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

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

***Abstract*— Proof-carrying hardware intellectual property (PCHIP) is a previously proposed framework for ensuring trustworthiness of third-party hardware IP through the development of formal proofs for security properties designed to prevent introduction of malicious behavior. Based on this framework, we introduce new approaches for assuring that the secrecy of internal information in a hardware design is not compromised by design flaws or malicious hardware Trojans. Specifically, we devise two PCHIP-based information flow tracking approaches, which enhance the formal PCHIP framework with secrecy tags and/or sensitivity levels in order to provide mechanisms for proving that sensitive information does not reach undesired sites. To assist in the development of data secrecy properties, we also introduce the concept of theorem generation functions, which enable generation of security theorems independent of the target circuit, thereby paving the way for proof automation. In addition, we enhance the PCHIP framework with a hierarchy-preserving methodology and we show its utility in preventing malicious data modification, which may indirectly result in sensitive information leakage, such as by modifying the secret key in a cryptographic core. This enhanced PCHIP framework also enables development of hybrid module libraries, which contain hardware description language code along with proofs of lemmas for these modules. These module libraries can then be used for hierarchically proving security properties in higher level designs, thereby reducing the proof development burden in the general PCHIP framework. Efforts toward automation of the proposed methodologies, as well as evaluation of their effectiveness in identifying design flaws or hardware Trojans in various cryptographic hardware designs are presented in part II of this paper series.**

***Index Terms*— Hardware trust, proof-carrying code, data secrecy protection, dynamic information assurance, information flow tracking.**

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.)*

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

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).

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.2017.2707323

I. Introduction

With sensitive data, such as user names, passwords, or banking information routinely being stored and communicated in our everyday use of electronic devices, data secrecy protection has become a key objective of computer security research. Encryption plays an important role in preventing the direct exposition of data to adversaries. However, culprits invest their utmost effort in getting access to such valuable data through various well-known or newly-discovered vulnerabilities of computer systems or through introduction of malicious software. Accordingly, numerous approaches have been devised toward reducing the likelihood of sensitive information leakage in such systems. Among them, information flow tracking (IFT) [1] is a powerful approach for preventing sensitive information from reaching untrusted sites. In this approach, labels representing secrecy/trust levels are assigned to data and operations on data are extended to include operations on their labels, based on predefined information flow policies. Access or propagation of data with sensitive labels is, thereby, restricted to trusted segments of the code or system and is forced to abide by the desired information flow policies.

IFT can be implemented at various levels and can approach the problem from different perspectives. Static IFT enforces information flow polices at compile time and does not introduce any overhead at runtime [2]. However, its scope of enforcement is limited. Dynamic IFT seeks to remove this limitation and implements IFT at runtime [3], at the cost of introducing performance and memory overhead in the system. To reduce this overhead and to make IFT more accurate, hardware-assisted IFT methodologies have also been introduced. For example, the authors in [4] proposed a dynamic IFT framework with all internal storage elements equipped with a security tag. Similarly, the authors in [5] focused on pointer-tainting in order to prevent both control data and non-control data attacks. Besides IFT, the hardware may also be enhanced to help with preventing information leakage, such as in the InfoShield architecture, which applies restrictions to operations on sensitive data [6]. Similarly, the RIFLE architecture is developed on top of an information flow security (IFS) instruction set architecture (ISA), where all states defined by the base ISA are augmented by labels [7]. More recently, a new software-hardware architecture was developed to support more flexible security policies, either for protecting sensitive data [8] or for preventing malicious operations by untrusted third-party OS kernel extensions [9]. Authors in [10] extended IFT to gate-level implementations by modifying the restrictive IFT operation definitions used in higher levels.

The aforementioned hardware-supported information tracking and protection schemes are based on the assumptions that the underlying hardware infrastructure is trusted and that any modifications to the hardware architecture have passed functionality and security verification. Yet the recent emergence of hardware Trojans and hardware-level back-doors as a plausible threat [11], [12] has casted doubt on the validity of these assumptions and has provided attackers with alternative leakage paths, which current IFT schemes do not cover [13], [14]. Indeed, globalization of the integrated circuit (IC) supply chain has intensified the concern that unknown and potentially malicious capabilities may be inserted in a circuit or system. While inclusion of such capabilities is possible at any stage [12], modifying the hardware at the design stage is far easier than tampering with its layout mask at fabrication. Additionally, hardware designers regularly use third-party hardware IPs to expedite the design process. Consequently, flaws or malicious capabilities in hardware IPs can easily expose the end system to attacks. *As a result, besides IFT in software, hardware-level information assurance is also required to ensure that the hardware itself obeys the desired information flow policies and does not leak sensitive data, and to establish system-level trust by detecting design flaws and/or hardware Trojans.*

To this end, in Part I of this paper series we introduce several variants of a formal methodology for assessing trustworthiness of a design described in a hardware description language (HDL). These variants build upon a previously proposed proof-carrying hardware IP (PCHIP) framework [15], which we enhance in order to support IFT and hierarchy-preserving proof development. Then, in Part II, we describe our efforts towards automating the enhanced PCHIP framework and we evaluate the applicability and effectiveness of its capabilities in detecting design flaws and/or hardware Trojans in various cryptographic cores. The remainder of this paper is organized as follows. Section II introduces the threat model considered in this work, Section III reviews related efforts, and Section IV provides an overview of the proposed PCHIP-based methodologies for data secrecy protection. In Section V, we review the principles of the original PCHIP framework. In Section VI, we provide details of the two formal models devised for supporting IFT within the PCHIP framework. In Section VII, we review the existing flattened formal model in the original PCHIP, we present the new hierarchy-preserving alternative, and we describe how the enhanced PCHIP framework can be used to prevent malicious data modification. Conclusions are drawn in Section VIII.

II. Threat Model

The methods proposed in this work seek to detect *sensitive information leakage*, caused either by intentional malicious modifications or by inadvertent flaws in the design of a circuit. Accordingly, our work mainly targets circuits handling sensitive information, such as cryptographic hardware. We focus on designs described as HDL code (e.g. soft hardware IP) and we assume that any leakage will occur in digital form, through *primary physical outputs* of the design, to which an adversary is expected to have access. Since the proposed methods are only able to reason in the digital domain, information leakage through side channels (timing, power, etc.) is not directly considered in our work. However, as we discuss in Part II of this paper series, a knowledgeable user may be able to leverage the proposed methods in order to reveal suspicious internal locations in the design, where the digital information leaked through side-channels may be originating from.

Our threat model assumes that the attack surface is the HDL description of the design. Accordingly, the attacker resides either at the site of a 3rd party IP provider, if the design is sourced externally, or in-house, if the design is performed internally. Evidently, a design sourced as 3rd party IP, which is later modified in-house, also falls within the scope of our methods. We emphasize that, while a large portion of the hardware security literature focuses on the inclusion of hardware Trojans at the fabrication stage, other stages of hardware design and fabrication are not immune to such threats [12]. In fact, the threat model wherein malicious capabilities are introduced through 3rd party IP has attracted similar levels of attention and numerous methodologies targeting hardware Trojans introduced at the design stage have been developed, as we briefly review in Section III. Indeed, as the use of third-party hardware IPs has become ubiquitous in contemporary hardware design, this threat is intensified as not all IP vendors can be trusted. Even in the case of a trusted hardware IP vendor, a single rogue adversary involved in the design, acquisition or utilization of the hardware IP can introduce malicious capabilities in the design. Moreover, undiscovered design flaws and in-house attacks can also compromise the hardware at the design stage. Therefore, developing methods for detecting sensitive information leakage due to intentional or inadvertent vulnerabilities in the design of an electronic circuit has become paramount.

III. Related Work

Several methodologies have been introduced for identifying hardware Trojans introduced at the HDL code of hardware designs and soft hardware IPs. For example, FANCI [16] uses statistical analysis to detect nearly unused logic in a design, which may be suspicious. Similarly, VeriTrust [17] searches for dedicated Trojan triggers in the design. While such methods are systematic, smart Trojans may still evade their checking mechanisms [18]. Alternatively, COTD [19] employs controllability and observability analysis of a gate-level netlist to mark potential Trojan gates. Through controllability- and observability-based clustering, it asserts that Trojan gates have significant inter-cluster distance from genuine gates, and therefore, may be distinguished. Other approaches employ test patterns, specifically generated and directed towards detecting hardware Trojans [20], [21]. Such methods have been gradually improving, yet are still faced with a very large space of options, which they need to intelligently prune. Furthermore, none of the above methods focuses on information leakage.

