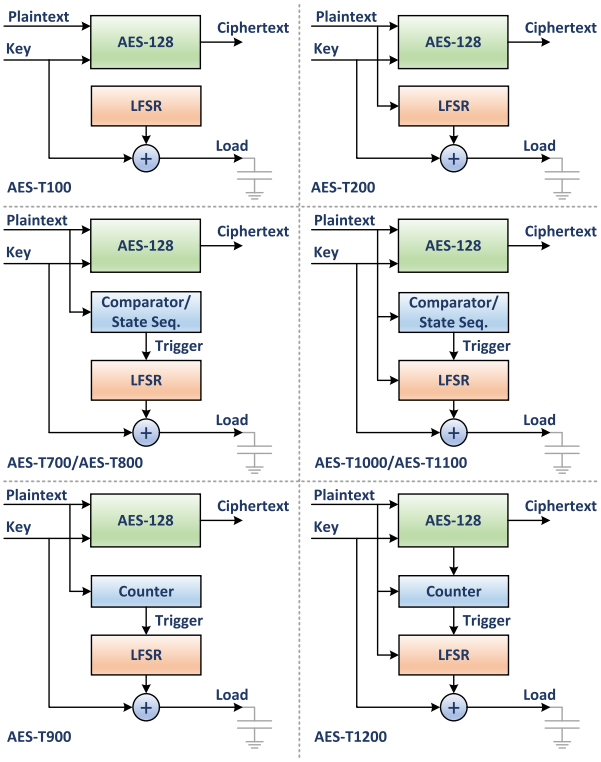

Fig. 10. First group of Trojan-infested AES designs with different triggers and LFSR initializations.

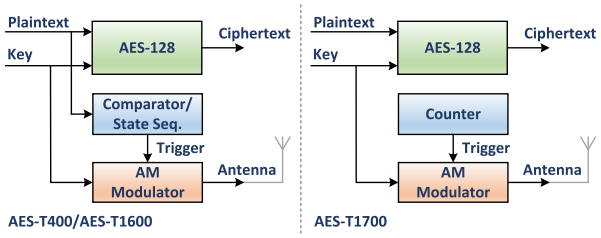

Fig. 11. Second group of Trojan-infested AES designs which leak the key by AM radio.

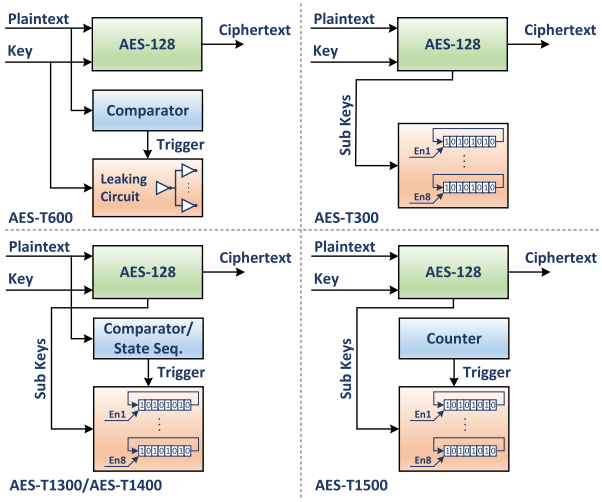

Fig. 12. Third group of Trojan-infested AES designs which leak the key by establishing an internal side channel.

automatically generated proofs for the security property theorems in Coq-IDE. The checking of proofs passes, confirming that this design abides by the information flow policies.

2) Group 1 of Trojan-Infested AES Designs: These Trojan-infested AES designs have an additional output in which the Trojan tries to leak 8 bits of the key using a CDMA like modulation [12], [16]; their block diagrams are shown in Fig. 10. The intention is to connect this output, named *load* in Fig. 10, to a large capacitive load, and create a power side channel. The Trust-Hub website provides this Trojan with various trigger conditions. As Fig. 10 shows, AES-T100 and AES-T200 are always on. AES-T700 and AES-T1000 are triggered by a specific plaintext input, while AES-T800 and AES-T1100 require a predefined sequence of plaintext values to be activated. AES-T900 and AES-T1200 are triggered after a predefined number of encryptions. Another variation in these Trojans is the value used to initialize the LFSR for generating pseudorandom numbers. In Trojans shown on the left column of Fig. 10, the LFSR is initialized using a predefined constant value. Trojans on the right column use a subset of bits from the plaintext input to initialize the LFSR instead.

