Using Security Invariant to Verify Confidentiality in Hardware Design

Shuyu Kong
Northwestern University
shuyukong2020@u.northwestern.edu

Yuanqi Shen
Northwestern University
yuanqishen2020@u.northwestern.edu

Hai Zhou
Northwestern University
haizhou@northwestern.edu

ABSTRACT

Due to the increasing complexity of design process, outsourcing, and use of third-party blocks, it becomes harder and harder to prevent Trojan insertion and other malicious design modifications. In this paper, we propose to deploy security invariant as carried proof to prevent and detect Trojans and malicious attacks and to ensure the security of hardware design. Non-interference with down-grading policy is checked for confidentiality. Contrary to existing approaches by type checking, we develop a method to model-check a simple safety property on a composed machine. Down-grading is handled in a better way in model-checking and the effectiveness of our approach is demonstrated on various Verilog benchmarks.

1. INTRODUCTION

Design and manufacture of complex semiconductor circuits and systems requires many steps, and one design could involve hundreds of engineers, typically distributed across multiple locations and organizations worldwide. Moreover, the deployment of IP blocks from different sources has become a common practice. The conventional processes and tools for design and manufacture of semiconductors at most ensure the correctness. However, to date, these processes do not provide confidence about whether the chip is altered such that it provides illegal sensitive information leakage. Such undesirable behavior can be due to a weakness in the design that results in an unintentional side channel or due to maliciously inserted Trojan hardware.

Existing protections from hardware Trojan insertion and side-channel attacks are mostly based on testing or circuit analysis that are passive, ad hoc, and afterthoughts based on individual designs. In this paper, we propose a proactive approach based on a novel design framework called *Invariant-Carrying Machine* that will increase the confidentiality and trustworthiness of hardware. It leverages the current practice of verification in hardware design, and is based on lightweight formal methods that are scalable for large designs.

GLSVLSI '17, May 10-12, 2017, Banff, AB, Canada
© 2017 ACM. ISBN 978-1-4503-4972-7/17/05...5.00
DOI: http://dx.doi.org/10.1145/3060403.3060456

The idea of carrying proof in hardware has been proposed in some recent work on "Proof-Carrying Hardware" [3, 6]. Drzevitzky 's approach only applies on configuration bit stream for FPGA. Even though Love 's approach can handle general designs in HDL, it requests the design being translated into another formal language, and a proof being constructed based on required security properties. Such requests incur extra workloads on designers and the scalability of formal methods could be a concern. Contrarily, ICM will use the native Verilog directly as the hardware model, and only add the inductive invariant in the Verilog description. Similar to the Proof-Carrying Code (PCC) for software security, Invariant-Carrying Machine (ICM) requests that each hardware design carries with it an inductive invariant. The invariant constitutes an assurance of the design, that is, it implies the security property or policy. The invariant must also be inductive.

In this paper, we mainly focus on one of the most important security property, confidentiality. We uses the notion of noninterference as a means to specify and prove hardware confidentiality. Noninterference requests the low confidential output will not be affected by the high confidential input. Based on the allowed amount of information leakage, noninterference can be further categorized into pure noninterference and relaxed noninterference. We develop strategies to handle both types and leverage state of the art formal verification tools to extract the inductive invariant held in state machine of the hardware system that can imply the noninterference property. In the experiment, we verify our approach on six small and median-sized behavioral verilog benchmarks revised from standard design.

2. ICM FRAMEWORK

In this paper, we are going to develop a hardware design framework called ICM (Invariant-Carrying Machine) for hardware assurance. As illustrated in Figure 1, the framework consists of a library of security policies and properties that are agreed among all parties, tools to aid the generation of inductive invariant for a given hardware design, and tools to check the validity of the invariant-carrying machine under the given security properties. If the hardware is an in-house design, the inductive invariant will be maintained through the whole design process and its validity will be checked after each big design step. If the hardware is an IP block, the user needs to check the validity of the inductive invariant before deploying the block. Hardware Trojans will be detected when validity checking fails. The whole approach is based on formal methods and follows the principles of the