TABLE I
SUMMARY OF PROPOSED PCHIP-BASED METHODOLOGIES

Method	Type	Complexity	Expressiveness	Ease of use
Static information flow tracking	IFT	Low	Low	Easy
Dynamic information assurance	IFT	Moderate	Moderate	Moderate
Flattened functional model	General	High	High	Hard
Hierarchical functional model	General	High	High	Moderate

Several language-level approaches have also been introduced, seeking to enforce information flow policies on hardware designs. Caisson [22] is a hardware description language with static information flow verification capabilities at design time. Sapper [23] provides a language which employs static analysis at compile-time but inserts dynamic checks in the resulting hardware in order to enforce information flow policies. SecVerilog [24] enforces information flow policies by introducing a type system. It is essentially Verilog, which has been extended with type annotations. SecVerilog is powerful and provides a very accurate information flow model by supporting dependent security types (i.e., labels defined as a function of signal values). Also, while such methodologies are very useful in designing secure hardware, such as microprocessors, their main focus is on storage and timing channels, not on possible hardware Trojans. Moreover, as explicitly stated in the threat model considered in [24], these methods assume a software-level adversary, as opposed to the hardware-level adversary considered in our work.

Finally, a method which detects hardware Trojans leaking sensitive information based on gate-level IFT is introduced in [25]. While this method takes advantage of precise IFT at the gate-level, it requires exploration of the entire signal/value space using a SAT solver and/or a model checker, which results in state explosion and limits its scope. Moreover, to evade detection in this approach, it is sufficient for an adversary to have only one operation which propagates the key to the output (e.g. eXclusive-OR), while hijacking all the encryption rounds in a cryptographic core.

IV. PROPOSED DATA SECRECY PROTECTION METHODS

The proof-carrying hardware IP (PCHIP) framework [15] introduced a formal methodology for ensuring trustworthiness of soft hardware IPs and designs delivered as code in HDL. PCHIP augments the hardware IP with machine-checkable formal proofs of security properties which are agreed upon by the IP developer and the IP consumer and which are crafted in a way that prevent inclusion of malicious capabilities violating these properties in the design. The IP consumer may, then, use an automatic proof checker to verify the proofs and ensure that the security properties are obeyed by the design.

The original PCHIP framework does not provide any means for enforcing information flow policies, which are particularly important for ensuring trustworthiness of cryptographic cores and hardware IPs that process sensitive data. Nevertheless, PCHIP is quite flexible and can serve as a foundation for developing an array of IFT methodologies, each of different complexity, expressiveness and ease of use. Accordingly, in this paper we describe four such PCHIP-based methodologies seeking to protect data secrecy in hardware designs. Table I contrasts these four methods, enabling a user to select the most appropriate one, based on the design requirements and the security properties that need to be proven.

The first two of these methods, which we introduce in Section VI, are specifically crafted to incorporate IFT in the PCHIP framework and aim at preventing sensitive information leakage. The *static information flow tracking* [26] method provides a starting point which demonstrates how the HDL code of a design can be converted to a formal theorem proving language representation wherein IFT is supported. This initial option is straightforward and easy to apply, but its scope is limited and it can only handle simple hardware designs. In contrast, the *dynamic information assurance* methodology [27] which we introduce next, adds support for more complex and pipeline designs, at the cost of somewhat increasing complexity and requiring further effort to apply. While early versions of these two approaches were presented as part of our initial efforts [26], [27], herein we extend them and present them in more detail, emphasizing their integration within the generalized PCHIP framework. Additionally, we introduce the notion of *security property theorem generation functions*. These functions can be used to generate security theorems for information flow policies, independent of the target circuit, and are an essential step towards automating the IFT framework. It is worth noting that these two models are geared towards manual conversion of the design to a formal theorem proving language representation and enforcement of information flow policies, and they intentionally rely on specific procedures for making such conversion simpler while retaining the essential requirements for performing IFT in the formal representation. Moreover, as compared to the original PCHIP framework which retains the entire functionality, these methods preserve the structure but only partially retain the functionality of the hardware design. For example, all binary operations are represented in the same way in the formal representation. While efforts towards automating these methods are presented in Part II of this paper series, we consider it essential to first describe the manual conversion-based models in order to help in grasping the basics of PCHIP-based IFT.

The last two of these methods, which we introduce in Section VII, are motivated by the observation that information leakage may occur not only through manipulation of information flow but also through manipulation of sensitive data, such as the secret key in a cryptographic core, which may not be directly detectable through our IFT methods. Therefore, in order to prevent such malicious modification of sensitive data and ensure integrity, we resort to a solution employing the general PCHIP framework which, as opposed to the two IFT methodologies described above, converts the entire functionality of the design to the formal representation. The first of these two methods utilizes the *flattened functional model* first introduced by the general PCHIP framework. The second method introduces a *hierarchical functional model*, which is based on the flattened model but preserves the hierarchy of the hardware design in the formal representation. For each of these two options, we describe how this framework can be utilized to develop security properties which can prevent malicious modification of sensitive data in hardware designs.

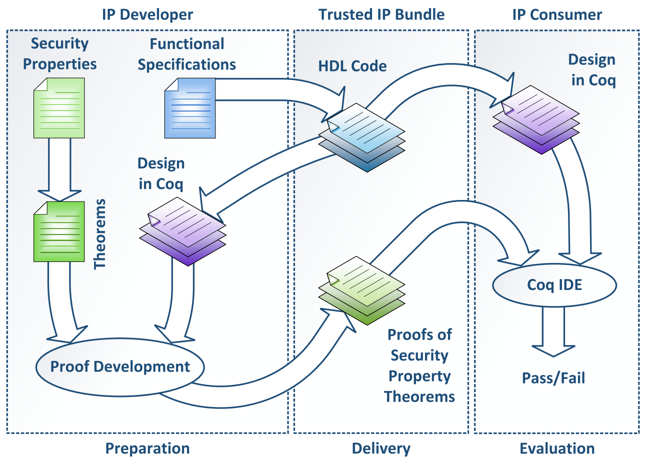

Fig. 1. PCHIP framework.

While either the flattened or the hierarchical functional model can be utilized for this purpose, the hierarchical approach makes proof development and reuse of previously proved lemmas significantly easier and is more appropriate for large designs. However, the flattened model may also provide some advantages. As we will demonstrate in Part II of this paper series, the flattened model allows us to build a fully automated PCHIP-based IFT framework. Also, flattened and hierarchical functional models can be mixed together, creating a combined model. In this model, larger modules are converted hierarchically while the modules instantiated inside them use the flattened approach. This provides a degree of flexibility to developers who may prefer to develop lemmas only for larger modules and do not want to overuse modules in the formal representation.

V. 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 the HDL code for a design, IP developers are required to develop and deliver another essential piece: formal proofs that the code abides by a set of security properties that are agreed upon by both the IP developer and the IP consumer. These properties do not necessarily impose restrictions on the details of implementation. Rather, they institute a high-level boundary of trusted functionality, which prevents misbehavior or unsolicited actions. For example, a security property for a microprocessor IP could be defined as follows: Each instruction is only allowed to access memory locations which are specified in the corresponding fields of its op-code [28]. This property prevents stealthy information leakage. However, it does not restrict the details of instruction implementation.

Mechanized proof development and checking requires a theorem-proving language and a proof-checking environment, such as Coq and CoqIDE [29], 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 [30]. Rather, it adds extra steps in parallel to the current design methodology, namely converting to Coq, stating security properties as theorems in Coq, constructing proofs for such theorems based on the hardware design, and delivering those proofs along with the HDL code to the IP consumer. PCHIP does not inflict IP consumers with much extra burden. Along with the IP developers, they need to agree on the desired security properties. The onerous task of proof development is, then, the responsibility of the IP developers. PCHIP can be employed in various types of hardware and can be adaptively modified to fit the requirements of the design and the IP consumer.