We used *VeriCoq-IFT* to convert these designs to their Coq representation and we evaluated the automatically generated proofs of the security property theorems in Coq-IDE. Thereby, we observe that checking of these proofs fails for the *load* output in all of these designs, indicating that *VeriCoq-IFT* can successfully detect the sensitive information leakage by these Trojans. Also, as expected, Coq successfully verifies the proofs of the security property theorems for *Ciphertext* outputs, as these do not leak any sensitive information.

3) Group 2 of Trojan-Infested AES Designs: This group of Trojan-infested AES designs leak the key by generating a radio-frequency (RF) signal on an output pin, which is intended to work as an antenna. They leak the key through amplitude modulation (AM), easily received and interpreted by a regular AM radio as meaningful beeps [17]. The difference between the Trojans in this group is the trigger mechanism, as shown in Fig. 11. AES-T400 is activated by a predefined plaintext value. AES-T1600 is triggered by a specific sequence of plaintext input and AES-T1700 is activated after a predefined number of encryptions.

Evaluation of these Trojan-infested AES designs using *VeriCoq-IFT* shows that checking the proofs of the security theorems for the *Antenna* output fails for these designs. Therefore, *VeriCoq-IFT* can successfully detect this group of Trojans. Since these Trojans do not modify the AES core itself, checking the proofs of security theorems for the *Ciphertext* output in these designs passes in Coq-IDE, showing that no sensitive information is leaked by these output ports.

4) Group 3 of Trojan-Infested AES Designs: The third group of Trojan-infested AES designs that the Trust-Hub website provides, shown in Fig. 12, do not have a dedicated output to create the side channel for leaking the key. AES-T600, one of the Trojan-infested designs in this group, creates an internal load using a few inverters to leak the entire key and is activated by a predefined plaintext input. Another type of

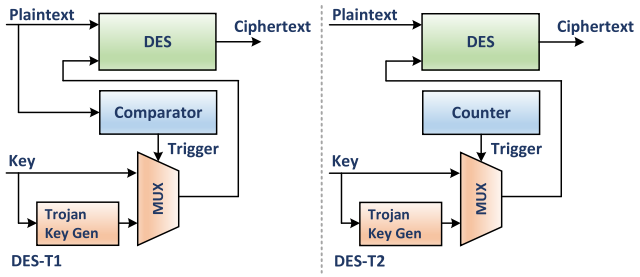

Fig. 13. DES Trojans which modify the secret key.

Trojan-infested design in this group generates 8 enable signals as a function of the plaintext input and 8 sub-keys. Each enable signal is used to activate rotation of the value in an 8 bit shift register, initialized with "10101010", to create a power side channel [12]. As Fig. 12 shows, this type of Trojans are provided with four trigger conditions similar to Trojans in the previous groups. AES-T300 is always on, AES-T1300 is activated by a predefined plaintext, AES-T1400 is triggered by a specific sequence of plaintext input, and AES-T1500 is activated after a predefined number of encryptions.

Since these Trojans do not use an explicit output, *VeriCoq-IFT* cannot automatically generate the information flow policy theorems for the internal load signals and shift registers. However, by marking the suspicious signals and registers as explained in Section II-B, we can instruct *VeriCoq-IFT* to generate such theorems for these signals. Following this step, checking the automatically generated proofs for the security theorems on these signals does not pass in Coq-IDE. This reveals that these signals obtain sensitive values, thereby alerting designers to perform further evaluations in order to either justify existence or remove such assignments.

