Secure Information Flow Analysis for Hardware Design: Using the Right Abstraction for the Job

Xun Li
Department of Computer Science
University of California, Santa Barbara
xun@cs.ucsb.edu

Mohit Tiwari
Department of Computer Science
University of California, Santa Barbara
tiwari@cs.ucsb.edu

Ben Hardekopf
Department of Computer Science
University of California, Santa Barbara
benh@cs.ucsb.edu

Timothy Sherwood
Department of Computer Science
University of California, Santa Barbara
sherwood@cs.ucsb.edu

Frederic T Chong
Department of Computer Science
University of California, Santa Barbara
chong@cs.ucsb.edu

ABSTRACT

Hardware designers need to precisely analyze high-level descriptions for illegal information flows. Language-based information flow analyses can be applied to hardware description languages, but a straight-forward application either conservatively rules out many secure hardware designs, or constrains the designers to work at impractically low levels of abstraction. We demonstrate that choosing the right level of abstraction for the analysis, by working on Finite State Machines instead of the hardware code, allows both precise information flow analysis and high-level programmability.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Hardware description languages*

General Terms

Languages, Security

Keywords

Information Flow Analysis, Hardware Security, Language-Based Automated Verification

1. INTRODUCTION

Embedded systems are increasingly being used in critical applications that require a high level of assurance. For example, systems used in banks, automobiles, aircraft, and smartphones can benefit from strong guarantees on how secret or untrusted information flows through the system, ensuring that secrets never leak to unclassfied outputs or that untrusted information never affects critical system data.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLAS '10 Toronto, Canada
Copyright 2010 ACM 978-1-60558-827-8 ...$10.00.

To provide such guarantees, designers of embedded systems often rely on information-flow analysis tools. Information-flow analysis is a versatile technique that associates information labels (such as secret/unclassified or trusted/untrusted) with various system inputs, and tracks how these labels propagate through the system to the outputs. Such tracking can then be used to ensure policies on information-flow such as non-interference, which requires that secret inputs have no visible effect on unclassified outputs.

While there exist many techniques to track information flows through software [17], little work has been done to aid hardware developers in analyzing information flows through hardware designs. In systems where the entire functionality is implemented in hardware, the hardware itself has to guarantee an end-to-end information flow policy. In systems which are programmable, all software-based security schemes build upon an interface provided by the hardware, hence it is also important to verify that the actual hardware implementation allows no covert channels. However, it is a non-trivial task to verify that a design complies with a given information-flow policy, as it could involve complex features such as caches, branch-predictors, status-bits, exception-handlers and pipelining. The situation becomes worse when we consider that hardware developers are not usually experts in information flow analysis.

Ideally, hardware designers should be able to design using familiar idioms, get early design-time feedback about information flows in the system, and quickly iterate through different options to create verifiably secure hardware designs. However, hardware designs have unique characteristics that make a direct application of traditional information flow tracking techniques too conservative to be useful. Hence in this paper, we posit that *by carefully choosing the right level of abstraction for analysis, we can analyze information flows through hardware designs precisely using automated, language-level techniques.*

We explore a new direction where we base the information-flow analysis on a pervasive idiom in hardware design, namely Finite State Machines, and use this fact to make our analysis more precise even as the developers focus on specifying functionality using a familiar high-level abstraction. State machines are widely recognized as a natural way to describe

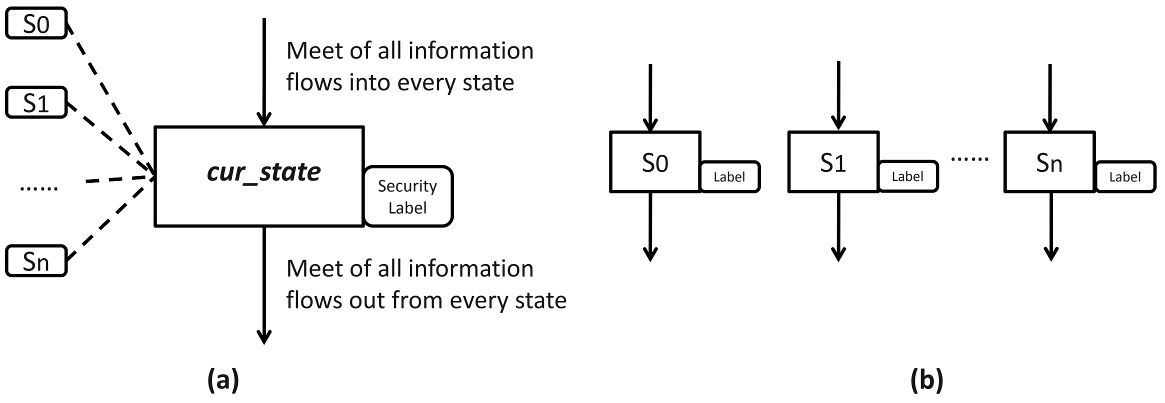

S0
S1
......
Sn
Meet of all information flows into every state
cur_state
Security Label
Meet of all information flows out from every state
(a)
S0
Label
S1
Label
......
Sn
Label
(b)

Figure 1: (a) Existing program analysis on state machines: States are represented as values of a single variable `cur_state`, associated with a single tag, and all information flows into and out from every individual state are combined. (b)Our proposed precise analysis: Every specific state is analyzed independently.

hardware controllers, and most commercial Computer-aided design (CAD) tools can extract state machines from Verilog/VHDL programs automatically. More recently, numerous state machine based languages and diagrams have been invented to explicitly express hardware as state machines [10, 19]. We propose that this trend towards expressing hardware explicitly as state machines not only retains the high-level programmability that is so desired by developers, but also allows more precise information-flow analysis.

A state machine can be expressed as a set of states and transitions among those states triggered by *signals* which are either inputs to the state machine or local data. The most natural way to implement a state machine using programming languages is to have a variable `cur_state` to store the current state, and `case`-style statements to decide state transitions based on `cur_state` and some other conditions. Figure 1(a) shows how conventional information analysis is applied to such state machines implementations. The value of `cur_state` can be one of S_0, $S_1 \ldots$, indicating the current state. When information flows are analyzed, `cur_state` is associated with a single security label. Such analysis does not take into consideration the fact that information flows are actually flowing through each individual state, hence there is no way to track the security labels of individual states when states are represented only as different values of the variable `cur_state`, making the analysis conservative. The *key insight* of our approach is that by analyzing hardware descriptions explicitly as state machines (i.e., as a reified set of individual states with accompanying transitions) rather than as an implicit state machine encoded using variables, the analysis can precisely track security labels for individual states. Figure 1(b) shows that we associate security labels with each individual state, and analyze information flows for every state independently, hence we are able to derive more precise information flow relations.

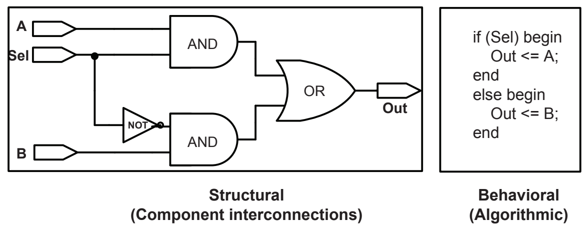
Specifically, this paper makes the following contributions:

- We investigate the problem of precise information-flow analysis of hardware descriptions without compromising programmability. We demonstrate that traditional information-flow tracking techniques are overly conservative for many critical hardware designs.
- We show that precise analysis can be performed on explicit state-machine based designs rather than conventional high-level descriptions.
- We outline a general framework for a state machine based hardware verification tool.

2. BACKGROUND

Simple hardware designs can be expressed *structurally* as a composition of hardware design primitives. For example, as shown in Figure 2, a multiplexer can be expressed as a composition of a pair of AND gate, one OR gate and an inverter. However, as the desired functionality grows in size and complexity, hardware design increasingly relies on CAD tools and higher-level hardware description languages and design patterns. Modern CAD tools require programmers to specify only the high-level behavior that is expected of the logic. For example, the above multiplexer can be simply expressed by the hardware designer using an `if-else` or a `switch-case` construct (as illustrated in Figure 2), while the CAD tool automatically synthesizes a gate-level implementation. While the difference in expressiveness is not evident when all we need is a multiplexer, being able to synthesize an adder by writing `a <= b + c` instead of a gate-level description of an adder frees the programmer to focus on higher-order issues and leaves low-level details to the CAD tool. Such *behavioral* modeling of hardware design has become predominant since creating large hardware designs such as the OpenSPARC CPU [1] in a completely structural manner becomes intractable quickly.

A design pattern that arises frequently in hardware description and synthesis is that of a Finite State Machine. State machines are useful because they naturally arise as a means of modeling many hardware controllers, allow designers to model the behavior of the digital system clearly under all input conditions, and guarantee that a design can be synthesized correctly. One can implement state machines implicitly in HDLs by using combinations of *if* and *case* statements. Alternatively, to help explicitly model digital system designs as state machines, programming languages and tools such as Statecharts [10] and Esterel [19] have been invented. Modeling hardware designs explicitly as state machines has become an increasingly recognized trend.



```
if (Sel) begin
   Out <= A;
end
else begin
   Out <= B;
end
```

Figure 2: Implementation of a 2-1 multiplexer using *structural description* and *behavioral description*. The function of a 2-1 multiplexer is to choose between input A and B based upon the value of input Sel. The structural description demonstrates how logic gates are connected to achieve multiplexing, while behavioral description describes the high level behaviors by using high level languages

3. RELATED WORK

3.1 Information Flow Security

Information flow policies can involve either confidentiality (secret data can never leak to unclassified users) or integrity (untrusted data can not affect trusted data), where both confidentiality and integrity are duals that can be captured by a general notion of Non-Interference [8]. Different approaches have been taken to enforce information flow policies at various levels of computer abstraction. Tag-based tracking at the virtual machine, architecture, or ISA levels is a popular dynamic solution, and tracks information flows through all registers and memory [2, 7, 3, 24, 4, 25, 16]. Projects such as LoStar [30], HiStar [29] and Flume [13] apply distributed information flow control (DIFC) [28] through general purpose operating systems abstractions. Programming language-based techniques either use specific type systems to represent security levels to enforce information flow policies statically [15, 26], or use program analysis techniques to derive information flow properties [5, 14]. A survey by Sabelfeld and Myers [17] gives a more complete list of related work in language based information security. As compared to the above research on information flow security (that aims to guarantee software security policies), our problem is to analyze information flow properties of the underlying hardware digital systems.

3.2 Program Analysis on HDLs

While there exists considerable research on functional verification of hardware, techniques for verifying information flow policies for hardware designs have been scarce. Functional verification of the hardware becomes harder with the increasing complexity of designs, and detecting errors through simulation becomes impractical due to long simulation time and limited coverage. Approaches based on program analysis have been proposed to help functional hardware verification, i.e, static analysis [9] and model checking [6]. In [11, 12] Hymans gives a design for static analysis of VHDL using abstract interpretations, while Schlicking further presents a framework for generating static analyzers on VHDL code [18]. These static analysis techniques are useful for deriving hardware specification properties and to ease testing and evaluation [27].

The closest related work on analyzing information flow properties for HDLs was proposed by Tolstrup et al [22, 23]. In this paper the authors describe techniques to identify information flows in synthesizable VHDL programs. Tolstrup's work is promising, but we find that for many hardware designs that multiplex different trust domains onto the same hardware, the analysis is imprecise and will label as unsafe designs that can be shown to have no illegal information flows. In the next section, we will give an example to show this imprecision and provide a novel method to analyze HDLs in a more precise way.

In another related work [21], we have proposed a technique that can *dynamically track* all information flows through hardware designs at the level of individual logic gates. Such Gate-Level Information Flow Tracking (GLIFT) can account for explicit, implicit, and timing channels in a unified manner, but the downside is that it requires the hardware design to be synthesized down into a netlist before its information flows can be tracked dynamically. Ideally, we would like the information flow analysis to be able to work statically on a behavioral description of the hardware, and enable quicker feedback and redesign without losing precision such as in extant higher-level analysis.

4. HARDWARE DESCRIPTION ANALYSIS

In this section, we will show that program analysis on conventional hardware description languages can be imprecise for certain types of digital systems, and we demonstrate how to perform precise analysis by raising the level of abstraction to explicit state machines.

4.1 Hardware Information Security

Hardware information security can be enforced in two ways. The first is by physical separation such that trusted components and untrusted components are physically separated and can never interact with each other, hence there is no way to leak information. While physical separation can completely guarantee hardware information flow policies, one needs to duplicate each resource into different copies for each partition. Such aggressive duplication is expensive and hence not practical in many situations.

The other solution is controlled separation. The same hardware resource is shared by all security partitions, but each partition uses the resource in a bounded way such that the boundaries of different partitions can never intersect. Examples of controlled separation include Time Division Multiple Access (TDMA) which divides a shared communication channel into separated time slots, and execution leases [20], which allow the execution of untrusted components within bounded time and using bounded resources. Analysing information flow policies for physically separated systems is trivial, while verifying the correctness of controlled separation is challenging. In the following paragraphs we will show how conventional program analysis can fail to precisely derive information flow policies for system descriptions using controlled separation.

4.2 Imprecise Program Analysis on Behavioral HDLs

We use the Execution Lease controller as an example hardware design and Verilog as an example hardware description language. Execution Leases are an architectural mechanism that enables trusted code to grant access to a limited amount of machine resources to untrusted code for a fixed amount of

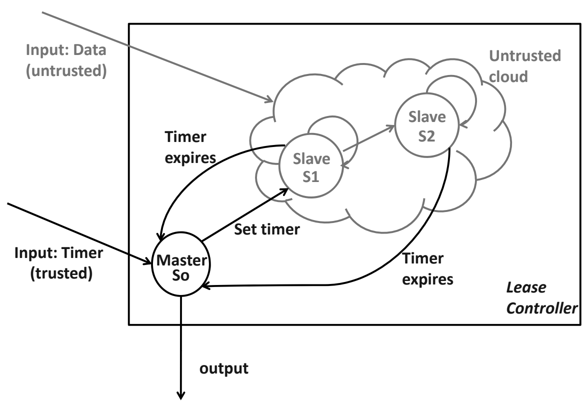


Figure 3: State machine diagram of the lease controller. The input to the lease controller consists of a trusted timer and untrusted data, and the output is generated by the master state. When a timer is specified, the state machine transfers to the untrusted cloud until the timer expires, at which time the state machine goes back to the master state.

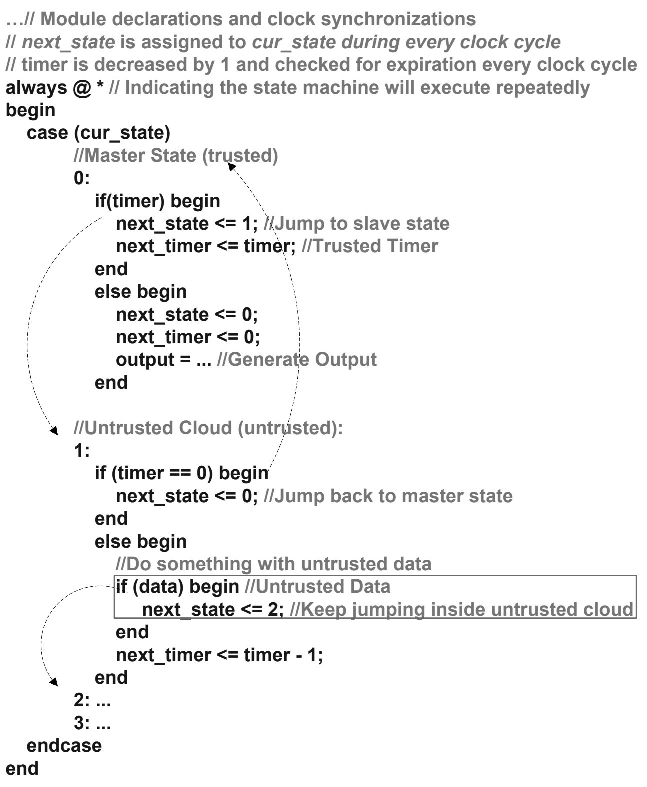
time. One can imagine a lease to be a sandbox of space and time which untrusted components can never go beyond. A lease starts by setting up a timer and transferring the control to untrusted components. After the timer expires, the control will be automatically transferred back to the trusted system.

Without losing generality, in this paper we assume a two-label security lattice with security labels *High* and *Low*, where *High* indicates secret (in terms of secrecy) or untrusted (in terms of trustiness), and *Low* represents unclassified or trusted. In such security lattice, information is allowed to flow from any data with a *Low* security label to data marked as *High*, while the other direction is illegal.

Figure 3 gives the simplified state machine diagram of an Execution Lease controller. As can be seen from the figure, the inputs to the lease controller include the $\mathtt{timer}_{low}$ which is a trusted value and $\mathtt{data}_{high}$ which is an untrusted value. The `output` is generated by state S_0. The lease mechanism works as follows: Some trusted component (master state S_0) initiates a lease to some untrusted cloud (simplified as two states in this example) by specifying a $\mathtt{timer}_{low}$ boundary. The control transfers to slave state S_1 which is inside the untrusted cloud, and either stays at S_1 or jumps to S_2 based on some untrusted data_{high} and loops inside the cloud until the $\mathtt{timer}_{low}$ expires. When the $\mathtt{timer}_{low}$ expires the control automatically transfers back to the master state no matter what the current state is.

The corresponding Verilog program is shown in Figure 4. The value of the state variable `cur_state` can be either 0 (master state) or $1, 2, 3\ldots$(in the untrusted cloud). In master state S_0, if the $\mathtt{timer}_{low}$ is activated, the state will transfer to S_1 by assigning `cur_state` to 1. In slave state S_1, if the `timer` expires, the state will transfer back to S_0, otherwise $\mathtt{data}_{high}$ is processed and $\mathtt{timer}_{low}$ will be decremented.

We apply conventional program analysis techniques on the program to identify how information flows between input and output. Figure 5 gives part of a simplified set of information flow analysis rules used by most previous type-system-based approaches. In this figure we only list two



```
…// Module declarations and clock synchronizations
// next_state is assigned to cur_state during every clock cycle
// timer is decreased by 1 and checked for expiration every clock cycle
always @ * // Indicating the state machine will execute repeatedly
begin
   case (cur_state)
         //Master State (trusted)
         0:
            if(timer) begin
               next_state <= 1; //Jump to slave state
               next_timer <= timer; //Trusted Timer
            end
            else begin
               next_state <= 0;
               next_timer <= 0;
               output = ... //Generate Output
            end

         //Untrusted Cloud (untrusted):
         1:
            if (timer == 0) begin
               next_state <= 0; //Jump back to master state
            end
            else begin
               //Do something with untrusted data
               if (data) begin //Untrusted Data
                  next_state <= 2; //Keep jumping inside untrusted cloud
               end
               next_timer <= timer - 1;
            end
         2: ...
         3: ...
   endcase
end
```

Figure 4: Behavioral Verilog Code for Lease Controller: State Transitions are made based on both the current state and other conditions. Arrows represent state transitions, and the highlighted statements are those that makes the state variable untrusted.

rules that derive how explicit (through assignment) and implicit (through branch) information flows along with a subtype rule that enables information flows from low security labels to high security labels: This rule-set only deals with ***if*** statements but ***case*** statements can be transformed into a set of ***if else*** statements hence share the same rules. In state S_1 of the lease controller, as highlighted by Figure 4, the state variable can either stay at S_1 or change into S_2 based on the value of $\mathtt{data}_{high}$, hence there is implicit information flow from $\mathtt{data}_{high}$ to the state variable `cur_state`. By applying Rule BRANCH, `cur_state` becomes *high.* Once the state variable becomes *high*, everything including `output` gets *high* according to Rule BRANCH extended to ***case*** statements since every assignment is based on the current state. Hence conventional program analysis will conclude that the output of the lease controller will be *high.* However, the lease mechanism guarantees that when the $timer_{low}$ expires, control will always transfer back to the master state–there is no way that $data_{high}$ can actually affect either the value or the timing of the output generated in the master state. Conventional program analysis techniques can not detect that the `timer` which triggers the control transferring back is *low* and hence security label of the master state and any information derived from the master state are also *low.*

4.3 State Machine Based Analysis

In the previous analysis `cur_state`, the variable keeping

$$\frac{\Gamma(x) = \tau\ var \qquad \vdash e : \tau}{\vdash x \mathrel{\mathbf{:=}} e : \overline{\tau}\ cmd} \quad (\text{ASSIGN})$$

$$\frac{\vdash e : \tau \qquad \vdash c_1 : \overline{\tau}\ cmd \qquad \vdash c_2 : \overline{\tau}\ cmd}{\vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \overline{\tau}\ cmd} \quad (\text{BRANCH})$$

$$\frac{\vdash p : \tau \qquad \tau <: \tau'}{\vdash p : \tau'} \quad (\text{SUBTYPE})$$

Figure 5: Conventional Analysis Rules (from [26]): rule ASSIGN formalizes explicit information flows through assignment statements, and rule BRANCH gives implicit flows through conditional statements. The rule SUBTYPE indicates that information can flow from low security labels into high security labels. $\Gamma(x)$ is defined as the mapping from a variable to its security label. Security labels can be either τ which is the security label of expressions, τ var as the security label of variables, or $\overline{\tau}$ cmd as the security label of commands. The overline in $\overline{\tau}$ cmd indicates contravariance.

track of the current state in the state machine, is associated with a single security label. Once `cur_state` becomes *high*, taint explosion will mark everything as *high*. However, `cur_state` being *high* does not necessarily mean every specific state of the state machine (S_0,S_1 ... in this case) contains *high* information. The root cause of this problem is that there is no way to track the security labels of individual states, because states are represented only as different values of the variable `cur_state`. The key insight of our approach is that by analyzing hardware descriptions explicitly as state machines (i.e., as a reified set of individual states with accompanying transitions) rather than as an implicit state machines encoded using variables, the analysis can precisely track the security labels for individual states. We informally describe below the resulting rules that allow for a more precise analysis [1].

To perform program analysis on the hardware descriptions with explicit state machine information we associate a security label with every specific state and extend conventional analysis rules to include the following **Basic Rules**:

1. Under some state S_0, if there is some condition e making the state transits to S_1 then there are information flows from both S_0 and e to S_1.

2. Under some state S_0, if there is any assignment to some variable x then there is information flow from S_0 to x.

To increase the precision of the analysis we propose two extra **Reduction Rules** which can reduce the conservative propagation of information flows. These reduction rules are the key rules to make our analysis precise:

1. If there is a transition from every state labeled as *high* to some state S_0 based on the same *low* signal, there should not be any information flow from *high* states to S_0 even there seem to be flows from every *high* state to S_0 according to Basic Rule 1. The reason is that the transition to S_0 will be executed in every *high* state, and hence is independent of any single *high* state.

2. If under every *high* state there is the same computation assigned to some variable x, then there should not be any information flow from *high* states to x even though there seems to be according to Basic Rule 2. The reason can be explained similarly to Reduction Rule 1.

[1]Our ongoing work involves formalizing and proving the soundness of an analysis based on these insights.

These reduction rules can be seen as generalizations of the BRANCH rule in Figure 5. Notice that this rule requires the security levels of the **then** and **else** branches to be raised to at least the level of the guard expression, but that these raised security levels are reduced at the join point of these two branches; i.e., the security level of the point immediately following the two branches is independent of the guard expression. This reduction is sound because only the two branches are control-dependent on the guard expression; the program point following the branches is not control-dependent on the guard expression. Our reduction rules similarly recognize the fact that if multiple states transition to the same new state under exactly the same condition, then that new state isn't control-dependent on any one of the original source states and allows the analysis to reduce the security levels accordingly.

Now we apply the new rules to the lease controller and we will get the following flows between inputs, states and outputs:

- $\{S_0, timer_{low}, data_{high}, S_1\} \rightarrow S_1$
- $\{S_1, data_{high}\} \rightarrow S_2$
- $\{S_1, S_2, timer_{low}\} \rightarrow S_0$
- $\{S_1, S_2\} \rightarrow timer$
- $\{S_0\} \rightarrow output$

The first flow gives the result that S_1 is labeled as *high* since *data* is *high*. The second gives the result that S_2 is also *high* for the same reason. The third one is reduced according to Reduction Rule 1: in both S_1 and S_2 the transition can go back to S_0 if `timer` expires. The fourth one is also reduced according to Reduction Rule 2: the computation assigned to `timer` is the same under every untrusted states S_1 and S_2. From the results of the above analysis, we can conclude the following security status for every state: S_1 and S_2 are *high* while S_0 is *low*. Finally the *output*, which comes out from *low* state S_0, now is correctly marked as *low*.

4.4 General Framework of Our Tool

In conclusion, we propose to explicitly model hardware descriptions as state machines such that we are able to analyze information flows through every individual state, and give more precise results than conventional techniques. Figure 6 presents the general framework for our proposed approach. The bottom part represents the existing framework in which hardware descriptions are written at either behavioral or structural abstraction, verified by conventional analysis tools, then synthesized down to physical implementations. To enable precise information flow analysis, we add another level above behavioral hardware descriptions which

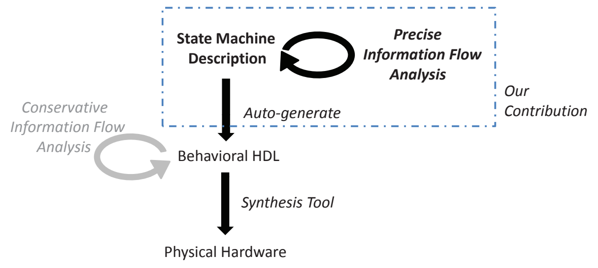

State Machine Description
Precise Information Flow Analysis
Our Contribution
Conservative Information Flow Analysis
Auto-generate
Behavioral HDL
Synthesis Tool
Physical Hardware

Figure 6: General Framework of Our Tool: The highlighted part is our contribution which allows one to explicitly model hardware designs as state machines and perform more precise information flow analysis. The gray part–conventional imprecise information flow analysis techniques are then removed from the framework.

allows one to describe hardware using state machine languages, verified using our proposed analysis tool, and compiled to conventional behavioral or structural code using the tool's back-end.

As a common pattern used in system designs, state machines are not limited to hardware designs, but can also be applied to model software systems such as software protocols. Analyzing state-machine-based software protocols by explicitly tracking information flows among individual states may also lead to more precise observations. Our future work seeks to build formal type system based analyses to express the reduction rules proposed in this paper, and explore potentials of such technique in software design.

5. REFERENCES

[1] OpenSPARC project. http://www.opensparc.net.

[2] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[3] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, pages 221 – 232, 2004.

[4] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA)*, June 2007.

[5] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[6] E.M.Clarke, O. Grumberg, and D.Peled. *Model Checking.* MIT Press, 2000.

[7] G.E.Suh, J.W.Lee, D.Zhang, and S.Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, 2004.

[8] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of IEEE Symposium on Security and Privacy*, 1982.

[9] C. Hankin. Program analysis tools. *International Journal on Software Tools for Technology Transfer*, 2(1), 1998.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 1987.

[11] C. Hymans. Checking safety properties of behavioral vhdl descriptions by abstract interpretation. In *9th International Static Analysis Symposium (SAS'02) (2002*, pages 444–460. Springer.

[12] C. Hymans. Design and implementation of an abstract interpreter for vhdl. *D.Geist and E.Tronci, editors, CHARME*, 2860 of LNCS, 2003.

[13] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. Frans, K. Eddie, and K. R. Morris. Information flow control for standard os abstractions. In *In SOSP*, 2007.

[14] J. McHugh and D. I. Good. An information flow tool for gypsy. In *IEEE Symposium on Security and Privacy*, pages 46–48, Apr. 1985.

[15] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. Software release. http://www.cs.cornell.edu/jif, July 2001.

[16] O.Ruwase, P.B.Gibbons, T.C.Mowry, V.Ramachandran, S.Chen, M.Kozuch, and M.Ryan. Parallelizing dynamic information flow tracking. In *SPAA'08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 35–45. ACM, 2008.

[17] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), Jan. 2003.

[18] M. Schlickling and M. Pister. A framework for static analysis of vhdl code. *7th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2007.

[19] E. Technologies. *The Esterel v7 Reference Manual, version v7.30 - initial IEEE standardization proposal edition.* 2005.

[20] M. Tiwari, X. Li, H. Wassel, F. Chong, and T. Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2009.

[21] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[22] T. K. Tolstrup. *Language-based Security for VHDL.* PhD thesis, Technical University of Denmark, 2006.

[23] T. K. Tolstrup, F. Nielson, and H. R. Nielson. Information flow analysis for vhdl. volume 3606 of LNCS, 2005.

[24] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *In MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254. IEEE Computer Society, 2004.

[25] G. Venkataramani, I. Doudalis, Y. Solihin, and

M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Fourteenth International Symposium on High Performance Computer Architecture (HPCA)*, pages 196–206, New York, NY, USA, 2008. ACM.

[26] D. Volpano and G. Smith. A type-based approach to pro-gram security. In *In Proceedings of the 7th International Joint Conference on the Theory and Practice of Software Devel-opment*, pages 607–621. Springer, 1997.

[27] S. Wilhelm. Efficient analysis of pipeline models for WCET computation. In *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, 2005.

[28] N. Zeldovich, S. Boyd-Wickizer, and D.Mazieres. Security distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–308, Apr. 2008.

[29] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[30] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *8th USENIX Sumposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2008.