Security Results for SIRRTL, A Hardware Description Language for Information Flow Security.

Andrew Ferraiuolo [*]

Cornell University

Abstract

This document establishes security results for SIRRTL, a secure variant of the FIRRTL intermediate language. We developed ChiselFlow, a variant of the Chisel hardware design language [1] for information flow security. ChiselFlow extends Chisel, a hardware description language embedded in Scala. ChiselFlow allows the hardeware designer to describe security policies about the hardware that are checked at design-time. Much like Chisel, ChiselFlow gains much of the expressive power of the rich host language, Scala. However, security enforcement is done by a small intermediate language called SIRRTL, so the trusted component of ChiselFlow is small. ChiselFlow emits SIRRTL, a variant of the FIRRTL intermediate language augmented with an information flow type system. ChiselFlow supports security policies that depend on the run-time values of signals, though these policies are checked purely at design-time by SIRRTL. In this document, we prove that well-typed SIRRTL modules enforce a timing safe variant of noninterference. We constructed the HyperFlow processor using ChiselFlow thereby establishing high assurance in the implementation of the processor.

1 Language

1.1 Syntax

Figure 1 shows a core syntax of SIRRTL that we use to establish security results. SIRRTL labels, ℓ, are pairs of confidentiality and integrity components (c, i). Label components include atomic security levels n which form a lattice. The notation $p_1 \geq p_2$ means that p_1 has higher authority than p_2. For confidentiality, $c_1 \geq c_2$ means that c_1 is more secret than c_2. Dually, for integrity, $i_1 \geq i_2$ means that i_2 is more trusted than i_2. For either confidentiality or integrity components, $p_1 \wedge p_2$ denotes the join of p_1 and p_2, and $p_1 \vee p_2$ denotes their meet. The greatest and least components are $\top$ and $\bot$ respectively. The syntax of label components also includes functions f that are fully

[*]The author is now at Google.

applied to some number of free variables in the hardware module $\vec{x}$. Components of this form can be used to express dependent labels that change at run time based on the values of signals. Dependent labels are important for the description of efficient hardware designs that allow the hardware to be shared by different security domains at run-time. Dependent labels of this form are similar to those in SecVerilog [4]. The lattice over label components is lifted to a lattice over security labels which is defined in Figure 2.

$n \in \mathcal{N}$	atomic principals
$x \in \mathcal{V}$	variable names
$\bar{x} \in \overline{\mathcal{V}}$	next-cycle symbols
$v \in \mathbb{N}$	integers

$$\begin{aligned}
i, c, p &::= n \mid \top \mid \bot \mid p \wedge p \mid p \vee p \mid f(\vec{x}) \\
\ell &::= (p, p) \\
e &::= v \mid x \mid \bar{x} \mid e \oplus e \mid \texttt{decl}(e, \ell) \mid \texttt{endo}(e, \ell) \\
\texttt{Prog}, s &::= \texttt{skip} \mid s; s \mid \texttt{when}(e)\ s\ \texttt{else}\ s\ \mid x \Leftarrow e
\end{aligned}$$

Figure 1: SIRRTL core syntax.

$$\begin{aligned}
(c_1, i_1) \sqcup (c_2, i_2) &\triangleq (c_1 \wedge c_2, i_1 \vee i_2) \\
(c_1, i_1) \sqcap (c_2, i_2) &\triangleq (c_1 \vee c_2, i_1 \wedge i_2) \\
(c_1, i_1) \sqsubseteq (c_2, i_2) &\iff c_2 \geq c_1 \text{ and } i_1 \geq i_2
\end{aligned}$$

Figure 2: Lattice over labels.

The syntax of expressions is mostly standard. Values v are finite bit-vectors. Variables, denoted x, represent sequential variables that define registers. For simplicity, the formal syntax of SIRRTL omits combinational variables, though in doing so it loses no expressive power – combinational variables can simply be replaced with the expressions that assign their values. The implementation of SIRRTL includes combinational variables. The syntax $\bar{x}$ represents a special symbol reserved for storing the next-cycle valuation of x. The symbol $\bar{x}$ cannot be written by the programmer; it is an auxilary symbol used in the semantics and typing judgments to capture delayed updates to the sequential variables. Binary operators are denoted $e_1 \oplus e_2$. The syntax $\texttt{decl}(e, \ell)$ and

$\texttt{endo}(e, \ell)$ respectively express declassification and endorsement of the expression e to the security level ℓ. Declassification and endorsement respectively relax the confidentiality and integrity level of the expression. Downgrades are controlled by the type system so that they enforce non-malleable information flow control [2]. The syntax of statements s is entirely standard except that `when` denotes a conditional statement. Programs, written `Prog` are single commands.