5) Group 4 of Trojan-Infested AES Designs: The last group of Trojan-infested AES designs that the Trust-Hub website provides consists of Trojans which do not leak sensitive information. These Trojans increase the power consumption of the design, with the intention of depleting the energy source (battery) more quickly, and use three different Trigger conditions: predefined plaintext (AES-T1800), specific sequence of plaintext input (AES-T500) and predefined number of encryptions (AES-T1900). Since these Trojans do not leak sensitive information, we exclude them from evaluation by *VeriCoq-IFT*. However, since the Trigger condition in two of these Trojans depends on the plaintext input, which is sensitive, we can use the same procedure as in Section IV-B.4 to check their safety. Such checking, however, requires design review by the IP consumers and cannot be performed automatically by *VeriCoq-IFT*.

C. Preventing Malicious Modification of Data

To evaluate our methodology for preventing malicious modification of data, we developed two Trojan-infested DES cores by slightly modifying the pipelined DES implementation, shown in Fig. 13. These Trojans set the secret key to a constant value once they are triggered and have a simple structure.

The first DES Trojan is activated by a specific plaintext value. Using *VeriCoq-IFT*, we converted this design to the Coq representation and verified the information flow policy theorem. Since assignment of a malicious (modified) key is triggered by the plaintext input, this Trojan alters the sensitivity level of the key (due to a conditional assignment based on the trigger value), which then propagates in the design and violates the information flow policies. Therefore, the proof of theorems generated by *VeriCoq-IFT* fails Coq-IDE evaluation and the Trojan is detected.

Unfortunately, *VeriCoq-IFT* cannot always detect a malicious modification of sensitive data. As an example, the second DES Trojan is activated after a predefined number of clock cycles and does not change the sensitivity level of the secret key, because the trigger in this case has a sensitivity level of zero. Therefore, checking the proof of the information flow policy theorem passes for this design and the Trojan evades detection. Other methods are also possible to hide these Trojans. Hence, malicious modification of data cannot always be prevented by enforcing information flow policies. To prevent such meddling, we resort to the general PCHIP framework. In this case, we utilized *VeriCoq-H* to convert the genuine pipelined DES core to the corresponding Coq representation and developed theorems ensuring the authenticity of key bits throughout the design. Proving these theorems was performed hierarchically, wherein lemmas are first proven in sub-modules and then applied in theorems of higher-level modules to construct proofs.

As an example, lines 65-69 in the partial Coq representation of Fig. 14 define a theorem named `key_sub1_genuine` stating that the first round of encryption is done by an eXclusive-OR operation on the first sub-key (`K1`) and the corresponding signal created from the plaintext input (`E`). Also, this theorem ensures that the LSB of the first sub-key is indeed coming from the corresponding key input, according to the specifications. To prove this theorem, we used theorems `crp_u0_X_eq_E_xor_k1` and `k1_bit48_decrypt0_genuine` which are proved in `module_des`, as seen in lines 27-50 of Fig. 14. To prove `crp_u0_X_eq_E_xor_k1` in `module_des`, we applied a theorem proved in `module_crp` which is a sub-module instantiated in `module_des`. Similarly, the proof of theorem `k1_bit48_decrypt0_genuine` uses theorem `k1_bit48_decrypt0_key_r` which subsequently applies a theorem proven in `module_key_sel`, a sub-module of `module_des`. For Trojan-infested DES designs which modify the secret key, evaluation of proofs of these theorems in Coq-IDE results in failure, thereby enabling their detection by this generalized PCHIP-based framework. Crucially, this success relies on the hierarchical approach, facilitated by *VeriCoq-H*, which makes proof construction more streamlined and facilitates the creation of design libraries consisting of modules and corresponding reusable lemmas.

