Cycle-Accurate Information Assurance by Proof-Carrying Based Signal Sensitivity Tracing

Yier Jin* and Bo Yang† and Yiorgos Makris‡
*Department of Electrical Engineering and Computer Science, The University of Central Florida
†Department of Electrical Engineering, Zhejiang University
‡Department of Electrical Engineering, The University of Texas at Dallas
{yier.jin@ucf.edu, bo.yang.ts@gmail.com, yiorgos.makris@utdallas.edu}

***Abstract*—We propose a new information assurance model which can dynamically track the information flow in circuit designs and hence protect sensitive data from malicious leakage. Relying on the Coq proof assistant platform, the new model maps register transfer level (RTL) codes written in hardware description languages (HDLs) into structural Coq representatives by assigning all input, output, and internal signal sensitivity levels. The signal sensitivity levels can be dynamically adjusted after each clock cycle based on proposed signal sensitivity transition rules. The development of data secrecy properties and theorem generation functions makes the translation process from security properties to Coq theorems independent of target circuits and, for the first time, makes it possible to construct a property library, facilitating (semi) automation of the proof. The proposed cycle accurate information assurance scheme is successfully demonstrated on cryptographic circuits with various complexities from a small-scale DES encryption core to a state-of-the-art AES encryption design prohibiting the leakage of sensitive information caused by hardware Trojans inserted in RTL codes.**

I. Introduction

The problem of maliciously intended modifications, commonly known as hardware Trojans, in manufactured integrated circuits (ICs) has recently garnered interest not only in academia but also in governmental agencies and industry. Partly because of design outsourcing and fabrication migration to low-cost areas around the world, and partly because of increased reliance on third-party intellectual property (IP) cores and electronic design automation (EDA) tools from various vendors, the integrated circuit supply chain is now considered far more vulnerable to malicious modifications than ever before. In essence, the fundamental concern is that hardware Trojan-infected chips might be capable of additional functionality that the designer, vendor, and customer are unaware of, and this functionality may be exploited by the perpetrator after chip deployment or FPGA implementation. Depending on the field of application, the consequences of such attacks can range from minor inconvenience to major catastrophes, such as sabotaging or incapacitating a chip, stealing sensitive data, etc. Motivated by this new circuit designing and semiconductor manufacturing concern, researchers have proposed various methods for ensuring the security and trustworthiness of integrated circuits, which mostly focus on post-silicon stage, targeting attacks in untrusted foundry. These methods can be categorized into three groups and can be applied based on the availability of resources: *reverse engineering*, *enhanced functional testing*, and *side-channel fingerprinting* [1].

Because of the time-to-market pressure and the request to lower design costs, circuit designers and system integrators rely more on third-party IP cores than ever before. However, a full analysis of all previously proposed hardware Trojan detection methods reveals that few efforts have been spent in the area of pre-silicon hardware Trojan detection [2], [3]. Lacking of trusted RTL designs hampers with the effectiveness of post-silicon hardware Trojan detection methods because trusted RTL codes are often preliminary requests in these methods. More specifically, the malicious modifications in the circuit design written in hardware description languages (HDLs) can easily invalidate the process of generating genuine side-channel fingerprints or predicting low probability events because no golden model is available given the threat from RTL hardware Trojans. As a result, the protection of RTL designs and the development of trusted HDL codes become an urgent task among our efforts to protect the whole IC supply chain.

II. Dynamic Information Assurance Scheme

The syntax similarity between hardware description languages and software programming languages opens the door that we may protect third-party IP cores by exploiting software program protection methods in which the well-known proof-carrying code (PCC) serves as a good example. Originally developed by G. Necula, PCC provides a way to determine whether codes from potentially untrusted sources are safe to execute [4]. The verification method is accomplished by establishing a formal, automatically verifiable proof showing that questionable codes obey a set of formalized properties.

Adapting principles of the PCC methodology in the hardware domain, we propose a cycle accurate information assurance scheme performing data sensitivity tracking and information leakage prevention, which supports various levels of circuit architectures, ranging from low-complexity single-stage designs to large-scale deep-pipelined circuits. Similar to the PCC in software domain, properties formalization and proofs generation of the proposed scheme are also constructed on the Coq proof assistant platform [5]. Further, a structural Coq formal logic and a signal sensitivity transition model are developed to represent circuit logic in the Coq platform and to track information flow within the circuit, respectively. 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 representative, leaving the circuit functionality unspecified. This signal sensitivity transition model is used to facilitate dynamic information tracking in multi-stage circuit designs.