Our full implementation of SIRRTL (and ChiselFlow) also securely supports record types (bundles), arrays, synchronous and asynchronous memories, module declarations and instantiations, and purely combinational wires, though the typing rules for these features do not fundamentally change the design of the type system, so we omit them to simplify the proofs.

1.2 Semantics

The big-step semantics of expressions, shown in Figure 1 is entirely standard aside from the rule for $\bar{x}$ which is similar to the evaluation of a conventional variable. The small-step semantics of statements, shown in Figure 4 is mostly standard. Hardware states, σ, are mappings from variables and next-cycle symbols to bit-vectors. Formally, states range over $(\mathcal{V} + \overline{\mathcal{V}}) \to \mathbb{N}$, or isomorphically, $(\mathcal{V} \to \mathbb{N}) \times (\overline{\mathcal{V}} \to \mathbb{N})$; they are are pairs including a function from variables to values and a function from next-cycle symbols to values. For simplicity, we use $\sigma(x)$ and $\sigma(\bar{x})$ to denote the valuation of either a variable or next-cycle symbol in σ. Assignments to a variable x cause updates to $\bar{x}$ rather than x to model the fact that updates to registers are delayed until the start of the clock cycle.

Variables are updated by the program semantics. As the program is evaluated, the program semantics constructs traces that have the syntax

$$t ::= \epsilon \mid (T, \sigma) \mid t_1; t_2$$

where T is a clock cycle counter represented by a positive integer. The program semantics operates on configurations of the form $\langle T, \sigma, s, t \rangle$. Transitions between configurations are denoted by $\to_{\mathcal{S}}$, in which $\mathcal{S}$ represents the statement that is the initial syntactic description of the program.

The rule, $S - Tick$ applies when the program has been fully evaluated to `skip`. This rule updates the contents of the registers on the new clock cycle. All variables $x_1, ..., x_n$ in the program $\mathcal{S}$, are updated to their corresponding next-cycle valuations stored in $\sigma x_1, ..., \sigma x_n$. The cycle counter is incremented, and a trace event that includes the clock cycle number and the state at the start of the cycle is emitted. The program re-starts evaluation from $\mathcal{S}$ to compute the values for the next cycle. The rule, $S - Eval$, applies when s is not `skip`, and it simply updates the statement and state according to the semantics of statements. This semantics is similar to that of the variant of SecVerilog with dependent types that are checked purely at compile-time [3].

1.3 Type Rules

Type environments Γ map variables and next-cycle symbols to labels. Because label components in SIRRTL include functions of program variables, the valuation of com-

$$\frac{}{\langle \sigma, n\rangle \Downarrow n}\ \textsc{S-Const} \qquad \frac{\sigma(x) = n}{\langle \sigma, x\rangle \Downarrow n}\ \textsc{S-Var} \qquad \frac{\sigma(\bar{x}) = n}{\langle \sigma, \bar{x}\rangle \Downarrow n}\ \textsc{S-VarNext}$$

$$\frac{\langle \sigma, e_1\rangle \Downarrow n_1 \qquad \langle \sigma, e_2\rangle \Downarrow n_2 \qquad n = n_1 \oplus n_2}{\langle \sigma, e_1 \oplus e_2\rangle \Downarrow n}\ \textsc{S-Op} \qquad \frac{\langle \sigma, e\rangle \Downarrow n}{\langle \sigma, \mathtt{decl}(e,\ell)\rangle \Downarrow n}\ \textsc{S-Decl}$$

$$\frac{\langle \sigma, e\rangle \Downarrow n}{\langle \sigma, \mathtt{endo}(e,\ell)\rangle \Downarrow n}\ \textsc{S-Endo}$$

Figure 3: Expression semantics.

