Provable Information Flows in RTL Circuit Representations

Eric Love
Advisors: Yiorgos Makris and Zhong Shao
May 2, 2011

Abstract

We present a new method of preventing hardware Trojans using a proof-based approach to reasoning about hardware trustworthiness. We build on previous work in this area by introducing a new semantic model of the Verilog hardware description language (HDL) and by incorporating ideas about information flow control into the proof framework. We discuss the prevention of information-leaking Trojans in a DES Encryption circuit as a motivation for the new proof framework.

1 Introduction

Due to an increasingly globalized hardware design and manufacturing process, system integrators rarely have complete control over the devices they produce. As a result, the security of those devices has become much more difficult to ensure. When an IC passes through many hands from specification to fabrication, or an IP core uses code from untrusted third parties, there is ample opportunity for a malevolent engineer to insert undesired functionality into the final product at some stage of the supply chain. These undesired additions and modifications are known as Hardware Trojans [11]. When the target platform is a mission-critical device, such as a missile guidance system, the consequences of unexpected interference can be disastrous. Even in general-purpose computers, a maliciously modified memory management unit can introduce a privilege escalation backdoor that no amount of software security can shut [4].

There has been much interest from the hardware community in diffusing the threat of Trojans [12]. Many techniques have been developed which rely on a “golden IC” model in which it is assumed that some genuine, Trojan-free chip exists against which untrusted copies can be compared. These methods have examined various features such as changes in power consumption caused by inserted circuitry [10, 9] or timing variations due to altered path delays.

More recently, some research has focused on adapting software-based security approaches to the hardware domain. This has been a particularly interesting area of exploration because the increasing prominence of hardware description languages means that more and more devices are constructed in a way that closely resembles the writing of computer programs. Furthermore, the widespread use of FPGAs means that hardware devices are now becoming reconfigurable, so an analogy to software is even more fitting. As such, attempts have been made to make use of research from the computer science community for the purposes of hardware security.

One such concept, known as Proof-Carrying Code (PCC), was proposed in the late-1990s as a means to formally reason about whether compiled machine code conforms to some predefined properties, such as the type safety rules of a functional programming language [8]. In PCC, a compiled binary is paired with a formal proof that it obeys these properties. The idea of using PCC-style techniques in hardware was first proposed by Drzevitzky et al., who in 2009 [1] demonstrated that the trace produced by a SAT solver could be used as a kind of “proof” of combinational equivalence between the combinational logic functions implemented in an FPGA bitstream and their predetermined specifications.

More recent work by Love et al. [6] has taken a first step toward the development of a much more general
framework for proving arbitrarily abstract behavioral properties about HDL circuit representations. The authors proposed a novel acquisition protocol for secure hardware modules written in Verilog wherein an IP consumer and IP vendor agree on a pre-defined set of security-related properties that the delivered module must satisfy. The vendor delivers to the consumer the HDL code for the completed module as well as a formal proof in the Coq language that the module obeys the agreed-upon properties. A framework in Coq for modeling the behavior of synchronous Verilog circuits was presented, and the authors demonstrated its utility in a sample design scenario.

Despite the success of this framework in some applications, the Coq formulation in [6] has many shortcomings that could be improved with a better semantic model of the Verilog HDL. In addition, the temporal logic in which security-related properties could be expressed does not cover all desirable property definitions. We describe one such property in the context of a DES encryption module and show how the proof framework could be expanded to express it.

2 Revised Formal Semantic Model of Verilog

Software PCC methods have traditionally been built around a semantic model of machine instructions in a particular architecture’s assembly language. This was sensible, as the goal of PCC was to reason about what a piece of binary executable code may or may not do. It was possible to consider software as a sequence of operations performed a strict order and to model how each operation updates the system’s state.

In hardware, however, such a model does not work. Aside from the fact that there is no obvious analogue to assembly instructions, we also cannot rely on a single piece of state being updated at each clock cycle. Verilog and other HDLs produce hardware where many computations may happen simultaneously and multiple pieces of state information may be updated at once. In place of memory (stores and heaps) and assignment instructions, we have wires that can be connected together with gates to form circuits (albeit still with a command called assign). Each may be one or more bits wide and can carry a signal that changes over time.

These factors have made development of a proof framework for HDLs more challenging by complicating the semantic model. Previous work in [6] presented one formulation directly in Coq, but this has proven to be inadequate for several reasons. First and foremost, it lacked a satisfactory treatment of modules and module instantiation, relying instead on an external proof management program to decide when proofs may be combined together modularly in larger systems. It also represented wires as either single bits carrying values in \{hi, lo\}, or natural numbers, and did model bus lines as vectors of such bits. For these reasons, we propose a new formal semantics of Verilog.

