Proof Carrying-Based Information Flow Tracking for Data Secrecy Protection and Hardware Trust

Yier Jin* and Yiorgos Makris†
*Department of Electrical Engineering, Yale University
†Department of Electrical Engineering, The University of Texas at Dallas
{yier.jin@yale.edu, yiorgos.makris@utdallas.edu}

***Abstract*—We discuss a new approach for protecting the secrecy of internal information in an Integrated Circuit (IC) from malicious hardware Trojan threats and, thereby, enhancing hardware trust. The proposed approach is based on Register Transfer Level (RTL) code certification within a formal logic environment. The key novelty lies in the introduction of a new semantic model for the Verilog Hardware Description Language (HDL) in the Coq theorem-proving platform, which facilitates tracking and proving secrecy labels of internal sensitive data and, by extension, security properties of the design. Additional framework enhancements include the ability to encapsulate submodule properties in the top module proof environment, thereby strengthening the ability of Coq representation to reason on hierarchically organized RTL code. We demonstrate the proposed framework on a DES encryption core, wherein we employ it to prevent secret information (e.g. round keys) leaking by hardware Trojans inserted at the RTL description of the circuit.**

I. Introduction

The increasingly globalized Integrated Circuit (IC) supply chain has recently given rise to questions regarding chip trustworthiness. The fundamental concern is the potential inclusion of malicious functionality, known as hardware Trojans, which may cause erroneous behavior, steal sensitive information, incapacitate, or even destroy a chip. The potential implications of this problem have resulted in a large body of current research on this topic, with the majority of the efforts focusing on prevention and detection at the post-silicon stage [1]–[6], assuming that the culprit will act at the manufacturing site. Of equal importance, however, is protection against threats that may exist earlier in the supply chain, and in particular in the acquisition of third party intellectual property (IP) cores. While most contemporary designs involve some form of third-party IP, and while the barrier to entrance for contaminating an HDL description is seemingly lower than for altering fabrication masks, little effort has been expended towards pre-silicon hardware Trojan prevention and detection [7]–[9].

Towards this end, in this work, we introduce an information flow tracking methodology for data secrecy protection. The proposed method borrows concepts from formal programming languages, such as proof carrying code [10] and decentralized labels [11], towards enhancing trustworthiness of third-party IP cores. The proposed method operates in two stages: (i) As shown in Figure 1, the IP vendor crafts proofs regarding the secrecy properties of certain data, as agreed upon with the consumer. These properties, written in a formal language, along with the circuit, which is enhanced with secrecy tags and is also expressed in the same formal language, are then passed as a bundle to the consumer. (ii) As shown in Figure 2, the IP consumer, uses a formal property checker to confirm that the circuit complies to the agreed-upon security properties.

We note that, since the data secrecy properties are formally proven, it is impossible for a hardware Trojan to violate a secrecy property without making the proof fail. Hence, the proposed method provides the IP consumer with a simple and powerful mechanism for continuously checking an acquired core throughout the design flow and ensuring that no hardware Trojan is inserted, not only by the IP vendor, but also by potential culprits within the IP consumer's own organization. Similarly, the proposed method provides the IP vendor with a defense mechanism against potential liability claims.

The remainder of this paper is organized as follows: In section II, we elaborate on the logistics of the proposed proof-carrying-based information flow tracking scheme for data secrecy protection. In section III, we provide details about the proposed semantic model for expressing a circuit in the Coq proof-assistant platform. Sensitivity properties for information flow tracking and property reasoning across hierarchical module instantiations are described in section IV. An example implementation of the proposed protection scheme and its effectiveness in detecting inserted hardware Trojans on a DES core are presented in section V. Finally, conclusions are drawn in section VI.