$$\begin{aligned}
\langle \sigma, x \Leftarrow e\rangle &\longrightarrow \langle \sigma[\bar{x} \mapsto \sigma(x)], \mathtt{skip}\rangle & & \\
\langle \sigma, \mathtt{skip}; s\rangle &\longrightarrow \langle \sigma, s\rangle & & \\
\langle \sigma, s_1; s_2\rangle &\longrightarrow \langle \sigma', s_1'; s_2\rangle & & (\text{if } \langle \sigma, s_1\rangle \longrightarrow \langle \sigma', s_1'\rangle) \\
\langle \sigma, \mathtt{when}(e)\ s_1\ \mathtt{else}\ s_2\rangle &\longrightarrow \langle \sigma, s_1\rangle & & (\text{if } \neg(\langle \sigma, e\rangle \Downarrow 0)) \\
\langle \sigma, \mathtt{when}(e)\ s_1\ \mathtt{else}\ s_2\rangle &\longrightarrow \langle \sigma, s_2\rangle & & (\text{if } \langle \sigma, e\rangle \Downarrow 0)
\end{aligned}$$

Figure 4: Command semantics.

ponents and labels both depend on the state, σ. We use the meta-syntax $C(p, \sigma)$ and $\mathcal{T}(\ell, \sigma)$ respectively to denote the valuations of components p and labels ℓ respectively. As in SecVerilog [4], the type rules apply only for type environments that are well-formed. In a well-formed type environment 1) no label depends on variables with more restrictive labels, and 2) variables that appear in labels cannot depend on labels. The second condition is more restrictive than the one in SecVerilog, which allows variables to have labels that depend on themselves. Let $fv(\ell)$ denote the free variables in ℓ. Formally, a type-environment is well-formed, written $\vdash \Gamma$ when,

Definition 1 (Well-Formedness of Environments)

$$\begin{aligned}
&\forall x \in \mathcal{V}.(\forall \sigma.\forall x' \in fv(\Gamma(x)). \\
&\qquad\qquad\qquad \mathcal{T}(\Gamma(x'), \sigma) \sqsubseteq \mathcal{T}(\Gamma(x), \sigma) \\
&\qquad\qquad\qquad \wedge fv(\Gamma(x')) = \emptyset)
\end{aligned}$$

Type judgements for expressions have the form $\Gamma; pc \vdash e : \ell$ which means that e is well-typed in typing environment Γ under program counter label pc. The type rules for expressions are mostly standard aside from next-cycle valuations of variables and for downgrades. The rule $T - NextVar$ computes the valuation of the label x on the following clock cycle by substituting each occurrence of a free variable in the label with its next-cycle symbol.

$$\frac{\begin{array}{c}\{x_1, ..., x_n\} = vars(\mathcal{S}) \\ \sigma' = \sigma[x_1 \mapsto \sigma(\bar{x_1})]...[x_n \mapsto \sigma(\bar{x_n})]\end{array}}{\langle T, \sigma, \texttt{skip}, t\rangle \rightarrow_{\mathcal{S}} \langle T+1, \sigma', \mathcal{S}, t; (T+1, \sigma')\rangle}\ \textsc{S-Tick}$$

$$\frac{s \neq \texttt{skip} \qquad \langle \sigma, s\rangle \rightarrow \langle \sigma', s'\rangle}{\langle T, \sigma, s, t\rangle \rightarrow_{\mathcal{S}} \langle T, \sigma', s', t\rangle}\ \textsc{S-Eval}$$

Figure 5: Program semantics.

Because labels in SIRRTL can depend on the run-time values of signals, SIRRTL relies on a static program analysis that that models the run-time behavior of the hardware. The notation $P(\eta) \Rightarrow Q$ means that the program analysis has derived that the proposition Q holds before executing control-flow graph node η.

The rules for downgrades enforce non-malleable information flow control and are similar to those used to enforce the same security condition in a recent functional programming language [2]. These rules require auxiliary definitions on labels. In particular, $(^{\rightarrow}\ell)$ is defined $(c, i)^{\rightarrow} \triangleq (c, \top)$, and it computes a label that has the confidentiality of ℓ, but is fully trusted. Similarly, $(^{\leftarrow}\ell)$ is defined $(c, i)^{\leftarrow} \triangleq (\bot, i)$ and it computes a label that has the integrity of ℓ but is fully public. The view of a label, $\Delta\ell)$ converts the integrity of a label to a confidentiality component, and is defined by $\Delta c, i) \triangleq (^{\rightarrow}i, \top)$. Dually, the voice of a label, $\nabla\ell)$ converts a confidentiality component to an integrity component, and it is defined by $\nabla c, i) \triangleq (\bot, c)$.