2.1 Verilog Language Syntax

Figure 1 gives the syntax for our model’s version of the Verilog language. The syntax of variables and expressions is fairly straightforward, and the only subtlety in those cases is that the use of square brackets indicates the selection of either a single bit from a bus, or a subsequence of bits if used with a colon. These only occur in \(R\)-expressions, which are so called because they may only be used on the right hand side of an assignment operator.

We also introduce the \{\} construct to concatenate two or more bus lines together into one. This requires the introduction of the \(P\)-list, which is a list of one or more \(R\)-values separated by commas. We reuse this later for formal parameters in module declaration.

For non-blocking assignment we allow only `posedge clk` to appear in the sensitivity list so that we ensure only synchronous circuits are modeled (i.e. register values may only be updated once every clock cycle).

For module declaration we use \(P\)-lists to declare formal parameters. For instantiation, we also give the ability to specify formal parameters by name when passing actual parameters. This is called a \(\gamma\)-list.

This syntax is not a full specification of the Verilog language, and some obvious constructs, such as `case` are missing. This is equivalent to if-statements, so we have omitted it, along with XOR, and other constructs easily representable with the given “abstract” syntax (we still specify some concrete syntax in order to show
what Verilog code should look like). We do not support initially statements, delays, or module parameterization.

The last item listed in the syntax is commands. These are introduced as an abstraction to refer to any one of various kinds of syntactic elements. We use them later in our discussion of semantics.

### 2.2 Types and Environments

We now give typing rules for Verilog. Our type system is built on top of the simply-typed λ-calculus, for which we assume the standard rules hold. We define a type value to represent binary values that can be either hi or lo. Other basic types include $\mathbb{N}$, the natural numbers, and $\mathbb{N}_n$, the natural numbers up to a constant.

We also define our type system to allow polytypes consisting of kinds, functions from one polytype to another, and types with universal and existential quantifiers over the kinds.

On top of these we then define some types to represent circuit elements. The bvec type models vectors of values by representing them as functions from bit index (type $\mathbb{N}_n$) to value. It is parametrized on an $n$ which specifies the length of the vector in bits. We define it as a function of time (a natural number in $\mathbb{N}$) onto bvecs of the same length, specified by parameter $n$. The abstract type wire is used to model both wire and reg elements.

### 2.3 Type Inference Rules

Inside of a module, every identifier refers to an object of some type. Expressions, too, have types, and we may derive these types for some module context $\Theta$ using the
set of inference rules in Figure 3. Outside of a module, we use the base context to derive that a module name is itself a type environment. This is the most basic of the type inference rules.

The other typing rules in Figure 3 cover basic expression types. The WireN rule says that if an identifier V names an input/output/reg/wire in a module’s declarations section, and is declared to be n bits wide, then we may infer that V has type wire n in that module. The Wire rule says the same thing for cases when there is no bit-width, so we assume only a single bit is declared.

In R-expressions, we can extract some subsequence of bits from a wire. Rule RExp2 tells us that the expression corresponding to this subsequence has a wire type that is as wide as the difference between the start bit and end bit of the sequence, while RExp1 handles the case where only one bit is extracted.

In RElop we define the type for an expression applying a binary operator to two subexpressions to be a wire whose width is equal to that of the smaller subexpression. We also have rules PConcat and EConcat to define types for wires concatenated with {}.

2.4 Expression Semantics

Once a term’s type has been established, we want to define its semantics. We do this first inductively over the syntactic E category, and then in a later section we will define how values are actually assigned to wires with assign, always, and module instantiation.

At the center of our semantic model is the operator \( \lbrack \theta \triangleright E : \text{wire } n \rbrack \). This operator takes a piece of syntax E for which the typing judgement \( \theta \vdash E : \text{wire } n \) holds and returns its meaning in the typing context (module) \( \theta \). Of course, this is not a direct map to signals or values, since the meaning of E in \( \theta \) depends on possibly all other elements of \( \theta \).

In the formal semantics community [2], this solved by simply introducing the concept of a \( \theta \)-environment called \( \rho \), which is a function from identifiers in \( \theta \) to values and which is passed to \( \lbrack \theta \triangleright E : \text{wire } n \rbrack \) as an argument. Thus we would consider \( \rho \) to be a \( \theta \)-environment if for all \( x \in \theta \), \( \rho(x) = z \) and \( \lbrack \theta(x) \rbrack \rho = z \).