II. Proposed Information Flow Tracking Scheme

The procedure for an IP vendor to specify data secrecy properties and provide proofs for them, as shown in Figure 1, involves a number of steps. The IP vendor first designs the circuit based on the functional specifications provided by the IP consumer, in the form of HDL code. Utilizing a formal semantic model and information flow tracking rules,

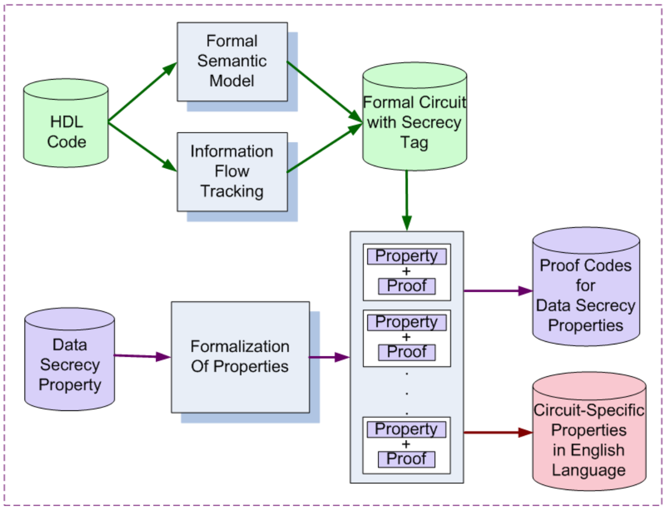
Formal Semantic Model
HDL Code
Information Flow Tracking
Formal Circuit with Secrecy Tag
Property + Proof
Property + Proof
Property + Proof
Proof Codes for Data Secrecy Properties
Data Secrecy Property
Formalization Of Properties
Circuit-Specific Properties in English Language

Fig. 1. Data secrecy property declaration and proof generation

978-1-4673-1074-1/12/$31.00 © 2012 IEEE

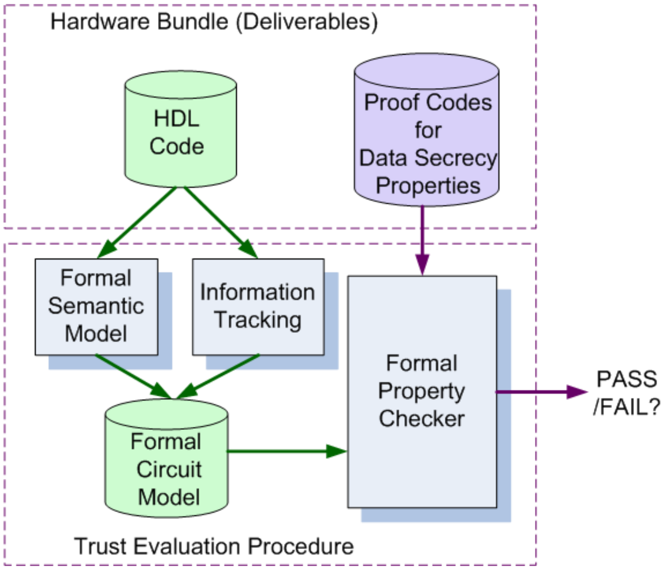

Fig. 2. Data secrecy property verification

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 data secrecy property from English text into formal theorems. Note that these data secrecy properties are agreed upon by both IP consumer and IP vendor. These formal theorems cannot be proven in a standalone fashion because parameters for theorems are missing at this stage. Subsequently, the converted formal logic is loaded into the formal theorems as parameters. The IP vendor, then, writes the proof code for the parameterized formal security theorems. Both the proofs and formal theorems are part of the final deliverable handed to the IP consumer.

Upon receiving the hardware bundle which includes the RTL code and proof codes for data secrecy property, as shown in Figure 2, the IP consumer regenerates the formal logic of the original circuit based on the same formal semantic model and information flow tracking rules. The regenerated semantic description is then combined with the provided proof codes to pass through the formal property checker. A “PASS” result means that the delivered IP core fulfills the agreed-upon data secrecy property. However, a “FAIL” signal warns the user that malicious circuit (or design errors) exist in the IP core, making it violate the data secrecy properties. Any system that contain IP cores which do not pass the property checker should then be treated as untrusted. Within the context of hardware Trojans, this scheme will enable detection of maliciously added code that attempts to leak sensitive data.

The proposed information tracking scheme is conceptually similar to the well-known software proof-carrying code (PCC) methods [10], enhanced with labels [11]. With regards to the complexity of this scheme, most of the burden falls on the IP vendors, who have to specify the security properties in formal logic and write the proofs. The IP consumer, however, only has a trivial task, since the proof checking is fast, automated and does not required extensive computational resources.