The typing rules for commands are standard except that the rule for assignments ensures that the expression can flow to the label of the *next-cycle-valuation* of the variable that is assigned.

2 Security Results

We now prove security results about SIRRTL. Namely, that well-typed hardware modules that do not contain downgrades enforce a timing-safe variant of observational determinism. The typing rules for downgrades in SIRRTL also resemble those from a recent software type system that enforces a security condition in the presence of downgrades called non-malleable information flow control [2]. We first define low-equivalence of hardware states before stating the main theorems. We define a full-evaluated security label as one which does not contain sub-components of the form $f(x)$. When two states, σ_1 and σ_2 are low-equivalent to an attacker at fully-evaluated security label L we write $\sigma_1 \approx_L \sigma_2$. Low-equivalence at level L is defined as follows,

$$\begin{aligned}\sigma_1 \approx_L \sigma_2 \triangleq \forall x \in \mathcal{V}.(&\mathcal{T}(\Gamma(x), \sigma_1) \sqsubseteq L \iff \mathcal{T}(\Gamma(x), \sigma_2) \sqsubseteq L) \\ &\wedge \mathcal{T}(\Gamma(x), \sigma_1) \sqsubseteq L \Longrightarrow \sigma_1(x) = \sigma_2(x)\end{aligned}$$

Traces are low-equivalent, written $t_1 \approx_L t_2$ when for each element of the trace, the corresponding clock cycle counters are equal, and the states are low-equivalent.

$$\textsc{T-Const}\ \frac{}{\Gamma; pc \vdash n : \bot} \qquad \textsc{T-Var}\ \frac{\Gamma(x) = \ell}{\Gamma; pc \vdash x : \ell}$$

$$\textsc{T-NextVar}\ \frac{\Gamma(x) = \ell \qquad \{x_1, ..., x_n\} = fv(\Gamma(x))}{\Gamma; pc \vdash \bar{x} : \ell[x_1 \mapsto \bar{x_1}]...[x_n \mapsto \bar{x_n}]} \quad \textsc{T-Op}\ \frac{\begin{array}{c}\Gamma; pc \vdash e_1 : \ell_1 \\ \Gamma; pc \vdash e_2 : \ell_2\end{array}}{\Gamma; pc \vdash e_1 \ \oplus \ e_2 : \ell_1 \sqcup \ell_2}$$

$$\textsc{T-decl}\ \frac{\begin{array}{c}\Gamma; pc \vdash e : \ell' \qquad P(\eta) \Rightarrow \ell^{\leftarrow} = \ell'^{\leftarrow} \wedge pc \sqsubseteq \ell \\ P(\eta) \Rightarrow \ell'^{\rightarrow} \sqsubseteq \ell^{\rightarrow} \sqcup \Delta(\ell' \sqcup pc)\end{array}}{\Gamma; pc \vdash \texttt{decl}(e, l) : l}$$

$$\textsc{T-endo}\ \frac{\begin{array}{c}\Gamma; pc \vdash e : \ell' \qquad P(\eta) \Rightarrow \ell^{\rightarrow} = \ell'^{\rightarrow} \wedge pc \sqsubseteq \ell \\ P(\eta) \Rightarrow \ell'^{\leftarrow} \sqsubseteq \ell^{\leftarrow} \sqcup \nabla(\ell' \sqcup pc)\end{array}}{\Gamma; pc \vdash \texttt{endo}(e, l) : l}$$

Figure 6: Type Rules: Expressions.

We now state the observational determinism theorem.

Theorem 1 (Observational Determinism) *If Γ is a type environment, s is a statement that does not contain downgrades, pc is a label, L is a fully-evaluated security label, and σ_1 and σ_2 are states, then*

$$\begin{aligned}&\vdash \Gamma \wedge \Gamma; pc \vdash s \wedge \sigma_1 \approx_L \sigma_2 \wedge \\ &\qquad\qquad\qquad \langle 0, \sigma_1, s, \epsilon\rangle \longrightarrow_{\mathcal{S}} \langle n_1, \sigma_1', s, t_1\rangle \wedge \\ &\qquad\qquad\qquad \langle 0, \sigma_2, s, \epsilon\rangle \longrightarrow_{\mathcal{S}} \langle n_2, \sigma_2', s, t_2\rangle \\ &\qquad\qquad\qquad \Longrightarrow \sigma_1' \approx_L \sigma_2' \wedge t_1 \approx_L t_2\end{aligned}$$

Before proving the observational determinism result, we first prove some useful lemmas. The first lemma states that low expressions other than downgrades do not contain high variables.

