Enhancing Security via Provably Trustworthy Hardware Intellectual Property

Eric Love*, Yier Jin†, and Yiorgos Makris*
*Departments of Electrical Engineering and Computer Science, Yale University, New Haven, CT 06520-8267, USA
†Department of Electrical Engineering, Yale University, New Haven, CT 06520-8267, USA

***Abstract*—We introduce a novel hardware intellectual property acquisition protocol, show how it can support the transfer of provably trustworthy modules between hardware IP producers and consumers, and discuss what it might mean for a device to be considered "secure." Specifically, we demonstrate the applicability of previous work in the software field of Proof-Carrying Code (PCC) to the problem of hardware trust and use it to combat the threat of hardware IP-level Trojans. We outline a semantic model representing the constructs permissible in a Verilog hardware description language (HDL) and show how this model can be used to reason about the trustworthiness of circuits represented at the register-transfer level (RTL). A discussion of "security-related properties" reveals how rules for trustworthy operation might be established for a particular design without necessarily specifying exact functionality. We then examine a hypothetical scenario involving a consumer with certain security needs and show how our system could be employed to guarantee that these needs are met by a hardware IP vendor's code.**

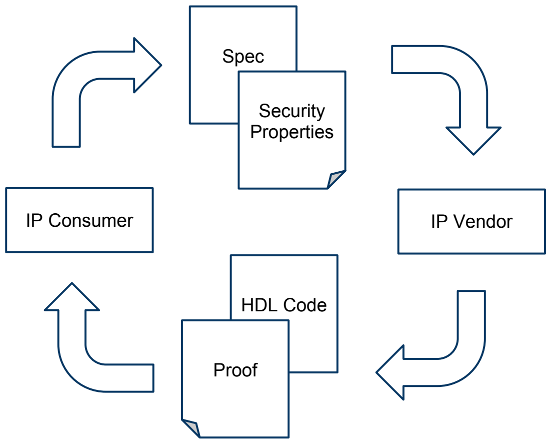

Fig. 1. Module Design and Acquisition Protocol

I. Introduction

The problem of hardware security has grown more important and more difficult with the emergence of an increasingly globalized design process. The tight control manufacturers once exerted over their devices is no longer possible when more complicated systems now employ hardware components from a variety of different suppliers whose trustworthiness is unknown [1], [2]. Researchers have, accordingly, devised techniques to diffuse the threat of malicious circuitry (a.k.a. hardware Trojans) being inserted into the supply chain, relying variously on physical, behavioral, and formal methods [3]–[8].

Our scheme is different from all previous approaches to the problem of hardware Trojans in that it does not concentrate on the physical level of chip layout, but focuses instead on the security of third-party Intellectual Property (IP) modules commonly used in contemporary designs. We imagine an attacker who makes malicious modifications to a module's HDL code in order to introduce the potential for undesired behavior. This module may then be sold for use in a larger system which, with the inclusion of tampered IP, becomes itself vulnerable to attack.

If, however, we can guarantee that certain carefully specified properties hold across the outputs of components from untrusted IP vendors, then we may be able to guard against certain types of undesirable or insecure behavior. These can include disruption of operation, manipulation of signals, or misuse of sensitive data. Each case requires different kinds of properties, but a strong specification can render many modes of attack significantly more difficult to implement. If these safeguards become integrated into the design process, then when an IP consumer asks for some module to be constructed, he will provide the vendor with not only a functional specification, but also a list of specific security-related properties that the desired module must obey. It is then the vendor's task to construct a formal proof demonstrating adherence to these properties. Figure 1 outlines such an interaction.

A similar idea has been proposed for software as Proof-Carrying Code (PCC) [9]. In its original form, PCC required the acceptance of a large, unverified code base at its core. As a solution to this problem, researchers have developed Foundational PCC (FPCC) which uses a universal logic framework to model the semantics of all possible assembly language instructions and is written in the same logical inference language used to write correctness proofs, thereby subjecting the entire system to validation by the proof checker [10]–[12]. Further work has led to the creation of a Certified Assembly Programming language (CAP) [13], [14], upon whose construction and application we model our reformulation of PCC for use with hardware IP.

