Chapter 10
IP Trust Validation Using Proof-Carrying Hardware

Xiaolong Guo, Raj Gautam Dutta, and Yier Jin

10.1 Introduction

A rapidly growing third-party Intellectual Property (IP) market provides IP consumers with high flexibility when designing electronic systems. It also reduces the development time and expertise needed to compete in a market where profit-windows are very narrow. However, one key issue that has been neglected is the security of hardware designs built upon third-party IP cores. Historically, IP consumers have focused on IP functionality and performance than security. The negligence toward development of robust security policies is reflected in the IP design flow (see Fig. 10.1), where IP core specification usually only includes functionality and performance measurements.

The prevailing usage of third-party soft IP cores in SoC designs raises security concerns as current IP core verification methods focus on IP functionality rather than IP trustworthiness. Moreover, lack of regulation in the IP transaction market adds to the predicament of the SoC designers and forces them to perform verification and validation of IPs themselves. To help SoC designers in IP verification, various methods have been developed to leverage enhanced functional testing and/or perform probability analysis of internal nodes for IP core trust evaluation and malicious logic detection [1, 2]. However, these methods were easily bypassed by sophisticated hardware Trojans [3–5]. Formal methods were also introduced for IP core trust evaluation [1, 6–10]. Among all the proposed formal methods, proof-carrying hardware (PCH), which originated from proof-carrying code (PCC), emerged as one of the most prevalent methods for certifying the absence of malicious logic in soft IP cores and reconfigurable logic [6–10]. In the PCH approach, synthesizable register-transfer level (RTL) code of IP core and informal security properties were

X. Guo • R.G. Dutta • Y. Jin (✉)
University of Central Florida, Orlando, FL 32816, USA
e-mail: guoxiaolong@knights.ucf.edu; rajgautamdutta@knights.ucf.edu; yier.jin@eecs.ucf.edu

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_10

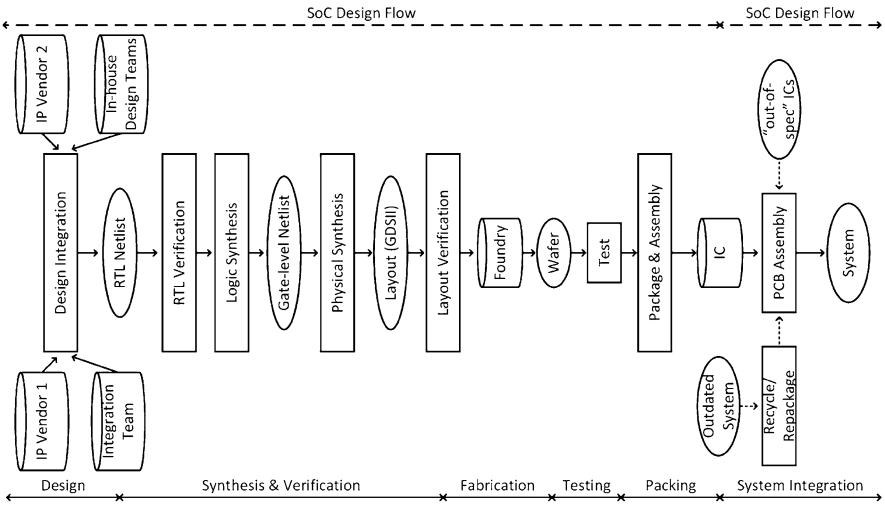

Fig. 10.1 IC design flow within the semiconductor supply chain

first represented in *Gallina*—the internal functional programming language of the Coq proof assistant [11]. Then, Hoare-logic style reasoning was used to prove the correctness of the RTL code in the Coq platform.

The rest of the chapter is organized as follows: In Sect. 10.2, we provide an overview of existing methods for IP protection, introduce the threat model, and provide some relevant background on two different formal verification approaches. In Sect. 10.3, we provide detailed explanation of the PCH method for ensuring trustworthiness of IP cores. Finally, Sect. 10.4 concludes the chapter.

10.2 Overview of Formal Verification Methods for IP Protection

To counter the threat of untrusted third-party resources, pre-silicon trust evaluation approaches have been proposed recently [1, 12, 13]. Most of these methods try to trigger malicious logic by enhancing functional testing with extra test vectors. Authors in [12] proposed a method to generate "Trojan Vectors" into the testing patterns, hoping to activate the hardware Trojans during the functional testing. In order to identify suspicious circuitry, unused circuit identification (UCI) [13] method analyzed the RTL code to find lines of code that are never used. However, these methods assume that the attacker uses rarely occurring events as Trojan triggers. Using "less-rare" events as trigger will void these approaches. This was demonstrated in [14], where hardware Trojans were designed to defeat UCI.

Admitting the limitations of enhanced functional testing methods, researchers started looking into formal solutions. Although at its early stage, formal methods have already shown their advantages over testing methods in exhaustive security verification [8, 9, 15, 16]. A multi-stage approach, which included assertion based verification, code coverage analysis, redundant circuit removal, equivalence analysis, and use of sequential Automatic Test Pattern Generation (ATPG), was adopted in [15] to identify suspicious signals for detecting hardware Trojans. This approach was demonstrated on an RS232 circuit and the efficiency of the approach in detecting Trojan signals ranged between 67.7 and 100 %. In [8, 9, 16], a PCH framework was used to verify security properties on soft IP cores. Supported by the Coq proof assistant [11], formal security properties were formalized and proved to ensure the trustworthiness of IP cores. In the following section in this chapter, we will explain the PCH approach for soft IP core verification in greater details. This method uses an interactive theorem prover and model checker for verifying the design.