However, in the case of Verilog, this point is slightly more subtle. Not only do the values of expressions in \( \theta \) depend on all wires in that environment, they potentially depend on the values those wires carried over the entire history of clock cycles \( t \). To solve this problem, we treat \( \rho \) as a function from identifiers V in \( \theta \) to signals, which are themselves functions from time to bits: \( \mathbb{N} \rightarrow \text{bvec } n \) for some \( n \). We thus have the following definition:

**Definition 1** \( \theta \)-environment \( \rho \) is a function of type \( \text{Id} \rightarrow \forall n. \text{signal } n \) such that for all \( V : \text{ld}, \lbrack \theta \triangleright V : \text{wire } n \rbrack \rho = \rho(V) : \text{signal } n \).

More intuitively, we say that if identifier V has type wire n in context \( \theta \), then \( \rho \) should associate it with a signal of width n.

We can then use this definition to define expression semantics for operators on these identifiers. In order to do so we also define a few helper functions on bvec that define how AND, OR, NOT, etc. depend on individual bit values. Figure 5 defines these functions.

For arithmetic operations, we are faced with a somewhat more delicate situation. We define in Figure 6 a function \( \text{nat}_n(v) \) which creates a natural number representation of a bvec n on which the normal definitions of + and - hold. This is admittedly a bit of a hack, and any proof of soundness of the proof rules would have to show that this definition is valid. This we defer to future work.

We also note, however, that systems which define custom adder circuitry (e.g. create a very fast Ling/Kogge-Stone adder to accelerate computation) may then prove their own arithmetic circuitry is equivalent to the definition in our model. Doing so may be useful if a proof specification requires the computation of \( e_1 + e_2 \), but the system designer wishes to use his own adder implementation instead of leaving this choice to the synthesis tool. The same applies to other arithmetic operations and corresponding circuits.

With these functions in place, we can now inductively define the expression semantics operator over E trees as shown in Figure 4. However, one important expression subtype, variables V that stand for bare wires and registers, is missing from the evaluation semantics. In order to give a complete evaluation semantics this type, we will need to explain module instantiation, non-blocking assignment (register update), and combinational assign statements. The definition can be split
$$M \equiv \langle \text{module}(P) \ D \ldots \ \text{endmodule} \rangle$$

$$D = \langle \ldots \ [IO] \ (\text{reg}[wire]) \ V[(n+1):0]; \ldots \rangle$$

$$\vdash M : \text{env}$$

$$M \vdash V : \text{wire} \ n$$

$$\Theta \vdash V : \text{wire} \ n \quad E = V$$

$$\Theta \vdash E : \text{wire} \ n$$

$$\Theta \vdash V : \text{wire} \ n \quad R = V[n_2:n_1]$$

$$n_1 < n_2 \leq n$$

$$\Theta \vdash R : \text{wire} \ (n_2-n_1)$$

$$\Theta \vdash R : \text{wire} \ n \quad \Theta \vdash P : \text{wire} \ n'$$

$$\Theta \vdash P, R : \text{wire} \ (n+n')$$

$$\Theta \vdash E_1 \& E_2 : \text{wire} \ n \quad \rho \ t =$$

and($$
\Theta \vdash E_1 : \text{wire} \ n \quad \rho \ t,$$
$$\Theta \vdash E_2 : \text{wire} \ n \quad \rho \ t)$$

$$\Theta \vdash E_1 | E_2 : \text{wire} \ n \quad \rho \ t =$$

or($$
\Theta \vdash E_1 : \text{wire} \ n \quad \rho \ t,$$
$$\Theta \vdash E_2 : \text{wire} \ n \quad \rho \ t)$$

$$\Theta \vdash \sim E : \text{wire} \ n \quad \rho \ t =$$

not($$
\Theta \vdash E : \text{wire} \ n \quad \rho \ t)$$

$$\Theta \vdash E_1 == E_2 : \text{wire} \ n \quad \rho \ t =$$

eq($$
\Theta \vdash E_1 : \text{wire} \ n \quad \rho \ t, $$
$$\Theta \vdash E_2 : \text{wire} \ n \quad \rho \ t)$$

$$\Theta \vdash E ? E_1 : E_2 : \text{wire} \ n \quad \rho \ t =$$

cond($$
\Theta \vdash E : \text{wire} \ n \quad \rho \ t,$$
$$\Theta \vdash E_1 : \text{wire} \ n \quad \rho \ t,$$
$$\Theta \vdash E_2 : \text{wire} \ n \quad \rho \ t$$)
2.5 Module Instantiation