A parallel concept of Proof-Carrying Hardware (PCH) was first proposed in [15], but the authors showed only that correctness proofs could be generated for FPGA bitstreams in order to provide assurance that the given gate configuration implements a specific boolean logic function and therefore did not allow for true functional variation. Furthermore, their method relied on a SAT solver rather than a formal high-level proof assistant tool, and thus more closely resembles formal verification than PCC. We, however, shift our analysis up to the register-transfer level (RTL), expanding the domain of provable specifications to include more complicated behavioral properties given in a temporal logic, achieving for hardware the same level of flexibility offered by PCC for software.

In our system, proofs are written in the Coq proof assistant language and are therefore easy to validate automatically, allowing the consumer to know very quickly whether or not the HDL code conforms to a given set of security-related properties. Just as with PCC and PCH, the computational burden of verification falls on the IP vendor, not the consumer. The vendor must make a significant investment of time in the construction of a proof, but the consumer's task of verifying it is trivial in comparison.

Our novel contribution is to create a set of definitions in the Coq [16] language (Section IV) that models the behavior of all possible statements in a domain-specific Verilog we specify in Section III. We also describe, in Sections IV and V, a set of rules to automatically generate the Coq representation of any given Verilog module for use in security compliance proofs. We then illustrate the usefulness of our framework by way of a contrived design scenario in Section VI. We present a model consumer with need for a specific component and imagine what sort of security requirements this consumer would have. The example covers formulation of security properties, translation into the temporal logic model we have implemented in Coq, sample HDL implementations, and the construction of proofs.

II. Design Process & Utility of Proof-Carrying Hardware Intellectual Property

If the consumer wishes to order a component from the IP vendor, our design framework requires that he decide upon a set of security properties in addition to the standard functional specification. Both parties must then agree upon a fixed translation of these properties into a formal mathematical codification in the theorem-proving language. As the vendor writes HDL code for the final product he also produces a formal proof as shown in Figure 1. This is not a type of testing procedure, but rather a new stage of the hardware design process to be carried out in addition to standard verification and debugging. Although we will see that the temporal logic used to specify security-related properties does resemble the syntax of many hardware assertion languages, the verification of these properties is not an assertion-based process. It is not at all necessary to test the module in simulation or emulate it on an FPGA to see that the properties are obeyed. Instead, the vendor need only construct a valid formal proof to show that these properties hold under all operational conditions.

This proof, once constructed, becomes a part of the finished package delivered to the IP consumer who, in turn, may then easily check the proof by running it through the Coq language interpreter. If the proof is valid, then he can accept the design, knowing that its operation stays within the functional boundaries set by the security property list. If the consumer is, say, a government or military organization, then he will have a strong reason to negotiate the production of such assurances. But it is also true that the vendor, too, will be able to assure himself that no in-house manipulations of the design have introduced functionality in violation of these safety rules.

It may also be the case that this delivered component will itself be incorporated into a larger system. It may therefore be worthwhile to consider, at the beginning of the design cycle, whether some additional properties may be of use when constructing a similar proof for the larger design into which this sub-component is later integrated.

As a continuation of that idea, we fully expect proofs to eventually be constructed modularly, much in the same fashion as IP cores themselves. As smaller components become embedded in larger systems, so too may the proofs of their respective security properties be used to demonstrate that the higher-level device is also subject to certain constraints in its operation. As some devices become standardized, and general consensus is reached on the sorts of relevant properties, a library of code-proof combinations will slowly be built. This will significantly simplify the task of proof construction while still maintaining the integrity of the framework. Additionally, some design teams may wish to engage a third-party proof-writer to construct a separate correctness proof in a strategy resembling N-version programming.

III. Provable Properties at the RTL-Level: A Verilog Subset for Safe Hardware

Because every statement in a module's HDL code must translate into a corresponding declaration in the theorem-proving language, it is necessary to specify this HDL and describe how such a translation might be carried out. We choose a fully functional subset of the Verilog language as our HDL, which we call Compact Verilog.