10.2.1 Threat Model

The IP protection methods in this chapter are based on the threat model that malicious logic is inserted by an adversary at the design stage of the supply chain. We assume that the rogue agent at the third-party IP design house can access the hardware description language (HDL) code and insert a hardware Trojan or backdoor to manipulate critical registers of the design. Such a Trojan can be triggered either by a counter at a predetermined time, by an input vector, or under certain physical conditions. Upon activation it can leak sensitive information from the chip, modify functionality, or cause a denial-of-service to the hardware. In this chapter, Trojans which can be activated by a specific "digital" input vector are only considered.

Meanwhile, verification tools (e.g., Coq) used in all methods are assumed to produce correct results. The existence of proofs for the security theorems indicates the genuineness of the design whereas its absence indicates the presence of malicious logic. However, the framework does not provide protection of an IP from Trojans whose behaviors are not captured by the set of security properties. Furthermore, there is also an assumption that the attacker has intricate knowledge of the hardware to identify critical registers and modify them in order to carry out the attack.

10.2.2 Formal Verification Methods

Formal methods have been extensively used for verification and validation of security properties at pre- and post-silicon stages [8, 9, 15–20]. These previous methods leverage one of the following two techniques, model checking and interactive/automated theorem proving, for design verification.

10.2.2.1 Theorem Prover

Theorem provers are used to prove or disprove properties of systems expressed as logical statements [21–28]. Over the years, several theorem provers (both interactive and automated) have been developed for proving properties of hardware and software systems. However, using them for verification on large and complex systems require excessive effort and time. Irrespective of these limitations, theorem provers have currently drawn a lot of interest in verifying security properties on hardware. Among all the formal methods, they have emerged as the most prominent solution for verifying large-scale designs.

One leading example of an interactive theorem prover is the open source tool called Coq proof assistant [11]. Coq is an interactive theorem prover/proof assistant, which enables verification of software and hardware programs with respect to their specification [25]. In Coq, programs, properties, and proofs are represented as terms in the *Gallina* specification language. By using the *Curry–Howard Isomorphism*, the interactive theorem prover formalizes both the program and proofs in its dependently typed language called the *Calculus of Inductive Construction (CIC)*. Correctness of the proof of the program is automatically checked using the built-in type-checker of Coq. To expedite the process of building proofs, Coq provides a library consisting of programs called *tactics*. However, existing Coq *tactics* does not capture properties of hardware designs and thus does not significantly reduce the time required for certifying large-scale hardware IP cores [6–8].

10.2.2.2 Model Checker

Model checking [29] is an automated method for verifying and validating models in software and hardware applications [30–42]. In this approach, a model (Verilog/VHDL code of hardware) $\mathscr{M}$ with an initial state s_0 is expressed as a transition system and its behavioral specification (assertion) ϕ is represented in a temporal logic. The underlying algorithm of this technique explores the state space of the model to find whether the specification is satisfied. This can be formally stated as, $\mathscr{M}, s_0 \models \phi$. If a case exists where the model does not satisfy the specification, a counterexample in the form of a trace is produced by the model checker [43, 44]. Recently, model checkers have been used for detecting malicious signals in third-party IP cores [15, 20]. The application of model checking techniques to SoCs, including symbolic approaches based on Reduced Order Binary Decision Diagrams (ROBDD) and Satisfiability (SAT) solving, has had limited success due to the state-space explosion problem [45]. For example, a model with n Boolean variables can have as many as 2^n states, a typical soft IP core with 1000 32-bit integer variables has billions of states.

Symbolic model checking using ROBDD is one of the initial approaches used for hardware systems verification [46–48]. Unlike explicit state model checking where all states of the system are explicitly enumerated, this technique model states (represented symbolically) of the transition system using ROBDD.

The ROBDD is a unique, canonical representation of a Boolean expression of the system. Subsequently, the specification to be checked is represented using a temporal logic. A model checking algorithm then checks whether the specification is true on a set of states of the system. Despite being a popular data structure for symbolic representation of states of the system, ROBDD requires finding an optimal ordering of state variables which is an NP-hard problem. Without proper ordering, the size of the ROBDD increases significantly. Moreover, it is memory intensive for storing and manipulating Binary Decision Diagrams (BDDs) of a system with a large state space.

Another technique called bounded-model checking (BMC) replaces BDDs in symbolic checking with SAT solving [49–51]. In this approach, a propositional formula is first constructed using a model of the system, the temporal logic specification, and a bound. The formula is then provided to an SAT solver to either obtain a satisfying assignment or to prove that no such assignment exists. Although BMC outperforms BDD based model checking in some cases, the method cannot be used to test properties (specification) when the bound is large or cannot be determined.

10.3 Proof-Carrying Hardware Framework for IP Protection

Various methods have been proposed in the software domain to validate the trustworthiness and genuineness of software programs. These methods protect computer systems from untrusted software programs. Most of these methods lay burden on software consumers to verify the code. However, *proof-carrying code* (PCC) switches the verification burden to software providers (software vendors/developers). Figure 10.2 outlines the basic working process of the PCC framework.

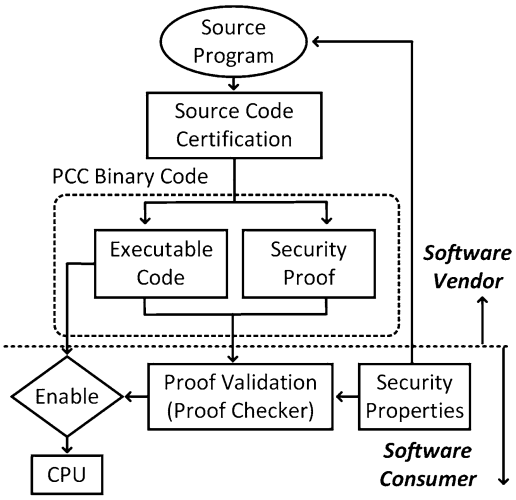

Fig. 10.2 Working procedure of the PCC framework [52]