While PCHIP establishes a very powerful formal framework for proving security properties of an IP, it does not provide any means for enforcing information flow policies. Nevertheless, the underlying representation of a hardware design in the Coq formal language can be flexibly varied to develop such a provision. In the following section, we describe how IFT capabilities can be added to the original PCHIP framework.

VI. Information Flow Tracking in PCHIP

In this section, we enhance the PCHIP framework to support IFT in the HDL description of an IP, with the intention of using this capability for analyzing the security of cryptographic hardware cores. We start with a static IFT methodology, targeting small and simple designs, and we then evolve it into a dynamic IFT methodology which can enforce information flow policies for larger, more complicated and pipelined hardware designs. For each methodology, we describe how to express the hardware design in the formal representation required for the verification of information flow policies.

A. Static Information Flow Tracking

The first formal model we introduce, named *Coq formal semantic model*, is used in the static IFT scheme and is developed in such a way that it can precisely describe the structure of the circuit but has loose restrictions on the functionality of any operators. In other words, the architecture of the circuit is accurately described in the formal semantic model but proof writers have flexibility in defining the functionality of the Coq circuit. This formal semantic model, as we will demonstrate later, is quite effective in statically tracking information flow inside the circuit in the PCHIP framework, supporting the target objective of data secrecy property verification. The formal semantic model includes preliminary definitions of signals, syntax of complex expressions, and semantics of operators.

1) Signal Definition: Values of signals are defined in an inductive set with two constructors, `hi` and `lo`, indicating high voltage level and low voltage level, respectively. Instead of defining one-bit signals and multi-bit buses separately, we unified both definitions under the bus scope, i.e., a one-bit signal is treated as a one-bit wide bus. The bus is then defined as a mapping of time, specified in clock cycles and given as a natural number, onto a `bus_value`, which is composed of a list of signal values. The natural number `t` defines an important property in temporal logic, namely that values of buses vary according to the system clock cycles. We also define a function to obtain the width of the bus, bus_length. These definitions are written in Coq as follows:

```
Inductive value := lo | hi.
Definition bus_value :=  list value.
Definition bus := nat -> bus_value.
Definition bus_length (b : bus) :=
  fun t : nat => length (b t).
```

2) Signal Operations: We construct bus handling methods in the formal semantic model which include logic operations such as and, or, xor, etc., as well as bus comparisons such as checking for bus equality, bus_eq, less-than comparison, bus_lt, etc. Conditional statements of Register Transfer Level (RTL) code, such as if..else, check whether signals are on or off. To incorporate this functionality in our formal semantic model, we add a special function, bus_eq_0, which compares the bus value to hi or lo. Note that Fixpoint defines a recursive function in Coq.

```
Fixpoint bv_bit_and  (a b : bus_value)
struct a : bus_value :=
  match a  with
 | nil => nil
 | la :: a' =>
    match b  with
   | nil => nil
   | lb :: b' => (v_and la lb)::(bv_bit_and a'
    b')
    end
  end.

Definition bus_bit_and (a b : bus) : bus :=
  fun t:nat => bv_bit_and (a t) (b t).

Fixpoint bv_eq_0 (a : bus_value)
struct a : value :=
  match a  with
 | hi :: lt => lo
 | lo :: lt => bv_eq_0 lt
 | nil => hi
  end.

Definition bus_eq_0 (a : bus) (t : nat) :
value := bv_eq_0 (a t).
```

3) Bus Slicing: In a circuit, it is often the case that operations are performed on certain bits of the bus, but not the entire bus. Most hardware description languages, therefore, provide quite flexible syntax to define bus length and bus bit-sequence. In order to support similar flexibility in our Coq semantic model, we developed two bus-slicing operations to shuffle data bits from lower bit positions to higher bit positions and vice-versa. Bit selection notations are also proposed to simplify code writing, such as [,] for the sliceD function defined below. The firstn and skipn functions used in these definitions get a number n and a list l, and return a list containing the first n elements of l, or a list skipping the first n elements of l, respectively.

```
Definition sliceA (b : bus) (p1 p2 : nat) : bus
:=  fun t : nat => firstn (p2-p1+1) (skipn
 (p1-1) b).

Definition sliceD (b : bus) (p1 p2 : nat) : bus
:=  fun t : nat =>
 rev (firstn (p1-p2+1)) (skipn p2 (rev b)).

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

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

4) Expressions: On top of signal definitions and operation rules, we build expressions to represent more complicated circuit logic. An expression is defined as an inductive set with operators to construct new expressions or combine expressions. Plenty of operators are supported, varying from basic logic operations (AND, OR, etc.) to sophisticated data manipulation (S-box mapping, permutation, etc.). The expression definition shown below is an excerpt from the complete expression definition, wherein operators have been chosen with particular attention to common tasks performed in cryptographic IP cores, since it is highly likely that data secrecy properties will have to be proven for such designs. A constant value list and a bus can be directly converted to expressions using the econv and econb constructors, respectively. The eand, eor and exor constructors connect two expressions to form a new expression, by performing logical AND, OR and eXclusive-OR operations, respectively. The exor_key also performs the eXclusive-OR operation with keys (or subkeys) as one input (details of this operation will be elaborated on in Section VI-A.8 and Section VI-B.2). The perm and sbox constructors are used to indicate permutation and S-box mapping operations. In this Coq semantic model, it is unnecessary to specify how the permutation and/or S-box mapping is actually performed. These structural constructors liberate the proof writers from tedious functional conversion, which may be unnecessary for data secrecy property checking and enforcement of information flow policies.

```
 Inductive expr :=
| econv : bus_value -> expr
| econb : bus -> expr
| eand : expr -> expr -> expr
| eor : expr -> expr -> expr
| exor : expr -> expr -> expr
| exor_key : expr -> expr -> expr
| enot : expr -> expr
| cond : expr -> expr -> expr -> expr
| perm : expr -> expr
| sbox : bus -> expr
 (* ... *)
```

Evaluation of expressions is recursively defined to calculate the value of an expression at a specified time (denoted by the t parameter) and return data of type bus_value, a list of values whose length depends on the width of the underlying bus. A close look at the eval function supports our claim that some expressions, such as perm, only denote that a permutation operation will be performed on the underlying bus, but do not specify the exact nature of the permutation. Of course, other expressions, such as eand which performs a logical AND on two sub-expressions, result in a case where both functionality and structure are fully specified.

```
Fixpoint eval (e : expr) (t : nat)
struct e : bus_value :=
  match e  with
 | econv v => v
 | econb b => b t
 | eand ex1 ex2 =>
    bv_bit_and (eval ex1 t) (eval ex2 t)
  | eor ex1 ex2 =>
      bv_bit_or (eval ex1 t) (eval ex2 t)
  | enot ex =>
      bv_bit_not (eval ex t)
  | cond cex ex1 ex2 =>
        match (bv_eq_0 (eval cex t))  with
        | hi => eval ex1 t
        | lo => eval ex2 t
  | perm ex => eval ex
  | sbox b => b t
   (* ... *)
   end
```

5) Coq Representation: The definition of signals, expressions and their semantic models paves the way for converting RTL circuits into Coq representation. When choosing code constructors, we sought to make the new semantic model user-friendly. The constructor `outb` is used to denote output signals of the module. Similarly, `inb` means input signals; `wireb` represents internal wire signals; and `regb` denotes the internal registers (note that, similar to HDLs, the `reg` type does not necessarily result in actual registers in the synthesized model). Two assignment constructors are also defined, namely `assign_*`, which works for combinational logic, and `nonblock_assign_*`, which is appropriate for non-blocking assignment in sequential logic. An extra notation is added to pile the code through the ';' symbol. The selection of the ';' mark is consistent with the syntax of HDLs.