This subset has three main components: combinational logic, sequential logic, and module declaration and instantiation. The combinational logic component consists of `assign` statements incorporating any of the standard bitwise logical and conditional operators, as shown in the complete syntactic specification below.

```
<assign-stmt> ::= <variable> "=" <assign-right>
<assign-right> ::= <expression> |
   <expression> "?" <expression> ":" <expression>
<expression> ::= <variable> |
   <expression> "|" <expression> |
   <expression> "&" <expression> |
   "~" <expression> | "(" <expression> ")"
```

In Compact Verilog we support only synchronous sequential circuits. This greatly decreases the complexity of many proofs, but should still be sufficient for a wide variety of applications.

```
<always-block> ::=
   "always @ (posedge clk)" <body>
<non-block-assign> ::= <var> "<=" <expr>
<body> ::= <stmt> |
   "begin" <block> "end"
<block> ::= <stmt> | <stmt> <block>
<stmt> ::= <non-block-assign> ";" |
   "if" <cond> <body> [<elseif>]+ [<else>]
<elseif> ::= "else if" <cond> <body>
<else> ::= "else" <body>
```

```
<expr> ::= "~" <expr> | "(" <expr> ")" |
   <expr> "&" <expr> | <expr> "|" <expr> |
   <expr> "+" <expr> |<expr> "-" <expr>
<cond> ::= <expr> "==" <expr> |
   <expr> "<" <expr> | <expr> "<=" <expr> |
   <expr> ">" <expr> | <expr> ">=" <expr> |
   <cond> "||" <cond> | <cond> "&&" <cond> |
   "!" <cond> | "(" <cond> ")"
```

Within the sequential logic we allow only `if`/`else` statements (with the same logical and control operators as in combinational `assign` statements) and non-blocking assignment statements. As for the declaration of signals themselves, we permit the `wire` and `reg` statements for both single-bit signals and bus lines. We also allow for module instantiations and definitions.

IV. Proof Framework in Coq

Given the HDL specification presented in the previous section, we derive a corresponding set of definitions in the Coq theorem language to model the functionality of circuits at the RT-level. This approach parallels [13]'s formulation of inference rules for the instruction set of CAP.

A. Combinational Logic

We first define a `value` as an inductive set with two constructors, called `lo` and `hi`, and a `signal` as a mapping of time, specified in clock cycles and given as a natural number, onto a `value`:

```
Inductive value := lo | hi.
Definition signal := nat->value.
```

On top of these we build "expressions" consisting of combinational logic and control operations on sets of signals. Also defined as an inductive set, these expressions are essentially equivalent to the parse tree generated by a Verilog compiler, representing logical and arithmetic operations as a network of symbols. These, in turn, are interpreted by the evaluate function `eval` which recursively maps the expression tree onto the values of signals at the specified time. Thus, for instance, the logical AND of two signals causes first one signal to be evaluated, followed by the second only if the first is `hi`. In this way, the `eval` function defines the operational semantics of expressions and is used to model the `assign` statement:

```
Fixpoint eval (e:expr)(t:nat) {struct e} :=
 match e with
  | (econs sig) => (sig t)
  | (and ex1 ex2) => match (eval ex1 t) with
      lo => lo | hi => (eval ex2 t) end
  | (or ex1 ex2) => match (eval ex1 t) with
      hi => hi | lo => (eval ex2 t) end
                  . . .
```

The definition of `eval` provides the proof-writer (the IP vendor) with a sufficiently precise definition of combinational logic functionality to prove useful theorems about the behavior of signals. To prove, for example, that a signal assigned to the logical AND of two other signals is low at a given clock cycle, he need only show that at least one input signal is also low at this time and then "unfold" the definition in Coq to reveal the underlying structural relationship between inputs and outputs.

For each Verilog `assign` we generate a corresponding proposition with the `assign` function we have written in Coq and express that proposition as a `Hypothesis` statement so that the code vendor may refer to it in his proof:

```
Definition assign : signal->expr->Prop :=
 fun (a:signal)(e:expr) =>
    forall (t:nat), (a t) = (eval e t).
```

This yields a proposition that the value of the assigned signal is equal to the value returned by calling `eval` on the expression to the right of the assignment operator, for which it also provides a Coq definition according to the rules outlined above. Assignment statements for bus signals are modeled with separate but analogous functions.