978-1-4799-0600-0

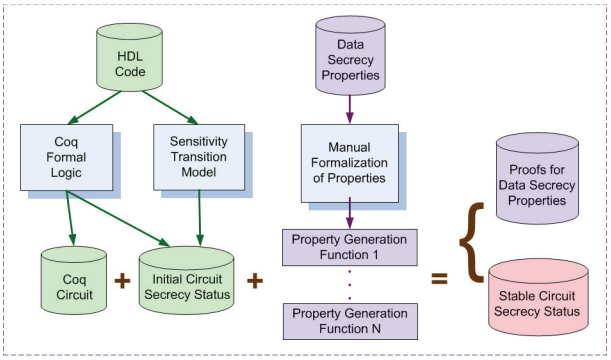

Fig. 1. Trusted Bundle Preparation by IP Vendors

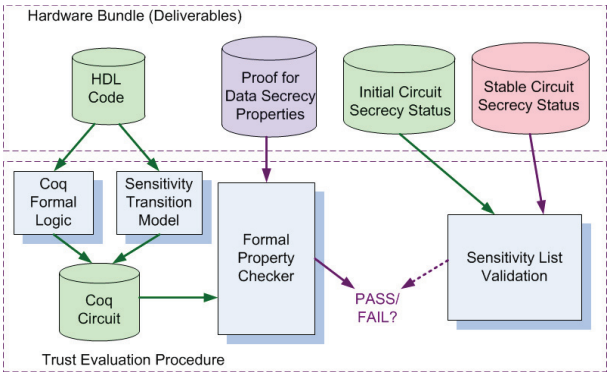

Fig. 2. Data Secrecy Property Verification by IP Consumers

The proposed information assurance scheme can be applied to any kinds of circuit designs, but it mostly 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 under protection. To achieve this goal, the proposed scheme assumes that the data secrecy property is agreed upon by both IP vendors and IP consumers beforehand so that the procedure of security property definition, a critical step when applying PCC in software domain, becomes trivial. The information assurance scheme, for the first time, sets up security property basis that is independent of target designs and is obeyed by any parties involved in the IP transaction process. Note that the security property basis with the characteristic of circuit functionality- and circuit structure-independence reveals our efforts toward constructing a security property library, a key component for proof automation.

Figure 1 illustrates the preparation process of the trusted bundle, which will later be delivered to IP consumers from IP vendors. As the figure shows, IP vendors will first design the circuit in the form of HDL codes based on the specification provided by IP consumers. Relying on the structural Coq formal logic and the signal sensitivity transition model, IP vendors then convert HDL codes into structural Coq representatives (aka Coq circuits). As a parallel step, IP vendors will translate the predefined data secrecy property from plain English text into formal theorems so that the Coq platform is able to recognize and later prove them.[1] Hardware IP vendors, unlike their counterparts in software domain, do not need to communicate with IP consumers to decide security properties. Furthermore, because the data secrecy property is independent of circuit functional specifications, IP vendors can perform the property conversion process before they finish writing HDL codes and store the converted Coq property in a property library, in the form of theorem generation functions, which we will introduce shortly, for other designs. The development of a Coq property library and the reuse of theorem proof contents lowers the workload for IP vendors and stimulates wider acceptance of the proposed proof-carrying based hardware IP protection scheme.

Note that IP consumers' absence from the duty for imposing security properties on IP cores does not mean that IP consumers lose control of the proof construction process. Rather, through defining the circuit initial secrecy status when the circuit is powered on (or is reset) and checking the stabilized circuit secrecy status after a finite number of operating clock cycles, IP consumers actively monitor the proof generation process for the data secrecy property. The circuit initial secrecy status and the stabilized status are in the form of signal sensitivity lists, named as the initial signal sensitivity list and the fix point signal sensitivity list, respectively.

Figure 2 shows the data secrecy property verification procedure performed by IP consumers upon receiving the delivered trusted bundle. As the first step, IP consumers will check contents of both signal sensitivity lists. The validity of the initial list is checked to ensure that sensitivity levels are appropriately assigned to all input/output/internal signals and are not changed by IP vendors. 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 Trojan channels that may leak sensitive information. Even though it is possible for IP consumers to detect information-leaking channels through either primary outputs or maliciously constructed side channels by scrutinizing the circuit stable status, in this paper we only discuss cases in which hardware Trojans try to illegally propagate sensitive information to primary outputs. Wider coverage of malicious data stealing will be presented in our later work.

Only if both signal sensitivity lists pass the initial checking, can IP consumers proceed to the next step to regenerate the Coq circuit from the delivered HDL codes based on the same Coq formal logic. Along with delivered theorem proofs, the reconstructed Coq circuit is loaded into a formal property checker, where an automatic checking process is performed. A "PASS" output from property checker provides evidence that HDL codes do not contain any malicious leaking channels prohibited by the data secrecy property. However, a "FAIL" result is a warning signal that the data secrecy property is breached because of malicious logics (or design faults) in the delivered IP cores. In the domain of hardware Trojan detection, the failure to pass the property checker leads to the detection of inserted hardware Trojans.

[1] In the scope of this paper, our information assurance scheme only supports the data secrecy property for IP cores and assumes that both IP consumers and IP vendors accept this property.