To handle module instantiation we define a substitution operator \( [d \mapsto \forall_n \text{signal } n] \) on environments \( \rho \):

\[
[x \mapsto y] \rho = \lambda z : \text{ld.} \begin{cases} 
  y & \text{if } z = x \\
  \rho(z) & \text{if } z \neq x
\end{cases}
\]

On top of this we can define a function to recursively build a new environment \( \rho' \) by evaluating the actual parameters of a module instance’s parameter list in enclosing environment \( \Theta \) and setting the formal parameters to map to the resulting signals in the new environment \( \rho' \). Following the possible instantiation syntax patterns:

\[
\text{makeEnv} (\Theta, \rho, (R_1 = V_1[\ldots], P_1), (R_2, P_2)) = [V_1 \mapsto [\Theta \triangleright R_2 : \text{wire } n] \rho t] \text{makeEnv}(\Theta, \rho, P_1, P_2)
\]

\[
\text{makeEnv} (\Theta, \rho, (R_1 = V_1[\ldots], P_1), R_2) = [V_1 \mapsto [\Theta \triangleright R_2 : \text{wire } n] \rho t]
\]

\[
\text{makeEnv} (\Theta, \rho, (V(R), \gamma)) = [V \mapsto [\Theta \triangleright R : \text{wire } n] \rho t] \text{makeEnv}(\Theta, \rho, \gamma)
\]

The evaluation of a wire \( V \) can then be evaluated as described in the previous section.

2.6 Sequential Logic: Semantics of Non-Blocking Statements

Suppose there is a test \( V \in \text{reg}(\Theta) \) to decide if identifier \( V \) belongs to module environment \( \Theta \)'s registers. If so, some method must be used to determine how \( [\Theta \triangleright V : \text{wire } n] \rho t \)'s value changes over time. This is completely determined by the contents of the module's \texttt{always @ (posedge clk)} statements, which selectively assign \( V \) new values at each clock cycle depending on some set of conditional expressions \( E \) in \texttt{if} statements. We therefore introduce the evaluation function \( [\Theta, S(\Theta) \triangleright V : \text{wire } n] \rho t \) which performs a conditional evaluation of \( V \) given a module context \( \Theta \) whose \texttt{always} block is \( S \).

\[
\text{nat}_n : \text{bvec } n \rightarrow \mathbb{N} \\
\text{nat}_n(v) = \sum_{i=1}^{n} v(i)2^{i-1} \\
\text{bin}_n(v) = \text{nat}_n^{-1}(v) \\
\text{plus} : \text{bvec } n \rightarrow \text{bvec } n \\
\text{plus}(v_1, v_2) = \text{bin}_n(\text{nat}_n(v_1) + \text{nat}_n(v_2)) \\
\text{minus} : \text{bvec } n \rightarrow \text{bvec } n \\
\text{minus}(v_1, v_2) = \text{bin}_n(\text{nat}_n(v_1) - \text{nat}_n(v_2))
\]

Figure 6: Arithmetic Operations
Figure 5: AND, OR, NOT, etc. Helper Functions for bvec s.

These rules give the meaning of a single if...else block containing potentially many sub-blocks and sequences of non-blocking assignments. The first case simply says that a lone non-blocking assignment implies that \( V' \) has the value \( E \) had in the previous cycle if it is the same variable \( V \) that appeared to the left of \(<=\). If \( V \neq V' \), then \( (V <= E;) \) preserves the value of \( V \) from one cycle to the next. Case (2) handles sequences of multiple non-blocking assignment statements using a similar test pattern, and Case (3) decides which branch of an if...else block should influence the update depending on the value of the conditional.