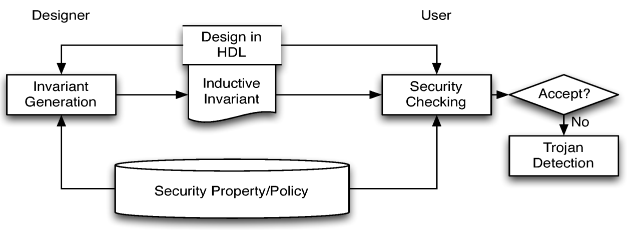

Figure 1: The Process Flow of Invariant-Carrying Machine for Hardware Assurance

famous and successful PCC (Proof-Carrying Code) [7] with modification for hardware design. An inductive invariant is used as the proof of hardware security. One critical observation is that the most expensive part of hardware correctness verification is the computation of an inductive invariant from the target property. With the request of inductivity, the checking of the invariant only requires three validations: the initial states satisfying the invariant, any state satisfying the invariant only going to states satisfying the invariant, and the invariant satisfying the security policy. None of the checks requests state unrolling, thus avoids the state explosion problem.

3. HARDWARE MODELED AS STATE MACHINE

Lamport [4] has shown and advocated that any computing system can be better modeled as a state machine. The behavior of a state machine is given by a subset of state changing sequences, each of which gives the sequence for a specific input. Therefore, we can specify the properties of state machine by temporal logic [4]. In temporal logic, we can use conventional logic to specify facts in one state (as in the initial condition), and use conventional logic with primed and non-primed variables to specify facts in two consecutive states (as in the transition relation).

In practice, it is unrealistic to implement a parser that can translate the hardware description language (HDL) like verilog of a large-scale design into state machine expressed in temporal logic because this would require the parser to handle the logical relation between executions of the complex behavioral hardware description. More specifically, the blocking assignments in always@ blocks happen sequentially as in the software programming language. This means that parser may have to compute the net effect of all the executions within the block to generate next state logic expression for each state variable. The cost and complexity could be very large. Therefore, a more realistic strategy is to borrow the idea of execution control flow from PCC and treat the set of blocking assignments as sequence of state transitions. Instead of generating the real hardware state transition, the clockwise transition, the parser will produce a statement-wise state transition. As a result, the original one-cycle state transition is divided into multiple state transition steps. Thus, it is necessary to introduce an extra state variable to represent and control the execution flow, e.g. PC. In addition, not only all the registers are treated as state variables, all the intermediate wires originally representing the netlists of the circuit may become state variables as well. In this way, each statement of execution can be regarded as one step transition.

The transition encoded with the execution flow actually represents a different hardware system. Due to many intermediate states generated from the revised transition, the property held in the original state machine is no longer an invariant in the state machine with parsed transition relation. It is meaningful to check the property only in the state which represents the end/start of one clock cycle computation in the original state machine because only these states exist in the original system. To make the invariant proof consistent, any original invariant *Inv* should be replaced by a new invariant *NInv* in the following way assuming PC is the state variable representing the execution flow and the state with $PC = 0$ indicates the start/end of the one clock cycle computation (we make this assumption by default in the rest of the paper).

$$\mathit{NInv} \triangleq (PC = 0 \Rightarrow \mathit{Inv})$$

The statement-wise transition relation is obviously much easier to generate since Verilog parser can treat one HDL statement as a step of transition and avoids handling the complex logical relation between multiple executions within a sequential always block. It is important to realize that when there are multiple combinational **always** blocks and **assign** statements, it is necessary to topologically sort all the blocks before translating the design.

4. VERIFICATION OF CONFIDENTIALITY

In this section, we discuss our specification of confidentiality property and our strategies of handling the transition relation parsed from the HDL in order to leverage formal verification methodologies to verify the confidentiality of the given hardware design.

4.1 Noninterference

Only a small subset of hardware requests confidentiality. However, these hardware are probably more important to protect than others. Intuitively, it requests that no secrecy is leaked to the attacker. The first question is how confidentiality can be specified and proved in our logic. A recent breakthrough in security research was the discovery by Clarkson and Schneider [2] that many security properties such as *noninterference* and *average response time* are not properties of a computational trace, but properties of a set of traces, called “hyperproperties”. They have shown that hyperproperties need second-order logic to express, which is out of the scope of conventional first-order logic.

Noninterference, in variety of forms, has been thoroughly studied for confidentiality. Generally speaking, a computing system may have both high and low confidential inputs and outputs, noninterference requests that the low confidential output will not be affected by the high confidential input. Formally, we can use a function $(O_l, O_h) = F(I_l, I_h)$ to represent the computing system, with O_l, O_h representing low and high outputs, and I_l, I_h low and high inputs. Then, noninterference means that for any $(O_l, O_h) = F(I_l, I_h)$ and $(O'_l, O'_h) = F(I'_l, I'_h)$, if $I_l = I'_l$ then $O_l = O'_l$.