III. Coq Formal Logic

The framework of the proposed cycle accurate information assurance scheme, depicted in Figures 1 and 2, shows that both IP vendors and IP consumers should perform the conversion process from HDL codes to their Coq representatives. This conversion step becomes a necessity because the proposed dynamic scheme is built on the Coq proof assistant platform, which only supports Coq formal language. Although additional efforts have been spent to develop a customized formal platform supporting syntax and semantics of Verilog (and VHDL) directly, we still presently rely on the Coq proof assistant platform and Coq proof checker.

The conversion process from HDL codes to Coq circuits, though critical, is no more than a mapping procedure to rewrite the circuit description from one language to another. This conversion process can be finished either manually or automatically, supported by two entities, a Coq-based formal semantic model to describe circuit architecture and a set of HDL-to-Coq mapping rules. The formal semantic model is designed only to reflect the circuit structure but not to specify the functionality for the following reasons: (1) The signal sensitivity transition model, which tracks the information flow throughout the Coq circuit, requires structural equality between HDL codes and Coq representatives; and (2) The signal sensitivity transition model does not impose any restrictions to the functionality of Coq circuits, that is, functional definitions in the Coq semantic model complicate the conversation process and add unnecessary workload for IP vendors in preparing a trusted bundle.[2] Examples of conversion process and Coq circuits are discussed in Section VI with all codes listed in Appendices A and B.

IV. Property Theorems Generation

A. Signal Sensitivity Transition Model

Defined by the structural Coq formal semantic model, signal values in Coq circuits represent their sensitivity levels, not electronic values. Circuit signals are not just treated qualitatively as normal (not containing sensitive data) or sensitive (containing sensitive data) [6] but are also quantitatively allocated number 0 if they are normal or are allocated positive integers if they are secure. A larger number indicates a higher level of sensitivity so that the underlying signal requires high-level protection. For example, noncritical controlling/data signals, such as input clock signal, data loading signal, etc., are set to value 0, whereas encryption/decryption key and plaintext are assigned positive integers. We then need a mechanism to depict the way in which signal sensitivity levels evolve during circuit operation. The task is finished by a newly developed signal sensitivity transition model that is constituted by a set of signal sensitivity transition rules. These rules put restrictions on how to upgrade/downgrade signal sensitivity levels during the circuit operation. This model allows dynamic updating of signal sensitivities after each clock cycle.

To ensure the integrity of the transition model and to prevent sensitive information leakage from illegal signal sensitivity downgrading operations, the Coq semantic model is adjusted to be conservative so that 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: sensitivity downgrading expressions and module instantiation.

- **Sensitivity downgrading expressions.** These expressions are defined under similar syntax to other expressions but often perform special operations with sensitive data involved. For example, the permutation operation is the only sensitivity downgrading expression in the Data Encryption Standard (DES) core while XORing with round keys is the only case for an Advanced Encryption Standard (AES) circuit. Case studies of both DES and AES encryption cores can be found in Section VI, with codes in the Appendix.
- **Module instantiation.** Almost all modern designs are of hierarchical structure with submodules instantiated to perform various functionalities from round key generation to memory control. It is quite difficult to track information flow in and out of submodules if the whole design is not flattened. 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 are called endogenous inputs in contrast to primary inputs, which are hereafter 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, including the sensitivity level downgrading operation.

B. 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 such that the circuit's entire sensitivity status at a specified time `t` is stored in a signal sensitivity list, where each element of the list represents the sensitivity level of one signal–input, output, or internal signal. IP consumers can easily check the validity of the secrecy property by defining the initial status of the sensitivity list–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 property, which serves as the basis of the proposed scheme, is independent of the circuit functionality and architecture, the generation of initial signal sensitivity list is closely related to the circuit structure and the functional specification. Guidelines are developed for IP consumers to initialize signal sensitivities for all signals, including inputs, outputs, and internal signals.

- **Input signals.** The assignment of input signal sensitivity levels, including both exogenous inputs (primary inputs) and endogenous inputs (submodules' outputs), can be divided into two steps: (1) Decide whether input signals

[2]Note that the structural formal model may not apply to other security properties in which complex Coq models are required.

contain secret information; and (2) Measure the sensitivity level of input signals if they contain secret data. The first task is mostly finished upon the analysis of circuit functionality and is relatively easy for IP consumers because the circuit specification originates from them. For example, a DES encryption core would have plaintext, key, clock signal, and reset signal as exogenous inputs and round keys and encryption round count as endogenous inputs, which are shown in Figure 3. From the DES specification, IP consumers can recognize that exogenous inputs (plaintext, key) 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 like clock signal, encryption round count do not contain sensitive information whose sensitivity levels are set to 0. After categorizing input signals into sensitive or normal, we proceed to the second task to decide the actual sensitivity levels for sensitive signals with a larger number indicating a higher sensitivity level and vice versa. The calculation process is closely related to the circuit architecture designed by IP vendors, particularly dominated by (pipeline-)stages of the circuit implementation. Although more complex sensitivity level determination algorithms will be developed as our research proceeds on, a sensitivity level downgrading counting method is used and is proved effective in our later demonstrations. According to this method, upon receiving the HDL codes and the description of circuit architecture, IP consumers will check all paths from sensitive inputs to primary outputs and count the number of sensitivity downgrading operations along each route. The sensitivity levels of input signals are set equal to the smallest count of sensitivity downgrading operations among all these paths. Because IP vendors may sabotage the circuit by adding extra sensitivity downgrading logics to "bleach" sensitive signals, all downgrading operations will be clearly marked with notes explaining why and how these operations are performed.