```
 Inductive code :=
| outb : bus -> code
| inb : bus -> code
| wireb : bus -> code
| regb : bus -> code
| assign_ex : bus -> expr -> code
| assign_b : bus -> bus -> code
| assign_case3 : bus -> expr -> code
| nonblock_assign_ex : bus -> expr -> code
| nonblock_assign_b : bus -> bus -> code
| codepile : code -> code -> code.

 Notation  " c1 ; c2 " := (codepile c1 c2)
  (at level 50,  left associativity).
```

6) Verilog-Coq Conversion Rules: Although multiple HDLs are available, we chose Verilog as the sample HDL to compose IP cores. The fundamental Verilog-to-Coq conversion rule which we need to obey is to keep the original code and destination code structurally the same. This forms the basis for the Verilog-to-Coq conversion methodology that we developed. For example, a combinational assign logic is mapped to an `assign_ex` statement and module instantiation is mapped to a `module_inst` statement as shown below.

Verilog code:

```
assign Lout = (roundSel == 0) ?  IP[33:64] :
 R;
```

Converted Coq representation:

```
assign_ex  Lout (cond (eq (econb roundSel)
  (econv (lo::lo::lo::lo::nil)))
  (econb ( IP @ [33, 64])) (econb  R));
```

Verilog code:

```
crp u0 (. P(out), . R(Lout), .K_sub(K_sub));
```

Converted Coq representation:

```
module_inst2in out  Lout K_sub;
```

7) Tracking the Information Flow: In itself, the new formal semantic model still cannot achieve the goal of facilitating theorem proving for information flow policies, since it is simply an alternative HDL to represent the circuit structure. Information-leaking hardware Trojans can still use signal bypassing strategies, which propagate sensitive internal data to primary outputs [13] or disseminate it through Trojan side channels [14], with little modification of the original circuit. However, the circuit description is now in a language that lends itself to formal reasoning. Therefore, by adding appropriate elements (i.e., secrecy tags) and logic for formally reasoning on these elements, we can now support IFT.

To facilitate tracking of internal sensitive data and proving of secrecy properties on this data in the new semantic model, we enhance circuit signals with an additional property, namely *sensitivity*. This property is akin to the existing *bus_value* property and it allows us to formally examine and prove signal integrity (from a security point of view) within the entire design. The chosen Coq platform for our formal semantic model comes in handy, since it is easy to enhance the Coq formal logic to support information flow tracking. The only significant change is that we need to extend the bus definition so that it will return a `bus_value*sensitivity` pair at a specified time `t` instead of just a `bus_value`. Sensitivity is defined as an inductive set with two constructors, `sensitive` and `non_sensitive`, indicating whether the signals are sensitive and need protection or not. Any output signals of the target circuit should be of `non_sensitive` secrecy tags at any time `t`. Otherwise we claim that data leakage channels exist in the circuit and the data secrecy properties are violated.

```
Inductive sensitivity := sensitive |
non-sensitive.
Definition bus := nat -> (bus_value *
sensitivity).
```

8) Security Policies: A security policy defines the legitimate propagation rules for data secrecy tags as the corresponding signals travel from inputs through bus operations to outputs. Three such rules are defined in the static IFT scheme: *(i) Signal assignment:* When a bus is the operand of a unary operator, the unary operator keeps the secrecy tag of the bus; *(ii) Logic/Functional operations:* For most of the logic/functional operations, if any of the input signals are of `sensitive` secrecy tag, then the operational outputs need protection and are assigned as `sensitive`; *(iii) Secrecy declassification:* In order to prevent leakage of sensitive internal data by illegitimate "declassification", it is important to restrict the ability of removing the `sensitive` tag to a few well-controlled operators. As a general rule, in cryptographic cores, eXclusive-OR operations between round keys and intermediate results are considered as sensitivity reducers. Similarly, since internal modules are verified independently, module instantiations are also considered as sensitivity reducers. Examples are provided in the context of the cryptographic cores on which IFT is demonstrated in part II of this paper series.

B. Dynamic Information Assurance

The methodology of Section VI-A provides a static method for reasoning about information flow policies in a hardware design. However, it has some restrictions. Specifically, reasoning about multiple rounds of cryptographic or declassifying operations in a design, such as in a pipelined implementation, is cumbersome and may not be accurate. In this section, we enhance this static methodology to be able to dynamically track the flow of information in a hardware design and eliminate such restrictions. Moreover, as we will describe in Part II of this paper series, PCHIP with dynamic IFT is a more suitable framework for automation. For this purpose, a second formal model, called *structural Coq formal logic*, is developed to represent circuit logic in the Coq platform and to dynamically track information flow within the circuit.

1) Basic Definitions: The structural Coq formal logic is defined in such a way that it can accurately map the data secrecy-related structure of the original circuit to its Coq representation, while leaving the circuit functionality unspecified. In contrast to the Coq formal logic of the static scheme, the Coq representations are significantly simplified in the new structural Coq formal logic, since circuit functionality does not need to be explicitly specified. As defined in the structural Coq formal model, signal values in Coq circuits represent their sensitivity levels, not electronic values. Furthermore, circuit signals are not just treated qualitatively as `non_sensitive` or `sensitive`, as in the static scheme, but are also quantitatively allocated integers: A number 0 means `non_sensitive` while a positive number indicates `sensitive`. A larger number relates to a higher level of sensitivity, implying that the underlying signal requires higher level of protection. For example, non-critical control/data signals, such as input clock signal, loading control, etc., are set to value 0, whereas encryption/decryption key and plaintext are assigned positive integers. As cryptographic algorithms usually consist of multiple rounds, this enables easier handling of pipeline designs. However, in simpler designs, two-state labels may also be employed for the same purpose. As we will introduce shortly, all signal sensitivities are managed in a central way (i.e., through a signal sensitivity list) and the definition of bus is only a number indicating its position in the list. Other definitions, such as bus slicing, expressions, etc., are similar to the formal model introduced in Section VI-A for the static scheme, with the exception that the calculation will only reflect updating of the bus sensitivity levels rather than the electronic values.

```
(* The definition of bus is only a number
indicating the position of the bus in
sensitivity tag list *)
Definition bus := nat.
```

2) Signal Sensitivity Transition Model: We, then, need a mechanism to depict the way in which signal sensitivity levels dynamically evolve when signals pass through circuit logic. This task is performed by a newly developed signal sensitivity transition model that involves a set of rules which put restrictions on how to upgrade/downgrade (or increase/decrease) signal sensitivity levels.

To ensure the integrity of the transition model and to prevent sensitive information leakage from illegal signal sensitivity downgrading operations, the signal sensitivity transition model is made conservative, i.e., only a small set of Coq circuit operations are allowed to downgrade signal sensitivity levels. The set of sensitivity downgrading operations can be further divided into two groups discussed below, *sensitivity downgrading expressions* and *module instantiation*.

a) Sensitivity downgrading expressions: These expressions are defined with similar syntax to other expressions but often perform special operations with sensitive data involved. In case of cryptographic circuits, the general rule for downgrading sensitivity of signals is to perform the eXclusive-OR operation between sensitive data and the key (or subkey). For example, eXclusive-OR of round keys with sensitive data is the only sensitivity downgrading expression for an advanced encryption standard (AES) core. We note that not all eXclusive-OR operations are sensitivity reducers. Also, sensitivity downgrading operations are design-specific and are determined based on a clean high-level architecture (or block diagram), independent of the IP vendor implementation. The IP consumer and the IP vendor should agree on these definitions beforehand, while the IP consumer should verify them in the final design. Extension and generalization of sensitivity downgrading operations through evaluation of a broader set of designs remains as an interesting topic of further investigation.

The recursive function `expr_sen_eval` provides detailed information on how to evaluate the sensitivity level of an expression. Specifically, for most operations the sensitivity level of an entire expression is equal to the highest sensitivity level among all operands and is obtained by calling the maximum number selection function `boptag`, with a few exceptions of sensitivity downgrading expressions.

```
Definition boptag (a b : nat) : nat := max a b.