III. Circuit Semantic Model in Coq

As depicted in Figure 2, the IP consumer uses a formal property checker to decide whether the underlying HDL code contains any hardware Trojans (or, for that matter, inadvertent design errors) which may violate the expected data security properties and leak sensitive internal information. Constructing proof-checkers in native HDLs would be extremely complicated, as these languages were not designed with such formal tasks in mind. Instead, we opted to use Coq, a well-developed proof-assistant platform [12], which is widely used for similar property-checking purposes in the software community. Therefore, we need a mechanism for expressing both the circuit and the proof codes in the Coq proof-assistant language, in order to use the formal property checker.

In order to convert HDL code to Coq-recognizable formal logic, we develop two entities: (i) a structural semantic model within the Coq platform, which allows us to represent hardware circuitry, and (ii) a set of rules for converting HDL to this Coq semantic model, as described below.

The structural semantic model is actually a new hardware description language represented in Coq formal logic Using this new HDL, we can express any IP core solely within the semantic model of the Coq platform. An important feature of this structural semantic model in Coq is that it can precisely describe the structure of the circuit but has loose restrictions on the functionality of the operators. In other words, the architecture of the circuit is accurately described in the Coq semantic model but designers (or proof writers) have flexibility in how to define the functionality of the circuit. Furthermore, as we will demonstrate later, this structural Coq semantic model is quite effective in tracking information flow inside the circuit, in support of the target objective of data secrecy property checking.

The set of HDL-to-Coq conversion rules were developed for the Verilog HDL currently, since we do not expect the designers to write their IP cores in our semantic model. Instead, using these rules, we can automate the process, making it transparent to them. Through this conversion, the Coq circuit resembles structurally the Verilog-described circuit.

In the rest of this section, we first introduce the new structural semantic model in a stepwise manner, starting with the preliminary definition of signals and proceeding with more complex expressions as well as the semantics of operators. We then finish the section with the key aspects of the Verilog-to-Coq conversion rules.

A. Signal Definition

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

```
Inductive value := lo | hi.
```

```
Definition bus_value := list value.
Definition bus := nat -> bus_value.
Definition bus_length (b : bus) :=
  Fun t : nat => length (b t).
```

B. Signal Operations

We construct bus handling methods in the semantic model. These methods include logic operations such as `and`, `or`, `xor`, etc., as well as bus comparisons such as checking for bus equality, `bus_eq`, less-than comparison, `bus_lt`, etc. Considering that in RTL code signals are often compared with 0 to decide the direction of branches in `if...else...` statements, we add a special function to compare the bus value with 0, `bus_eq_0`.

```
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’ => (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).
```

C. Bus Slicing

It is often the case that, in a circuit, operations are performed on certain bits of the bus but not the entire bus. Most hardware description languages therefore provide quite flexible syntax to define bus length and bus bit-sequence. In order to support similar flexibility in our Coq semantic model, we developed two bus-slicing operations to shuffle data bits from lower bit positions to higher bit positions and vice-versa. Bit selection notations are also proposed to simplify code writing.

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

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

D. Expressions

On top of the signal definitions and operation rules, we build expressions to represent more complicated circuit logic. An expression is defined as an inductive set with operators to construct new expressions or combine expressions together.

Plenty of operators are supported, varying from basic logic operations (AND, OR, etc.) to sophisticated data manipulation (S-box mapping, permutation, etc.). The expression definition shown below is an excerpt from the complete expression definition, wherein operators have been chosen with particular attention to common tasks performed in cryptographic IP cores, since it is highly likely that data secrecy properties will have to be proven for such designs. A constant value list and a bus can be directly converted to expressions using the `econv` and `econb` constructors, respectively. The `eand`, `eor` and `exor` constructors connect two expressions to form a new expression, by performing logical AND, OR and XOR operations, respectively. The `perm` and `sbox` constructors are used to indicate permutation and S-box mapping operations. In this structural semantic model, it is unnecessary to specify how the permutation and/or S-box mapping is actually performed. These structural constructors liberate the proof writers from tedious functional conversion, which may be unnecessary for data secrecy property checking.