B. Sequential Logic

The fundamental inductive structure used to define sequential logic in Coq is what we have called the `updateblock`. Like the expression definition for combinational logic, `updateblocks` are constructed as trees of operations on signals and bus lines. In this case, the permitted operations are non-blocking assignment (to an expression), and conditional assignment for bus lines, as shown below:

```
Inductive updateblock :=
 | upd : signal->expr->nat->updateblock
 | upd_bus : bus->bus->nat->updateblock
 | upd_bus_cond : expr->bus->bus->
    bus->nat->updateblock
 | upd_bus_add : bus->bus->bus->
    nat->updateblock
 | upd_bus_sub : bus->bus->bus->
    nat->updateblock
 | updcons : updateblock->
    updateblock->updateblock.
```

Every block which appears within a Verilog `always` statement generates a corresponding hypothesis in our Coq model to capture the meaning of non-blocking assignment. As an example, suppose the following assignment is to take place, as a result of some condition, in clock cycle n:

```
x <= x + 1;
```

For this expression, our semantic model would evaluate to a proposition that the value of `x` in cycle $n+1$ is equal to $x+1$ (note that `(x n)` represents the value of x at cycle n and `S n` is the successor function):

```
(x (S n)) = (x n) + 1
```

V. Automatic Proof Validation

Figure 2 shows the procedure for proof checking and module validation by the consumer upon receipt of the product from the vendor. The proof is first stripped of any circuit definitions declared with the `Hypothesis` statement in Coq. These are the generated Verification Hypotheses used to model Verilog code based on the semantic representation described in the previous section. These hypotheses, once declared, admit a proposition as true so that it may be used as a precondition for a proof. They must therefore be deleted at the start of proof checking and then regenerated automatically from the provided HDL code. This is a necessary step because otherwise there is no guarantee that the circuit behavior defined in the IP vendor's proof actually matches that of the coded circuit. We also reject any lines containing the keyword `admit` which tells the Coq interpreter to accept a proposition as true without proof.

The IP consumer recombines this "clean" version of the proof with the regenerated Coq circuit model and the framework of definitions already described. At this point, the entire assemblage of Coq code is given to the interpreter to be checked. The consumer simply executes the Coq interpreter program, and if execution passes through to the end of the proof, then the proof is valid and the code obeys the security-related properties.

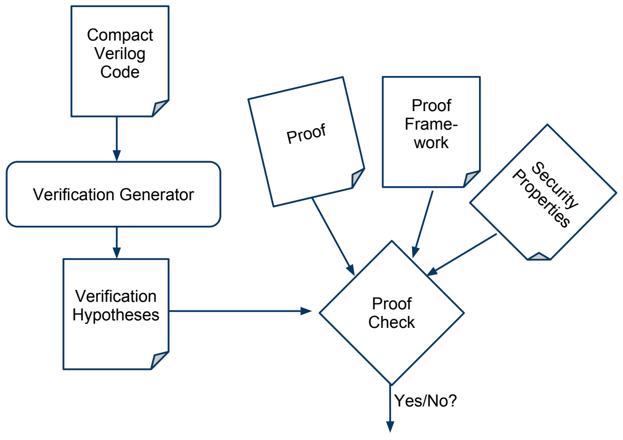

Fig. 2. Automated Verification

VI. Example Design Scenario

In order to demonstrate the capabilities of our proposed methodology, we describe a sample design scenario where an assurance of trustworthiness is desired. By way of this example we will show how intelligently-selected security-related properties can prevent certain kinds of malicious behavior, how these properties are translated into a formal logic, how the vendor of code that conforms to these properties can construct a correctness proof, and, finally, how the consumer can check this proof against the code.

