VeriCoq: A Verilog-to-Coq Converter for Proof-Carrying Hardware Automation

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

***Abstract*—Proof carrying hardware intellectual property (PCHIP) introduces a new framework in which a hardware (semiconductor) Intellectual Property (IP) is accompanied by formal proofs of certain security-related properties, ensuring that the acquired IP is trustworthy and free from hardware Trojans. In the PCHIP framework, conversion of the design from a hardware description language (HDL) to a formal representation is an essential step. Towards automating this process, herein we introduce *VeriCoq*, a converter of designs described in Register Transfer Level (RTL) Verilog to their corresponding representation in the Coq theorem proving language, based on the rules defined in the PCHIP framework. *VeriCoq* supports most of the synthesizable Verilog constructs and is the first step towards automating the entire framework, in order to simplify adoption of PCHIP by hardware IP developers and consumers and, thereby, increase IP trustworthiness.**

I. Introduction

Economy globalization has recently resulted in a highly geographically dispersed integrated circuit (IC) design and fabrication flow. Concomitant with this dispersion, this flow has become very vulnerable to inclusion of malicious capabilities, a.k.a. hardware Trojans, which are not known to the designer and user of the IC, but which can be exploited by a knowledgeable adversary. Given the wide reach of technology in every aspect of our everyday life, the impact of such hardware Trojans can be disastrous. Accordingly, intense research efforts have been invested in preventing and/or detecting hardware Trojan inclusion in various phases of IC design and fabrication [1, 2].

In the fast paced IC industry, time to market is a crucial factor in investment return. Hence, design reuse and utilization of previously developed designs in the form of hardware IPs (in-house or third party) is inevitable. 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 exercise the design capabilities exhaustively. Furthermore, since soft IPs are also widely used in FPGA-based designs, 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. Given this intensified threat, prevention and/or detection of hardware Trojans in soft IPs has become extremely important.

A few approaches, such as FANCI [3] and VeriTrust [4], sought to address the problem of soft-IP hardware Trojan identification at the design stage. While 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. Within 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, in order to prevent the insertion of hardware Trojans in a design. 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.

Although PCHIP is a very promising framework, its broad adoption faces a few challenges. First, developing security properties is not straightforward. While a few such properties have been introduced for microprocessors [7] and cryptographic hardware [8, 9], they are usually specific to each design, and cannot be reused for others. Second, converting HDL code to a formal representation, such as the Coq [11] language used in PCHIP[1], 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, making the barrier to entrance rather high for IP developers.

Evidently, automating the PCHIP framework to the extent possible could make it more appealing and could help in its broader utilization, leading to lower risk in hardware IP acquisition. Towards this goal, in this paper we introduce *VeriCoq*, a Verilog-to-Coq converter based on the rules developed in the PCHIP framework. *VeriCoq* supports most of the synthesizable Verilog constructs and converts parameters, arrays, module hierarchy and module instantiations effectively to their Coq representation. While automating the entire PCHIP framework is a much broader endeavor and may not be completely feasible, given the strenuous details of proof construction, *VeriCoq* is a fundamental step towards this end. Crucially, it not only automates the conversion process, but also makes proof construction by IP developers and proof checking by IP consumers less perplexing, since both can rely on the common Coq representation of the Verilog code, which is now automatically generated by *VeriCoq*.

II. Proof-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

[1]Coq is a popular theorem proving tool used extensively by the software research community. Other automated theorem provers can also be utilized in PCHIP framework by adjusting the conversion rules and the converter.