The case \( (S S') \) is somewhat more complicated because registers that \( S' \) does not update only retain their previous values if \( S \) also does not update them. This problem may be solved by introducing a predicate \( upd(\Theta, \rho, t, S, V) \) which is true iff \( S \) assigns \( V \) a new
value at time \( t - 1 \):

\[
\text{upd}(\Theta, \rho, (V \leq E), V') = \text{True} \text{ iff } (V = V')
\]

\[
\text{upd}(\Theta, \rho, (S(V \leq E)), V') = (V = V') \lor \text{upd}(\Theta, \rho, S, V')
\]

\[
\text{upd}(\Theta, \rho, (\text{if}(E) S_1 \text{ else } S_2), V') =
\begin{align*}
& (T(\Theta, E, \rho, t) \land \text{upd}(\Theta, \rho, S_1)) \lor \\
& (\neg T(\Theta, E, \rho, t) \land \text{upd}(\Theta, \rho, S_2))
\end{align*}
\]

\[
\text{upd}(\Theta, \rho, (S_1 S_2, V) = \\
\text{upd}(\Theta, \rho, S_1, V) \lor \text{upd}(\Theta, \rho, S_2, V)
\]

We can then add a fourth case to the sequential update evaluation operator:

\[
[\Theta, (S_1 S_2) \triangleright V : \text{wire } n] \rho t =
\begin{cases}
[\Theta, S_2 \triangleright V : \text{wire } n] \rho t & \text{if } \text{upd}(\Theta, \rho, S_2, V) t \\
[\Theta, S_1 \triangleright V : \text{wire } n] \rho t & \text{otherwise}
\end{cases}
\]

This definition of the semantic evaluation operator has implicitly relied on the existence of an assertion logic built on the primitive Prop type. In all such cases where the result of the evaluation operator depends on a conditional expression, there will necessarily be an undesirable intersection of evaluation with proof writing. This is because the value of the conditional proposition on which the operator depends can only be determined via a proof that the proposition is True or False.

In case (4) of the non-blocking semantics, for example, the expression \([\Theta, (S_1 S_2) \triangleright V : \text{wire } n] \rho t\) can only be simplified to \([\Theta, S_2 \triangleright V : \text{wire } n] \rho t\) by proving that the proposition \(\text{upd}(\Theta, \rho, S_2, V) t\) is True. However, this is not a terrible flaw, since the entire motivation for defining \([\Theta, \triangleright : \text{wire } n] \rho t\) is to use it in a proof. In the next section, we give a more explicit description of this assertion logic.

### 3 Assertion Language for Property Specification

Having established a model for the semantics of Verilog, we would like now to describe what kinds of properties this model can be used to prove. To that aim, we describe an assertion language on Verilog semantics as (4) shown in Figure 7.

Our assertion language is basically equivalent to predicate logic with an additional operator \([E]_n\), to reference the \(\text{bvec}\) returned by an expression \(E\) at time \(n\). In this formulation, we implicitly quantify over all possible environments \(\rho\) in the assertion language.

This assertion language is certainly powerful enough to express all the temporal logic specifications used by Love, Jin, and Makris [6] in their register-file copy controller example. For this they showed inductively that a controller module performed a sequence of read and write operations that transferred the contents of one memory bank to another. However, it is not clear that other useful properties could be expressed in either their logic, or our assertion language. The next subsection considers one kind of property that might be difficult to express.

#### 3.1 Motivating Example: A DES Encryption Core

It has been shown [3] that hardware Trojans in cryptographic circuits could pose a threat by leaking sensitive information such as the key or plaintext. Consider the DES Encryption Core in Figure 8. The path in red indicates a potential short-circuiting of the encryption function in the middle. One can imagine a simple MUX (i.e., a Verilog conditional statement) that looks for a specific trigger value on one of the internal signals—one
that is unlikely to show up in a testbench—and then leaks the plaintext directly to the output:

```verilog
assign desOut = trigger_cond ? DesIn : FP;
```

It is not easy to reason about the existence of such paths given the current logic. This is because the relevant security property is structural rather than behavioral. The rest of this paper explores a technique to express these kinds of desirable security properties.

4 Decentralized Information Flow Control

This section describes a method from the computer science community, Decentralized Information Flow Control (DIFC), for reasoning about information flow in software systems. We first examine the origins of DIFC and then discuss a specific implementation in the Flume operating system. Finally, the next section describes how these ideas can be used to enhance hardware security and how information flow guarantees can be integrated into the Verilog proof framework described above.

4.1 Myers/Liskov DIFC Model for Software

Much in the way that PCC was devised in the mid-90s to reason about the safety of code from a potentially untrusted source, the idea of DIFC was proposed as a response to an increasingly networked world in which “programs containing untrusted code are the rule rather than the exception,” as Andrew Myers [7] so succinctly explained it.