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

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

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

E. Coq Semantic Model

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

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

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

F. Verilog-Coq Conversion Rules

Since the Coq semantic model we developed has similar syntax with Verilog code, the fundamental Verilog-to-Coq conversion rule which we need to obey is to keep the original code and destination code structurally the same. This forms the basis for the Verilog-to-Coq conversion methodology that we developed. For example, a combinational assign logic is mapped to a `assign_ex` statement and module instantiation is mapped to `module_inst` statement as shown below.

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

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

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

Converted Coq formal logic:
  module_inst2in out Lout K_sub;
```

IV. Module Instantiation and Information Flow

A. Information Flow

In itself, the new semantic model can still not achieve the goal of protecting internal sensitive signals and designing trusted IP cores, since it is simply an alternative HDL for representing the circuit structure. Information leaking hardware Trojans can still use signal bypassing strategies, which propagate internal sensitive data to primary outputs [13] or disseminate it through Trojan side channels [14], with little modification in the original circuit. However, the circuit description is now in a language that lends itself to formal reasoning. Therefore, by adding the right elements (i.e. tags) and logic for formally reasoning on these elements, we can now support our cause.

Towards enabling the new semantic model to facilitate tracking of internal sensitive data and proving of secrecy properties on this data, we incorporate an additional property, namely *sensitivity*, to circuit signals. This property is akin to the existing *value* property and it allows us to formally examine and prove signal integrity (from a security point of view) within the entire design. More specifically, we explicitly define signal sensitivity on top of the existing semantic model, in order to support information flow tracking. Having chosen the Coq platform for our semantic model comes in handy, since it is relatively straightforward to enhance the Coq formal logic to support information flow tracking. The only significant change is that we need to extend the bus definition so that it will return a `value*sensitivity` pair at a specified time `t` instead of just a `value`. Sensitivity is defined as an inductive set with two constructors, `secure` and `normal`, indicating whether the signals are sensitive and need protection or not[1]. A bus with a `secure` tag is, then, not allowed to propagated to a primary output or a Trojan side channel.

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

Now that each bus has a sensitivity tag, we need to define the propagation rules (i.e. tag algebra) for those tags as the corresponding signals travel from inputs through bus operations to outputs. Three operation rules are defined.

```
Definition uoptag
  (a : sensitivity) : sensitivity := a.
Definition boptag
(a b : sensitivity) : sensitivity :=
  match a with
  | secure => secure
  | normal => match b with
             | secure => secure
             | normal => normal
             end
  end.
Definition rmtag
  (a : sensitivity) : sensitivity := normal.
```

The `uoptag` function deals with the case where a bus is the operand of a unary operator. In this case, as the definition indicates, the unary operator reserves the sensitivity tag of the bus. The definition for binary operators, `boptag`, is similar to the OR logic, where the output signal tag is `secure` as long as one of the input signals is `secure`. The only way to switch the `secure` tag back to a `normal` tag is through the `rmtag` function. As we will see shortly in the DES case-study, permutation and module instantiation are the only legal

[1] More complex multi-level sensitivity schemes can also be defined, though the corresponding operator algebra will also become more complex.

operations which are allowed to call the rmtag function to remove secure tags. It is important to restrict the ability of removing the secure tag to a small number of well-controlled operators, in order to prevent leakage of internal sensitive data by "declassification".

B. Module Instantiation

Modern circuits are typically designed hierarchically to better manage the integration of IP cores into large systems and SoC designs. The hierarchical architecture simplifies testing of circuit designs but, in our case, poses an additional challenge on how to transfer security properties from lower level modules to higher level modules so that proofs can be constructed across module instantiation interfaces. In our scheme, since the security property we need to prove for modules at different levels remains the same, i.e. data secrecy, we follow a bottom-up property transferring solution to solve this problem. The procedure to perform this bottom-up security checking in a hierarchical system is listed below:

1) Check the data secrecy protection property of modules at the bottom level.
2) Move to modules in higher levels. Redefine the sensitivity tags of signals at the interface between the current module and modules at the lower levels. Check the security of the module.
3) Repeat step (2) until the top level module is reached.

The DES core example in the next section demonstrates the effectiveness of this method to ensure the data security property across levels in a hierarchical designs.

V. DES Example

In order to demonstrate the capability of the semantic model to support information flow tracking and module instantiation towards protecting internal sensitive information and preventing leakage of secret information (e.g. encryption key, plaintext) by hardware Trojans, we employ a DES core written in Verilog [15]. In this example, we show how the Verilog code is converted into Coq formal logic, how the information flow tags are added to the Coq DES circuit description, how the data secrecy property that we are interested in proving is constructed and, finally, how the proof checker detects the presence of hardware Trojans.

A. DES Circuit in Coq

The architecture of the DES core is shown in Figure 3, wherein the top module, des.v, instantiates two submodules, the Feistel function, crp.v and the key generator, key_sel.v, to perform round encryption/decryption and round key generation, respectively.

Based on the Verilog-to-Coq conversion rules and the Coq semantic model, all three modules (des.v, crp.v and key_sel.v) are converted into their Coq equivalent. Part of the converted Coq code for des.v is shown below.