Lemma 1 *For all fully-evaluated security labels L, states σ, and expressions e that do not contain downgrades,*

$$\begin{aligned}&\vdash \Gamma \wedge \Gamma \vdash e : \ell \wedge \mathcal{T}(\ell, \sigma) \sqsubseteq L \\ &\qquad\qquad\qquad \Longrightarrow \forall x \in vars(e). \mathcal{T}(\Gamma(x), \sigma) \sqsubseteq L\end{aligned}$$

Proof. By induction on the structure of expressions. □

The next lemma states that low labels evaluate to the same concrete label in low-equivalent states.

$$\frac{}{\Gamma; pc \vdash \texttt{skip}}\ \textsc{T-Skip} \qquad \frac{\Gamma; pc \vdash s_1 \qquad \Gamma; pc \vdash s_2}{\Gamma; pc \vdash s_1; s_2}\ \textsc{Seq}$$

$$\frac{\begin{array}{c}\Gamma; pc \vdash e : \ell \\ \Gamma; pc \sqcup \ell \vdash s_t \\ \Gamma; pc \sqcup \ell \vdash s_f\end{array}}{\Gamma; pc \vdash \texttt{when}(e)\ s_t\ \texttt{else}\ s_f}\ \textsc{T-When}$$

$$\frac{\begin{array}{c}\Gamma; pc \vdash e : \ell \qquad \{x_1, ..., x_n\} = fv(\Gamma(x)) \\ \ell' = \Gamma(x)[x_1 \mapsto \bar{x_1}]...[x_n \mapsto \bar{x_n}] \\ P(\eta) \Rightarrow \ell \sqcup pc \sqsubset \ell'\end{array}}{\Gamma; pc \vdash x \Leftarrow e}\ \textsc{T-Assign}$$

Figure 7: Type Rules: Statements.

Lemma 2 *If Γ is a type environment, ℓ is a label, L is a fully-evaluated label, and σ_1 and σ_2 are states, then*

$$\sigma_1 \approx_L \sigma_2 \wedge \vdash \Gamma \wedge \mathcal{T}(\ell, \sigma_1) \sqsubseteq L \\ \Longrightarrow \mathcal{T}(\ell, \sigma_1) = \mathcal{T}(\ell, \sigma_2)$$

Proof. Let $\ell = (c, i)$ and $L = (c', i')$. By the definition of $\sqsubseteq$, $c' \geq C(c, \sigma_1)$ and $C(i, \sigma_1) \geq i'$. We show that $C(c, \sigma_1) = C(c, \sigma_2)$ by induction on the structure of c. The argument that $C(i, \sigma_1) = C(i, \sigma_2)$ is exactly dual, and the result that $\mathcal{T}(\ell, \sigma_1) = \mathcal{T}(\ell, \sigma_2)$ follows directly.

Case $c = n, c = \top, c = \bot$: trivial.

Case $c = f(\vec{x})$: Let x_i be some variable in $vecx$. By assumption, $\sigma_1 \approx_{(c',i')} \sigma_2$. By $\vdash \Gamma$, $\mathcal{T}(\Gamma x_i, \sigma_1) \sqsubseteq \ell$ and $\mathcal{T}(\Gamma x_i, \sigma_2) \sqsubseteq \ell$ By transitivity of $\sqsubseteq$, $\mathcal{T}(\Gamma(x_i), \sigma_1) \sqsubseteq L$ and $\mathcal{T}(\Gamma(x_i), \sigma_2) \sqsubseteq L$. By definition of $\approx_L$, $\sigma_1(x_i) = \sigma_2(x_i)$. The same is true for all other variables in $\vec{x}$, and so $C(f(\vec{x}), \sigma_1) = C(f(\vec{x}), \sigma_2)$

Case $p_1 \wedge p_2$: $\mathcal{T}(p_1 \wedge p_2, \sigma_1) = \mathcal{T}(p_1, \sigma_1) \wedge \mathcal{T}(p_2, \sigma_2)$. By assumption $\mathcal{T}(p_1 \wedge p_2, \sigma_1) \geq c'$, hence $\mathcal{T}(p_1, \sigma_1) \geq c'$ and $\mathcal{T}(p_2, \sigma_2) \geq c'$. By induction hypothesis, $\mathcal{T}(p_1, \sigma_1) = \mathcal{T}(p_1, \sigma_2)$ and $\mathcal{T}(p_1, \sigma_1) = \mathcal{T}(p_1, \sigma_2)$. Hence, $\mathcal{T}(p_1 \wedge p_2, \sigma_1) = \mathcal{T}(p_1 \wedge p_2, \sigma_2)$

Case $p_1 \vee p_2$: Similar to the case $f(\vec{x})$, by inspection of the free variables in $p_1 \vee p_2$. □

The next lemma states that low expressions evaluate to the same value in low-equivalent states.