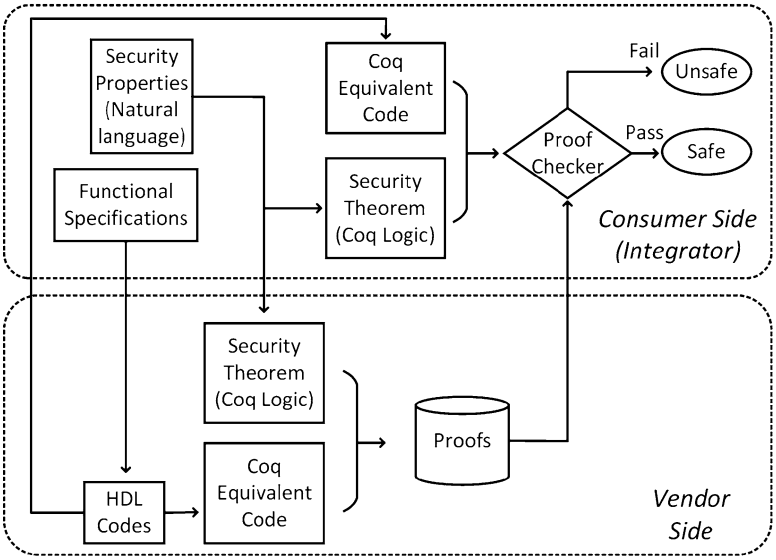

Fig. 10.3 Working process of the PCH framework [17]

During the *source code certification* stage of the PCC process, the software provider verifies the code with respect to the *security property* designed by the software consumer and encodes the formal proof of the security property with the executable code in a *PCC binary file*. In the *proof validation* stage, the software consumer determines whether the code from the potentially untrusted software provider is safe for execution by validating the PCC binary file using a proof checker [52].

A similar mechanism, referred to as Proof-Carrying Hardware (PCH), was used in the hardware domain to protect third-party soft IP cores [8–10]. The PCH framework ensures trust-worthiness of soft IP cores by verifying a set of carefully specified security properties. The working procedure of the PCH framework is shown in Fig. 10.3. In this approach, the IP consumer provides design specifications and informal (written in natural language) security properties to the IP vendor. Upon receiving the request, the IP vendor develops the RTL design using a hardware description language (HDL). Then, semantic translation of the HDL code and informal security properties to Gallina is carried out. Subsequently, Hoare-logic style reasoning is used for proving the correctness of the RTL code with respect to formally specified security properties in Coq. As Coq supports automatic proof checking, it can help IP customers validate proof of security properties with minimum efforts. Moreover, usage of the Coq platform by both IP vendors and IP consumers ensures that the same deductive rules will be used for validating the proof. After verification, the IP vendor provides the IP consumer with the HDL code (both original and translated versions), formalized security theorems of security properties, and proofs of these security theorems. Then, the proof checker in Coq is used by the IP consumer to quickly validate the proof of security theorems on the

translated code. The proof checking process is fast, automated, and does not require extensive computational resources.

10.3.1 Semantic Translation

The PCH based IP protection method requires semantic translation of circuit design in HDL to Coq's specification language, *Gallina*. Consequently, a *formal-HDL* is developed in [10], which includes a set of rules to enable this translation. These rules can help represent basic circuit units, combinational logic, sequential logic, and module instantiations. The *formal-HDL* is further extended in [53] to capture hierarchical design methodology, which is used for representing large circuits such as SoC. A brief description of the *formal-HDL* is given below

- *Basic Circuit Units:*
  In the *formal-HDL*, basic circuit units are the most important components and they include signals and buses. During the translation, three digital values are used for signals: high, low, and unknown. To represent sequential logic, a *bus* type is defined as a function, which takes timing variable *t* and returns a list of signal values as shown in *Listing 10.1*. All circuit signals are of *bus* type and their values can be modified either by a blocking assignment or a nonblocking assignment (shown in *Listing 10.1*). Moreover, inputs and outputs are also defined as *bus* type.

Listing 10.1 Basic Circuit Units in Semantic Model

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

- *Signal Operations:* Logic operations such as *and*, *or*, *not*, and *xor*, as well as bus comparison operations such as checking for bus equality: *bus_eq* and less-than: *bus_lt* are designed to handle *bus* in *Gallina*. The conditional statement of RTL code such as *if... else...* checks whether signals are on or off. To incorporate this functionality in Coq, a special function, *bus_eq_0*, which compares the bus value to *hi* or *lo* is added.

Listing 10.2 Signal Operations in Semantic Model

```
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).
```

- *Combinational and Sequential Logic:* The definition of signals, expressions, and their semantics paves the way for converting RTL circuits into Coq representatives. Combinational and sequential logic are higher level logic descriptions constructed on top of buses. The keyword *assign* of the *formal-HDL* is used for blocking assignment, while *update* is mainly used for nonblocking assignment. During the blocking assignment the bus value will be updated in the current clock cycle and in the nonblocking assignment the bus value will be updated in the next clock cycle.

Listing 10.3 Signal Operations in Semantic Model

```
Fixpoint assign (a:assignblock)(t:nat) {struct a} :=
    (* Blocking assignment *)
match a with
| expr_assign bus_one e => bus_one t = eval e t
| assign_useless => True
| assign_cons a1 a2 => (assign a1 t) /\ (assign a2 t)
end.
Fixpoint update (u:updateblock)(t:nat) {struct u} :=
    (* Nonblocking assignment *)
match u with
| (upd_expr bus exp) => (bus (S t)) = (eval exp t)
| (updcons block1 block2) => (update block1 t) /\ (update block2 t)
| upd_useless => True
end.
```

- *Module Definitions:* Module definition/instantiation is critical when dealing with hierarchical circuit structures, but it is never a problem for Verilog (and VHDL), as long as interfacing signals and their timing are correctly defined. Concerning the task of security property verification, however, treating a sub-module as a functional unit by ignoring its internal structure may cause problems. Security properties that are proven for the top level module and all its sub-modules do not

guarantee that the same properties will hold for the whole hierarchical design, where attackers can easily insert hardware Trojans to maliciously modify the interface without violating security properties proven for all modules separately. As a result, the operation of module definition/instantiation should be defined in a way that the details of sub-modules are accessible from the top level module so that any security properties, if proven, remain valid for the whole design. Thus, in PCH we flatten the hierarchical design such that the sub-modules and their interfaces are transparent to the top module. *module* and *module-inst* are key words for module definitions and instantiations. In [53], a new syntax for representing modules is introduced in Coq, which preserves the hierarchical structure and does not require design flattening.