```
Definition des : code :=
  outb desOut;
  inb desIn;
  inb key;
```

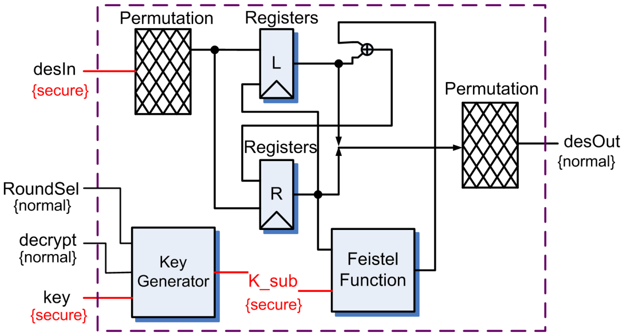

Fig. 3. Architecture of DES circuit

```
  inb decrypt;
  inb roundSel;
  inb clk;
  wireb K_sub;
...
  assign_ex Lout (cond (eq (econb roundSel)
    (econv (lo::lo::lo::lo::nil)))
    (econb (IP @ [33, 64])) (econb R));
  assign_ex Xin (cond (eq (econb roundSel)
    (econv (lo::lo::lo::lo::nil)))
    (econb (IP @ [1, 32])) (econb L));
  assign_ex Rout (econb (bus_bit_xor Xin out));
  assign_ex FP (econb (bus_app Rout Lout));
  module_inst2in out Lout K_sub;
  nonblock_assign_ex L (econb Lout);
...
```

B. Property Proof

For each module, we need to denote the information tags and prove the data secrecy property. Due to limited space, we only show the process for the top module, des.v, which also demonstrates the process of module instantiation. Similar steps are followed hierarchically for the other two modules.

For the des.v module, among all input, output and internal signals, the input key (key), input plaintext (desIn) and internal generated round keys (K_sub) require protection. As shown in Figure 3, we added a {secure} tag on these signals. Reflected in Coq formal logic, three axioms are added to reflect the semantics that the key, desIn and K_sub have secure sensitivity tags *in all* clock cycles. The separation of signal property denotation axioms and circuit code constitutes a key characteristic of the Coq platform. These axioms act as *preconditions* for all security properties extracted from the Coq circuit representation.

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

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

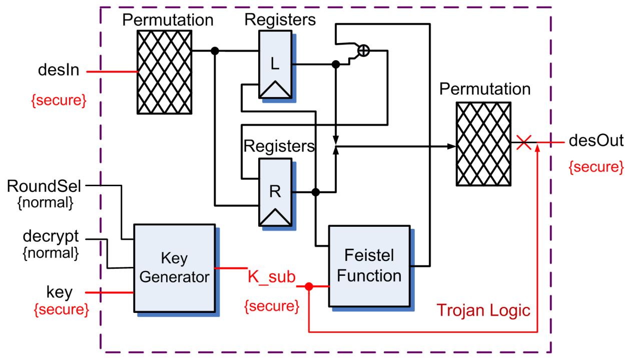

Fig. 4. Architecture of Trojan-infested DES circuit

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

If the Coq DES circuit and the preconditions can prove the `no_leaking_des` theorem, we can then declare that the delivered HDL code is trusted in protecting data secrecy. Not surprisingly, the theorem is proven (details of writing the proof in Coq are omitted due to space limitations). From a top-level perspective, this implies that the `{normal}` tag under the `desOut` output denotes no sensitive data is leaked in Figure 3. Also, from an IP acquisition perspective, this implies that a customer can be given the code for this DES core, along with the proof, and independently verify them using a Coq property checker, through the flow depicted in Figure 2.

C. Hardware Trojan Detection

In order to demonstrate the ability of the proposed information flow tracking scheme to detect hardware Trojans, we insert a hardware Trojan at the RT-level description of the DES core in our example. When triggered, this Trojan bypasses the internal round key directly to the primary output. Figure 4 shows the impact of the inserted Trojan on the information flow. Previous work in [13] has already demonstrated that if the Trojan is only triggered by rare everts, it is difficult to detect its existence. Yet in the proposed scheme, Trojan detection is independent of the triggering condition, since it happens through a formal theorem proving approach, rather than actual application of stimuli to the circuit. The converted Coq formal logic from the Trojan-infested HDL code, combined with the preconditions, lead us to the conclusion that the output `desOut` is of `secure` tag (as shown in Figure 4). Reflected in Coq formal logic, the `no_leaking_des` theorem cannot be proven, an evidence that the inserted Trojan is detected.