He argued that in such a setting it is often desirable for one application to work with data from many different sources that mutually distrust each other. For example, he imagines a hypothetical application called WebTax that takes financial data from a user, “Bob”, and uses data from a secret proprietary database owned by the “Preparer” to produce completed a tax return form for Bob. Bob does not trust the program not to release his confidential financial information to the Preparer, but the Preparer also does not want its secret database of tax optimization rules to be given to Bob. Nevertheless, useful computations need to be performed using data from both Bob and the tax database simultaneously. The WebTax application should be able to accept Bob’s data, solicit responses for appropriate queries to the database, compute a tax return and release the resulting information to Bob.

In order to ensure that data from one source is not accidentally sent to another without explicit permission, Myers introduced a notation for representing multiple principals who each authorize a limited set of other principals to read their data, but not necessarily release it. They do so by applying policies to data listing which other principals they authorize to read it. A policy is a principal followed by a set of readers, and might look like one of these examples:

- $o_1 : r_1, r_2, r_3$
- Bob : Bob
- Preparer : Preparer

This second policy says that Bob authorizes only himself to read his data, and no one else may see it. The third policy is what the Preparer might apply to the rules in the database. When the WebTax application contains Bob’s data, his policy prevents it from releasing it to the Preparer, just as the Preparer’s policy prevents it from revealing its secret database to Bob.

When WebTax processes Bob’s financial data using rules from the Preparer’s database, the resulting data it computes contains information from both Bob and the Preparer. Thus that data should be subject to both policies. To keep track of which policies apply to any piece of data, the Myers DIFC model uses the concept of a label, which is a list of policies in form:

$$L = \{o_1 : r_1, r_2, r_3; o_2 : r_2\}$$

Since principals $o_1$ and $o_2$ specify different lists of readers, data labeled with $L$ may only be read by those readers common to both lists. In this case, data with label $L$ may only be read by $r_2$. When WebTax uses data labeled by Bob as $\{Bob : Bob\}$ and the Preparer as $\{Preparer : Preparer\}$, then the results it computes should be labeled by the join or union of these two:

$$\{Bob : Bob\} \sqcup \{Preparer : Preparer\} = \{Bob : Bob; Preparer : Preparer\}$$
These labels form a lattice structure where \( L_1 \sqsubseteq L_2 \) is true for two labels when every policy \( I \) in \( L_1 \) is less restrictive than some policy \( J \) in \( L_2 \). Policy \( I \) is less restrictive than \( J \) (\( I \sqsubseteq J \)) if every reader in \( J \) is also in \( I \).

The intermediate computation results that are not allowed to be seen by either Bob or the Preparer are checked by a program that the Preparer has authorized to determine whether they should be declassified. When the declassifier program determines that it is safe, the resulting data has only Bob’s label, so it can be sent back to Bob.

So far we have seen labels that list a set of authorized readers. These are called secrecy labels, since they represent the extent to which data must be kept secret. But there is also another kind of label called an integrity label that is used to state which principals certify that the data in question should be trusted. This may be thought of as a kind of guarantee on the maximum number of sources that have somehow influenced some information. When data with different integrity labels is combined, the resulting data takes the intersection, rather than the union, of the integrity policies of the two sources. This concept also features prominently in [7].

Finally, Myers concretized some of this theory by extending the Java programming language syntax to create a language called Jif (Java Information Flow) that allows objects to have labels.

### 4.2 A Tag/Capability-Based DIFC Model in the Flume OS

DIFC has also been implemented in more concretely practical settings, such as operating systems. The Flume OS [5] uses a scheme similar to the Myers/Liskov DIFC model to enforce information flow policies between OS processes that may communicate with each other. By enforcing DIFC at the OS level, Flume allows policies to be maintained even if some process is compromised or contains a security flaw. If the OS is running a web server consisting of multiple processes serving different clients, the server can rely on the OS to keep data from different clients separate.

Instead of a principals-and-readers model, Flume uses
tags and capabilities. A tag is simply a marker that indicates membership in some security class. For example, a Unix user could label all his files with a tag he created. A capability represents the permission to read, write, or declassify data. For every tag \( t \), there exist two capabilities \( t^+ \) and \( t^- \). A process with the \( t^+ \) capability can “taint” itself with the tag \( t \) to read secret information with tag \( t \). Only a process with the \( t^- \) capability can remove the \( t \) from its taint once it has read information with \( t \).

For each process, the OS maintains a list of tags called a label (in reference to Myers/Liskov). An API call exists by which a process can request to add a tag to or remove a tag from its label according to its set of capabilities. A partial ordering \( \sqsubseteq \) exists on labels.