The underlying formal language of the Coq proof assistant, *Gallina*, is based on dependently typed lambda calculus and it defines both types and terms in the same syntactical structure. During the translation process, syntax and semantics of the HDL are translated to *Gallina* using the *formal-HDL*.

10.3.2 Data Protection Through Information Flow Tracking

Among all potential RT-level malicious modifications, sensitive information protection has been a research topic within the cybersecurity domain for decades. Various approaches have been developed, relying on safe languages and software level dynamic checks, to detect buffer overflow attacks and format string vulnerabilities. These methods suffer from the limitation that they either have high false-alarm rates or would cause significant performance overhead. Taking these limitations into consideration, researchers invented new information protection schemes based on hardware–software co-design, where the hardware infrastructure is actively involved in dynamic information flow tracking. This new trend has proven successful in improving detection accuracy and lowering performance overhead, at the cost of hardware level modifications. For example, authors in [54] proposed a dynamic information flow tracking framework with all internal storage elements equipped with a security tag.

Authors in [55] focused on pointer tainting to prevent both control data and non-control data attacks. Besides information flow tracking, the hardware is also enhanced to help prohibit information leakage, such as in the InfoShield architecture [56], which applies restrictions to operations on sensitive data. 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 [57]. More recently, a new software–hardware architecture was developed to support more flexible security policies, either to protect sensitive data [58] or to prevent malicious operations from untrusted third-party OS kernel extensions [59].

Two formal information flow tracking methodologies were also developed, namely *static information flow tracking* [60] and *dynamic information assurance*

[9], which address the challenge of hardware Trojans, capable of leaking sensitive information. These two schemes follow the concept of proof-carrying hardware IP (PCHIP) [8] to enhance the trustworthiness of third-party IP cores.

These two formal methods are particularly geared toward secret information protection, counteracting RTL hardware Trojan attacks in hardware soft IPs, and preventing unintended design backdoors. These two methods differ in complexity and the approach they take to track information in the design. Static information flow tracking scheme is suitable for small designs, and requires less effort in proof development, while dynamic information assurance scheme considers the requirements of more complex and pipelined designs, and needs much more effort in constructing the proofs of security theorems. Designers can adopt these two methodologies based on their requirements.

The *static information flow tracking scheme* and the *dynamic information assurance scheme* are integrated with the PCH IP protection framework and they accept the data secrecy properties as the security property. Furthermore, because the target data secrecy properties are independent of circuit functional specifications, IP vendors may translate the properties from natural language to formal theorems without specifying target circuits and can store the translated formal theorems in a property library for similar designs. The development of a Coq property library and the reuse of theorem-proof contents lowers the burden for IP vendors and stimulates wider acceptance of the proposed proof-carrying based hardware IP protection method. Property formalization and proof generation of both schemes are performed using the Coq proof assistant platform [11].

10.3.2.1 Static Information Flow Tracking Scheme

For the static information flow tracking scheme [60], the IP vendor first designs the circuit based on the functional specifications provided by the IP consumer, in the form of HDL codes. Utilizing a formal semantic model and static information flow tracking rules, the IP vendor then converts the circuit from HDL code into formal logic. In parallel, the IP vendor uses the property formalization constraints to translate the agreed-upon data secrecy properties from natural language to formal theorems. The IP vendor will then try to construct proofs for the translated theorems within the context of the target circuit. Even though the IP vendor is responsible for both circuit design and theorem proving, given a set of well-defined theorems, it is not possible to prove the theorems with a Trojan-infected circuit containing the prohibited information leakage paths. Both formal theorems and their proofs are part of the final deliverable handed to the IP consumer.

Upon receiving the hardware bundle which includes the HDL code and theorem-proof pairs for data secrecy properties, the IP consumer regenerates the formal logic of the original circuit based on the same formal semantic model and static information flow tracking rules. The IP consumer also checks whether the security theorems (in formal language) accurately represent the data secrecy properties (in

natural language). The security theorems and related proofs will then be combined with the regenerated formal logic to pass through an automatic proof checker. If no exceptions are raised, then we claim that the delivered IP core fulfills the agreed-upon data secrecy properties. However, any errors during the proof checking process warn the user that malicious circuits (or design flaws) may exist in the IP core, making it violate the data secrecy properties.

10.3.2.2 Dynamic Information Assurance Scheme

The static scheme is effective in detecting data leakage caused by hardware Trojans and/or design faults. It also requires less effort for constructing proofs. However, the static scheme is limited by the fact that it can only check circuit trustworthiness statically. To overcome this shortcoming of the static scheme and to achieve high-level hardware Trojan detection capability, a dynamic information assurance scheme is later developed [9].