Even though the plain noninterference is a security policy that is easy to express, it never has been widely enforced in real systems. The reason is that the plain noninterference is too strict to make a computing system satisfying this property. Even a simple prototype of the login process,

$$\text{IF } H = g \text{ THEN } l = 1 \text{ ELSE } l = 0,$$

where H representing the high confidential password, g the low guess, and l the low output, violates the property. However, in this example, even though the low output l depends on H, unless the domain of H is small, it will be almost impossible to learn H from (g, l) pairs.

One possible way to evaluate the confidentiality level is through quantitative information flow analysis that uses probabilistic analysis to calculate the amount of leaked information. However, such probabilistic analysis is very expensive and heavily depends on the specifics of the target system. Alternatively, we decide to use relaxed noninterference with downgrading policies [5] as the confidentiality policies in our framework. *Downgrading* is used to specify allowed information flow from a high security level to a low security level, which is also called *declassification* for confidentiality. Intuitively, *relaxed noninterference* is the same as noninterference that forbids information flow from high security to low security, with the exception of the given downgrading. A security policy can be modeled as a *downgrading function F* which involves at least one high-security input.

4.2 Self-composition

After settling on the relaxed noninterference with downgrading policies for confidentiality, we need to consider how to specify them and how to prove them. We come to an approach based on *self-composition* [1, 8]. Intuitively, noninterference means that for any computational trace of the system, there must exist another trace where even the high security inputs have different values, the low security behavior is the same. *Self-composition* composes two copies of the original system (with input and variable renaming). Therefore, any property about two possible traces in the original system can be stated as a property about one trace in the composed system.

Formally, a state machine $M = (\bar{X}, \bar{I}, \bar{O}, \mathit{Init}, TR)$ can be described by propositional logic formulas: $\bar{X}$ is all the internal state variables, $\bar{I}$ and $\bar{O}$ are all the inputs and outputs, $\mathit{Init}(\bar{X})$ is the initial condition and $TR(\bar{I}, \bar{X}, \bar{O}, \bar{X}')$ is the transition relation where $\bar{X}'$ represents the next state. Meanwhile, inputs and outputs are categorized by their security sensitivity such that $\bar{I} = \bar{I}^l \vee \bar{I}^h$ and $\bar{O} = \bar{O}^l \vee \bar{O}^h$.

Given M, we can characterize self-composed state machine $M_{comp} = (\bar{X}_{comp}, \bar{I}_{comp}, \bar{O}_{comp}, \mathit{Init}_{comp}, TR_{comp})$ as follows:

$$\bar{X}_{comp} \triangleq \bar{X} \vee \bar{X}_c, \ \bar{O}_{comp} \triangleq \bar{O} \vee \bar{O}_c, \ \bar{I}_{comp} \triangleq \bar{I}_l \vee \bar{I}_h \vee \bar{I}_h^c$$

$$\mathit{Init}_{comp} \triangleq \mathit{Init} \wedge \mathit{Init}_c \bigwedge\nolimits_{x \in X} (x = x_c),$$

$$TR_{comp} \triangleq TR \wedge TR_c$$

The pure noninterference can also be specified by the following propositional logic.

$$\mathit{Inv} \triangleq \bigwedge\nolimits_{o \in O_l} (o = o_c)$$

Then, the proof obligation is to show that every reachable state in the transition state machine M satisfies *Inv*, which is symbolically represented as $M \models \mathit{Inv}$. Now, the noninterference proof problem is reduced to a model checking problem. Ideally, we can take advantage of formal verification tools and apply state-of-the-art model checking algorithm to verify the query $M \models \mathit{Inv}$ and compute the fixed point as our inductive invariant. Note the above self-composition is for the clock-wise transition relations. If execution flow is encoded and TR is statement-wise, TR_{comp} should be revised to support synchronization at the end of one clock cycle executions since one copy may run faster than the other copy. Besides, Init_{comp} and *Inv* should be revised as follows:

$$\mathit{Init}_{comp} \triangleq \mathit{Init} \wedge \mathit{Init}_c \bigwedge\nolimits_{x \in X} (x = x_c) \wedge (PC = PC_c = 0)$$

$$\mathit{Inv} \triangleq (PC = PC_c = 0) \Rightarrow \bigwedge\nolimits_{o \in O_l} (o = o_c)$$

4.3 Transition Relation With Downgrading

The proof obligation presented in Section 4.2 does not take downgrading policy into consideration. Our next step is to show how to revise the composed transition relation to prove the relaxed noninterference of a given design.