Processes have also set of endpoints, such as file handles, network sockets, and IPC structures, from which they can read or to which they can write. Each endpoint also has a label associated with it, and a process may only change its label if it is “safe” to do so given the labels of the process’s current endpoints. Information cannot be read from a secret source by a process with an endpoint to a public network socket, for instance.

5 Adding Flow-Control to the Verilog Model

This section describes how the tag-capability model of Flume applies to hardware security. In the DES encryption example, we wanted to prevent the plaintext from bypassing the actual encryption operations and appearing directly at the output. A tag-based model will help us reason about this property because we can mark sensitive plaintext with tags but only give the declassification capability to certain internal modules that perform various parts of the encryption functionality. In this way we can make it very difficult for plaintext to take a shortcut to the output. Of course, the model we present here applies to a broader class of problems, including those where integrity guarantees are the most useful kind.

<table>
<thead>
<tr>
<th>Primitive Types</th>
<th>( \tau \in \text{Tag} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Labels</td>
<td>( \ell ::= \text{Tag} \to \text{Prop} \mid \top \mid \bot \mid \ell \sqcup \ell \mid \ell \sqcap \ell \mid \ell - \ell \mid {\tau}_L \mid L_S(\Theta,[E]_t) \mid L_I(\Theta,[E]_t) )</td>
</tr>
<tr>
<td>Assertions</td>
<td>( \mathcal{A} ::= \ldots \mid \ell = \ell \mid \ell \subseteq \ell \mid \tau = \tau )</td>
</tr>
</tbody>
</table>

Figure 9: Assertion Language Extensions for Information Labels

5.1 Formulation Based on Flume

We now describe an information flow control policy for Verilog circuits based on the tag-capability model used in Flume. Our formulation assigns secrecy and integrity labels to \texttt{bvec}s, as opposed to \texttt{wire}s or \texttt{signals}. This decision allows us to reason about information flow in an extremely fine-grained way. The label of a \texttt{signal} can change from one clock cycle to the next, depending on the sources of information contributing to its individual \texttt{bvec} values. We define a set of inference rules for labels just as we did for types. However, the arguments to each rule will depend on the semantic evaluation of the terms they contain. This is necessary in order to achieve fine granularity.

We verify information flow by adding labels to the assertion language. In this way we allow for security property specifications that stipulate what kind of label the information coming out of a module may contain. In the DES case, we would make an assertion that the label of output \texttt{DesOut} contains no tags. Unlike other DIFC implementations where labels are enforced by a compiler or a runtime environment, ours is unique because we enforce labeling restrictions in security proofs.

Just as in Flume, we define a label as a set of tags \( \ell = \{\tau_1, \ldots, \tau_n\} \). Formally, however, we model a label a special variety of assertion-language predicate having type \( \ell : \text{Tag} \to \text{Prop} \). The function \( \ell(\tau) = \text{True} \) iff \( \tau \in \{\tau_1, \tau_2, \ldots, \tau_n\} \). The special labels \( \top \) and \( \bot \) represent the labels with all tags and no tags, respectively. That is, for all tags \( t \), \( \top(\tau) = \text{True} \) and \( \bot(\tau) = \text{False} \).
In the previous subsection we introduced a function \( \ell \).

5.2 New Information Label Inference

The only noteworthy aspect of this figure is the use of functions \( \ell \) and \( \ell' \) to define secrecy and integrity labels for conditional expressions. These both use a \( \lambda \) abstraction to select a different set of tags depending on the value of the condition. In both cases, we take the union or intersection with the condition’s label in order to account for implicit information flow. Just as a more complicated definition was necessary to account for the semantics of an identifier expression \( V \), so too will it be necessary to provide separate labeling rules for values carried by \( V \) depending on whether it is a simple wire, a register, or a parameter to a module instance. In this case, we can simply reuse the existing semantic operator definition by inserting labels:

\[
L_S ([\Theta \triangleright V : \text{wire } n] \rho t) = \begin{cases} 
L_S(\Theta, [V]_t) & \text{if } V \in \text{inputs}(\Theta) \\
L_S(\Theta, [E]_t) & \text{if } \text{assign}(\Theta, V, E) \\
L_S^e(\Theta, S(\Theta), [V]_t) & \text{if } V \in \text{regs}(\Theta) \\
L_S^f(\Theta, P', [V]_t) & \text{if } \text{inst}(\Theta, \Theta', P, P') 
\end{cases}
\]

The first two cases are relatively straightforward. If \( V \) is a module input, then under a proper labeling scheme there should be a function \( L_S(\Theta, [V]_t) \) from module context \( \Theta \) that gives the label of \( V \) at different times. Alternatively, there will be an assertion \( A \) that can be used in a proof to specify this value. (As with any set of proof premise assertions, care must be taken to ensure that \( L_S(\Theta, [V]_t) \) is not specified twice under two contradictory assertions, as this would make the proof logic unsound.) The second rule similarly asserts that the label of some \( V \) appearing in an \texttt{assign} should be the label of the expression to which it assigned.