This dynamic scheme supports various levels of circuit architectures, ranging from low-complexity single-stage designs to large-scale deeply pipelined circuits. Similar to the static scheme, the dynamic scheme also focuses on circuits dealing with sensitive information, such as cryptographic designs, because it sets data secrecy as the primary goal and tries to prevent illegal information leakage from IP cores. Within the dynamic scheme, all signals are assigned values indicating their sensitivity levels. These values will be updated after each clock cycle according to their original values and the updating rules defined by the signal sensitivity transition model. Since the sensitivities of all circuit signals are managed in a sensitivity list, two sensitivity lists are of interests for data secrecy protection: the initial sensitivity list and the stable sensitivity list. The initial sensitivity list reflects the circuit status after initialization or powered-on mode when only some input signals contain sensitive information, such as plaintext and encryption keys. The stable sensitivity list, on the other hand, indicates the circuit status when all internal/output signals are of fixed sensitivity levels.

Similar to the static scheme, IP vendor will also translate the agreed-upon data secrecy properties from natural language to property generation functions, which can later help to generate formal theorems. Meanwhile, different from the static scheme, IP consumers will first check the contents of the initial signal sensitivity list and the stable signal sensitivity list, which represent the circuit's initial secrecy status and the stabilized status, respectively. The validity of the initial list is checked to ensure that sensitivity levels are appropriately assigned to all input/output/internal signals. The circuit's stable sensitivity status contains complete information of the distribution of sensitive information across the whole circuit, so the stable list will then be carefully evaluated to detect any backdoors that may leak sensitive information. After both signal sensitivity lists pass the initial checking, IP consumers proceed to the next step of proof checking. A "PASS" output from the automatic proof checker provides evidence that HDL codes do not contain any

malicious channels to leak information. However, a "FAIL" result is a warning that some of the data secrecy properties are breached in the delivered IP cores.

10.3.3 Hierarchy Preserving Verification

The above mentioned PCH frameworks treat the whole circuit design as one module and prove security properties on them [8–10, 60]. That is, the entire design is first flattened before translating the HDL code of the design into the formal language and proving it with respect to formal security theorems. Design flattening increases the complexity of translating HDL code into *Gallina*. It also adds to the risk of introducing errors during the code conversion process. Due to flattening, a verification expert has to go through the entire design in order to construct proofs of security theorems, which significantly increases the workload for design verification. Also, any updates to the HDL code will significantly change the proof for the same security property. Moreover, the PCH framework prevents proof reuse, i.e., proofs constructed for one design cannot be used in another design even though the same IP modules are used. All of these limitations prohibit a wide usage of the PCH framework in modern SoC designs.

To overcome these limitations, the *Hierarchy-preserving Formal Verification* (HiFV) framework is developed for verifying security properties on SoC designs in [53]. The HiFV framework is an extension of the PCH framework. In the HiFV framework, the design hierarchy of the SoC is preserved and a distributed approach is developed for constructing proofs of security properties. In the distributed approach, security properties are divided into sub-properties in such a way that each sub-property corresponds to an IP module of the SoC. Proofs are then constructed for these sub-properties and the security property for the SoC design is proven through the integration of all proofs from sub-properties. Similar to PCH, the HiFV framework requires semantic translation of the HDL code and informal security properties to *Gallina*. For proving the trustworthiness of the HDL code of the SoC, Hoare-logic is used. Similar to other PCH methods, the HiFV framework is carried out in Coq.

As mentioned earlier, before building the formal model for the SoC system, the syntax and semantics should be defined and then shared by any parties who need to design or check the proof. In addition, *interface* and *module* are incorporated in the *formal-HDL* to preserve the design hierarchy of the SoC. That is, in order to make distributed proof construction applicable on hierarchical designs, an *interface* is developed in the HiFV framework which makes the verification process flexible and efficient for the proof writer. To define the *interface*, information about each IP and its corresponding I/O are needed, such as the name, number, and data type. By using the *interface*, the management of the plenty of formal modules would be much easier in the verification house side. The structure of the *interface* is shown in Fig. 10.4. Through the *interface*, an IP module within an SoC can access other

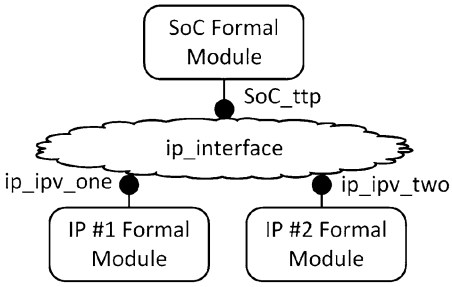

Fig. 10.4 Structure of the SoC with *interface*

modules such as IP #1 Formal Module or IP #2 Formal Module in the figure. The *ip_ipv_one*, *ip_ipv_two*, and *SoC_ttp* are the name of the corresponding *interfaces*.

The distributed proof construction process uses Hoare-logic, where the trustworthiness of the SoC *formal-HDL* code is determined by ensuring that the code operates following the constraints of the pre-condition and the post-condition. The pre-condition of the *formal-HDL* code is the initial configuration of the design and the post-condition is the security theorem. Meanwhile, in order to overcome the scalability issue, a distributed proof construction approach is developed, which is dedicated for SoC designs with hierarchical structures. This approach makes the HiFV framework scalable by reducing the time required for proof construction, proof correction, and proof modification.

In the HiFV framework, the translated HDL code of the SoC, formal security theorems, and the initial configuration of the design is represented as a Hoare Triple (Eq. (10.1)).

$$(\phi)CoqEquivalentCode\_SoC(\psi) \tag{10.1}$$

In this equation, ϕ is the pre-condition corresponding to the initial configuration of the design. The translated HDL code of the SoC design hierarchy in *Gallina* is given by *CoqEquivalentCode_SoC*. In the process of translation, modules in the SoC HDL code, which correspond to IPs from different vendors, are also translated. The post-condition is given by ψ which represents the formal security theorem.