Intuitively, relaxed noninterference verification is equivalent to the strong noninterference verification if the sensitive information is never leaked through the downgrading function. Accordingly, our goal is to encode this condition into the state transition relation. This requires the same output value of any corresponding computation blocks in both copies that perform the downgrading function. For the sake of simplicity in demonstration but without loss of generality, we assume only one downgrading function is provided and is performed by at most one execution block only once. We make this assumption default in the rest of the paper. It is trivial to extend the idea to more complex cases.

Firstly, we have the observation that it is relatively straightforward to incorporate the downgrading function into the proof obligation if the transition relation is coarse-grained without execution flow. In order to build the state transition TR_F with downgrading policy F, we only need to revise the transition relation to force the output of the downgrading function of one copy(minor copy) equal to that of the other copy(major copy) instead of minor copy's own downgrading function output. On the other hand, when the transition relation is generated by a parser and has execution control flow, the two copies may not be synchronous during the execution. So it is relatively hard to enforce the same output of the downgrading function from two copies. To solve this problem, we introduce an extra boolean state variable, *flag* to indicate whether the major copy has already performed the downgrading function or not. The transition for *flag* is as follows assuming *dp* and cond^F are the execution point and corresponding branch condition under which the downgrading is performed respectively.

$$\begin{aligned} TR^{\mathit{flag}} \triangleq & \quad PC = 0 \Rightarrow \neg \mathit{flag}' \wedge (PC = \mathit{dp} \wedge \mathit{cond}^F) \Rightarrow \mathit{flag}' \\ \wedge & \quad \neg(PC = 0 \vee PC = \mathit{dp}) \Rightarrow \mathit{flag}' = \mathit{flag} \end{aligned}$$

To construct the new composed transition relation TR^F_{comp} from TR_{comp} to support downgrading function F, the transition logic of the control state variable from the minor copy PC_c should be changed. If the minor copy is about to perform the downgrading, it first needs to check whether the flag is raised, meaning the major copy has already been through the downgrading in the same clock cycle. If the flag is raised, it will take the output value of the downgrading function from the major copy and proceed forward. If the flag is not raised, it will wait until either the flag is raised or the major copy reaches the end of a clock cycle computation. Denote *Fvar* as the variable assigned to the output value of the downgrading function F. Accordingly, new PC_c and *Fvar* transition wil be as follows:

$$\mathit{cond1} \triangleq PC_c = \mathit{dp} \wedge \mathit{cond}_c^F \wedge \neg \mathit{flag} \wedge PC \neq \mathit{maxPC} + 1$$

$$\mathit{cond2} \triangleq PC_c = \mathit{dp} \wedge \mathit{cond}_c^F \wedge \mathit{flag} = \mathit{TRUE}$$

$$PC'_c = \begin{cases} PC_c & \mathit{cond1} \\ \mathit{same\ as\ in\ old}\ TR_{comp} & \mathit{otherwise} \end{cases}$$

$$\mathit{Fvar}'_c = \begin{cases} \mathit{Fvar} & \mathit{cond2} \\ \mathit{same\ as\ in\ old}\ TR_{comp} & \mathit{otherwise} \end{cases}$$

where maxPC is the original last execution point in one clock cycle and $\mathit{maxPC}{+}1$ is the extra execution point to synchronize two copies at the end of a clock cycle.