In order to compare the *hierarchical* to the *flattened* functional model, we also converted the DES core to the Coq representation using the flattened option and we developed the corresponding proof for the same theorem. The details of the converted code can be found in the supplementary material.

```
Require Import Vericoq_ift.
(* ... *)
Module Type module_des.
Parameters key_r desIn_r (* ... *) : wire.
(* ... *)

Declare Module crp_u0 : module_crp.
Declare Module crp_u1 : module_crp.
(* ... *)
Declare Module key_sel_uk : module_key_sel.

Definition instantiate
  (desOut desIn key decrypt clk : bus) (t:nat) :=
(* ... *)
.

Theorem k1_bit48_decrypt0_key_r:
  forall (t : nat) (desOut desIn key decrypt clk : bus),
  module_des.instantiate desOut desIn key decrypt clk t ->
  decrypt t = lo::nil ->
  K1 [(48 - 48), (48 - 48)] t = key_r [28, 28] t.
Proof.
  (* ... *)
  apply key_sel_uk.k1_bit48_decrypt0_genuine with (* ... *)
Qed.

Theorem k1_bit48_decrypt0_genuine:
  forall (t : nat) (desOut desIn key decrypt clk : bus),
  module_des.instantiate desOut desIn key decrypt clk t ->
  module_des.instantiate desOut desIn key decrypt clk (S t) ->
  decrypt t = lo::nil ->
  decrypt (S t) = lo::nil ->
  K1 [(48 - 48), (48 - 48)] (S t) = key [28, 28] t.
Proof.
  (* ... *)
  apply k1_bit48_decrypt0_key_r
  with desOut desIn key decrypt clk.
  apply H0. apply H2.
Qed.

Theorem crp_u0_X_eq_E_xor_k1:
  forall (t : nat) (desOut desIn key decrypt clk : bus),
  module_des.instantiate desOut desIn key decrypt clk t ->
  crp_u0.X t = bv_xor (crp_u0.E t) (K1 t).
Proof.
  intros.
  apply crp_u0.X_eq_E_xor_K_sub
  with out0 (IP [(64 - 33), (64 - 64)]).
  (* ... *)
Qed.

End module_des.

Module Type module_des_top.

Parameters desOut desIn key decrypt clk : bus.
Declare Module des_top : module_des.

Axiom des: forall (t:nat),
  des_top.instantiate desOut desIn key decrypt clk t.

Axiom decrypt0: forall (t: nat),
  decrypt t = lo::nil.

Theorem key_sub1_genuine: forall (t : nat),
  des_top.crp_u0.X (S t) =
  bv_xor (des_top.crp_u0.E (S t)) (des_top.K1 (S t)) /\
  des_top.K1 [(48 - 48), (48 - 48)] (S t) = key [28, 28] t.
  (* /\ statements for authenticity of other key bits ... *)
Proof.
  intros.
  split.
  apply des_top.crp_u0_X_eq_E_xor_k1
  with desOut desIn key decrypt clk.
  apply des.
  apply des_top.k1_bit48_decrypt0_genuine
  with desOut desIn decrypt clk.
  apply des. apply des.
  apply decrypt0. apply decrypt0.
Qed.

End module_des_top.
```

Fig. 14. Partial hierarchical proof construction to prevent malicious modification of data in the pipelined DES core.

While proof development style varies across developers (e.g., by proving different intermediate lemmas or employing different Coq tactics), the key advantage of the hierarchical functional model is that it provides separate `Module Type` definitions for each Verilog module. This modularity enables development of lemmas inside a module, independent of the higher-level module that instantiates it and without the need to track and globally define local module signals, as required by the flattened model. In turn, this supports lemma reusability, which can significantly shorten proof development time.

As an additional experiment to corroborate the advantages of the hierarchical over the flattened functioal model, we also developed theorems to protect the key in a basic RSA core. A clean and Trojan-infested version of an RSA core, which replaces the secret key based on a trigger condition, can be found on TrustHub [14] in VHDL format. After converting both versions to Verilog and applying the proposed method, we concluded that checking the corresponding proof passes for the clean design but fails for the Trojan-infested one. The codes for both the hierarchical and the flattened functional model, as well as the pertinent theorems and proofs, are provided in the supplementary material.