The security theorem is divided into lemmas (Eq. (10.2)), which are post-conditions for individual IP modules. In Eq. (10.2), post-condition for IPs (lemmas) are represented as ψ_i $(1 \leq i \leq n)$, $n = maximum\ number\ of\ IP\ modules\ required\ to\ prove\ the\ security\ theorem$ and ψ is the security theorem. These lemmas correspond to those IP modules that are required to satisfy the security theorem.

$$\psi := \psi_1 \wedge \psi_2 \cdots \wedge \psi_n \tag{10.2}$$

Similarly, the pre-condition of the SoC design (ϕ) and the translated HDL code of the SoC design (*CoqEquivalentCode_SoC*) are divided according to Eqs. (10.3) and (10.4). Here, (ϕ_i) and (*CoqEquivalentCode_IPmodule_i*) $(1 \leq i \leq n)$ represent the pre-conditions and translated HDL code of each IP module of the

SoC, respectively.

$$\phi := \phi_1 \wedge \phi_2 \cdots \wedge \phi_n \tag{10.3}$$

$$\begin{aligned} CoqEquivalentCode\_SoC := {} & CoqEquivalentCode\_IPmodule\_1 \\ & \wedge CoqEquivalentCode\_IPmodule\_2 \ldots \\ & \wedge CoqEquivalentCode\_IPmodule\_n \end{aligned} \tag{10.4}$$

The HDL code of the IP core is certified to be trustworthy only if it satisfies the pre-condition and the post-condition. When all the modules of IP cores satisfy the post-conditions (lemmas), we can state that the security theorem is proven for the SoC design.

$$(\phi_i) CoqEquivalentCode\_IPmodule\_i(\psi_i) \tag{10.5}$$

The distributed approach of proof construction also enables proof reuse. After certifying the trustworthiness of each IP core of the SoC, the proofs can be stored in a library and accessed by the trusted third party (TTP) verification house for verification of other SoC designs in which the same IP modules are used and similar security properties are applied. In this way the HiFV framework further reduces the time for verifying complex designs.

As a summary, in this approach, the previously developed PCH framework is extended into the SoC design flow and largely simplified the process for proving security properties through a hierarchical proof construction procedure. To reduce the workload for circuit verification, the proof of the security properties for individual IPs can be encapsulated and reused in proving security properties at the SoC level. Also, in the hierarchical framework, the amount of updates that need to be done to existing proofs when SoC designs are modified is significantly lowered. The developed HiFV framework paves the way for large-scale circuit design security verification.

10.3.4 Integrating Theorem Prover and Model Checker

Although the HiFV hierarchical approach improves scalability of the previous PCH method, it still suffers from the challenge of proof construction. Meanwhile, model checkers such as Cadence IFV cannot be used for verifying systems with large state space because of the space explosion problem. As the number of state variables (n) in the system increases, amount of space required for representing the system and the time required for checking the system increases exponentially ($T(n) = 2^{O(n)}$) (Fig. 10.5).

To further overcome the scalability issue and to verify a computer system, an *integrated formal verification framework* (see Fig. 10.6) is introduced in [61], where

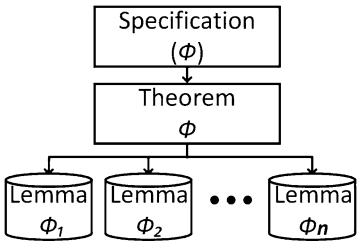

Fig. 10.5 Security specification (ϕ) decomposed into lemmas

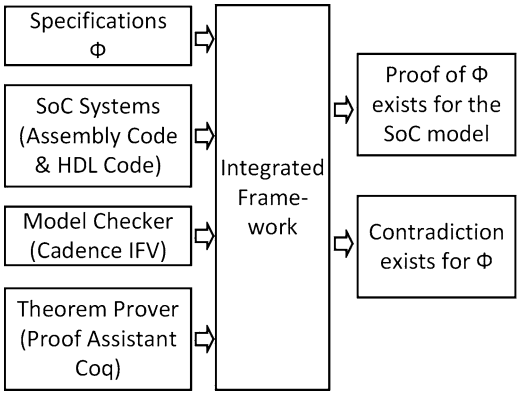

Fig. 10.6 Integrated formal verification framework

the security properties are checked against SoC designs. In this framework, the theorem prover is combined with a model checker for proving formal security properties (specifications). Moreover, the hierarchical structure of the SoC is leveraged to reduce the verification effort.

Some efforts have been made to combine theorem provers with model checkers for verification of hardware and software systems [62, 63]. These methods try to overcome the scalability issue of both techniques. That is, both model checkers and theorem provers cannot scale well to formally verify large-scale circuit designs. Some of the popular theorem provers such as higher order logic (HOL Light) and prototype verification system have integrated model checkers. These tools have been used for functional verification of hardware systems. For the first time, this combined technique has been extended toward verification of security properties on third-party IP cores and SoCs [61].

In the integrated framework, the hardware design, represented in a hardware description language (HDL), and the assembly level instructions of a vulnerable program, is first translated to *Gallina*, which is similar to other PCH methods. Then, the security specification is stated as a formal theorem in Coq. In the following step, this theorem is decomposed into disjoint lemmas (see Fig. 10.5) based on sub-modules. These lemmas are then represented in the Property Specification Language (PSL) specification language and are called sub-specifications. Subsequently, the Cadence IFV verifies the sub-modules against the corresponding sub-specifications. Sub-modules are functions, which have less number of state variables and are

connected to primary output of the design. These functions are always from the bottom level of SoC and have rare dependency relationship with each other.

The HDL code of a large design consists of many such sub-modules. If the sub-modules satisfy the sub-specifications, lemmas are considered to be proved. Checking the truth value of the sub-specifications with a model checker eliminates the effort required for proving the lemmas and translating the sub-modules to Coq. Upon proving these sub-modules, Hoare-logic is then used to combine proof of these lemmas to prove the security theorem of the entire system in Coq.