Of particular noteworthiness in this example is the relative freedom granted to the HDL coder in deciding how to implement the desired circuit, following as a consequence of the higher level of sophistication allowed in our property specification model as compared to others. We develop an abstract notion of a "protocol" which delimits a range of acceptable behaviors. This is in contrast to [15]'s proposal for Proof-Carrying Hardware which allowed only for proofs that an FPGA layout implements a specific boolean logic function, requiring a level of specificity that precludes any functional differences between implementations and does not really bring the full potential of software PCC into the hardware domain.

A. Register File Copy Controller

Our example is the following: suppose that the client needs a circuit which controls access to two register-files. Moreover, suppose that this controller is required to have a special mode called "copy" which, when activated by a special flag signal, `CF`, causes the controller to transfer the contents of one register file into the other. The sequence of reads and writes is not important, and neither are the addresses at which any individual value is stored; it is only required that each value in the first register file be copied, unchanged, to some location in the second. The illustration in Figure 3 shows how such a component appears in block form.

A possible application for this module could be in an automatic teller machine (ATM) where it will be used to create and maintain two lists of account numbers for transaction processing. Such a setting provides ample motivation for strengthening security, since a nefarious hardware coder could exploit his control of the circuitry for financial gain or to obtain access to otherwise confidential information.

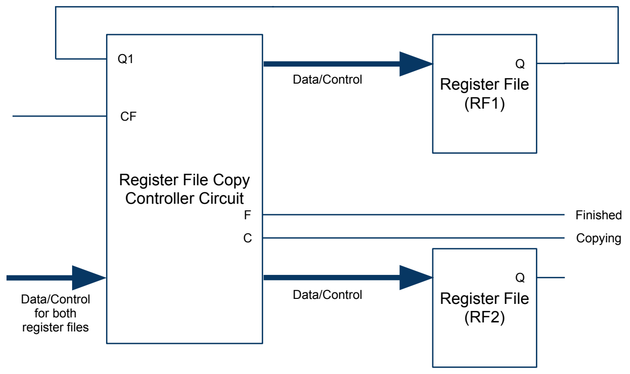

Fig. 3. Register File Controller Module

B. Choosing Security-Related Properties

One can easily imagine several types of behavior a malevolent supplier might introduce into his implementation of this circuit; he could scan incoming data for a specific trigger value to activate a special mode, selectively block certain registers from being copied, enter an infinite loop on yet another trigger, and so on. With carefully-crafted security properties, however, the consumer can successfully safeguard against each of these.

Being aware of these possible modes of attack, the consumer will probably choose a set of properties such as the following: 1.) **Stability**: do not enter copy mode unless the copy flag has been raised, 2.) **Transparency**: when not in copy mode, simply pass control signals through to both RFs, and 3.) **Termination-Transfer**: when the copy flag is raised, enter copy mode, transfer all values, unmodified from RF1 to RF2, and then exit copy mode within a certain predefined number of cycles.

These properties outline the limits of acceptable circuit behavior, and so a proof of compliance with them will guard against the kinds of attacks enumerated in the previous section; any circuit engaging in such behavior clearly breaks the rules.

We will now see how these properties may be translated into a formal mathematical logic. Below is our rendering of the specification into a set of Coq theorems (the bodies of each proof are blank initially–these are to be filled in by the vendor). Stability is easily expressed as a proposition that we must remain outside of copy-mode in all cycles for which the controller is not already in copy mode and for which the copy flag is low:

```
Theorem stable_c : forall t:nat, t > 0 ->
  c t = lo -> cf t = lo -> c (S t) = lo.
```

The definition of Transparency is similarly straightforward in that it simply asserts an equality of the input and output control signals.

```
Theorem transparency : forall t:nat,
  t > 0 -> c t = lo ->
  a1 t = a1_in t /\ d1 t = d1_in t /\
  we1 t = we1_in t /\
  a2 t = a2_in t /\ d2 t = d2_in t /\
  we2 t = we2_in t.
```

The last property exhibits significantly greater complexity, revealing where our framework can be most versatile. We define Termination-Transfer–which contains the bulk of our specification–as a hierarchy of sub-properties. For example, to define the operation of reading from an address a, we create a property called `read` which asserts that the value sent on the address line to RF1 during the stated clock cycle is equal to a. We also pass a variable, X, to capture the value returned from RF1, allowing us to refer to this value when we show that it is written to RF2.

```
Definition read := fun (a n t X : nat) =>
  (a1 (t+n)) = a /\ (q1 (t + n)) = X.
```