V. Discussion

VeriCoq-IFT provides an automated PCHIP framework for information flow tracking in hardware cores, which is highly effective in preventing infestation of malicious capabilities seeking to leak sensitive data. However, similar to any other information flow tracking methodology, *VeriCoq-IFT* relies on trusted and accurate labeling of the initial sensitivity values and declassifying operations, in order to effectively track the flow of sensitive information in the design. In Section II, we described a procedure to find the initial sensitivity values of sensitive signals. Nevertheless, to make the best use of *VeriCoq-IFT*, designers and IP consumers should carefully review the *VeriCoq-IFT* labels in the design and should be able to justify them with precise reasoning based on a clean high-level architecture or block diagram of the design.

The scope of protection provided by PCHIP is limited by what is covered by the agreed upon security properties. Therefore, a carefully selected set of security properties is required, commensurate with the protection level sought by each application. Additionally, while enforcement of information flow polices can effectively protect the design against leakage of sensitive information, the scope of these security properties pertains only to information flow. For example, such security properties and, consequently, *VeriCoq-IFT*, are not geared towards guaranteeing functional correctness of the hardware design. Moreover, tampering with sensitive information cannot always be captured by enforcing information flow policies. Nevertheless, the general PCHIP framework is extensive enough to address such aspects of security requirements in a cryptographic hardware IP design by combining *VeriCoq-IFT* with the original PCHIP method, enhanced with *VeriCoq-H* to convert the functionality of a design to a hierarchy-preserving Coq representation whereon formal reasoning on functionality can be hierarchically performed.

Moreover, the generalized PCHIP-based framework is mainly geared towards design-level attacks which use primary outputs to digitally leak sensitive information. Stealing information through other means, such as side-channels (e.g. power, timing, etc.), is not directly addressed through the proposed methods. Nevertheless, such side channel-based Trojans may still be detected indirectly, as we presented through a few examples in Sections IV-B.2 to IV-B.4. This happens when a primary output, for which *VeriCoq-IFT* automatically generates security theorems, is proven by the proposed method to carry sensitive information due to malicious design modifications. If this signal is the source from which the side channel-based hardware Trojan obtains the sensitive data, such as in the MOLES [16] side-channel attack, the Trojan is indirectly detected. The same holds true when an internal signal is used as the source of leakage for the side-channel attack. In this case, however, the user needs to annotate the internal signals

which are deemed suspicious, so that *VeriCoq-IFT* will automatically generate the corresponding security theorems. Such user guidance is not inconsistent with the practice required by many other hardware Trojan detection methodologies such as FANCI [18], VeriTrust [19] or ATPG-based methods for Trojan detection [20], [21]. Moreover, these methods focus on identifying nearly unused circuits or rare triggers as signatures for hardware Trojans and do not incorporate a notion of information flow. Hence, detection of a trigger is the only mechanism through which these methods may detect a side channel-based hardware Trojan. However, as summarized in Table I, several among the hardware Trojans that we studied in our experiments are always on and, therefore, evade these Trojan detection methodologies, while still being detected by the PCHIP-based solutions proposed herein.

Finally, our conservative information flow model implemented by *VeriCoq-IFT* is less accurate as compared to recently introduced methodologies such as SecVerilog [22] and the gate-level based information flow tracking for hardware Trojan detection [23]. Therefore, even though we have not encountered this situation in our experiments, it is possible that it may introduce false positives during circuit evaluation. SecVerilog, on the other hand, is not geared towards hardware Trojans, which is specifically considered in our methods, while hardware Trojan detection based on gate-level information flow tracking relies on model checking which is limited due to the state explosion problem in larger designs. Overall, the proposed work constitutes a powerful alternative approach for addressing data secrecy protection based on PCHIP-based information flow tracking, whose limitations may be further mitigated through further research and development.