The integrated formal verification framework helps in protecting a large-scale SoC design from malicious attacks. Given that an interactive theorem prover (e.g., Coq) requires lots of effort to manually verify the design and that a model checker suffers from scalability issues, these two techniques are combined together through the decomposition of the security property as well as the design in such a way that the model checker can verify those sub-modules which have much less state variables. Consequently, the amount of effort required for translating the design from HDL to *Gallina* and proving the security theorem in Coq is reduced.

10.4 Conclusion

In this chapter, we explain our interactive theorem proving based PCH approach for security property verification of hardware IP cores. We also describe application of the framework for preventing information leakage from soft IPs. To overcome scalability and reusability issues of original PCH method, a design hierarchy preserving scheme was then introduced that incorporates both model checking and interactive theorem proving for verification.

Acknowledgements This work has been partially supported by the National Science Foundation (NSF-1319105), the Army Research Office (ARO W911NF-16-1-0124), and Cisco.

References

1. M. Banga, M. Hsiao, Trusted RTL: Trojan detection methodology in pre-silicon designs, in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2010), pp. 56–59
2. A. Waksman, M. Suozzo, S. Sethumadhavan, FANCI: identification of stealthy malicious logic using boolean functional analysis, in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security, CCS'13* (2013), pp. 697–708
3. D. Sullivan, J. Biggers, G. Zhu, S. Zhang, Y. Jin, FIGHT-metric: Functional identification of gate-level hardware trustworthiness, in *Design Automation Conference (DAC)* (2014)
4. N. Tsoutsos, C. Konstantinou, M. Maniatakos, Advanced techniques for designing stealthy hardware trojans, in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE* (2014)

5. M. Rudra, N. Daniel, V. Nagoorkar, D. Hoe, Designing stealthy trojans with sequential logic: A stream cipher case study, in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE* (2014)
6. S. Drzevitzky, U. Kastens, M. Platzner, Proof-carrying hardware: Towards runtime verification of reconfigurable modules, in *International Conference on Reconfigurable Computing and FPGAs* (2009), pp. 189–194
7. S. Drzevitzky, M. Platzner, Achieving hardware security for reconfigurable systems on chip by a proof-carrying code approach, in *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip* (2011), pp. 1–8
8. E. Love, Y. Jin, Y. Makris, Proof-carrying hardware intellectual property: a pathway to trusted module acquisition. IEEE Trans. Inf. Forensics Secur. **7**(1), 25–40 (2012)
9. Y. Jin, B. Yang, Y. Makris, Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing, in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2013), pp. 99–106
10. Y. Jin, Y. Makris, A proof-carrying based framework for trusted microprocessor IP, in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2013), pp. 824–829
11. INRIA, The Coq proof assistant (2010), http://coq.inria.fr/
12. F. Wolff, C. Papachristou, S. Bhunia, R.S. Chakraborty, Towards Trojan-free trusted ICs: problem analysis and detection scheme, in *IEEE Design Automation and Test in Europe* (2008), pp. 1362–1365
13. M. Hicks, M. Finnicum, S.T. King, M.M.K. Martin, J.M. Smith, Overcoming an untrusted computing base: detecting and removing malicious hardware automatically, in *Proceedings of IEEE Symposium on Security and Privacy* (2010), pp. 159–172
14. C. Sturton, M. Hicks, D. Wagner, S. King, Defeating UCI: building stealthy and malicious hardware, in *2011 IEEE Symposium on Security and Privacy (SP)* (2011), pp. 64–77
15. X. Zhang, M. Tehranipoor, Case study: detecting hardware trojans in third-party digital ip cores, in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2011), pp. 67–70
16. Y. Jin, Design-for-security vs. design-for-testability: A case study on dft chain in cryptographic circuits, in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2014), pp. 19–24
17. X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, P. Mishra, Pre-silicon security verification and validation: a formal perspective, in *Proceedings of the 52Nd Annual Design Automation Conference, DAC'15* (2015), pp. 145:1–145:6
18. F.M. De Paula, M. Gort, A.J. Hu, S.J. Wilton, J. Yang, Backspace: formal analysis for post-silicon debug, in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design* (IEEE Press, New York, 2008), p. 5
19. S. Drzevitzky, Proof-carrying hardware: Runtime formal verification for secure dynamic reconfiguration, in *2010 International Conference on Field Programmable Logic and Applications (FPL)* (2010), pp. 255–258
20. J. Rajendran, V. Vedula, R. Karri, Detecting malicious modifications of data in third-party intellectual property cores, in *Proceedings of the Annual Design Automation Conference, DAC '15* (ACM, New York, 2015), pp. 112:1–112:6
21. J. Harrison, Floating-point verification, in *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Proceedings*, ed. by J. Fitzgerald, I.J. Hayes, A. Tarlecki. Lecture Notes in Computer Science, vol. 3582 (Springer, Berlin, 2005), pp. 529–532
22. S. Owre, J.M. Rushby, N. Shankar, PVS: a prototype verification system, in *11th International Conference on Automated Deduction (CADE)* (Saratoga, NY), ed. by D. Kapur. Lecture Notes in Artificial Intelligence, vol. 607 (Springer, Berlin, 1992), pp. 748–752
23. D. Russinoff, M. Kaufmann, E. Smith, R. Sumners, Formal verification of floating-point RTL at AMD using the ACL2 theorem prover, in *Proceedings of the 17th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation, Paris, France* (2005)
24. J.-D. Quesel, S. Mitsch, S. Loos, N. Aréchiga, A. Platzer, How to model and prove hybrid systems with KeYmaera: a tutorial on safety. Int. J. Softw. Tools Technol. Transfer **18**, 67–91 (2016)