Lemma 3 *If Γ is a type environment, e is an expression, pc is a label, L is a fully-evaluated security label, and σ_1 and σ_2 are states, then*

$$\sigma_1 \approx_L \sigma_2 \wedge \vdash \Gamma \wedge \Gamma \vdash e : \ell \wedge \mathcal{T}(\ell, \sigma_1, \sqsubseteq)L \wedge$$
$$\langle \sigma_1, e\rangle \Downarrow n_1 \wedge \langle \sigma_2, e\rangle \Downarrow n_2$$
$$\Longrightarrow n_1 = n_2$$

Proof. By Lemma 2, $\mathcal{T}(\ell, \sigma_2) = \mathcal{T}(\ell, \sigma_1) \sqsubseteq L$. By Lemma 1, for all x in e, $\mathcal{T}(x, \sigma_1) \sqsubseteq L$ and $\mathcal{T}(x, \sigma_2) \sqsubseteq L$. Since $\sigma_1 \approx_L \sigma_2$, $\sigma_1(x) = \sigma_2 x$. Since this is true for all x in e, $n_1 = n_2$. □

We now prove that SIRRTL enforces observational determinism for individual statements.

Theorem 2 (Single-Statement Obsevational Determinism) *If Γ is a type environment, s is a statement that does not contain downgrades, pc is a label, L is a fully-evaluated security label, and σ_1 and σ_2 are states, then*

$$\vdash \Gamma \wedge \Gamma; pc \vdash s \wedge \sigma_1 \approx_L \sigma_2 \wedge$$
$$\langle \sigma_1, s\rangle \longrightarrow_* \langle \sigma_1', skip\rangle \wedge \langle \sigma_2, s\rangle \longrightarrow_* \langle \sigma_2', skip\rangle$$
$$\Longrightarrow \sigma_1' \approx_L \sigma_2'$$

Proof. *Case $s_1; s_2$:* If s_1 =`skip`, then $\langle \sigma_1, s_1; s_2\rangle \to \langle \sigma_1', s_2$ and $\langle \sigma_2, s_1; s_2\rangle \to \langle \sigma_2', s_2$ so $\sigma_1 = \sigma_1'$ and $\sigma_2 = \sigma_2'$. By assumption, $\sigma_1 \approx_L \sigma_2$ and so $\sigma_1' \approx_L \sigma_2'$. By the induction hypothesis, execution of s_2 from σ_1' and σ_2' results in low-equivalent states.

If s_1 ≠`skip`, then $\langle \sigma_1, s_1; s_2\rangle \to \langle \sigma_1'', s1'; s_2$ and $\langle \sigma_2, s_1; s_2\rangle \to \langle \sigma_2'', s1''; s_2$ where $\langle \sigma_1, s_1\rangle \to \langle \sigma_1'', s1'$ and $\langle \sigma_2, s_1\rangle \to \langle \sigma_2'', s1''$. By the induction hypothesis $\sigma_1'' \approx_L \sigma_2''$ But SIRRTL statements clearly do not diverge, so for some σ_1''' and σ_2''', $\langle \sigma_1', s'1 \to_* \langle \sigma_1''', \texttt{skip}\rangle$ and $\langle \sigma_2', s'1 \to_* \langle \sigma_2''', \texttt{skip}\rangle$. By the induction hypothesis $\sigma_1''' \approx_L \sigma_2'''$. And since s_2 is eventually evaluated from σ_1''' and σ_2''' in two executions, by the induction hypothesis, $\sigma_1' \approx_L \sigma_2'$