- **Internal signals and output signals.** All signals other than input signals are treated as normal with a sensitivity level 0 because all internal and output signals are of preset (or random) values, which surely do not contain any sensitive information after the circuit is reset or powered off. Exceptions happen in 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 nicely solved because memory is always treated as a submodule with all outputs categorized as endogenous inputs under the reshuffling strategy.

The determined 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.

C. Theorem Generation Function

In this section, we will introduce the process of converting the data secrecy property from English text to Coq theorems in the scope of the proposed information assurance scheme. Because the data secrecy property itself is independent of the target circuit but the value of the initial sensitivity list is determined by circuit functionality and implementation architecture, we propose the concept of theorem generation function, for the first time, which takes Coq circuits and sensitivity lists as parameters and generates formal data secrecy theorems for target circuits in Coq platform. The proposal of theorem generation function is a breakthrough in the field of proof-carrying based hardware IP protection because it simplifies the theorem generation process for IP vendors but provides equivalent safety controlling capability for IP consumers through the definition of initial sensitivity list. The theorem generation function also separates the work of circuit development and security property theorem generation, a major step towards EDA tools development for theorem and proof auto-generation and makes it possible to integrate proof construction into the standard IC supply chain.

Before digging into details of the theorem generation function, we need to introduce a special sensitivity list that represents stabilized circuit security status, if it exists.

Definition 1: Fix point sensitivity list.

A fix point sensitivity list is a special sensitivity list containing circuit secrecy status at a specified time `t` with the key characteristic of stability. Denoting `coq_circuit` as the converted Coq circuit and `fix_list` as the fix point sensitivity list, if `fix_list` represents the current circuit secrecy status, then the circuit status will not change until the circuit is reset (or is powered off). In Coq platform, if the signal sensitivity updating function is `update_sensitivity`, the stability characteristic is presented in the form

```
update_sensivitity coq_circuit fix_list t
            = coq_circuit fix_list (t+1).
```

The data secrecy property, if described in English, means "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: (1) There exists a fix point sensitivity list; (2) The fix point sensitivity list is achievable from a legitimate initial sensitivity list; and (3) The circuit secrecy status defined by a fix point sensitivity list is trusted. These three subproperties will then be translated into Coq theorems relying on three theorem generation functions as long as the target circuit and the initial sensitivity list are both specified.

- **Theorem Generation Function I: Existence.** The existence of fix point sensitivity list is a preliminary request ensuring that data secrecy property can be proved for the target circuit. If we cannot find one or more fix point sensitivity lists, we believe that the target circuit is untrusted because sensitive data can be leaked freely.
- **Theorem Generation Function II: Accessibility.** The existence property demonstrates the availability of fix point sensitivity list for the target circuit, but it does not provide evidence that the fix point sensitivity list is accessible from the circuit's initial secrecy status. The second theorem tries to solve the problem that given the initial sensitivity list, the circuit will finally achieve stable

status after finite clock cycles.

- **Theorem Generation Function III: Trustworthiness.** Evaluating the trustworthiness of the derived fix point sensitivity list is the most critical step when validating data secrecy property. Because the fix point sensitivity list contains complete secrecy status of the target circuit, the goal of the trustworthiness theorem is to ensure that no sensitive information has leaked through primary outputs when the target circuit is stuck in the stable status.[3]

V. Prior Work

The first expansion of the PCC methodology in trusted hardware designs appeared in [7], where the authors introduced the application of Proof-Carrying Hardware (PCH) in the increasing prominence of FPGAs and reconfigurable devices. A proof is generated to demonstrate that an agreed-upon specification function is combinationally equivalent to the FPGA implementation (aka FPGA bitstream file). However, this IP protection approach can be only applied in FPGA domain, and it is also limited by the need to specify exact Boolean functionality.

To better implement security property verification in IP protection, the authors in [8] presented a new Proof-Carrying Hardware Intellectual Property (PCHIP) framework that helps guarantee that specified security properties are fulfilled by HDL codes. An IP acquisition and delivery protocol is also proposed on the Coq proof assistant platform to ensure the trustworthiness of purchased IP cores from untrusted IP vendors. PCHIP, for the first time, provides a general IP protection framework relying on security property proofs, but it does not specify details of security properties for individual design. Along this direction, the authors in [6] enhanced the Coq representative to include an information flow tracking property, a mechanism supporting the proof generation to demonstrate that the underlying IP cores will not leak sensitive data. However, the proposed data secrecy protection scheme, though effective in detecting any kinds of data leakage caused by hardware Trojans and/or design faults, suffers from the limitation that it cannot be implemented on multi-stage designs directly and can only check circuit trustworthiness in a static way.