The write operation is defined in a similar fashion, stipulating that write-enable is high during cycle $t + n$ and that the value sent on the data line to RF2 is equal to some value X. In defining the complete transfer operation predicate, we link the read and write properties together, asserting that the read X is also the X to be written. But before we can describe the top-level transfer property, we complete the `write` definition by specifying write-uniqueness. Given as the `unique` property, this asserts that a value, once stored, will not be overwritten. That is, there exists no index $nm > n$ in the current copy at which write-enable is high and the same address is sent to RF2.

Finally, `transfer` is defined to indicate a counter index nf by which all possible addresses have been read from RF1 and written to RF2. We make this assertion with a universal quantifier over all addresses a, requiring the existence of some index n such that at time $t + n$ we read the value X from address a and for which there also exists some other index nw at which we write X to RF2:

```
Definition transfer := fun (t nf : nat) =>
  forall a:nat, a <= regs -> exists n:nat,
   n > 0 /\ n < nf /\ exists X:nat,
   (read a n t X) /\ exists nw:nat, nw > 0
   /\ nw < nf /\ (write X nw t nf).
```

In specifying the security properties as we have done with a complicated property-tree, we have paralleled the work presented in [13], which also constructs an elaborate series of quantified predicates in order to define a "valid free list" for a memory allocator. We claim that this structural similarity provides evidence for our framework's unique success in porting the flexibility of PCC into the hardware domain.

C. Proving Security Compliance

To see what might constitute an acceptable implementation of the Register File Copy Controller circuit, we have crafted two security-compliant examples. The first performs the copy operation by sequentially reading register values from RF1, saving them for one clock cycle, and then writing them to the register at the same address in RF2. The second completes this task in reverse order, counting down from the highest register address to the lowest. Due to space constraints, we omit the code for both circuits as it can be inferred from this description and the architecture of the block diagram in Figure 3.

Once the circuit has been coded, the first step of any proof construction is the generation of Verification Hypotheses. These are the Coq propositions described in Section IV and used to represent HDL statements in a formal proof. They are generated automatically from the Compact Verilog code according to the previously outlined rules. Figure 4 shows examples of both combinational and sequential Compact Verilog code (taken from one of our sample implementations) and their corresponding Coq Verification Hypothesis representation.

With this generation having been completed, we may now begin construction of a proof. The first two required properties are trivial, so we will not describe their proofs here. For the more complex Termination-Transfer rule, however, we are forced to adopt a more elaborate plan of attack; just as the property itself was stated as a combination of smaller definitions, so too will the proof be constructed from a set of more primitive lemmas. Although this proof is much too large to be presented in its entirety, we will give a high-level overview of our approach.

The method of induction on clock cycle informs our general technique. Most lemmas rely on a "transition cycle" t which marks the transition into copy-mode, and an index n which counts a certain number of cycles after this transition. Thus, if the transition occurs at time 15, then time 18 could be represented as $t = 15$ and $n = 3$.

At the center of these proofs is the `count_all` lemma, showing inductively that the circuit remains in copy mode until the current write address reaches its highest possible value, and that this write address is always two less than the count index n:

```
Lemma count_all : forall t:nat, t>0 ->
  cf t = hi -> c t = lo -> forall n:nat,
  n < S regs -> cur_read(t + S n) = n /\
  cur_write(t+S(S n)) = n /\
  c (t+S(S n)) = hi /\ cprev (t+S(S n)) = hi.
            . . .
```