Case $x \Leftarrow e$: We have $\langle \sigma_1, x \Leftarrow e\rangle \to \langle \sigma_1[\bar{x} \mapsto n_1], \texttt{skip}\rangle$ and $\langle \sigma_2, x \Leftarrow e\rangle \to \langle \sigma_2[\bar{x} \mapsto n_2], \texttt{skip}\rangle$ Let $\ell' = ell[x_1 \mapsto \bar{x_1}]...[x_n \mapsto \bar{x_n}]$. We first consider the case in which $\mathcal{T}(\ell', \sigma_1) \sqsubseteq L$. By assumption, $\sigma_1 \approx_L \sigma_2$ and by Lemma 2, $\mathcal{T}(\ell', \sigma_2) = \mathcal{T}(\ell', \sigma_1) \sqsubseteq L$ By T-ASSIGN, $\Gamma; pc \vdash e : \ell$ and $\ell \sqcup pc \sqsubseteq \ell'$. By lattice properties, $\ell \sqsubseteq \ell'$ and $\ell \sqsubseteq L$. By Lemma 3, $n_1 = n_2$. Because $\sigma_1[\bar{x} \mapsto n_1] = \sigma_1'$ and σ_1 agreee on values of all variables other than $\bar{x}$, $\sigma_1 \approx_L \sigma_1'$. Similarly, $\sigma_2 \approx_L \sigma_2'$ and by transitivity, $\sigma_2' \approx_L \sigma_1'$.

We now consider the case in which $\mathcal{T}(\ell', \sigma_1) \not\sqsubseteq L$ We first show that $\mathcal{T}(\ell', \sigma_2) \not\sqsubseteq L$ If $\mathcal{T}(\ell', \sigma_2) \sqsubseteq L$, then by Lemma 2, $\mathcal{T}(\ell', \sigma_1) \sqsubseteq L$ which violates our assumption. By $\vdash \Gamma$, $\bar{x} \notin fv(Ga(\bar{x}))$. Because σ_1 and $\sigma_1[\bar{x} \mapsto n_1] = \sigma_1'$ agree on valuations of all variables other than $\bar{x}$, $\mathcal{T}(\ell', \sigma_1') \not\sqsubseteq L$. Similarly, $\mathcal{T}(\ell', \sigma_2') \not\sqsubseteq L$. Hence, $\sigma_1 \approx_l \sigma_1'$, and $\sigma_2 \approx_l \sigma_2'$, and by transitivity, $\sigma_1' \approx_l \sigma_2'$.

Case `when`$(e)s_1elses_2$*:* By T-COND, $\Gamma; pc \vdash e : \ell$. We first consider the case in which $\mathcal{T}(\ell, \sigma_1) \sqsubseteq L$. By Lemma 2, $\mathcal{T}(\ell, \sigma_2) = \mathcal{T}(\ell, \sigma_1) \sqsubseteq L$. By Lemma 3,

$\langle \sigma_1, e\rangle \Downarrow n$ and $\langle \sigma_2, e\rangle \Downarrow n$ for some n. By the semantics, either $\langle \sigma_1, s\rangle \rightarrow \langle \sigma_1, s1$ and $\langle \sigma_2, s\rangle \rightarrow \langle \sigma_2, s1$ or $\langle \sigma_1, s\rangle \rightarrow \langle \sigma_1, s2$ and $\langle \sigma_2, s\rangle \rightarrow \langle \sigma_2, s2$, but both executions take the same path. If the branch is taken, then by the induction hypothesis, $\langle \sigma_1, s\rangle \rightarrow_* \langle \sigma_1', \texttt{skip}$ and $\langle \sigma_2, s\rangle \rightarrow_* \langle \sigma_2', \texttt{skip}$ for some σ_1', σ_2' such that $\sigma_1' \approx_L \sigma_2'$. It is similar if the branch is not taken.

We now consider the case in which $\mathcal{T}(\ell, \sigma_1) \not\sqsubseteq L$. By Lemma 7, $\mathcal{T}(\ell, \sigma_2) \not\sqsubseteq L$. By T-COND, $\Gamma; pc \vdash s_1$. Let $pc' = pc \sqcup \ell$. Then $pc' \not\sqsubseteq L$ by lattice properties. Let x be some variable assigned in s_1. By T-ASSGN, $pc' \sqcup \ell'$ where $\ell' \neq \Gamma(x)[x_1 \mapsto \bar{x_1}]...[x_n \mapsto \bar{x_n}]$. By T-NEXTVAR, $\Gamma(\bar{x}) = \ell'$, and so $pc' \sqsubseteq \Gamma(\bar{x})$, and $Ga(\bar{x}) \not\sqsubseteq L$. The same is true for all other variables assigned in s_1 and for all variables assigned in s_2. Let $\langle \sigma_1, s_1\rangle \rightarrow_* \langle \sigma_1'', \texttt{skip}$. Because only high variables are assigned in s_1, σ_1 and σ_1'' may only disagree on high variables, and so $\sigma_1'' \approx_L \sigma_1$. Similarly, $\sigma_2'' \approx_L \sigma_2$. Becuase $\sigma_1 \approx_l \sigma_2$, by transitivity twice, $\sigma_1'' \approx_L \sigma_2''$.