Fixpoint expr_sen_eval (e : expr) (sl :
code_sen)
struct e : bus_expr_sen :=
  match e  with
  | econv v =>  O
  | econb b => nth b sl 0
  | eand ex1 ex2 => boptag (expr_sen_eval ex1
   sl) (expr_sen_eval ex2 sl)
  | eor ex1 ex2 => boptag (expr_sen_eval ex1
   sl) (expr_sen_eval ex2 sl)
  | exor ex1 ex2 => boptag (expr_sen_eval ex1
   sl) (expr_sen_eval ex2 sl)
  | eand_bit b => nth b sl 0
  | eor_bit b => nth b sl 0
  | exor_bit b => nth b sl 0
  | eplus ex1 ex2 => boptag (expr_sen_eval ex1
   sl) (expr_sen_eval ex2 sl)
  | eminus ex1 ex2 => boptag (expr_sen_eval ex1
   sl) (expr_sen_eval ex2 sl)
  | enot ex => expr_sen_eval ex sl
  | eapp b1 b2 => boptag
     (nth b1 sl 0) (nth b2 sl 0)
  | perm ex => expr_sen_eval ex sl
  | exor_key b key => lowertag
     (boptag (nth b sl 0) (nth key sl 0))
   (* ... *)
```

As may be observed in the definition of the boptag function, it selects the maximum of two sensitivity levels. As an example, consider the evaluation of the logical AND operation, as defined in line 6 of the expr_sen_eval function, where each operand is recursively evaluated and the maximum of the two sensitivities is selected as the result of the evaluation. The lowertag function reduces the sensitivity level by 1 and is used for the evaluation of sensitivity downgrading operations, e.g. exor_key in the above definition, as opposed to the regular xor which is evaluated similar to other binary operations. For the evaluation of a constant and a bus, 0 and the current sensitivity level of the bus in the sensitivity list is returned, respectively, as seen in lines 4-5 of this function.

b) Module instantiation: Almost all modern designs are of hierarchical structure with submodules instantiated to perform various functionalities, from round key generation to memory control. Unless the entire design is flattened, this hierarchy makes it quite difficult to track information flow in and out of submodules in our formal environment. To prevent attacks targeting the interface between higher level modules and their submodules, we propose a sensitivity reshuffling strategy under which output signals from submodules are denoted as input signals of the top module. These signals (outputs from submodules) are called *endogenous* inputs, as opposed to primary inputs which, hereafter, are called *exogenous* inputs. All sensitivity assigning and transition rules that apply to exogenous inputs are also valid for endogenous inputs. This reshuffling strategy eliminates the relation between inputs and outputs of submodules, so that submodules can adjust signal sensitivity levels independently, including sensitivity level downgrading operations. This method of handling submodules and their outputs makes the manual conversion of the design to its Coq representation simpler. A flattened, automated method for handling module instantiations, which does not require defining endogenous and exogenous inputs, will also be presented in part II of this paper series where we discuss framework automation efforts.

3) Implicit Information Flow: Implicit information flow is also considered in our sensitivity transition model as attackers may leverage conditional statements to indirectly leak internal information. To solve this problem, a conservative signal sensitivity transition rule is applied, such that the output of the conditional statement maintains the sensitivity of any of the inputs, including the condition clause. In our dynamic scheme, this rule is further specified by the expr_sen_eval function in Coq as follows:

```
Fixpoint max_list (l :  list nat) : nat :=
  match l  with
 | nil =>  O
 | a :: rl => max a (max_list rl)
  end.

Fixpoint expr_sen_eval (e : expr) (sl :
code_sen)
struct e : bus_expr_sen :=
  (* ... *)
 | cond cex ex1 ex2 => max_list
     ((expr_sen_eval cex sl) ::
     (expr_sen_eval ex1 sl)  ::
     (expr_sen_eval ex2 sl)  :: nil)
  (* ... *)
```

The max_list function selects the maximum in a list of numbers. Using this function, the expr_sen_eval function selects the maximum sensitivity among the conditional clause and the branches when evaluating conditional statements.

This conservative information flow approach may result in false positives, i.e., the design is secure, but the proof for security theorems cannot be constructed and our framework marks it as insecure. To resolve this limitation, further research and development of advanced and more accurate PCHIP-based information flow models is required.

4) Signal Sensitivity List: To facilitate the operation of the signal sensitivity transition model, all signal sensitivity levels in the target circuit are managed in a centralized way. The entire sensitivity status of a circuit at a specified time t is stored in one signal sensitivity list, wherein each element represents the sensitivity level of one input, output, or internal signal. IP consumers can easily check the validity of the data secrecy property by defining the initial status of the sensitivity list, i.e., the starting point from which the sensitivity information spreads across the whole circuit, and then monitoring sensitivity levels of all output signals. Although the data secrecy properties, which serve as the basis for the proposed scheme, are independent of the circuit functionality and architecture, generation of the initial signal sensitivity list is closely related to the circuit structure and its functional specification. Guidelines are developed below for both IP vendors and IP consumers to initialize sensitivities for all signals, including inputs, outputs, and internal signals.

a) Input signals: The assignment of input signal sensitivity levels, including both exogenous inputs (primary inputs) and endogenous inputs (submodule outputs), can be divided into two steps: (i) deciding whether input signals contain secret information, and (ii) computing the sensitivity level of input signals if they contain secret data.

The first task is mostly completed upon analysis of circuit functionality and is relatively easy according to the circuit specification. For example, a DES encryption core would have plaintext, key, clock signal, round count, and reset signal as exogenous inputs and round keys as endogenous inputs. From the DES specification, IP vendors/consumers can recognize that exogenous inputs (plaintext, key, round count) and endogenous inputs (round keys) contain sensitive information so that their sensitivity levels should be positive integers requiring protection against information leakage attacks. Other inputs, such as the clock signal, do not contain sensitive information, so their sensitivity levels are set to 0.

After categorizing input signals into sensitive or non-sensitive, we proceed to the second task, which is to decide the actual sensitivity levels for sensitive signals, with larger numbers indicating higher sensitivity. The calculation process is closely related to the circuit architecture designed by IP vendors, particularly dominated by (pipeline) stages of the circuit implementation. While we considered more complex sensitivity level determination algorithms, we opted for a simple sensitivity-level downgrading counting method which is proven very effective in our later demonstrations. According to this method, after producing the HDL code and constructing the circuit architecture, the IP vendor will check all paths from sensitive inputs to primary outputs and will count the number of sensitivity downgrading operations along each route. The sensitivity levels of input signals are then set to the smallest count of sensitivity downgrading operations among all paths. Choosing the smallest count ensures that the output sensitivity becomes zero upon evaluation of a legitimate design in Coq representation.

Upon receiving the HDL code and the description of the circuit architecture, the IP consumer will check the validity of input signal sensitivity levels using the same method. Because the IP vendor may sabotage the circuit by adding extra sensitivity downgrading logic to "bleach" sensitive signals, and make the outputs non-sensitive, all downgrading operations must be clearly annotated with notes explaining why and how these operations are performed. We would like to emphasize that the IP consumer should perform this validation on a high-level architecture or block diagram of the intended implementation. In other words, IP consumers compute the initial sensitivity values using a clean block diagram of the design, and utilize these values in the HDL code delivered by the IP vendor to verify the proofs. Checking the validity of sensitivity reducing operations is also performed similarly, based on a clean block diagram of the design. This enables an independent review of initial sensitivity values and sensitivity reducing operations.

b) Internal signals and output signals: All signals other than input signals are treated as non-sensitive with level 0. This is because all internal and output signals have preset (or random) values when the circuit is reset or powered off, since in this case they do not contain any sensitive information. Exceptions may occur for storage elements. For example, non-volatile memory can keep stored values at power-off mode and may already contain sensitive information at the moment the circuit is powered on. This problem is solved because memory is always treated as a submodule with all outputs categorized as endogenous inputs under the reshuffling strategy.

As an example application of the above guidelines, an initial signal sensitivity list for a DES core is shown below, where the plaintext input, the key, the round count, and the internally generated sub-key are located at the zeroth, the first, the third, and the fifth position, respectively.