VI. Demonstrations

To demonstrate the capability of the proposed cycle accurate information assurance scheme in protecting data secrecy and also provide concrete evidences that the proposed method can be applied to any IP cores with various circuit sizes and complexities, we employ two circuit designs in our demonstration, a DES encryption core representing small-scale, one-stage circuit logic and an AES encryption core as an example of medium-scale, multi-stage design. The DES example shows a rare case in which the initial signal sensitivity list is also a fix point sensitivity list. The AES example, in contrast, demonstrates a general situation for multi-stage designs that the fix point sensitivity list is different from the initial list. In both examples, we show how HDL codes are converted into Coq formal logic, how the initial sensitivity lists are constructed, and how three theorems are constructed from the theorem generation functions. Circuit descriptions, both in HDL codes and the converted Coq circuits, generated data secrecy theorems, and their proofs can be found in the Appendix.

[3]In this paper, we try to prevent data leakage from primary outputs and leave more thorough analysis on the fix point sensitivity list to future work.

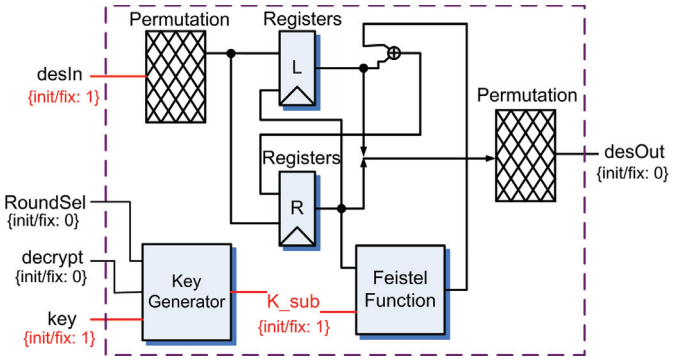

Fig. 3. Diagram of DES Encryption Core and Signal Sensitivity Levels

A. DES Encryption Core

The architecture of DES encryption core is shown in Figure 3, where the top module instantiates two submodules performing the functionalities of Key Generation and Feistel transformation, respectively. Figure 3 also shows that two register files (`L` and `R`) are used to store intermediate encryption results for the purpose of area optimization.

Coq DES Circuit Generation/Conversion. Based on the Coq formal logic and HDL-to-Coq conversion rules, the converted Coq DES circuit is of the same structure as that described by HDL codes. No functionality is specified for the Coq circuit.

Initial Sensitivity List. The analysis of the DES specification tells us that `desIn` and `key` among exogenous inputs and the endogenous input `K_sub` contain sensitive information, meaning their sensitivity levels should be positive. Further, the DES circuit only finishes one round of DES encryption operation, so we assign level 1 to all sensitive signals in accordance with the rule developed in Section IV-B. For internal/output signals, signal initialization rules already set their sensitivity levels to 0. That is, both `L` and `R` registers as well as the output `desOut` are of initial sensitivity level 0. Part of the circuit initial security status is also shown in Figure 3.

Theorems Generation. Theorems generation and proofs construction are key parts of the proof-carrying based IP protection scheme because they provide mathematical evidence that data secrecy is fulfilled by the delivered IP core and they are part of the trusted bundle prepared by IP vendors. Assisted by the three theorem generation functions, we can easily generate theorems representing the existence, accessibility, and trustworthiness of a stabilized circuit secrecy status.

Because of page limitation, constructed theorems and their proofs can be found in Appendix A.

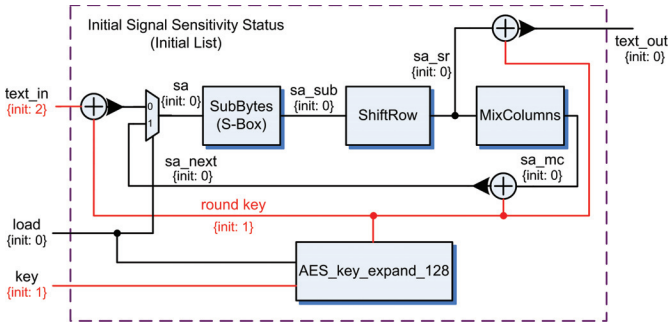

Fig. 4. AES Circuit Architecture and Initial Sensitivity Status

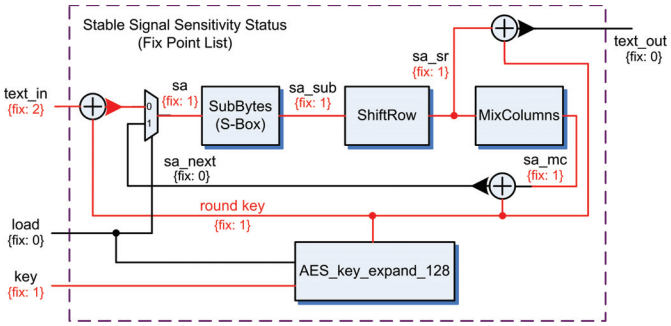

Fig. 5. Stable Sensitivity Status of the AES Circuit

B. AES Encryption Core

The diagram of the AES encryption core is shown in Figure 4, where the top module only instantiates *AES_key_expand_128* to generate round keys. `SubBytes` (non-linear byte substitution), `ShiftRow` (row shifting), and `MixColumns` (column mixing), though shown in an abstract way, represent top-level logics but not module instantiations.