To summary, the final desired version of transition relation, initial condition and relaxed noninterference property

Table 1: experiment results for both strict and relaxed noninterference verification

benchmarks	# of clauses in inductive invariant		# of sat queries		model checking time (s)	
	strict	relaxed	strict	relaxed	strict	relaxed
updown-cntr.v	46	27	1090	1447	1.56	3.73
LFSR.v	114	44	5513	6728	14.24	68.62
fib.v	148	157	7059	12025	72.2	132.39
multiplier.v	209	218	12431	12891	187.62	199.47
gcdEuclid.v	176	198	9787	10017	104.42	158.05
crc.v	208	242	17875	20242	230.45	257.86

are expressed as:

$$\begin{aligned} TR_{final} &\triangleq TR^{F}_{comp} \wedge TR^{flag} \\ Init_{final} &\triangleq Init_{comp} \wedge \neg flag \wedge (PC = PC_c = 0) \\ Inv_{final} &\triangleq (PC = 0 \wedge PC_c = 0) \Rightarrow \bigwedge\nolimits_{o \in O_l} (o = o_c) \end{aligned}$$

Our approach to enforce the same downgrading function output of two copies can avoid false negative, that is, report secure when the hardware is actually insecure under the given downgrading policy. This guarantees the baseline security checking criterion. However, we can not avoid false positive, that is, report insecure when the hardware is actually secure under the given downgrading policy. In this sense, our approach to verify the relaxed noninterference is over conservative and it is necessary to conduct further security checking when the a result of "insecure" is reported.

5. EXPERIMENT RESULTS

We have implemented a Verilog parser that currently can translate a behavioral specification into a state machine given by its initial condition and transition relation. We leverate the state-of-art model checking tool CTIGAR (Counterexample To Induction-Guided Abstraction-Refinement) [?] to verify and compute the inductive invariant of the confidentiality property from the state machine. CTIGAR is an SMT-based model checker which extracts the inductive invariant by finding counterexamples and incrementally refining the property. CTIGAR is more efficient than many other model checking algorithms because it avoids unnecessary state unrolling.

We verify our approach on 6 small and median-sized behavioral verilog benchmarks revised from standard design. We first have a short description of each of our benchmarks:

- **updown-cntr.v**: 8-bit bidirectional counter and the direction is depending on a 1-bit input.
- **LFSR.v**: 8 bit linear-feedback shift registers.
- **fib.v**: sequentially compute the nth fibonacci number, where n is an 8-bit input.
- **multiplier.v**: sequentially compute the multiplication of two 16-bit inputs.
- **gcdEuclid.v**: compute greatest common divisor between two 16-bit inputs using Euclid Algorithm.
- **crc.v**: serial cyclic redundancy check $x^{16}+x^{15}+x^{2}+x^{0}$.

In all the benchmarks, we define the downgrading function as well as the security level of each input and output. Experimental results are shown in table 1. Compared with small-scale designs like **updown-cntr.v**, larger designs like **crc.v** require more sat queries and longer model checking run time to generate inductive invariants with more clauses. This is because larger designs have more execution blocks and will be translated into longer transition relations.

6. CONCLUSION

In this paper, we have introduced the framework of Invariant Carrying Machine for hardware assurance and investigated one of the most important security policy, confidentiality. We focused on relaxed noninterference with downgrading policies that could be proved by inductive invariant on self-composition of the machine. Experimental results on small- and median-sized designs have demonstrated the effectiveness of our approach.

7. ACKNOWLEDGEMENT

This work is partially supported by NSF under CNS-1651695 and CNS-1441695, and by SRC under 2014-TS-2559.

8. REFERENCES

[1] Barthe, G., D'Argenio, P. R., and Rezk, T. Secure information flow by self-composition. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations* (Washington, DC, USA, 2004), CSFW '04, IEEE Computer Society, pp. 100–.

[2] Clarkson, M. R., and Schneider, F. B. Hyperproperties. *J. Comput. Secur. 18*, 6 (Sept. 2010), 1157–1210.

[3] Drzevitzky, S., Kastens, U., and Platzner, M. Proof-carrying hardware: Towards runtime verification of reconfigurable modules. In *Reconfigurable Computing and FPGAs, 2009. ReConFig '09. International Conference on* (Dec 2009), pp. 189–194.

[4] Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Publishing Company, 2002.

[5] Li, P., and Zdancewic, S. Downgrading policies and relaxed noninterference. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2005), POPL '05, ACM, pp. 158–170.

[6] Love, E., Jin, Y., and Makris, Y. Proof-carrying hardware intellectual property: A pathway to trusted module acquisition. *IEEE Transactions on Information Forensics and Security 7*, 1 (2012), 25–40.

[7] Necula, G. C. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages* (1997).

[8] Terauchi, T., and Aiken, A. Secure information flow as a safety problem. In *Proceedings of the 12th International Conference on Static Analysis* (Berlin, Heidelberg, 2005), SAS'05, Springer-Verlag, pp. 352–367.