```
Definition des_sen_initial : code_sen :=
    1::1::0::1::0::1::0::0::0::0::0::0::0::
    0::0::nil.
```

The initial signal sensitivity list, combined with the signal sensitivity transition model, helps both IP vendors and IP consumers track the progress of how sensitive information is propagated and finally absorbed inside the chip.

5) Theorem Generation Functions: To facilitate conversion of data secrecy properties from natural language to Coq theorems and to assist with automation of the framework, the dynamic information flow tracking scheme introduces a new concept, namely *theorem generation functions*.

A theorem generation function takes Coq circuits and sensitivity lists as parameters and generates data secrecy theorems in the Coq platform. The use of theorem generation functions is a breakthrough in the field of proof-carrying based hardware IP protection. It simplifies the theorem generation process for the IP vendor and it provides a safety control capability to the IP consumer, who may later check the validity of the initial sensitivity list. Theorem generation functions also separate the tasks of circuit design and theorem generation, a major step toward EDA tool development for theorem and proof auto-generation, and makes it possible to integrate the proof construction task into the standard IC supply chain.

Before delving into the details of theorem generation functions, we need to introduce a special sensitivity list that represents circuit security status in a stable mode, if one exists.

Definition 1 (Stable sensitivity list): A stable sensitivity list is a special sensitivity list containing circuit secrecy status at a specified time `t`, with the key characteristic of stability, meaning that further evaluations of the code (at larger timestamps `t`) do not change the sensitivity levels of the signals. Denoting `coq_circuit` as the converted Coq circuit and `stable_list` as the stable sensitivity list, if `stable_list` represents the current circuit secrecy status, then the circuit status will not change until the circuit is reset (or is powered off). In the Coq platform, if the signal sensitivity evolvement function is `update_sensitivity`, the stability characteristic is presented in the form:

```
update_sensivitity coq_circuit stable_list
                                = stable_list.
```

Described in natural language, in the scope of the dynamic scheme, the data secrecy property means that "no sensitive data has leaked through primary outputs". Supported by the signal sensitivity transition model, the "no leakage" property can be further elaborated into three subproperties: (i) there exists a stable sensitivity list for the target circuit implementation, (ii) the stable sensitivity list is achievable from a legitimate initial sensitivity list, and (iii) the circuit secrecy status defined by a stable sensitivity list is trusted. As long as the target circuit and the initial sensitivity lists are both specified, these three subproperties will, then, be translated into Coq theorems using three theorem generation functions.

a) Theorem generation function I (existence): The existence of a stable sensitivity list is a prerequisite before proving any data secrecy property for the target circuit. If we cannot find one or more stable sensitivity lists, we believe that the target circuit is untrusted because sensitive data can be leaked freely. In the Coq platform, given the Coq circuit `coq_circuit`, the signal sensitivity list `sen_list` and the signal sensitivity evolvement function `update_sensitivity`, we define the theorem generation function for the stable sensitivity list existence property as:

```
Theorem stable_list_existence:
update_sensitivity coq_circuit sen_list =
  sen_list.
```

After plugging in the actual Coq circuit and the corresponding sensitivity list, we can generate and prove the existence theorem.

b) Theorem generation function II (accessibility): The existence property demonstrates the availability of a stable sensitivity list for the target circuit, but it does not provide evidence that the stable sensitivity list is accessible from the circuit's initial secrecy status. The second theorem seeks to solve this problem by proving that, given an initial sensitivity list, the circuit will finally achieve stable status after a finite number of clock cycles. Having the Coq circuit `coq_circuit`, the initial signal sensitivity list `ini_list`, a finite natural number `n`, the multiple-step evolvement function `check_sensitivity`, and a known stable sensitivity list `stable_list`, we define the theorem generation function for the accessibility property as follows:

```
Theorem stable_list_accessibility:
  forall t : nat, t > n ->
   (check_sensitivity t coq_circuit ini_list)
                                    = stable_list.
```

The selection of the number `n` is arbitrary, as long as it is a finite number; typically this number is bounded by the count of circuit stages. It is also possible that `n=0`, such that the initial signal sensitivity list is itself a stable sensitivity list, a case which sometimes exists in small-scale circuits with simple architecture.

c) Theorem generation function III (trustworthiness): Evaluating the trustworthiness of the derived stable sensitivity list is the most critical step when validating a data secrecy property. Because the stable sensitivity list contains the complete secrecy status of the target circuit, the goal of the trustworthiness theorem is to ensure that no sensitive information is leaked through primary outputs when the target circuit has reached the stable status. The evaluation of the trustworthiness process is shown below:

```
Theorem no_leaking_output:
 nth output stable_list = 0.  (* For all
   outputs *)
```

6) Data Protection of Intermediate Status: The above developed three theorems do not fully cover the entire working status of target circuits because the intermediate secrecy status between the initial status and the stable status are left unprotected. Although the intermediate status may only last a few clock cycles, the attackers may still be able to leak internal information during this short transition period. To overcome this limitation, a fourth theorem generation function is developed to formally prove that no sensitive information is leaked before the circuit reaches its stable status.

a) Theorem generation function IV (trustworthiness of intermediate status): The intermediate status during the transition procedure which evolves from the initial status to the stable status also needs protection to prevent information leakage. In the Coq platform, given the Coq circuit `coq_circuit` and the intermediate signal sensitivity list `intermediate_list`, we define the theorem generation function for the intermediate sensitivity list security property as:

```
Theorem no_leakage_intermediate:
 nth output intermediate_list=0.  (*For all
   outputs*)
```

In the next section, we demonstrate how the general PCHIP framework can be utilized to prevent malicious modification of sensitive data which may leak secrets indirectly.

VII. Preventing Malicious Modification of Sensitive Data

Even if the data secrecy properties generated through the aforementioned formal theorems are proven to hold, attackers may still be able to indirectly compromise security. Specifically, this can be achieved by inserting hardware Trojans which replace sensitive data, such as the encryption/decryption keys (or sub-keys), with a value known only to the attacker, or which create a new key with much less entropy than the original key, thereby reducing the effort required for cryptanalysis of the ciphertext. Such Trojans may rely on a rare trigger [13] for activation; hence, functional and structural testing may not detect their existence.

The PCHIP-based IFT capabilities introduced in Section VI cannot fully detect such attacks. The underlying reason for this shortcoming is the conservative information flow model employed in our approach for addressing implicit data flows. Another reason is that only the structure of the circuit is precisely considered in the conversion of the HDL code to its corresponding Coq representation, while the functionality of the operations and signal values are abstracted away. This is done to make proof development for information flow policy theorems simpler, as the exact functionality is not crucial in ensuring that sensitive data does not reach primary outputs.

While one possible approach would be to label critical data as non-sensitive and all other signals as sensitive, and then verify that the critical data remains non-sensitive [25], our conservative information flow model prevents us from using it. For example, consider loading the key register based on a load signal in the following code:

```
always @( posedge clk)  begin
  if (load == 1'b1)
   key_reg <= key_in;
end
```

In this case, `load` should be sensitive and `key_reg` non-sensitive. During code evaluation, the implicit flow from `load` to `key_reg` results in the the key register to also become sensitive. While this is a legitimate operation, it violates the above rule. This is not a problem in [25], since flip-flops in the gate level model only propagate tags from the input to the output [31] and do not consider the implicit data flow from the load signal.

In order to address this limitation and develop theorems to prevent malicious modification of sensitive data using the PCHIP framework, we resort to a general PCHIP-based solution wherein the entire functionality and structure of the HDL code needs to be converted to the Coq representation. For this purpose, the newly-introduced IFT capabilities need to be combined with the scope of the original PCHIP framework [15], [28]. Therein, however, the conversion process, which we review for the sake of completeness below, flattens the entire design hierarchy, thereby making proof development very cumbersome. Therefore, we also introduce a new conversion methodology which preserves the design hierarchy and reduces the burden of proof construction. While we chose to convert the entire functionality of the design to the Coq representation, the possible trade-off between the amount of functionality preserved in the conversion process and the level of protection which can be provided is an interesting topic for further investigation.

A. Flattened Functional Model

This model flattens the design hierarchy in a way that the precise functionality of the hardware design is converted to its corresponding Coq representation.

1) Basic Circuit Elements: In the flattened functional model, circuit elements are defined in the same way as what we presented in Section VI-A.1. In other words, all signals in the design are considered as `bus` in Coq representation.

2) Module Definitions: In this model, we flatten the design hierarchy and convert the module definitions in the Verilog source code to an inductive type in the Coq representation. We create a constructor for the module and we consider module inputs and outputs as parameters of this constructor. The body of the module is created in a function named `module_inst`, which is described in the following.

3) Local Signals: Local signals are widely defined and used inside modules in HDLs such as Verilog. However, Coq does not provide a flexible way of defining local variables inside functions. Therefore, in this conversion method, local signals are passed as additional arguments to the module type and instance definitions. Consequently, when instantiating a module, we need to define global variables in the Coq representation and pass them to the `module_inst` function.

4) Module Instantiations: Due to the flattened hierarchy in the Coq representation, we consider sub-modules as parameters of the `module` type definition, similar to the local signals. Therefore, when instantiating a module in the Coq representation, we need to appropriately instantiate its sub-modules too, and pass them as parameters to the `module_inst` function.

5) Expressions and Verilog Operations: We build expressions using the same method as presented in Section VI-A.4. However, a difference here is that an expression covers only the basic operations built into the Verilog language, and does not include higher level operations such as permutation or sbox, as was the case for the formal model of Section VI-A. Such higher level operations, which are mainly implemented as sub-modules, are handled by module definition and instantiation. We also define the `eval` function to precisely implement the operations in the Coq representation for evaluating the expressions.

6) Conditional, Combinational and Sequential Statements: The models in Sections VI-A and VI-B, which are devised for information flow tracking, consider combinational and sequential statements in the same way. In contrast, when converting the entire functionality into a flattened model, we need to distinguish between them in the Coq representation. Therefore, we define two distinct inductive types for conditions in a combinational and a 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` (with \ notation) and `aifcons` (with `;` notation) 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 seamlessly converted to Coq. These conditional constructors constitute the base structure of the code in its Coq representation. We also unroll `case` statements and treat them as consecutive `if..else` statements.

```
Inductive ifblock :=
| noif : updateblock->ifblock
| ifsimple : expr->ifblock->ifblock
| ifelse : expr->ifblock->ifblock->ifblock
| ifcons : ifblock->ifblock->ifblock.

Inductive aifblock :=
| anoif : assignblock->aifblock
| aifsimple : expr->aifblock->aifblock
| aifelse : expr->aifblock->aifblock->aifblock
| aifcons : aifblock->aifblock->aifblock.
```

To distinguish between combinational and sequential assignments, we define 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.

```
Inductive assignblock :=
| expr_assign : bus->expr->assignblock.

Inductive updateblock :=
| upd_expr : bus->expr->updateblock.
```

To evaluate combinational and sequential conditional blocks we define `adoif` and `doif` functions which are used in the Coq representation. The `assign` and `update` functions evaluate the expression and assign the result to the corresponding `bus` in the current or the next clock cycle, accordingly.

```
Fixpoint doif (i : ifblock)(t : nat)  struct
i :=  match i  with
  | (noif up) => (update up t)
  | (ifsimple exp ifb) =>
      match (eval exp t)  with
       | hi::nil => (doif ifb t)
       | lo::nil =>  True
       | x::nil =>  True
       | _ =>  True  end
  | (ifelse exp ifb1 ifb2) =>
      match (eval exp t)  with
       | hi::nil => (doif ifb1 t)
       | lo::nil => (doif ifb2 t)
       | x::nil => (doif ifb2 t)
       | _ =>  True  end
  | ifcons if1 if2 => doif if1 t /\ doif if2 t
end.

Fixpoint adoif (ai : aifblock)(t : nat) struct
ai:=  match ai  with
  | (anoif a_s) => assign a_s t
  | (aifsimple exp ifb) =>
      match (eval exp t)  with
       | hi::nil => (adoif ifb t)
       | lo::nil =>  True
       | x::nil =>  True
       | _ =>  True  end
  | (aifelse exp ifb1 ifb2) =>
       match (eval exp t)  with
        | hi::nil => (adoif ifb1 t)
        | lo::nil => (adoif ifb2 t)
        | x::nil => (adoif ifb2 t)
        | _ => True end
  | aifcons aif1 aif2=> adoif aif1 t /\ adoif
   aif2 t
 end.
```

For further clarification of the conversion process, we list below an excerpt from the Verilog code of a DES core and its partial conversion to the Coq representation. Verilog code:

```
 module des(desOut, desIn, key, decrypt,
  roundSel, clk);
// ...
 assign Lout = (roundSel == 0) ? IP[33:64] : R;
 assign Xin  = (roundSel == 0) ? IP[01:32] : L;
// ...
 always @( posedge clk)
          L <= #1 Lout;
 always @( posedge clk)
          R <= #1 Rout;

crp u0(. P(out), . R(Lout), .K_sub(K_sub));

key_sel u1(.K_sub(K_sub), . K(key),
  .roundSel(roundSel), .decrypt(decrypt));
 endmodule
```

Converted Coq representation:

```
 Inductive module :=
| module_des : bus->bus->bus->bus->bus->bus->
    bus->bus->bus->bus->bus->bus->bus->bus->
    module->module->module
| module_key_sel : bus->bus->bus->bus->bus->
    bus->bus->bus->bus->bus->bus->bus->bus->
    bus->module (* Other modules ... *)
.

 Fixpoint module_inst (m:module) (t:nat) :=
  match m  with
  | (module_des desOut desIn key decrypt
  roundSel clk
      Lout R IP Xin L Rout out  FP
     module_key_sel_u1
     module_crp_u0) =>
       (adoif (
         (anoif (expr_assign  Lout
           (cond (eeq (econb roundSel)
           (econv (lo::nil))) (econb ( IP [33,
            64]))
           (econb  R))));
         (anoif (expr_assign  Xin
           (cond (eeq (econb roundSel)
           (econv (lo::nil))) (econb ( IP [1,
            32]))
           (econb  L))));
            (* ... *)
       ) t) /\
       (doif (
         (noif (upd_expr L (econb Lout)))  \
         (noif (upd_expr R (econb Rout)))) t) /\

         (module_inst module_key_sel_u1 t) /\

         (module_inst module_crp_u0 t)
 |  (* Other modules ... *)
 end.
```

As we mentioned, in this conversion local signals and sub-modules are considered as parameters of the module definition. That is why `module_des` gets so many `bus` parameters and two `module` parameters. The appropriate constructors are used to convert combinational and sequential statements to the Coq representation.

In principle, this flattened model, which is employed by the original PCHIP framework, can be utilized to prove general security properties including the ones geared towards preventing malicious data modification in a hardware design. However, the corresponding Coq representation can quickly become very complicated due to the flattened hierarchy, as seen in the DES code excerpt example. Furthermore, this approach limits reusability of theorems proven for a module, thereby making proof development much more difficult, as the entire design needs to be reasoned upon monolithically. To overcome these limitations, we developed a hierarchy-preserving methodology which we present in the next section.

B. Hierarchical Functional Model

This new hierarchy-preserving model follows the general principles for flattened functional model which was described in Section VII-A. However, it takes advantage of the `Module Type` constructs in Coq to preserve the hardware design hierarchy.

For each module in the Verilog code of the hardware design we define its corresponding module in the Coq representation using the `Module Type` construct. Sub-modules instantiated inside this module are defined using `Declare Module` inside the `Module Type` definition of the Coq representation. We define local signals of the module as `Parameters` inside this `Module Type`. We also define a function named `instantiate` and we create the body of the module using similar constructs as those defined in Section VII-A. The following shows the DES core of Section VII-A, this time converted to the Coq representation through this hierarchy-preserving model.

```
 Module  Type module_key_sel.
   (* module definition ... *)
End module_key_sel.

 (* Other module definitions ... *)

 Module  Type module_des.

 Declare  Module crp_u0 : module_crp.
 Declare  Module key_sel_uk : module_key_sel.

 Parameters  K_sub IP FP L R Xin Lout Rout out :
 bus.