With this fact in place, the lemma `read_eq` easily follows, establishing that the current read address remains one less than

```
assign we2 = (cprev & c) ? 1'b1 : (c ? 1'b0 : we2_in);
```

```
                   . . .
Hypothesis assign_we2 : (assign we2
  (cond (and (econs cprev) (econs c))
    (econs Vdd)
    (cond (econs c)
      (econs Gnd)
      (econs we2_in)))).
                   . . .
```

```
             . . .
if (cur_write == 5'b11111 & ~cf) begin
       cur_write <= 5'b00000;
       cur_read <= 5'b00000;
       c <= 1'b0;
       cprev <= 1'b0;
       stored_value <= 32'd0;
end
             . . .
```

```
                           . . .
(ifelse (and (bus_eq cur_write (const regs)) (not (econs cf)))
     (noif (updcons
       (updcons
       (updcons
         (upd_bus cur_read (const 0) t)
         (upd_bus cur_write (const 0) t))
       (updcons
         (upd c (econs Gnd) t)
         (upd cprev (econs Gnd) t)))
       (upd_bus stored_value (const 0) t)))
                           . . .
```

Fig. 4. Top: A Compact Verilog combinational assign statement and corresponding Coq definition. Bottom: Sequential Compact Verilog code and Coq definition.

the write address for the duration of copying. Other lemmas are then constructed on top of these, proving for example that the uniqueness sub-property holds on all writes and that the sequence of operations performed in copy mode is a complete transfer.

It is easy to imagine a parallel proof for the second circuit and, indeed, we have constructed one using the same structure of lemmas. To do this, we simply rewrote most of the lemmas in a manner consistent with the new direction of operation, changing definitions too where appropriate.

VII. Conclusion

While traditional approaches to hardware security have focused on leveraging assertion-based testing and formal verification methods, we have shown that work done by computer science researchers on PCC can be successfully translated to the domain of hardware trustworthiness in order to provide a definitive guarantee that HDL code obeys a set of security-related properties.

By assigning vendors the task of constructing compliance proofs for their hardware IP, we allow consumers to know quickly and easily that the hardware they purchase operates within the parameters they have chosen as provable security properties. With a set of well-formulated and proven properties, the consumer will know that he cannot be the victim of certain varieties of attack, as it will be impossible to prove adherence to the rules for any module that engages in the undesired behavior.

It is not difficult to imagine an extension of our framework for use in other applications beyond the example design scenario presented above. We believe that the current needs of many hardware IP-consuming organizations could be better served with such a framework for provably trustworthy hardware acquisition as an established component of the design cycle. Future work will include the production of an automated verification generator, expansion of Compact Verilog into the full Verilog language, a more thorough analysis of the soundness of our inference rules, and the development of a better behavioral circuit model in the theorem-proving language.

References

[1] Defense Science Board (DSB) study on High Performance Microchip Supply, "http://www.cra.org/govaffairs/images/2005-02-hpms_report_final.pdf," 2005.

[2] S. Adee, "The hunt for the kill switch," *IEEE Spectrum*, vol. 45, no. 5, pp. 34–39, 2008.

[3] 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.

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

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

[6] 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.

[7] M. Potkonjak, A. Nahapetian, M. Nelson, and T. Massey, "Hardware Trojan horse detection using gate-level characterization," in *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, 2009, pp. 688–693.

[8] 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.

[9] 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.

[10] A. W. Appel, "Foundational proof-carrying code," *Foundations of Intrusion Tolerant Systems*, pp. 247–256, 2003.

[11] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni, "A syntactic approach to foundational proof-carrying code," *Journal of Automated Reasoning*, vol. 31, pp. 191–229, 2003.

[12] A. W. Appel and D. McAllester, "An indexed model of recursive types for foundational proof-carrying code," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 5, pp. 657–683, 2001.

[13] D. Yu, N. A. Hamid, and Z. Shao, "Building certified libraries for pcc: Dynamic storage allocation," in *Science of Computer Programming*, 2003, pp. 363–379.

[14] X. Feng, Z. Shao, A. Vaynberg, S Xiang, and Z. Ni, "Modular verification of assembly code with stack-based control abstractions," *SIGPLAN Notes*, vol. 41, no. 6, pp. 401–414, 2006.

[15] S. Drzevitzky, U. Kastens, and M. Platzner, "Proof-carrying

hardware: Towards runtime verification of reconfigurable modules," *International Conference on Reconfigurable Computing and FPGAs*, pp. 189–194, 2009.

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