978-1-4799-8391-9/15/$31.00 ©2015 IEEE

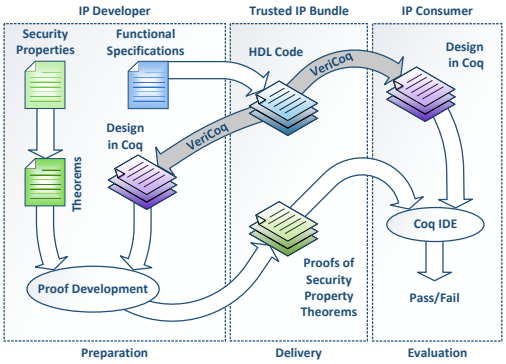

Fig. 1. PCHIP framework

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. As another example, security properties might impose restrictions on the flow of information in a design [8, 9] to avoid propagation of sensitive information to unauthorized sites within the chip and eventual leakage.

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 [12]. 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. *VeriCoq* seeks to assist with a portion of these extra tasks, as shown by the shaded arrows in Fig. 1, by making the conversion to Coq representation effective, reliable, quick and automatic, with minimum user intervention.

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. Consumers, upon receiving the HDL code and the proofs, may utilize *VeriCoq* to convert the design to its Coq representation and check the validity of proofs in the CoqIDE environment. In this sense, *VeriCoq* is helpful to both IP developers and IP consumers. In the next section, we introduce *VeriCoq* and we provide details regarding conversion of Verilog code to Coq.

III. VeriCoq

VeriCoq converts RTL designs described in Verilog to Coq, according to the rules defined in the PCHIP framework. Although conversion of Verilog constructs to an equivalent Coq representation may seem straightforward, a closer look at these two languages reveals fundamental differences which add significant challenges to the task. In the following, we describe how *VeriCoq* handles Verilog constructs and creates their equivalent in Coq. To this end, we use the simple Verilog code shown in Fig. 2 and its Coq representation shown in Fig. 3 to explain the details of the conversion.

Basic Circuit Elements in Coq Representation: The PCHIP framework defines `value` as an inductive type which can be `lo`, `hi`, or `x`. A `bus_value` is represented as a list of `value`. This allows treating single-bit and multi-bit signals in the same manner, without the need for additional definitions.

```
Inductive value := lo|hi|x.
Definition bus_value := list value.
Definition bus := nat -> bus_value.
```

PCHIP also defines `bus` as a function of `nat` to `bus_value` which allows bringing the notion of time to the Coq representation, so that we can handle sequential statements. All signal types in a design, such as **`reg`** and **`wire`**, are considered as `bus` in Coq representation. PCHIP considers arrays as **`list`** `bus` which *VeriCoq* identifies and handles accordingly.

Module Definitions: *VeriCoq* flattens the design hierarchy and 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 3-8 in Fig. 3 show the created `module` type definition for the modules defined in the Verilog source code of Fig. 2. Then, as shown in lines 10-58 of Fig. 3, *VeriCoq* also creates the `module_inst` function which constitutes the body of the modules. More details on the structure of this function are provided below. Flattened design hierarchy in Coq may make developing proofs for security properties more difficult. To solve this problem, designers can selectively convert parts of the design to Coq and develop proofs incrementally.

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* 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 signals `compl_result` and `and_result` which are locally defined in the `my_alu` module of Fig. 2. As line 13 of Fig. 3 shows, these two 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, there is no need to treat it as a module parameter. *VeriCoq* can correctly identify such local signals and accurately create their equivalent Coq description.