25. A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant* (MIT Press, Cambridge, 2013)
26. U. Norell, Dependently typed programming in Agda, in *Advanced Functional Programming* (Springer, Berlin, 2009), pp. 230–266
27. R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, S.F. Smith, *Implementing Mathematics with the Nuprl Proof Development System* (Prentice-Hall, Upper Saddle River, 1986)
28. L.C. Paulson, Isabelle: the next 700 theorem provers, in *Logic and Computer Science*, vol. 31 (Academic Press, London, 1990), pp. 361–386
29. E.M. Clarke, O. Grumberg, D. Peled, *Model Checking* (MIT press, Cambridge, 1999)
30. T.A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Software verification with blast, in *Model Checking Software*, (Springer, Berlin, 2003), pp. 235–239
31. J. O'Leary, X. Zhao, R. Gerth, C.-J.H. Seger, Formally verifying ieee compliance of floating-point hardware. Intel Technol. J. **3**(1), 1–14 (1999)
32. M. Srivas, M. Bickford, Formal verification of a pipelined microprocessor. IEEE Softw. **7**(5), 52–64 (1990)
33. T. Kropf, Introduction to Formal Hardware Verification (Springer, Berlin, 2013)
34. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood, seL4: formal verification of an os kernel, in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles* (ACM, New York, 2009), pp. 207–220
35. S. Chaki, E.M. Clarke, A. Groce, S. Jha, H. Veith, Modular verification of software components in C. IEEE Trans. Softw. Eng. **30**(6), 388–402 (2004)
36. H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M.F. Kaashoek, N. Zeldovich, Using crash hoare logic for certifying the fscq file system, in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP'15* (ACM, New York, 2015), pp. 18–37
37. M. Vijayaraghavan, A. Chlipala, N. Dave, Modular deductive verification of multiprocessor hardware designs, in *Computer Aided Verification* (Springer, Cham, 2015), pp. 109–127
38. A.A. Mir, S. Balakrishnan, S. Tahar, Modeling and verification of embedded systems using cadence SMV, in *2000 Canadian Conference on Electrical and Computer Engineering*, vol. 1 (IEEE, New York, 2000), pp. 179–183
39. M. Kwiatkowska, G. Norman, D. Parker, Prism: probabilistic symbolic model checker, in *Computer Performance Evaluation: Modelling Techniques and Tools* (Springer, Berlin, 2002), pp. 200–204
40. G.J. Holzmann, The model checker spin. IEEE Trans. Softw. Eng. **23**(5), 279 (1997)
41. D. Beyer, M.E. Keremoglu, Cpachecker: a tool for configurable software verification, in *Computer Aided Verification* (Springer, Berlin, 2011), pp. 184–190
42. A. David, K. G. Larsen, A. Legay, M. Mikučionis, Z. Wang, Time for statistical model checking of real-time systems, in *Computer Aided Verification* (Springer, Berlin, 2011), pp. 349–355
43. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in *Computer Aided Verification*, (Springer, Berlin 2000), pp. 154–169
44. C. Baier, J. Katoen, *Principles of Model Checking* (MIT Press, Cambridge, 2008)
45. A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, Y. Zhu, Symbolic model checking using sat procedures instead of BDDs, in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference* (ACM, New York, 1999), pp. 317–320
46. R.E. Bryant, Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
47. R.E. Bryant, Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **100**(8), 677–691 (1986)
48. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Nusmv 2: an opensource tool for symbolic model checking, in *Computer Aided Verification* (Springer, Berlin, 2002), pp. 359–364

49. E. Clarke, A. Biere, R. Raimi, Y. Zhu, Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (2001)
50. A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, Y. Zhu, Bounded model checking Adv. Comput. **58**, 117–148 (2003)
51. S. Qadeer, J. Rehof, Context-bounded model checking of concurrent software, in *Tools and Algorithms for the Construction and Analysis of Systems* (Springer, Berlin, 2005), pp. 93–107
52. G.C. Necula, Proof-carrying code, in *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1997), pp. 106–119
53. X. Guo, R.G. Dutta, Y. Jin, Hierarchy-preserving formal verification methods for pre-silicon security assurance, in *16th International Workshop on Microprocessor and SOC Test and Verification (MTV)* (2015)
54. G.E. Suh, J.W. Lee, D. Zhang, S. Devadas, Secure program execution via dynamic information flow tracking, in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI (2004), pp. 85–96
55. S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, R. Iyer, Defeating memory corruption attacks via pointer taintedness detection, in *Proceedings. International Conference on Dependable Systems and Networks, 2005. DSN 2005* (2005), pp. 378–387
56. W. Shi, J. Fryman, G. Gu, H.-H. Lee, Y. Zhang, J. Yang, Infoshield: a security architecture for protecting information usage in memory, in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006* (2006), pp. 222–231
57. N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani, D. August, RIFLE: an architectural framework for user-centric information-flow security, in *37th International Symposium on Microarchitecture, 2004. MICRO-37 2004* (2004), pp. 243–254
58. Y.-Y. Chen, P. A. Jamkhedkar, R.B. Lee, A software-hardware architecture for self-protecting data, in *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS'12* (2012), pp. 14–27
59. Y. Jin, D. Oliveira, Extended abstract: trustworthy SoC architecture with on-demand security policies and HW-SW cooperation, in *5th Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-5)* (2014)
60. Y. Jin, Y. Makris, Proof carrying-based information flow tracking for data secrecy protection and hardware trust, in *IEEE 30th VLSI Test Symposium (VTS)* (2012), pp. 252–257
61. X. Guo, R.G. Dutta, P. Mishra, Y. Jin, Scalable soc trust verification using integrated theorem proving and model checking, in *IEEE Symposium on Hardware Oriented Security and Trust (HOST)* (2016), pp. 124–129.
62. S. Berezin, *Model checking and theorem proving: a unified framework*. Ph.D. Thesis, SRI International (2002)
63. P. Dybjer, Q. Haiyan, M. Takeyama, Verifying haskell programs by combining testing, model checking and interactive theorem proving. Inf. Softw. Technol. **46**(15), 1011–1025 (2004)