□

We now prove that well-typed SIRRTL modules enforce a timing safe variant of observational determinism

Theorem 1 (Observational Determinism) *If Γ is a type environment, s is a statement that does not contain downgrades, pc is a label, L is a fully-evaluated security label, and σ_1 and σ_2 are states, then*

$$\begin{aligned}\vdash \Gamma \wedge \Gamma; pc \vdash s \wedge \sigma_1 \approx_L \sigma_2 \wedge &\\ &\langle 0, \sigma_1, s, \epsilon\rangle \longrightarrow_{\mathcal{S}} \langle n_1, \sigma_1', s, t_1\rangle \wedge\\ &\langle 0, \sigma_2, s, \epsilon\rangle \longrightarrow_{\mathcal{S}} \langle n_2, \sigma_2', s, t_2\rangle\\ &\implies \sigma_1' \approx_L \sigma_2' \wedge t_1 \approx_L t_2\end{aligned}$$

1

Proof. By cases on the semantic rules for programs

Case S-TICK By induction on the value of T. The base case is $T = 0$, and we have

$$\begin{aligned}\langle 0, \sigma_1, s, \epsilon\rangle \longrightarrow_{\mathcal{S}} \langle 1, \sigma_1', s, (1, \sigma_1')\rangle \wedge\\ \langle 0, \sigma_2, s, \epsilon\rangle \longrightarrow_{\mathcal{S}} \langle 1, \sigma_2', s, (1, \sigma_2')\rangle\end{aligned}$$

where

$$\sigma_1' = \sigma_1[x_1 \mapsto \sigma_1(\bar{x_1})]...[x_n \mapsto \sigma_1(\bar{x_n})]\sigma_2' \quad = \sigma_1[x_1 \mapsto \sigma_2(\bar{x_1})]...[x_n \mapsto \sigma_2(\bar{x_n})]$$

Let $\bar{x_i}$ be some next-cycle symbol such that $x_i \in \{\bar{x_1}, ..., \bar{x_n}\}$. If $\mathcal{T}(\Gamma(\bar{x_i}), \sigma_1) \sqsubseteq L$ then $\mathcal{T}(\Gamma(\bar{x_i}, \sigma_2) \sqsubseteq L$ by Lemma 2. By Lemma 3, $\sigma_1(\bar{x_i}) = \sigma_2(\bar{x_i})$. The same is true for all other next-cycle symbols in $\{\bar{x_1}, ..., \bar{x_n}\}$. Since $\sigma_1' \approx_L \sigma_2$, and σ_2' and σ_1' agree on low symbols $\bar{x_i}$, $\sigma_1' \approx_L \sigma_2'$.

We now consider the general case. We have

$$\begin{aligned}\langle n, \sigma_1, s, t_1\rangle \longrightarrow_{\mathcal{S}} \langle n+1, \sigma_1', s, t_1; (n+1, \sigma_1')\rangle \wedge\\ \langle n, \sigma_2, s, t_2\rangle \longrightarrow_{\mathcal{S}} \langle n+1, \sigma_2', s, t_2; (n+1, \sigma_2')\rangle\end{aligned}$$

By the induction hypothesis, $\sigma_1' \approx_L \sigma_2'$ and $t_1 \approx_L t_2$. So $t_1; (n+1, \sigma_1') \approx_L t_2; (n+1, \sigma_2')$

Case S-EVAL*:* Follows directly from Theorem 2.

$\square$

References

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, 2012.

[2] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. Nonmalleable Information Flow Control. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17.

[3] Andrew Ferraiuolo, Weizhe Hua, Andrew C Myers, and G Edward Suh. Secure information flow verification with mutable dependent types. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 6. ACM, 2017.

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