VI. Conclusion

While the majority of contemporary research in hardware Trojan prevention and detection focuses on post-silicon actions, the problem of untrusted, potentially Trojan-infested RT-Level code is becoming of equal importance. Indeed, the extended use of third party IP has intensified such concerns, necessitating similar pre-silicon hardware Trojan prevention and detection methods. To this end, we introduced a new information flow tracking scheme based on the principles of proof-carrying code. A new semantic model for expressing an RTL circuit description in the theorem-proving language Coq through a set of Verilog-to-Coq conversion rules, provides a formal framework for reasoning on data secrecy properties and, thereby, increasing hardware trustworthiness. Using a genuine and a Trojan-infested version of a DES circuit, we demonstrated that the proposed information flow tracking method can definitively prove or disprove security properties of IP cores and, thereby, detect hardware Trojans that violate these properties. Future expansions of this framework will focus on enhancing the Coq semantic model to support functional-level code conversion, as well as on automated security property extraction.

References

[1] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, “Trojan detection using IC fingerprinting,” in *IEEE Symposium on Security and Privacy*, 2007, pp. 296–310.

[2] Y. Jin and Y. Makris, “Hardware Trojan detection using path delay fingerprint,” in *IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008, pp. 51–57.

[3] R. M. Rad, X. Wang, M. Tehranipoor, and J. Plusquellic, “Power supply signal calibration techniques for improving detection resolution to hardware Trojans,” in *IEEE/ACM International Conference on Computer-Aided Design*, 2008, pp. 632–639.

[4] R. Rad, J. Plusquellic, and M. Tehranipoor, “Sensitivity analysis to hardware Trojans using power supply transient signals,” in *IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008, pp. 3–7.

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

[6] M. Tehranipoor and F. Koushanfar, “A survey of hardware Trojan taxonomy and detection,” *Design Test of Computers, IEEE*, vol. 27, pp. 10–25, 2010.

[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] M. Banga and M.S. Hsiao, “Trusted RTL: Trojan detection methodology in pre-silicon designs,” in *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*, june 2010, pp. 56 –59.

[9] 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*, 2012, (to appear).

[10] 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.

[11] A. C. Myers and B. Liskov, “Protecting privacy using the decentralized label model,” *ACM Transactions on Software Engineering and Methodology*, vol. 9, 2000.

[12] INRIA, “The coq proof assistant,” September 2010, `http://coq.inria.fr/`.

[13] Y. Jin, N. Kupp, and Y. Makris, “Experiences in hardware Trojan design and implementation,” in *IEEE International Workshop on Hardware-Oriented Security and Trust*, 2009, pp. 50–57.

[14] L. Lin, M. Kasper, T. Guneysu, C. Paar, and W. Burleson, “Trojan side-channels: Lightweight hardware Trojans through side-channel engineering,” in *Cryptographic Hardware and Embedded Systems*, vol. 5747 of *LNCS*, pp. 382–395. Springer-Verlag Berlin, 2009.

[15] *http://www.opencores.org/projects.cgi/web/des/overview*.