Parameters: *VeriCoq* supports Verilog numeric parameters which are often defined within modules. Since such parameters can be modified by each module instance, *VeriCoq* considers them as additional parameters when defining the module in its Coq representation. It also tracks the parameter definitions in

```
1  // Simple ALU for VeriCoq demonstration
2  module my_alu (result, src1, src2,
3                 clk, rst, op_select);
4  parameter ALU_BIT_LEN = 8;
5  output [1:ALU_BIT_LEN] result;
6  input [1:ALU_BIT_LEN] src1;
7  input [1:ALU_BIT_LEN] src2;
8  input clk;
9  input rst;
10 input op_select;
11 wire [1:ALU_BIT_LEN] compl_result;
12 wire [1:ALU_BIT_LEN] and_result;
13 reg [1:ALU_BIT_LEN] result;
14
15 always @(rst)
16   if (rst)
17     result = 8'h00;
18
19 always @(posedge clk) begin
20   if (!rst)
21     case (op_select)
22       0: result = compl_result;
23       1: result = and_result;
24     endcase
25 end
26
27 defparam add1.BIT_LEN = ALU_BIT_LEN;
28 defparam and1.BIT_LEN = ALU_BIT_LEN;
29
30 my_add add1 (.result (compl_result),
31              .src1 (~src1), .src2 (8'h01), .clk(clk));
32 my_and and1 (.result (and_result), .src1 (src1),
33              .src2 (src2), .clk(clk));
34
35 endmodule  // ALU
36
37 // Adder module
38 module my_add (result, src1, src2, clk);
39 parameter BIT_LEN = 4;
40 output [1:BIT_LEN] result;
41 input [1:BIT_LEN] src1;
42 input [1:BIT_LEN] src2;
43 input clk;
44 reg [1:BIT_LEN] result;
45
46 always @(posedge clk)
47   result[1:BIT_LEN] <= src1[1:BIT_LEN] + src2[1:BIT_LEN];
48 endmodule  // Adder
49
50 // And module
51 module my_and (result, src1, src2, clk);
52 parameter BIT_LEN = 4;
53 output [BIT_LEN:1] result;
54 input [BIT_LEN:1] src1;
55 input [BIT_LEN:1] src2;
56 input clk;
57 reg [BIT_LEN:1] result;
58
59 always @(posedge clk)
60   result[BIT_LEN:1] <= src1[BIT_LEN:1] & src2[BIT_LEN:1];
61 endmodule  // And
```

Fig. 2. Verilog source code of a simple ALU

the Verilog source code and passes the correct values for these parameters when creating module instances. As an example, `BIT_LEN` is defined as a parameter in module `my_add` of Fig. 2. Consequently, in lines 6 and 43 of Fig. 3, this Verilog module parameter is defined as a `nat` parameter for the `module_my_add` constructor and its body is defined in the `module_inst` function.

Module Instantiations: To support hierarchy, *VeriCoq* tracks module instantiations inside a module and defines them as parameters of the `module` definition in the Coq representation. For example, the `my_alu` module in Fig. 2 instantiates two modules named `add1` and `and1`. As lines 5 and 14 in Fig. 3 show, these modules are added to the definition of the `my_alu` module in its Coq representation.