Coq AES Circuit Generation/Conversion. The procedure to convert AES HDL code into AES Coq circuit is similar to that in the DES case, with the difference being that the AES core is of much larger circuit size and so are the converted Coq circuits. For example, there are 95 signals in the AES circuit versus 15 signals defined in the DES core, so the sensitivity list in the AES design is longer than that in the DES design (see Appendix B).

Initial Sensitivity List and Fix Point Sensitivity List. The diagram of the AES design in Figure 4 also shows the initial sensitivity levels of all circuit signals. Internal and output signals are set to 0, whereas both endogenous and exogenous inputs are assigned sensitivity levels depending on whether or not they contain sensitive information. The key input and internal round key are assigned sensitive level 1. Plaintext input `text_in` is assigned level 2 because the plaintext, before propagating to the output `text_out`, is XORed with round keys twice.

The fix point list of the AES core is shown in Figure 5. A comparison between Figure 4 and Figure 5 shows that the fix point sensitivity list is different from the initial sensitivity list in the AES design, a normal case for most medium- to large-scale designs.

Theorems Generation. With the fix point sensitivity list available, the task to generate theorems to prove the consistence of the AES design with the data secrecy property is simple and straightforward if assisted by the theorem generation functions. Three theorems are generated to prove the existence, accessibility, and trustworthiness of the circuit stable status, with details shown in Appendix B.

VII. Conclusion

Compared with the work on preventing and detecting hardware Trojans in post-fabrication chips, where many researchers have already proposed possible solutions, the field of hardware IP cores protection attracts less attention. The increasing reliance on third-party IP cores for hierarchical designs makes it more important to protect IP cores against RTL Trojans. The problem is also critical in the cryptography domain, where IP cores run encryption/decrypiton operations serving as the basis of system security. To protect IP cores dealing with sensitive information, we proposed a cycle accurate information assurance scheme within the scope of proof-carrying based hardware protection, which can dynamically track the spreading of sensitive information across the whole circuit in a cycle accurate way and therefore detect any malicious information leakage behaviors. Theorem generation functions are also proposed to lower the workload of proof preparation and pave the way toward the construction of a security property library, a key step in proof generation automation.

Acknowledgements

This research was partially supported by the U.S. Army Research Office (ARO) under grant W911NF-12-1-0091.

References

[1] Y. Jin and Y. Makris, "Hardware Trojans in wireless cryptographic ICs," *IEEE Design and Test of Computers*, vol. 27, pp. 26–35, 2010.

[2] M. Banga and M.S. 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.

[3] S. Drzevitzky and 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 (ReCoSoC)*, 2011, pp. 1–8.

[4] 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.

[5] INRIA, "The coq proof assistant," September 2010, `http://coq.inria.fr/`.

[6] Y. Jin and 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.

[7] S. Drzevitzky, U. Kastens, and M. Platzner, "Proof-carrying hardware: Towards runtime verification of reconfigurable modules," in *International Conference on Reconfigurable Computing and FPGAs*, 2009, pp. 189–194.