The third case requires the introduction of a new function \( L_S^e \) to evaluate the label of a register. We can give its definition over \texttt{always} blocks \( S \) recursively as shown in Figure 12. Note that in case (3), the label of the updated register \( V \) is combined with the label of the conditional \( E \) since \( V \) has “seen” at least one bit of \( E \). This is another example of implicit information flow as Myers and Liskov defined it. An analogous function \( L_S^f \) is defined for integrity labels by replacing \( \sqcup \) with \( \sqcap \) in the above formulas.

We define a new function \( L_S^f \) which represents the secrecy label of module instance outputs using a derived context \( \Theta \triangleright \Theta' \). This context is equivalent to \( \Theta \) in every way except that it labels its inputs with the labels given to the actual parameters by the enclosing context \( \Theta \). We thus have an assertion that may be used as a proof rule:

\[
V \in \text{inputs}(\Theta') \\
L_S(\Theta \triangleright \Theta', [V]_t) = L_S(\Theta, [V]_t)
\]
\begin{align*}
L_S(\Theta, [E_1 \text{ op } E_2]_t) &= L_S(\Theta, [E_1]_t) \cup L_S(\Theta, [E_2]_t) \\
L_S(\Theta, [E?E_1 : E_2]_t) &= L_S(\Theta, [E]_t) \cup L_S(\Theta, [E_1 \text{ op } E_2]_t) \\
L_S(\Theta, [E]_t) &= L_S(\Theta, [E_1]_t) \cap L_S(\Theta, [E_2]_t) \\
\ell^I_1(\Theta, E, E_1, E_2) &= \lambda \tau : \text{Tag.} (L_I(\Theta, [E_1]_t) \land \Vert E \Vert = \text{hi}) \\
\ell^I_2(\Theta, E, E_1, E_2) &= \lambda \tau : \text{Tag.} (L_I(\Theta, [E_1]_t) \land \Vert E \Vert = \text{lo}) \\
\ell^S_1(\Theta, E, E_1, E_2) &= \lambda \tau : \text{Tag.} (L_S(\Theta, [E_1]_t) \land \Vert E \Vert = \text{hi}) \\
\ell^S_2(\Theta, E, E_1, E_2) &= \lambda \tau : \text{Tag.} (L_S(\Theta, [E_1]_t) \land \Vert E \Vert = \text{lo})
\end{align*}

Figure 11: Label Semantics over Expression Trees (Subscript \(n\) represents clock cycle)

Analogous assertions also exist in the proof framework to show that all the other properties of \(\Theta \gg \Theta'\) are the same as those of \(\Theta\).

The function \(L^0_S\) also \textit{subtracts} any tags which the instantiated module is allowed to \textit{declassify}:

\[ L^0_S(\Theta \gg \Theta', P, [V]_t) = L_S(\Theta \gg \Theta', [V]_t) - \text{declassifiers}(\Theta') \]

Here \textit{declassifiers}(\(\Theta\)) is a set of tags that the module with context \(\Theta'\) is allowed to declassify. This captures the idea that tags can be removed from the label of some piece of data if that data passes through a module in possession of the right to make that data public. Generally, this “right” is determined by an assertion predicate that grants \(\Theta\) permission to declassify a tag \(\tau\) if it satisfies some property \(A\).

As before, a parallel rule exists for the integrity labels of instantiated modules where \(-\text{declassifiers}(\Theta')\) is replaced by \(\uplus \text{certifiers}(\Theta')\). In the case of integrity labels, we grant some modules the right to \textit{ certify} data by adding a tag \(\tau\) to its integrity list.

\section{Conclusion}

We have seen some of the history of Trojan detection and prevention, and examined how DIFC came about and why it is applicable to proof-carrying hardware. Much work remains to be done, however, since this new proof framework has yet to be practically demonstrated like the Coq formulation in [6]. Furthermore, much has yet to be said on the matter of choosing the correct formal properties for specific circuits, and on lessening the burden of manual proof construction. A semantic
model that does not limit circuits to synchronous designs would also be desirable.

References