VeriCoq automatically creates an axiom for the top module of the Verilog source code, representing the top module instantiation. For this purpose, *VeriCoq* creates the appropriate variables, parameters and module instantiations. As lines 60-77 in Fig. 3 show, to instantiate `my_alu`, *VeriCoq* defined the required variables and created parameters with their corresponding values assigned in the Verilog source code. It also created two module instances, namely `module_my_and` and `module_my_add`, in order to instantiate `module_my_alu`.

```
1  Require Import Vericoq.
2
3  Inductive module :=
4   | module_my_alu : bus->bus->bus->bus->bus->bus->
5     bus->bus->nat->module->module->module
6   | module_my_add : bus->bus->bus->bus->nat->module
7   | module_my_and : bus->bus->bus->bus->nat->module
8  .
9
10 Fixpoint module_inst (m:module) (t:nat) :=
11  match m with
12   | (module_my_alu result src1 src2 clk rst op_select
13      and_result compl_result ALU_BIT_LEN
14      module_my_and_and1 module_my_add_add1) =>
15       (adoif (
16        (aifsimple (econb rst) (
17         (anoif (expr_assign result
18          (econv (lo::lo::lo::lo::
19            lo::lo::lo::lo::nil))))
20        ))
21       ) t)
22     /\
23      (doif (
24        (ifsimple (enot (econb rst)) (
25         (ifelse (eeq_case (econb op_select)
26                  (econv (hi::nil)))
27          (
28           (noif (upd_expr result
29            (econb and_result)))
30          )
31          (
32           (ifsimple (eeq_case (econb op_select)
33                                (econv (lo::nil))) (
34            (noif (upd_expr result (econb compl_result)))
35           ))
36          )
37         )
38        ))
39      ) t)
40     /\
41      (module_inst module_my_and_and1 t) /\
42      (module_inst module_my_add_add1 t)
43   | (module_my_add result src1 src2 clk BIT_LEN) =>
44     (doif (
45      (noif (upd_expr
46       (result [(BIT_LEN - 1), (BIT_LEN - BIT_LEN)])
47       (eadd (bus_length
48              (result [(BIT_LEN - 1), (BIT_LEN - BIT_LEN)]) t)
49        (econb (src1 [(BIT_LEN - 1), (BIT_LEN - BIT_LEN)]))
50        (econb (src2 [(BIT_LEN - 1), (BIT_LEN - BIT_LEN)])))))
51     ) t)
52   | (module_my_and result src1 src2 clk BIT_LEN) =>
53     (doif (
54      (noif (upd_expr (result [(BIT_LEN - 1), (1 - 1)])
55       (eand (econb (src1 [(BIT_LEN - 1), (1 - 1)]))
56       (econb (src2 [(BIT_LEN - 1), (1 - 1)])))))
57     ) t)
58 end.
59
60 Variable src1 src2 clk rst op_select : bus.
61 Definition ALU_BIT_LEN := 8.
62 Variable result : reg.
63 Variable and_result compl_result : wire.
64 Definition and1_BIT_LEN := ALU_BIT_LEN.
65 Variable and1_result : reg.
66 Definition add1_BIT_LEN := ALU_BIT_LEN.
67 Variable add1_result : reg.
68
69 Axiom my_alu: forall (t:nat),
70  module_inst (module_my_alu result src1 src2 clk rst
71       op_select and_result compl_result
72      (module_my_and and_result src1 src2 clk  and1_BIT_LEN)
73      (module_my_add compl_result
74       (fun t => (eval (enot (econb src1)) t))
75       (fun t => (eval (econv (lo::lo::lo::lo::
76         lo::lo::lo::hi::nil)) t)) clk  add1_BIT_LEN)
77   ) t.
```

Fig. 3. *VeriCoq* generated Coq code for simple ALU

Part Selection: Selecting a part of a bus is common in Verilog statements. Since `bus_value` is defined as a list, PCHIP defines a function to select a portion of a list and uses the `[ , ]` notation to represent it. A challenge 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* normalizes indexes in part 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. 2 shows two methods of defining and using ranges. Specifically, the LSB of the `result` bus has index `BIT_LEN` in `my_add` module, while it has index 1 in `my_and` module. *VeriCoq* normalized the LSB to index 0 in both cases as seen in lines 46, 48 and 54 of the converted code in Fig. 3.

Expressions and Verilog Operations: PCHIP defines an inductive type `expr` in order to build expressions based on basic mathematical and logical operations of Verilog. It then defines constructors to build expressions based on these operations. *VeriCoq* converts these operations to their equivalent in Coq. PCHIP also defines the `econv` and `econb` constructors to convert `bus_value` and `bus` types to `expr`. For example, we point out the *add* operation in line 47 of Fig. 2, which is converted as lines 47-50 in Fig. 3. Constructor `eadd` represents the *add* operation whose first parameter is a number and determines the length of the result. As mentioned earlier, `econb` is used to convert `src1` and `src2` from `bus` to `expr`, which is the type of the other two parameters of `eadd`. Similarly, the logical *and* operation in line 60 of Fig. 2 is converted as lines 55-56 of Fig. 3. However, `eand` only gets two parameters of type `expr`. PCHIP also defines the `eval` function in order to compute the result of expressions and utilizes it in assignments and conditions.