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

APPENDIX A
CASE I: DES CIRCUIT

A. Full Version of HDL Codes

```
module des(desOut, desIn, key,
                decrypt, roundSel, clk);
output  [63:0]  desOut;
input   [63:0]  desIn;
input   [55:0]  key;
input   decrypt;
input   [3:0]   roundSel;
input   clk;

wire    [1:48]  K_sub;
wire    [1:64]  IP, FP;
reg     [1:32]  L, R;
wire    [1:32]  Xin;
wire    [1:32]  Lout, Rout;
wire    [1:32]  out;

assign Lout = (roundSel == 0) ? IP[33:64] : R;
assign Xin  = (roundSel == 0) ? IP[01:32] : L;
assign Rout = Xin ^ out;
assign FP = { Rout, Lout};

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

always @(posedge clk)
        L <= #1 Lout;
always @(posedge clk)
        R <= #1 Rout;

// Select a subkey from key.
key_sel u1( .K_sub( K_sub ),
         .K( key ),
         .roundSel( roundSel ),
         .decrypt( decrypt ));

// Perform initial permutation
assign IP[1:64] = {
desIn[06], desIn[14], desIn[22], desIn[30],
desIn[38], desIn[46], desIn[54], desIn[62],
desIn[04], desIn[12], desIn[20], desIn[28],
desIn[36], desIn[44], desIn[52], desIn[60],
desIn[02], desIn[10], desIn[18], desIn[26],
desIn[34], desIn[42], desIn[50], desIn[58],
desIn[00], desIn[08], desIn[16], desIn[24],
desIn[32], desIn[40], desIn[48], desIn[56],
desIn[07], desIn[15], desIn[23], desIn[31],
desIn[39], desIn[47], desIn[55], desIn[63],
desIn[05], desIn[13], desIn[21], desIn[29],
desIn[37], desIn[45], desIn[53], desIn[61],
desIn[03], desIn[11], desIn[19], desIn[27],
desIn[35], desIn[43], desIn[51], desIn[59],
desIn[01], desIn[09], desIn[17], desIn[25],
desIn[33], desIn[41], desIn[49], desIn[57] };

// Perform final permutation
assign desOut = {FP[40], FP[08], FP[48], FP[16],
FP[56], FP[24], FP[64], FP[32], FP[39], FP[07],
FP[47], FP[15], FP[55], FP[23], FP[63], FP[31],
FP[38], FP[06], FP[46], FP[14], FP[54], FP[22],
FP[62], FP[30], FP[37], FP[05], FP[45], FP[13],
FP[53], FP[21], FP[61], FP[29], FP[36], FP[04],
FP[44], FP[12], FP[52], FP[20], FP[60], FP[28],
FP[35], FP[03], FP[43], FP[11], FP[51], FP[19],
FP[59], FP[27], FP[34], FP[02], FP[42], FP[10],
FP[50], FP[18], FP[58], FP[26], FP[33], FP[01],
FP[41], FP[09], FP[49], FP[17], FP[57], FP[25] };

endmodule
```

B. Full Version of Coq Representatives

```
Definition desIn : bus      := 0.     (* #0 *)
Definition key : bus        := 1.     (* #1 *)
Definition decrypt : bus    := 2.     (* #2 *)
Definition roundSel : bus   := 3.     (* #3 *)
Definition clk : bus        := 4.     (* #4 *)
Definition K_sub : bus      := 5.     (* #5 *)
Definition IP : bus         := 6.     (* #6 *)
Definition FP : bus         := 7.     (* #7 *)
Definition L : bus          := 8.     (* #8 *)
Definition R : bus          := 9.     (* #9 *)
Definition Xin : bus        := 10.    (* #10 *)
Definition Lout : bus       := 11.    (* #11 *)
Definition Rout : bus       := 12.    (* #12 *)
Definition out : bus        := 13.    (* #13 *)
Definition desOut : bus     := 14.    (* #14 *)

Definition des_signals : signal :=
  outb desOut &      (* #14 *)
  inb desIn &        (* #0 *)
  inb key &          (* #1 *)
  inb decrypt &      (* #2 *)
  inb roundSel &     (* #3 *)
  inb clk &          (* #4 *)
  wireb K_sub &      (* #5 *)
  wireb IP &         (* #6 *)
  wireb FP &         (* #7 *)
  regb L &           (* #8 *)
  regb R &           (* #9 *)
  wireb Xin &        (* #10 *)
  wireb Lout &       (* #11 *)
  wireb Rout &       (* #12 *)
  wireb out.         (* #13 *)

Definition des : code :=
 assign_ex Lout (cond (eq (econb roundSel)
  (econv (0))) (econb (IP @ [33, 64])) (econb R));
 assign_ex Xin (cond (eq (econb roundSel)
  (econv (0))) (econb (IP @ [1, 32])) (econb L));

  assign_ex Rout (exor (econb Xin) (econb out));
  assign_ex FP (eapp Rout Lout);

  module_inst2in out Lout K_sub;

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

  module_inst3in K_sub key roundSel decrypt;

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

C. Data Secrecy Theorems and Proofs

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

Lemma fp_list_existence :
 update_sensitivity des des_sen_initial = des_sen_initia
Proof.  intros.  reflexivity.  Qed.

Definition des_sen_stable : code_sen:=des_sen_initial.

Theorem fp_list_accessability :
  forall t : nat, t > 0 -> (check_sensitivity t
            des des_sen_initial) = des_sen_stable.
Proof.
  intros. induction H. reflexivity.
  unfold check_sensitivity. rewrite fp_list_existence.
  simpl. apply IHle.
Qed.

Fixpoint nth (n:nat) (l:list nat) {struct l}:nat:=
    match n, l with   | O, x :: l' => x
                      | O, other => 999
                      | S m, nil => 999
                      | S m, x :: t => nth m t end.

Theorem no_leaking : nth desOut des_sen_stable = 0.
Proof.  trivial.  Qed.  (*Property Proved*)
```