VI. Conclusion

In part II of this paper series we described our efforts towards automating the extended PCHIP frameworks whose foundations we presented in part I. We first introduced *VeriCoq-IFT*, which provides a fully automated PCHIP framework for enforcing dynamic information flow policies, wherein designers only need to annotate the HDL code with minimal required information, such as the initial sensitivity levels and the declassifying operations. Based on this information, *VeriCoq-IFT* automatically generates the Coq representation, theorems and proofs required for enforcing information flow policies. In our experiments, *VeriCoq-IFT* successfully revealed design flaws and/or malicious capabilities hidden inside various genuine and Trojan-infested DES and AES implementations. We then presented *VeriCoq-H*, a method for converting the exact functionality of a design from HDL to its Coq representation, based on the hierarchy-preserving functional model described in part I. We demonstrated how it can be used to hierarchically develop proofs of security properties for a design and we detailed its utility through an example of preventing and/or capturing malicious modification of sensitive data in hardware cores. Notably, *VeriCoq-H* facilitates proof construction and development of hybrid hardware module libraries containing HDL code and proof of reusable lemmas for the module, thereby significantly reducing the burden of proof writing.

In our ongoing research, we plan to expand the capabilities of *VeriCoq-IFT* and *VeriCoq-H* by adding support for a few additional Verilog constructs, such as `task` and `generate`. Enhancing the information flow model and reducing its limitations by improving its accuracy, as well as studying the trade-off between the degree of functionality preserved in the converted code in the Coq representation and the protection level achieved, is another direction of our current efforts. We also plan to explore further automation possibilities in the general PCHIP framework for various hardware designs, such as microprocessor IPs and communication cores. Expansion of the automated PCHIP framework will help its wide adoption by the hardware design community, resulting in more secure and trustworthy third-party IP acquisition transactions.

References

[1] INRIA. *The COQ Proof Assistant*, accessed on Jun. 1, 2016. [Online]. Available: http://coq.inria.fr/

[2] M.-M. Bidmeshki and Y. Makris, "Toward automatic proof generation for information flow policies in third-party hardware IP," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, May 2015, pp. 163–168.

[3] *Data Encryption Standard (DES), Federal Information Processing Standards Publication*, NIST, Gaithersburg, MD, USA, 1999.

[4] *Announcing the Advanced Encryption Standard (AES) Federal Information Processing Standards Publication*, vol. 197, Nat. Inst. Standards Technol., Gaithersburg, MD, USA, 2001, pp. 1–51.

[5] M.-M. Bidmeshki and Y. Makris, "VeriCoq: A Verilog-to-Coq converter for proof-carrying hardware automation," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2015, pp. 29–32.

[6] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," in *Proc. ACM Design Autom. Conf. (DAC)*, 2015, p. 112:1–112:6.

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

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

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

[10] Y. Jin, B. Yang, and Y. Makris, "Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing," in *Proc. IEEE Int. Symp. Hardware-Oriented Security Trust (HOST)*, Jun. 2013, pp. 99–106.

[11] *OpenCores*, accessed on May 21, 2016. [Online]. Available: http://opencores.org/

[12] L. Lin, M. Kasper, T. Guneysu, C. Paar, and W. Burleson, "Trojan side-channels: Lightweight hardware trojans through side-channel engineering," in *Proc. Cryptograph. Hardw. Embedded Syst. (CHES)*, 2009, pp. 382–395.

[13] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Scalable SoC trust verification using integrated theorem proving and model checking," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, May 2016, pp. 124–129.

[14] *Trust-Hub*, accessed on Jan. 10, 2016. [Online]. Available: https://www.trust-hub.org/

[15] D. Rudolf. *Development and analysis of block ciphers and the DES system*, accessed on Oct. 10, 2015. [Online]. Available: http://homepage.usask.ca/~dtr467/400/

[16] L. Lin, W. Burleson, and C. Paar, "MOLES: Malicious off-chip leakage enabled by side-channels," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, 2009, pp. 117–122.

[17] A. Baumgarten, M. Steffen, M. Clausman, and J. Zambreno, "A case study in hardware trojan design and implementation," *Int. J. Inf. Secur.*, vol. 10, no. 1, pp. 1–14, 2011.

[18] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of stealthy malicious logic using Boolean functional analysis," in *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*, 2013, pp. 697–708.