Constants and Expressions in Module Instantiations: Verilog allows instantiating modules by connecting input ports directly to a constant value or an expression. However, signals, constants and expressions have different types in Coq representation. To support such module instantiations in Coq, whenever *VeriCoq* finds a connection to a constant or an expression in module instantiations, it creates anonymous functions converting a `bus_value` or `expr` to `bus`. For example, we point out the instantiation of `add1` in lines 30-31 of Fig. 2, which is converted to lines 73-76 in Fig. 3. *VeriCoq* created two anonymous functions, one to negate `src1` and the other to convert constant `8'h01` to `bus`.

Conditional, Combinational and Sequential Statements: PCHIP defines two distinct inductive types for conditions in a combinational or sequential block. To simplify working with the converted code, unconditional statements are considered a special case of conditionals without any condition. For sequential blocks, `noif`, `ifsimple` and `ifelse` constructors are used for no condition, if, and if-else statements, respectively. Constructors `anoif`, `aifsimple` and `aifelse` are used similarly to represent combinational blocks. Constructors `ifcons` and `aifcons` are used to link such statements together and create Coq code blocks in sequential and combinational cases. Since these constructors expect corresponding if blocks as their action, nested conditional statements can be converted to Coq seamlessly. The conditional constructors constitute the base structure of the code in its Coq representation.

To distinguish between combinational and sequential assignments, PCHIP defines two inductive types through the `expr_assign` and `upd_expr` constructors, respectively. The difference between these two is that in a combinational assignment, the computed result affects the left side in the current clock cycle, while in a sequential one, the result is computed for the next clock cycle. Lines 15-17 in Fig. 2 show a combinational block whose Coq representation is given in lines 16-20 of Fig. 3. Likewise, lines 46-47 of Fig. 2 are considered a sequential block and converted to lines 45-50 in Fig. 3. `adoif` and `doif` are functions which PCHIP defines to evaluate combinational and sequential conditional blocks.

VeriCoq also recognizes **`case`** statements as conditional blocks. To make the conversion process less complicated, *VeriCoq* unrolls **`case`** structures in Verilog and treats them as consecutive if-else conditions, while considering their appearance in combinational or sequential blocks. As an example, *VeriCoq* unrolled the **`case`** block of lines 21-24 in Fig. 2 and converted it to lines 25-37 in Fig. 3

Fig. 3 shows the complete code generated by *VeriCoq* for the simple example of Fig. 2, which is directly usable in the Coq environment to develop proofs for desired security properties. The conversion is quick and takes less time than compiling Verilog source codes for running a simulation. The first line of Fig. 3 imports a Coq library containing the PCHIP definitions. We are planning to extend the capabilities of *VeriCoq* by adding support for a few other Verilog statements such as **`generate`**, **`function`** and **`task`**.

IV. Conclusion

PCHIP introduces a new framework for enhancing trustworthiness of hardware IPs by accompanying their HDL code with formal proofs regarding their security properties. These properties are designed to confine the IP behavior in a definite boundary and prevent the introduction of hardware Trojans in the design. However, PCHIP adds extra steps in the hardware IP development process, among which conversion of the HDL code to a formal theorem proving language, such as Coq, is one of the essential ones. Towards automating this process, in this paper we introduced *VeriCoq*, a Verilog-to-Coq converter based on the rules developed in the PCHIP framework and we used a simple ALU example to describe how the various Verilog constructs are handled. *VeriCoq* supports all of Verilog's essential statements and ongoing work is extending its capabilities into a richer set. *VeriCoq* is a first attempt towards automation of the entire PCHIP framework. We continue our efforts in this direction in order to reduce the extra burden and make it more appealing for the hardware community to adopt and utilize the PCHIP framework towards enhancing trustworthiness of 3rd party hardware IP.

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] 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.
[2] 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.
[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 PART 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] T. Braibant and A. Chlipala, "Formal verification of hardware synthesis," in *Computer Aided Verification*. Springer, 2013, pp. 213–228.