 Definition instantiate (desOut desIn key
 decrypt roundSel clk : bus) (t:nat) :=
       (adoif (
         (anoif (expr_assign  Lout
           (cond (eeq (econb roundSel)
           (econv (lo::nil))) (econb ( IP [33,
            64]))
           (econb  R))));
         (anoif (expr_assign  Xin
           (cond (eeq (econb roundSel)
          (econv (lo::nil))) (econb ( IP [1,
           32]))
          (econb  L))));
          (* ... *)
      ) t)
      /\
      (doif (
        (noif (upd_expr  L (econb  Lout)))    \
        (noif (upd_expr  R (econb  Rout)))
              ) t)
      /\
        (key_sel_uk.instantiate
          K_sub key roundSel decrypt t) /\
        (crp_u0.instantiate out  Lout K_sub t).
 End module_des.
```

C. Theorems Preventing Malicious Modification of Data

By maintaining design hierarchy, this new formal model introduced in this work has the advantage that it eliminates the need for keeping track of local signals and module instantiations inside modules. By comparing the Coq code excerpts of the flattened and hierarchical models, it is evident that the hierarchical model is much more readable and easier to follow. Using this model, we can develop various lemmas inside each `Module Type` in the Coq representation. These lemmas can, then, be applied hierarchically to prove theorems in higher-level modules. This procedure paves the way for developing hybrid module libraries containing both the HDL code and various lemmas usable for proving security theorems in higher-level modules. Such libraries greatly reduce the proof development effort required in the PCHIP framework and make PCHIP far more appealing to the hardware design community.

Moreover, based on this model, we can now also simplify the development of security properties which aim to protect the design against malicious modification of sensitive data. This is achieved by introducing theorems which track legitimate operations on such data, according to the design specification, as we present using a cryptographic core example in part II of this paper series. In conjunction with the IFT methodologies described in Section VI, this capability provides a comprehensive solution for evaluating and assessing data secrecy protection in hardware IPs.

VIII. Conclusion

While the majority of contemporary research in hardware Trojan prevention and detection focuses on post-tapeout threats, the problem of untrusted, potentially Trojan-infected RTL code is becoming equally important. Indeed, the increasing reliance on third-party IP cores for hierarchical designs calls for solutions to protect IP cores from inclusion of malicious functionality. The problem is particularly critical in the cryptography domain, where IP cores are used to run encryption/decryption operations and serve as the basis of system-level security. To address this problem, we augmented the previously proposed proof-carrying hardware IP (PCHIP) framework through the introduction of various information flow tracking approaches aiming to protect data secrecy in hardware IPs. To this end, new formal models have been developed for expressing an RTL circuit description in the Coq theorem-proving language through a set of HDL-to-Coq conversion rules, supporting formal reasoning on data secrecy properties and, thereby, increasing hardware trustworthiness. Theorem generation functions and proof libraries have also been introduced to reduce the burden of proof preparation and to pave the way towards construction of a security property library, a key step towards automating proof generation. In addition, we presented a hierarchy-preserving model of the original PCHIP framework, facilitating formal reasoning on security properties to prevent indirect leakage of sensitive information through malicious data modification. In Part II of this paper series, we focus on automation of the extended PCHIP framework and we demonstrate its application and effectiveness in detecting hardware Trojans and/or design flaws in the RTL description of third party IP cores.

References

[1] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Proc. ACM Symp. Oper. Syst. Principles (SOSP)*, 1997, pp. 129–142.

[2] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[3] L. C. Lam and T. Chiueh, "A general dynamic information flow tracking framework for security applications," in *Proc. Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2006, pp. 463–472.

[4] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2004, pp. 85–96.

[5] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer, "Defeating memory corruption attacks via pointer taintedness detection," in *Proc. Int. Conf. Dependable Syst. Netw. (DSN)*, 2005, pp. 378–387.

[6] W. Shi, J. Fryman, G. Gu, H.-H. Lee, Y. Zhang, and J. Yang, "Infoshield: A security architecture for protecting information usage in memory," in *Proc. Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2006, pp. 222–231.

[7] N. Vachharajani *et al.*, "RIFLE: An architectural framework for user-centric information-flow security," in *Proc. Int. Symp. Microarchitecture (MICRO)*, 2004, pp. 243–254.

[8] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee, "A software-hardware architecture for self-protecting data," in *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*, 2012, pp. 14–27.

[9] Y. Jin and D. Oliveira, "Extended abstract: Trustworthy SoC architecture with on-demand security policies and HW-SW cooperation," in *Proc. Heterogeneous Archit. Workloads Workshop SoCs*, 2014, pp. 1–2.

[10] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2009, pp. 109–120.

[11] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware Trojans: Lessons learned after one decade of research," *ACM Trans. Design Autom. Electron. Syst. (TODAES)*, vol. 22, no. 1, pp. 6:1–6:23, 2016.

[12] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan, "Hardware Trojan attacks: Threat analysis and countermeasures," *Proc. IEEE*, vol. 102, no. 8, pp. 1229–1247, Aug. 2014.

[13] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware Trojan design and implementation," in *Proc. IEEE Int. Workshop Hardw.-Oriented Secur. Trust (HOST)*, Sep. 2009, pp. 50–57.

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

[15] 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.

[16] 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)*, Sep. 2013, pp. 697–708.

[17] J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu, “Veritrust: Verification for hardware trust,” in *Proc. ACM Design Autom. Conf. (DAC)*, 2013, pp. 61:1–61:8.
[18] J. Zhang, F. Yuan, and Q. Xu, “Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware Trojans,” in *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*, 2014, pp. 153–166.
[19] H. Salmani, “COTD: Reference-free hardware trojan detection and recovery based on controllability and observability in gate-level netlist,” *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 2, pp. 338–350, Feb. 2017.
[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. Int. Workshop Cryptograph. Hardw. Embedded Syst. (CHES)*, 2009, pp. 396–410.
[22] X. Li *et al.*, “Caisson: A hardware description language for secure information flow,” in *Proc. ACM Conf. Program. Lang. Design Implement. (PLDI)*, 2011, pp. 109–120.
[23] X. Li *et al.*, “Sapper: A language for hardware-level security policy enforcement,” in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2014, pp. 97–112.
[24] 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.
[25] W. Hu, B. Mao, J. Oberg, and R. Kastner, “Detecting hardware trojans with gate-level information-flow tracking,” *Comput.*, vol. 49, no. 8, pp. 44–52, 2016.
[26] 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)*, 2012, pp. 252–257.
[27] Y. Jin, B. Yang, and Y. Makris, “Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing,” in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, Apr. 2013, pp. 99–106.
[28] Y. Jin and Y. Makris, “A proof-carrying based framework for trusted microprocessor IP,” in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, May 2013, pp. 824–829.
[29] *INRIA. The COQ Proof Assistant*, accessed on Jun. 1, 2016. [Online]. Available: http://coq.inria.fr/
[30] T. Braibant and A. Chlipala, “Formal verification of hardware synthesis,” in *Proc. Comput. Aided Verification*, 2013, pp. 213–228.
[31] W. Hu *et al.*, “Gate-level information flow tracking for security lattices,” *ACM Trans. Design Autom. Electron. Syst.*, vol. 20, no. 1, p. 2, 2014.

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 post-deployment 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 was awarded the DoE Early CAREER Award in 2016 and was the best paper award recipient of DAC’15, ASP-DAC’16, and HOST’17.

Xiaolong Guo received double bachelor’s degrees from Beijing University of Posts and Telecoms (BUPT), and University of London (UL) in 2010. In 2013, he received the M.S. degree from BUPT. He is currently working toward the Ph.D. degree in electrical engineering with the University of Central Florida, Orlando, FL. 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.

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 a Ph.D. candidate 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.

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, San Diego, in 1998 and 2001, respectively. After spending a decade on the faculty of Yale University, he joined UT Dallas where he is now 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 serves as an Associate Editor of IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY and *IEEE Design & Test* of computers periodical and he has also served as a guest editor for *IEEE Transactions on Computers* and IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. He is a recipient of the 2006 Sheffield Distinguished Teaching Award, as well as Best Paper Awards from the 2013 Design Automation and Test in Europe (DATE’13) conference and the 2015 VLSI Test Symposium (VTS’15).