Appendix B
Case II: AES Circuit

A. Segments of HDL Codes

```
`include "timescale.v"
module aes_cipher_top(clk, rst, ld, done, key,
                         text_in, text_out );
input   clk, rst;
input   ld;
output  done;
input   [127:0] key;
input   [127:0] text_in;
output  [127:0] text_out;

wire    [127:0] w;
wire    [31:0]  w0, w1, w2, w3;
reg     [127:0] text_in_r, text_out;
reg     [7:0]   sa00, sa01, sa02, sa03;
reg     [7:0]   sa10, sa11, sa12, sa13;
reg     [7:0]   sa20, sa21, sa22, sa23;
reg     [7:0]   sa30, sa31, sa32, sa33;
...

assign w3 = w[127:96];
assign w2 = w[95:64];
assign w1 = w[63:32];
assign w0 = w[31:0];

always @(posedge clk)
  if(!rst)  dcnt <= #1 4'h0;
  else
    if(ld)  dcnt <= #1 4'hb;
  else
    if(|dcnt)  dcnt <= #1 dcnt - 4'h1;

always @(posedge clk)
  done <= #1 !(|dcnt[3:1]) & dcnt[0] & !ld;
always @(posedge clk)
  if(ld) text_in_r <= #1 text_in;
always @(posedge clk) ld_r <= #1 ld;

// Initial Permutation (AddRoundKey)
always @(posedge clk) sa33 <= #1 ld_r ?
    text_in_r[007:000] ^ w3[07:00] : sa33_next;
always @(posedge clk) sa23 <= #1 ld_r ?
    text_in_r[015:008] ^ w3[15:08] : sa23_next;
...
```

B. Segments of Coq Representatives

```
Definition aes : code :=
  assign_b w3 (w[127, 96]);
  assign_b w2 (w[ 95, 64]);
  assign_b w1 (w[ 63, 32]);
  assign_b w0 (w[ 31,  0]);

(* // Misc Logic *)
 nonblock_assign_ex dcnt (cond (enot (econb rst))
  (econv 0) (cond (econb ld) (econv 10)
  (cond (eor_bit dcnt) (eminus (econb dcnt)
  (econv 1)) (econb dcnt))));
 nonblock_assign_ex done
  (eand (eand (enot (eor_bit (dcnt[3,1])))
  (econb (dcnt[0,0]))) (enot (econb ld)));
 nonblock_assign_ex text_in_r
  (cond (econb ld) (econb text_in)
  (econb text_in_r));
 nonblock_assign_b ld_r ld;

(* // Initial Permutation (AddRoundKey) *)
 nonblock_assign_ex sa33
  (cond (econb ld_r) (exor_key (text_in_r[7,0])
  (w3[7,0])) (econb sa33_next));
 nonblock_assign_ex sa23
  (cond (econb ld_r) (exor_key (text_in_r[15,8])
  (w3[15,8])) (econb sa23_next));
...
```

C. Data Secrecy Theorems and Proofs

```
Definition aes_initial_list : code_sen :=
0 :: 0 :: 0 :: 1 :: 2 :: 0 :: 0 :: 0 :: 0 :: 0 ::
0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 ::
0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 ::
0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 ::
0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 ::
0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 ::
0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 ::
0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 ::
0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 ::
0 :: 0 :: 0 :: 0 :: 0 ::nil.

Definition aes_stable_list : code_sen :=
0 :: 0 :: 0 :: 1 :: 2 :: 0 :: 0 :: 0 :: 0 :: 0 ::
2 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 ::
1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 0 :: 0 :: 0 ::
0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 ::
0 :: 0 :: 0 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 ::
1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 ::
1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 ::
1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 ::
1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 :: 1 ::
1 :: 0 :: 0 :: 0 :: 0 :: nil.

Lemma aes_sen_stable : update_sensitivity
          aes aes_stable_list = aes_stable_list.
Proof.
  intros. vm_compute. reflexivity.
Qed.

Lemma stable_code_sen_chk :
 forall t:nat, check_sensitivity t aes aes_stable_list
                                = aes_stable_list.
Proof.
  intros.
  induction t. vm_compute. reflexivity.
  unfold check_sensitivity. fold check_sensitivity.
  rewrite IHt.
  apply aes_sen_stable.
Qed.

Lemma stable_state : forall t:nat, t = 5 ->
 check_sensitivity t aes aes_initial_list
                  = aes_stable_list.
Proof.
  intros.
  rewrite H. vm_compute. reflexivity.
Qed.

Theorem fp_list_accessability :
 forall t : nat, t > 5 ->
 (check_sensitivity t aes aes_initial_list)
                    = aes_stable_list.
Proof.
  intros. induction H. vm_compute. reflexivity.
  unfold check_sensitivity. fold check_sensitivity.
  rewrite IHle.
  apply aes_sen_stable.
Qed.

Fixpoint nth (n:nat) (l:list nat) {struct l} :nat:=
    match n, l with
      | O, x :: l' => x
      | O, other => 999
      | S m, nil => 999
      | S m, x :: t => nth m t
    end.

Theorem no_leaking_1 : nth done aes_stable_list = 0.
Proof.  trivial.  Qed.  (*Property Proved*)

Theorem no_leaking_2 :
      nth text_out aes_stable_list = 0.
Proof.  trivial.  Qed.  (*Property Proved*)
```