[19] J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu, "Veritrust: Verification for hardware trust," in *Proc. ACM Design Autom. Conf. (DAC)*, 2013, p. 61:1–61:8.
[20] S. Saha *et al.*, "Improved test pattern generation for hardware trojan detection using genetic algorithm and Boolean satisfiability," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst. (CHES)*, 2015, pp. 577–596.
[21] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "MERO: A statistical approach for hardware trojan detection," in *Proc. Cryptograph. Hardw. Embedded Syst. (CHES)*, 2009, pp. 396–410.
[22] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2015, pp. 503–516.
[23] W. Hu, B. Mao, J. Oberg, and R. Kastner, "Detecting hardware trojans with gate-level information-flow tracking," *Computer*, vol. 49, no. 8, pp. 44–52, 2016.

Mohammad-Mahdi Bidmeshki (S'11) received the B.Sc. and M.Sc. degrees in computer engineering from Sharif University of Technology, Tehran, Iran, in 2004 and 2006, respectively. He is currently pursuing the Ph.D. degree in computer engineering with The University of Texas at Dallas. His current research includes hardware-based security, trusted hardware design, formal methods in security and verification, and the applications of machine learning in computer security.

Xiaolong Guo received double bachelor's degrees from the Beijing University of Posts and Telecoms (BUPT) and the University of London in 2010 and the M.S. degree from BUPT in 2013. He is currently pursuing the Ph.D. degree in electrical engineering with the University of Central Florida, Orlando, FL, USA. His current research interests include design of scalable verification methods for hardware IP protection, trusted SoC verification, cyber security, formal methods, program synthesis, and secure language design.

Raj Gautam Dutta received the B.Tech. degree in electronics and communication from Visvesvaraya Technological University, India, in 2007, and the M.S. degree in electrical engineering, with an emphasis on control systems, from the University of Central Florida, USA, in 2011, where he is currently pursuing the Ph.D. degree with the EECS Department. His current research interests include development of security solutions for semiconductor soft IP cores by using formal verification techniques, design of attack detection and mitigation software for autonomous systems, and synthesis of robust controllers for cyber-physical systems.

Yier Jin received the B.S. and M.S. degrees in electrical engineering from Zhejiang University, China, in 2005 and 2007, respectively, and the Ph.D. degree in electrical engineering from Yale University in 2012. He is currently an Assistant Professor with the ECE Department, University of Central Florida.

His research focuses on the areas of trusted embedded systems, trusted hardware intellectual property (IP) cores and hardware–software co-protection on computer systems. He proposed various approaches in the area of hardware security, including the hardware Trojan detection methodology relying on local side-channel information, the postdeployment hardware trust assessment framework, and the proof-carrying hardware IP protection scheme. He is also interested in the security analysis on Internet of Things (IoT) and wearable devices with particular emphasis on information integrity and privacy protection in the IoT era. He received the DoE Early CAREER Award in 2016 and is a Best Paper Award Recipient of DAC'15, ASP-DAC'16, and HOST'17.

Yiorgos Makris (SM'08) received the Diploma degree in computer engineering from the University of Patras, Greece, in 1995, and the M.S. and Ph.D. degrees in computer engineering from the University of California at San Diego, in 1998 and 2001, respectively. After spending a decade on the Faculty of Yale University, he joined The University of Texas at Dallas, where he is currently a Professor of Electrical Engineering, leading the Trusted and RELiable Architectures (TRELA) Research Laboratory. His research focuses on applications of machine learning and statistical analysis in the development of trusted and reliable integrated circuits and systems, with particular emphasis in the analog/RF domain. He was a recipient of the 2006 Sheffield Distinguished Teaching Award and Best Paper Awards from the 2013 Design Automation and Test in Europe (DATE'13) Conference and the 2015 VLSI Test Symposium (VTS'15). He serves as an Associate Editor of IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY and IEEE DESIGN & TEST periodical, and he has also served as a Guest Editor of IEEE TRANSACTIONS ON COMPUTERS and IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS.