Secure Information Flow Verification with Mutable Dependent Types

Andrew Ferraiuolo, Weizhe Hua, Andrew C. Myers, G. Edward Suh
Cornell University
Ithaca, NY 14850, USA
af433@cornell.edu, wh399@cornell.edu, andru@cs.cornell.edu, suh@csl.cornell.edu

ABSTRACT

This paper presents a novel secure hardware description language (HDL) that uses an information flow type system to ensure that hardware is secure at design time. The novelty of this HDL lies in its ability to securely share hardware modules and storage elements across multiple security levels. Unlike previous secure HDLs, the new HDL enables secure sharing at a fine granularity and without implicitly adding hardware for security enforcement; this is important because the implicitly added hardware can break functionality and harm efficiency. The new HDL enables practical hardware designs that are secure, correct, and efficient. We demonstrate the practicality of the new HDL by using it to design and type-check a synthesizable pipelined processor implementation that support protection rings and instructions that change modes.

1. INTRODUCTION

Modern processors are complex systems that implement a large number of instructions and features, rely on subtle optimizations, and are built by large teams. In such complex systems, bugs and security vulnerabilities are inevitable. Processor errata [4] and security attacks exploiting these bugs [12] illustrate the problem. Security vulnerabilities in hardware are often subtle, and difficult to amend after fabrication. Therefore, there is need for formal and automated methods for catching and preventing vulnerabilities during the hardware design process. With increasing interest in hardware accelerators, specialization, and external IPs, the need for effective design-time tools for security verification is growing.

This paper presents a new version of the secure hardware description language (HDL) SecVerilog [15] with novel type system extensions. We refer to this new version of SecVerilog as SecVerilogLC. SecVerilog uses an information flow control (IFC) type system to check security properties of hardware implementations at design time. Previously proposed secure HDLs [5, 6, 15] do not provide sufficient support for sharing hardware resources among security levels. This sharing is essential for building efficient and practical systems. We show that this limitation can be addressed by making clock cycles explicit in the HDL and redesigning the type system accordingly. We demonstrate the capabilities of our new version of SecVerilog through a prototype hardware design.

Information flow control (IFC) constrains the movement of data in HDL code based on a security policy provided by the hardware designer. A security policy is expressed by annotating variables in the code with security labels (such as "trusted" or "untrusted") and specifying how information can be propagated among these labels (e.g., untrusted signals cannot affect trusted ones). The constraints are enforced by a type system, which is fast, detects vulnerabilities without simulations, and can ensure a formal security property [8]. Previous studies show that information flow control (IFC) at the HDL level is a promising approach for statically and automatically checking security properties of hardware at design time [5, 6, 15].

HDL-level IFC is lightweight in terms of both designer effort and hardware overhead. By contrast, the effort required for conventional verification methods is often significant. For example, verifying an Intel Pentium 4 processor took a dedicated team with specialized verification knowledge multiple years [1]. Writing security policies in SecVerilog is similar to writing type declarations—it does not require manual proofs as with theorem provers or temporal logic specifications as with model checkers. Because checking is done statically, overhead in the final design is minimal.

However, a naive application of IFC to an HDL would lead to inefficient hardware implementations. Each hardware component would be able to handle information of only a single security level, so hardware would need to be duplicated for the trusted and untrusted parts of the system. Prior secure HDLs support shared hardware by using either nested states [5, 6] or dependent types [15]. Nested states can only support coarse-grained sharing; execution can be time-multiplexed between trusted and untrusted instantiations of hardware, but physical duplication (e.g., of registers) is still required.

Dependent types enable fine-grained sharing of registers by allowing the security level of individual registers to depend on the run-time value of another signal. However, the previous dependently-typed secure HDL, SecVerilog [15], limits expressiveness and has serious practical limitations. First, it does not reason about cycle-by-cycle updates to registers, and this often causes the type system to reject secure designs. Second, unless the type system is carefully designed, using dependent types for information flow labels can lead to subtle security vulnerabilities because they allow security labels to change at run time. If the label of a register changes (such as from untrusted to trusted), but the value stored in the register does not also change, the label no longer describes the register's content accurately. In this case, the label change could allow the untrusted value to corrupt critical signals. This incorrect change in labeling is known as *implicit downgrading* [15].

The state-of-the-art solution to the implicit downgrading problem in secure HDLs is dynamic clearing [15]. Dynamic clearing is a compiler mechanism that inserts run-time logic that clears a register when its security label changes. Unfortunately, dynamic clearing has significant practical limitations: 1) it precludes practical hardware designs where either hardware is shared by multiple security levels or the initial state must be controlled by the hard-

DAC '17, June 18-22, 2017, Austin, TX, USA
© 2017 ACM. ISBN 978-1-4503-4927-7/17/06.
DOI: http://dx.doi.org/10.1145/3061639.3062316

ware designer, 2) it produces hardware that differs from what is described by the HDL code, and 3) it may damage integrity.

In this paper, we propose novel extensions to secure HDLs [15] to address the implicit downgrading problem. Addressing this problem requires fundamental changes to the design of hardware description languages. In particular, the proposed language is redesigned to explicitly distinguish sequential and combinational variables, and the propagation of values on clock edges is made explicit in the language. This version of SecVerilog allows designers to specify how label changes should be handled, and is expressive enough to describe a secure implementation of mode switches.

Our novel type system relies on two key observations: 1) implicit downgrading occurs only for sequential variables, and 2) the signals which determine both the labels and the values of registers during the *next* clock cycle (i.e., on updates to registers and their labels) are available statically. Given these observations, we introduce an explicit notion of cycle-by-cycle transitions into the syntax, semantics, and type system of our language in a way that is natural to hardware designers. The type system statically checks that changes in the security level of a module (i.e., label changes) are handled in the code. The language allows the designer to describe how label changes should be handled when explicit intervention is required, and label changes often require no new code to be written. Because the designer must explicitly describe how to handle insecure label changes, the behavior of the code is clear to the designer and matches that of the synthesized hardware.

The main contributions of this paper are as follows:

- A novel secure HDL that avoids implicit downgrading, yet is powerful enough to express practical hardware designs.
- The introduction of clock-cycle updates into the syntax, semantics, and type system that improves expressiveness.
- The first dependent type system for information flow control which supports types that refer to mutable variables and verifies label changes fully statically at design time. Our approach is also applicable to software IFC systems.
- A demonstration of the new language through a processor pipeline design that includes support for mode switches between security levels.

2. NEW TYPE SYSTEM

2.1 Background

In hardware description languages (HDLs) that use information flow control (IFC) [5, 6, 15], the types of variables (signals) are annotated with security labels such as T (for trusted) or U (for untrusted). The hardware designer also provides a security policy that expresses how information is permitted to flow among these security levels. For example, a security policy may specify that flow is allowed from trusted (T) signals to untrusted (U) ones, but flow is not allowed in the other direction. This policy protects the integrity of signals labeled T. The security type system then enforces the security policy based on the annotations (security labels). In this integrity protection example, the type system can statically guarantee that untrusted signals do not affect trusted ones. Confidentiality protection can be provided by preventing secret (S) signals from affecting public (P) ones. Figure 1 shows an example in which the signals `creg` and `trst` are labeled T and the signal `untr` is labeled U. The assignment on line 3 is rejected by the type system because it would cause an illegal flow U→T, but the assignment on line 5 is permitted since both signals are trusted.

```
1   reg [31:0] {T} creg, [31:0] {U} untr, [31:0] {T} trst;
2   ...
3       creg <= untr; // not allowed
4       creg <= trst; // allowed
5   ...
6   reg {T} mode;
7   // mode_to_lb(0) = T, mode_to_lb(1) = U
8   reg [31:0] {mode_to_lb(mode)} gpr;
9   ...
10  if (mode == 1'b0) creg <= gpr;
11  ...
```

Figure 1: SecVerilog code example.

```
1   reg {f(label)} data;
2   reg {f(next_label)} next_data;
3   always@(posedge clk)
4       data  <= next_data;
5       label <= next_label;
6   end
```

Figure 2: A label propogation example.

However, many practical hardware designs cannot be implemented efficiently with such simple labels. Labeling a component T means it can only be used exclusively by one security level. If the same functionality were needed for another security level, the hardware module would have to be duplicated. To design efficient hardware, it is essential that hardware resources can be shared among multiple security levels over time. In SecVerilog [15], sharing is permitted through *dependent types* such as the label of `gpr` on line 10. This label, `mode_to_lb(mode)`, is a function of the signal `mode`; the label is T when the `mode` bit is 0 and U otherwise. Even though the label of `gpr` depends on the run-time value of `mode`, the assignment on line 10 can still be type-checked statically. Because the assignment happens under a branch in which `mode` is 0, the type system can infer that the label of `gpr` is `mode_to_lb(0) = T` in this context. Prior secure HDLs [5, 6] also used nested states to allow sharing among levels. However, nested states cannot be used to describe registers like the `gpr` which are shared by different security levels over time. We use a dependent type system since it permits fine-grained sharing.

SecVerilog [15] has limited expressive power because it does not reason about updates to values on clock edges. Figure 2 illustrates the problem through a simple example in which a variable and its corresponding label are updated at the same time. In the example, the label of `data` depends on the value of the variable `label` (since it is `f(label)`). The label of `next_data` similarly depends on `next_label`. The code is obviously secure because both value and label change together. Propogating labels in this way is necessary for components such as pipeline registers and on-chip networks.

However, SecVerilog fails to typecheck this secure code because it does not distinguish between updates that occur in the current clock cycle and the next clock cycle. The assignment on line 4 is rejected because it is handled as though the update to `data` happens immediately. Indeed, if `data` were updated without propogating its label, security could be violated. However, by the time the contents of `data` are updated (i.e., on the clock edge), its label has also been updated. Our new version, SecVerilogLC, introduces a notion of cycle-by-cycle updates to registers. SecVerilogLC can detect that on clock edges `label` is updated to `next_label` and can conclude on line 4 that it is safe to allow `next_data` to be stored in `data`.

Unfortunately, dependent types introduce subtle security vulnerabilities when the variables on which they depend can change. Figure 3 illustrates this problem. The signal `shared` has dependent label `mode_to_lb(v)`, `v` is trusted, and `trst` and `untr` are labeled in the same way as before. This code is clearly insecure; on line 5, an untrusted value is stored in the shared register and this untrusted

```
1   // mode_to_lb(0) = T, mode_to_lb(1) = U
2   reg {T} v, {T} trst, {U} untr;
3   reg {mode_to_lb(v)} shared;
4   ...
5       if (v == 1'b1) shared <= untrusted;
6       else           trusted <= shared;
7   ...
```

Figure 3: Implicit downgrading example.

value is directly copied into the trusted variable on line 6. These lines can be executed in sequence over two clock cycles. This type of leakage through changes in dependent types is known as *implicit downgrading* [15]. In information flow control, secret information may be explicitly *downgraded*, i.e., released to the public, when the designer deems this release necessary and secure [9]. However, when downgrading is implicit, it represents a potential security vulnerability that the designer is unaware of.

The state-of-the-art solution to the implicit downgrading problem is *dynamic clearing* [15] — the compiler automatically inserts logic to clear dependently labeled registers whenever the labels of these registers are changed [14]. Dynamic clearing has severe practical limitations. It can cause hard-to-detect functional errors because it adds extra logic in the background that is not specified in the code. The added clearing logic causes the simulations and synthesized hardware to differ from what the designer would expect.

Dynamic clearing also makes it impossible to describe many hardware designs. We illustrate these limitations by describing the complications that dynamic clearing causes for the design of a widely-used processor feature — a privileged kernel mode and a user mode. Naturally, the labels for many processor resources will depend on the control register that indicates the current mode. General purpose registers (GPRs) should have labels that depend on the mode, since the trustworthiness of their contents depends on the mode that wrote them last. The program counter (`pc`) will also have a label that depends on the mode. Pipeline registers should have the same mode-dependent labels to reflect the privilege level of in-flight instructions. When the mode switches from user (`U`) to privileged (`T`), all of these registers would be dynamically cleared—whether the hardware designer wants this behavior or not.

Dynamic clearing prevents legitimate communication between security levels. For example, a system call instruction in the above processor example will trigger a label change from `U` (user) to `T` (privileged). Typically, some of the GPRs are used to pass information such as a system call number or arguments from the user mode to the privileged supervisor mode. Automatically clearing the GPRs during this mode switch breaks the functionallity of system calls. Instead, the secure design language should allow the designer to *explicitly* downgrade the label of a register in certain cases so that its value can be preserved on a label change.

Here, and in other cases, dynamic clearing damages integrity. If the pipeline registers were automatically cleared on a mode change, in-flight instructions would likely be converted erroneously into NOPs. More generally, dynamic clearing can remove secrets and protect confidentiality, but when a trusted register is expected to contain a specific value, replacing it with a zero violates integrity.

Ideally, the security type system must be precise enough so that it only requires explicit handling of label changes only if necessary for security. Dynamic clearing conservatively erases data on *any* label change. For example, a label switch from `T` to `U` on a return from a system call is not a concern for integrity; restoring a PC value from a saved one (in the *epc* register in MIPS) should not require explicit downgrading.

Our novel dependent type system is expressive enough to describe all of the above hardware while securely avoiding the implicit downgrading problem. Communication among security levels in the GPRs is permitted by explicit downgrading. The type system precisely tracks the direction of label changes allowing our design to load a trusted value into the *pc* on entry to the kernel, and restore a saved *pc* on re-entry to userspace. The new type system permits design choices. For example, we can think of two correct implementations of mode switching: 1) pipeline the labels along with the regular pipeline registers, and 2) stall the pipeline until all in-flight instructions are drained. Our type system supports both designs.

```
1   wire com {T} mode_switch;
2   assign mode_switch = decode_out[4];
3
4   reg seq {U} epc;
5   reg seq {T} mode;
6   reg seq {mode_to_lb(mode)} pc;
7   // mode_to_lb(0) = T, mode_to_lb(1) = U
8   always@(seq) begin
9       if (rst) pc <= 16'b0;
10      else if (mode_switch && (next mode == 1'b0))
11      pc <= `SYSCALL_PC_VAL; //switch to kernel mode
12      else if (mode_switch)
13          pc <= epc; //return to user mode
14      ...
15  end
```

Figure 4: PC during mode switches.

2.2 Approach

Our type system securely supports changes in dependent labels 1) by making the propagation of signals on clock edges explicit in the syntax, semantics, and type system, 2) by introducing a syntax for testing labels for the next clock cycle, and 3) by using the type system to statically establish that registers are securely updated along with their labels. Figure 4 shows code for a PC register that securely handles mode changes in the concrete syntax of SecVerilogLC.

Notably, only sequential logic can be implicitly downgraded through label changes since combinational logic is not stateful [15]. For this reason, combinational and sequential logic are separated in the language. In SecVerilogLC, sequential and combinational variables are explicitly separated through type annotations `com` (on line 1) and `seq` (on lines 4–6). Sequential and combinational signals are type-checked differently. For example, an assignment to a trusted combinational signal such as `mode_switch` defined on line 2 is secure as long as the value that is assigned is also trusted.

However, sequential signals (registers) such as `pc`, are type-checked in a different way. The values assigned to registers must be type-checked based on the *new* label of the register for the *next clock cycle*. This ensures that the new label of the register accurately reflects the security level of its contents. As an example, for the assignment on line 11 to type-check, the type system must prove that the label of `` `SYSCALL_PC_VAL `` is permitted to flow into the new label of `pc`, which is dependent upon the value of `mode` in the *next* clock cycle. In the example, the label can be statically determined to be `mode_to_lb(0)` (`T`) as we explain in the following paragraph.

SecVerilogLC supports a new operator, `next`, which when applied to a variable, gives the value it will take during the *next* cycle. For example, it is applied to `mode` on line 10, where it evaluates to the value of `mode` during the next cycle. The branch on line 10 is taken when there is a mode switch and the next-cycle value of `mode` is `0`, indicating a switch to kernel mode. The type system can thus infer that on line 11, the label of `pc` during the next cycle is `T`, and the assignment is safe as long as `` `SYSCALL_PC_VAL `` is trusted. On line 13, the branch was not taken, so the next-cycle label of `pc` must be `U`, and the assignment is safe. When the `next` operator is applied to registers (declared `seq`), the next-cycle value is available from the combinational input to the register. The next operator is useful both for implementing necessary access controls and assisting the type system.

Since type checking depends on whether variables are sequential or combinational, the syntax and semantics of SecVerilogLC ensure that the `com/seq` labels are accurate (i.e., that variables labeled `com` are not sequential). In SecVerilogLC, the clock signal is implicit, and sequential logic is written by describing its combinational input, as is done on lines 8–15. In the semantics, the statements describing sequential logic are treated as the combinational input to a register. Restrictions are then placed on combinational wires which ensure that they are in fact combinational: 1) there are no combinational loops, and 2) there are no inferred latches (see Section 2.3 for more details).

2.3 Language

Figure 5 shows a core abstract language capturing the essential elements of SecVerilogLC programs. SecVerilogLC supports most of the implementation of Verilog. Expressions e include integer literals (n), combinational variables (w, declared `com` in the concrete syntax), sequential variables (r), the `next` operator applied to expressions (written `next` e), and binary ($e \oplus e$) and unary (`uop` e) operators. The syntax also includes special symbols denoting the next-cycle values of sequential variables (r'), that are not written explicitly by the programmer, but are used in the semantics and typing rules. Programs *Prog* contain a combinational statement c and a list of sequential statements s. Combinational statements permit sequential composition since their meaning is order-dependent, and sequential statements do not. The syntax of both kinds of statements is otherwise standard.

$$\begin{aligned}
e &::= n \mid w \mid r \mid r' \mid \texttt{next}\ e \mid e \oplus e \mid \texttt{uop}\ e \\
c &::= \texttt{skip} \mid \texttt{if}(e)\ c\ \texttt{else}\ c\ \mid w := e \mid c;\ c \\
s &::= \texttt{skip} \mid \texttt{if}(e)\ s\ \texttt{else}\ s\ \mid r := e \\
\mathit{Prog} &::= c, \vec{s}
\end{aligned}$$

Figure 5: SecVerilogLC core syntax.

Formally, security labels represent security levels $\ell \in \mathcal{L}$ where $\mathcal{L}$ is a lattice with relation $\sqsubseteq$ [3]. Labels `T` and `U` introduced earlier are examples of security levels. Information is permitted to flow from `T` to `U` if and only if $T \sqsubseteq U$. The syntax of security labels is

$$\tau ::= l \mid f(\vec{w}, \vec{r}) \mid \tau \sqcup \tau$$

and includes levels ℓ, pure (side-effect free) functions of variables $f(\vec{w}, \vec{r})$, and the join of labels $\tau \sqcup \tau$, where the join is the least upper bound of its arguments. The dependent labels are the functions f which are applied to program variables.

Aside from the `next` operator, which has already been introduced (and simply substitutes occurrences of r with r' in e for `next` e), the semantics of expressions is standard. The small-step semantics of sequential commands is defined on configurations $\langle \sigma, r \rangle$. Here, σ is a mapping from (sequential and combinational) variables (r and w) and symbols identifying next-cycle values of sequential variables(r'). This semantics of commands is standard other than the rule for assignment to sequential variable r, which updates the value of r' in σ rather than r directly.

The small-step semantics of programs shown in Figure 6 is defined on configurations $\langle \sigma, c, \vec{s} \rangle$ where σ is a global store, c is the next combinational command to be evaluated, and $\vec{s}$ is the list of sequential commands to be evaluated in parallel. Rule Tick applies when the commands have been fully evaluated. Here, C and $\mathcal{S}$ are the commands as written in the original program before execution. The notation $\sigma[r \mapsto v]$ denotes the substitution of value v for r in σ. This rule propagates the next-cycle values to registers by substituting each sequential value r_i with its corresponding next-cycle value r'_i. This rule also copies C and $\mathcal{S}$ back into the configuration, so that the logic is re-evaluated during the next clock cycle.

$$\frac{c = \texttt{stop} \qquad \forall s \in \vec{s}.s = \texttt{stop} \\ \sigma' = \sigma[r_1 \mapsto \sigma(r'_1)]...[r_n \mapsto \sigma(r'_n)]}{\langle \sigma, c, \vec{s} \rangle \to \langle \sigma', \mathcal{C}, \mathcal{S} \rangle}\ \textsc{Tick}$$

$$\frac{c \neq \texttt{stop} \qquad \langle \sigma, c \rangle \to \langle \sigma', c' \rangle}{\langle \sigma, c, \vec{s}, \rangle \to \langle \sigma', c, \vec{s} \rangle}\ \textsc{Comb}$$

$$\frac{c = \texttt{stop} \qquad \exists s_i \in \vec{s}.s_i \neq \texttt{stop} \\ \langle \sigma, s_i \rangle \to \langle \sigma', s'_i \rangle \qquad \vec{s'} = \{s'_1, ..., s'_n\}}{\langle \sigma, c, \vec{s} \rangle \to \langle \sigma', c, \vec{s'} \rangle}\ \textsc{Seq}$$

Figure 6: Program semantics.

Rule Comb evaluates the command c and applies until it has been fully evaluated. Rule Seq applies when c has been evaluated, but there is some sequential command which has not been fully evaluated. In this rule, each of the sequential commands that have not been fully evaluated simultaneously take a single step.

Programs $c, \vec{s}$ have three well-formedness requirements not captured by the typing rules. First, for every combinational variable w assigned in c, w must be assigned in some node along some path to each node: 1) which is a postdominator of c, and 2) in which c is live. This ensures that there are no inferred latches and that variables are defined before they are read. Second, to prevent combinational loops, each sequential variable w is assigned a depth in the circuit graph denoted $\mathit{depth}(w)$. If w is assigned from e and $\mathit{vars}(e)$ are the variables in e, then for all v in $\mathit{vars}(e)$, $\mathit{depth}(v)$ must be lower than $\mathit{depth}(w)$. Third, each sequential variable r is assigned in at most one $s \in \vec{s}$. This ensures that the program is deterministic.

Typing judgments for sequential commands have the form $\Gamma, C, pc \vdash s$, meaning that s is well-typed under type environment Γ, constraint context C, and program counter label pc.[1] Type judgments for combinational commands have nearly the same form. Here Γ maps variables to labels. The program counter label is used to prevent implicit flows, as is standard in language-based IFC [8]. To make reasoning about dependent types more precise, typing judgments use a context C that keeps track of constraints known to hold during execution of the programs. For a control-flow graph (CFG) node η, $C(\bullet\eta)$ signifies constraints that hold before executing node η. Hence, $C(\bullet\eta) \Rightarrow P$ means that these constraints entail proposition P. Context C is determined by the Hoare rules in [15], straightforwardly extended to also reason about symbols r'. The form and meaning of type judgments for expressions is standard.

The most interesting typing rules of SecVerilogLC are for assignments, shown in Figure 7. In the rule T-AsgnCom for assignment to combinational variables, assignment $:=_\eta$ is annotated with its CFG node η, and the constraint context C must entail that the label of the expression from which w is assigned (τ) can flow into the label of w (that is, $\Gamma(w)$). The pc label is used to prevent implicit flows in the standard way [8]. For brevity, we show only the

[1] The label pc is an abstraction used by the type system, and is *not* the program counter register. We use pc to refer to the label and `pc` to refer to the register.

cases in which dependent labels are not self-referential, hence the premise that w is not a free variable in its own type: $w \notin fv(\Gamma(w))$. In SecVerilogLC, self-referential labels are supported in the same way as they are in SecVerilog [15]. Cyclic dependencies through dependent labels are prevented by a well-formedness requirement for type environments as in SecVerilog [14].

$$\frac{\begin{array}{c} w \notin fv(\Gamma(w)) \\ \Gamma \vdash e : \tau \qquad C(\bullet\eta) \Rightarrow \tau \sqcup pc \sqsubset \Gamma(w) \end{array}}{\Gamma, C, pc \vdash w :=_{\eta} e}\ \textsc{T-AsgnComb}$$

$$\frac{\begin{array}{c} \Gamma \vdash e : \tau \qquad r \notin fv(\Gamma(r)) \qquad \vec{r} = fv(\Gamma(r)) \\ \tau' = \Gamma(r)\{\vec{r'}/\vec{r}\} \qquad C(\bullet\eta) \Rightarrow \tau \sqcup pc \sqsubset \tau' \end{array}}{\Gamma, C, pc \vdash r :=_{\eta} e}\ \textsc{T-AsgnSeq}$$

Figure 7: Interesting typing rules for commands.

Rule T-AsgnSeq for sequential assignments differs subtly from the rule for combinational assignments. Assignments to sequential variables describe the values which they store at the start of the next clock edge. Therefore, the label of the expression τ should be permitted to flow into the label of the sequential variable during the next clock cycle τ'. This new label may depend on the new values of other sequential variables. Since the actual values of variables are not known a priori, τ' is determined by simultaneously substituting each sequential variable in $\Gamma(r)$ with its corresponding next-cycle value symbol r'. Thus, $\tau' = \Gamma(r)\{\vec{r'}/\vec{r}\}$, and $C(\bullet\eta)$ must contain sufficient facts to fulfill the proof obligation that this flow is safe.

3. EVALUATION

3.1 Methodology

We evaluated SecVerilogLC by using it to type-check the processing pipeline of a processor that implements a subset of the MIPS ISA. The processor has four cores that communicate over a ring network. Each processor has a five-stage bypassing pipeline with private data and instruction caches. The processor was functionally evaluated with 166 unit test vectors. Our processor supports a privileged kernel mode and a unprivileged user mode.

The security policy protects the integrity of the kernel mode and ensures that the only point of entry into kernel mode is the SYSCALL instruction. The security policy only requires the two security levels discussed in examples throughout this paper: $T \sqsubseteq U$. The implementation is similar to the one described in Section 2.1. The trusted (T) status register mode indicates the current privilege mode. We chose an implementation with a single mode register rather than one which uses per-stage mode bits, as this most closely resembles conventional implementations. The pipeline registers, GPRs, and the PC have labels that depend on the value of mode; the label is T when mode is 0 and it is U when mode is 1. Note that no previously proposed security type system for HDLs [5, 6, 15] can support mode changes both securely and correctly.

Our prototype uses explicit downgrading in three places. The first is to modify the trusted (T) mode bit during the SYSCALL instruction, which takes place in user mode. During a SYSCALL, a control signal indicating that mode should change is propagated from the decode stage, through the intermediate stages, to the writeback stage, where the mode switch actually takes place. The input to mode depends on outputs from the memory stage, which are untrusted since the processor is in user mode. The statement that downgrades the mode bit is guarded by a branch (i.e., access control) that checks for the control signal that indicates that a SYSCALL is actually taking place. This access control permits entry to kernel mode only through the SYSCALL instruction. The other two uses of downgrading permit the contents of two general purpose registers to remain intact on a SYSCALL so they can be used as arguments to the system call handler. The other GPRs are cleared.[2]

3.2 Type Checking

Our labeled processor as described above passes type-checking suggesting that the implementation conforms to the specified security policy. Before building the labeled implementation, we first designed a comparable processor that is not labeled or type-checked, but which we felt was secure. We use this unlabeled implementation for comparisons pertaining to annotation burden and overhead in the final, synthesized hardware.

The process of labeling and type-checking the processor revealed a potential security vulnerability that we did not identify while implementing the unlabeled version of the processor. We implemented checks on an update to the pc register to ensure that the program counter is set only to a predefined constant when there is a switch to the kernel mode, and restored to the epc register on a return to the user mode. However, the update to pc was also controlled by an enable signal that depends on the fetch (F) stage; the design allowed the F stage to stall the pc update on cases such as an instruction cache miss. Unfortunately, because this enable signal depends on the current mode, it means that untrusted instructions from the F stage may possibly delay updates to the pc value. More alarmingly, our initial implementation allowed the delayed enable signal to prevent the update to the pc entirely while still escalating the privilige level. To remove this security vulnerability, our labeled implementation *always* updates the pc register on label changes. This is both secure and functionally correct because any stalls from the F stage during a label change are spurious. For the evaluation, we modified the unlabeled processor so that it is secure and the comparisons are fair.

The prototype shows that the restrictions placed on label changes by SecVerilogLC are precise; the designer only needs to take action on label changes that are dangerous. For example, the type system requires that during a SYSCALL instruction, each GPR is explicitly cleared, or explicitly downgraded. However, no change is required on the SYSRET instruction. On the SYSCALL instruction, the labels of the GPRs change from U to T. Since the new levels indicate a higher level of integrity than can accurately be attributed to the old values stored in the GPRs, the values must be updated. However, on the SYSRET instruction, the labels change from T to U and the GPR contents can remain—it is permissible (though conservative) for trustworthy data to be considered untrusted. Implicit downgrading is not precise in this sense as it requires dependently typed registers to be cleared on *any* change to their labels [14] rather than just changes that correspond to label downgrading.

3.3 Overhead

To evaluate the overhead of using the proposed type system in terms of programmer effort as well as the area and the clock period of the hardware design, we compare two implementations of the processor pipeline: one type-checked with SecVerilogLC, and one that we believe is secure, but that has not been type-checked. We refer to this second implementation as the baseline processor.

[2] A more realistic implementation of the SYSCALL instruction might save the contents of the GPRs in the region of memory reserved for storing context. The corresponding SYSRET instruction would then restore this saved context into the GPRs before returning to user space. We expect that our type system would also support such an implementation.

The baseline processor pipeline is implemented in 1,487 lines of code. To support the syntax of SecVerilogLC and pass type checking, 271 lines were changed. Most line changes (257) were to add `com/seq` annotations or labels. We believe that the `com/seq` annotations could be added automatically as it is easy to identify if a variable is combinational or sequential in Verilog. Most of these lines require both a `com/seq` annotation and a label, so the annotation burden is only slightly increased from that of SecVerilog [15]. Only a total of 14 lines (less than 1% of the code) were added to handle explicit downgrades or make invariants explicit for the type system. Some code was refactored to convince the type system that certain statments are true when the built-in analysis cannot automatically infer the invariants. These cases include a change to the `pc` update logic and a change to register file writes.

To evaluate the overhead that SecVerilogLC adds to the area and the clock frequency of the hardware, we used our SecVerilog-to-Verilog compiler for both designs. We synthesized both designs with Synopsys Design Compiler using the TSMC 65nm process and a target clock period of 2ns. Both reach the target clock period. The area of the baseline design is $29,638\mu m^2$ and the area of the labeled design is $29,843\mu m^2$, an area overhead of about 0.7%. There are two main sources of area overhead. The first is the addition of muxes for checking labels. This is the true overhead of using the proposed type system. A second source of area overhead is an implementation artifact that can be fixed. In our current implementation of SecVerilogLC, registers generate flip-flops without built-in enable signals, whereas the baseline processor uses multiple types of flip-flops. For this reason, the baseline processor maps more efficiently to standard cells. SecVerilogLC can be improved by adding a syntax for describing registers with enable signals.

4. RELATED WORK

Information flow control was first applied to hardware at the gate level by GLIFT [7, 10, 11]. The earliest GLIFT [11] approach inserts additional run-time logic and has prohibitive overhead in terms of performance, area, and energy. Later work applies GLIFT to simulations of circuits [10] by enumerating over the state space. Because the state space can be large for practical designs, prior work checks only the subset of that state space that is reachable by a particular piece of software. Compared to run-time IFC, SecVerilogLC provides static IFC at design time that checks information flow for all possible cases.

Sapper [5], Caisson [6], and SecVerilog [15] all apply information flow type systems at the HDL level. Caisson and Sapper support sharing among security levels through nested states. Both languages describe hardware as a composition of FSMs and resemble continuation-passing-style languages. Sapper and Caisson support transitions from high states (in which high or low variables may be modified) to low states (in which only low variables may be modified) through the use of linear continuations as proposed by Zdancewic et al. [13]. In practice, nested states can be used to describe controller FSMs that multiplex between physically separate modules for storing trusted and untrusted data. However, neither Sapper nor Caisson provide a mechanism for reusing storage elements for both trusted and untrusted data, so both languages require duplication of state elements. SecVerilogLC enables secure sharing of hardware at fine granularity without duplication of storage elements. This work builds on the dependent types in SecVerilog [15], but redesigns the language to securely enable flexible sharing policies by making combinational/sequential variables and clock cycles explicit. We expect that this language enforces the same security property as SecVerilog [15], i.e., observational determinism [13].

SecVerilogLC is the first to securely support mutable dependent labels in a language for information flow control in either hardware or software. Zhang et al. [16] proposed a software language that supports dynamic IFC labels in software programs. However, labels in this language are immutable, whereas SecVerilogLC supports mutable labels. While our language is designed for hardware, we note that the proposed approach can also be applied to support mutable dependent types in imperative software languages for IFC. Condit et al. [2] proposed Deputy, a language with a dependent type system for writing safe C programs. The authors use this language to prevent buffer overflows. The approach of Condit et al.has not been used for information flow control or for hardware.

5. CONCLUSION

This paper proposes novel extensions to secure HDLs. This new type system is expressive enough to allow efficient hardware with complex, fine-grained sharing among security levels while ensuring the security of such sharing. We evaluated this language by designing a MIPS processor with support for privilege levels, user and kernel modes, and found that the proposed HDL could express and verify efficient processor design that was not supported by the state-of-the-art secure HDL. Security type checking also revealed a security vulnerability in our prototype that we did not expect.

6. ACKNOWLEDGMENTS

This work was partially sponsored by NSF grant CNS-1513797 and NASA grant NNX16AB09G. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of NSF or NASA.

7. REFERENCES

[1] Bob Bentley. Validating the Intel Pentium 4 Microprocessor. In *Proceedings of the Design Automation Conference (DAC)*, 2001.

[2] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent Types for Low-level Programming. ESOP'07.

[3] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 1976.

[4] Advanced Micro Devices. Revision Guide for AMD Athlon 64 and AMD Opteron Processors, 2005.

[5] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A Language for Hardware-level Security Policy Enforcement. In *ASPLOS*, 2014.

[6] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A Hardware Description Language for Secure Information Flow. In *PLDI*, 2011.

[7] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Theoretical Analysis of Gate Level Information Flow Tracking. In *DAC*, 2010.

[8] Andrei Sabelfeld and Andrew C. Myers. Language-based Information-flow Security. *IEEE Journal on Selected Areas in Communications*, 2006.

[9] Andrei Sabelfeld and David Sands. Declassification: Dimensions and Principles. *JCS*, 2009.

[10] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security. In *ISCA*, 2011.

[11] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete Information Flow Tracking from the Gates Up. In *ASPLOS*, 2009.

[12] Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning, 2009.

[13] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *CSFW*, 2003.

[14] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for efficient control of timing channels. Technical Report http://hdl.handle.net/1813/36274, Cornell University, 2014.

[15] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *ASPLOS*, 2015.

[16] Lantien Zheng and Andrew C. Myers. Dynamic Security Labels and Static Information Flow Control. In *IJIS*, 2007.