A Hardware Design Language for Timing-Sensitive Information-Flow Security

Danfeng Zhang Yao Wang G. Edward Suh Andrew C. Myers

Cornell University
Ithaca, NY 14853

zhangdf@cs.cornell.edu yao@csl.cornell.edu suh@csl.cornell.edu andru@cs.cornell.edu

Abstract

Information security can be compromised by leakage via low-level hardware features. One recently prominent example is cache probing attacks, which rely on timing channels created by caches. We introduce a hardware design language, SecVerilog, which makes it possible to statically analyze information flow at the hardware level. With SecVerilog, systems can be built with verifiable control of timing channels and other information channels. SecVerilog is Verilog, extended with expressive type annotations that enable precise reasoning about information flow. It also comes with rigorous formal assurance: we prove that SecVerilog enforces timing-sensitive noninterference and thus ensures secure information flow. By building a secure MIPS processor and its caches, we demonstrate that SecVerilog makes it possible to build complex hardware designs with verified security, yet with low overhead in time, space, and HW designer effort.

Categories and Subject Descriptors B.6.3 [*Hardware/Design Aids*]: Hardware Description Languages

Keywords Timing channels, hardware description language, information flow control, dependent types

1. Introduction

Information flow control offers a powerful way to prevent improper release and modification of information in complex systems. It has been applied successfully at multiple levels of abstraction: the language level, the operating system level, and the hardware level. Because hardware behavior can create additional information flows that violate a security policy, it is not enough to control information flow only at the software level.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.
Copyright © 2015 ACM 978-1-4503-2835-7/15/03…$15.00.
http://dx.doi.org/10.1145/2694344.2694372

Recent work has demonstrated the danger of hardware-level information flows by showing that timing channels can be used to communicate sensitive information between processes and even across virtual machines. For example, cache probing attacks (e.g., [15, 30]) exploit the timing channel that arises because accesses to memory locations by one process affect the cache, and thereby observably affect the timing behavior of later accesses by other processes. The cache is not the only problem. Attacks have also been shown that exploit timing channels arising from other components: instruction and data caches [2], branch predictors and branch target buffers [3], and shared functional units [44].

Motivated by such vulnerabilities, we have developed a method for designing hardware that correctly, precisely, and efficiently enforces secure information flow. This method is based on a new hardware description language (HDL) called SecVerilog, which adds a security type system to Verilog so that hardware-level information flows can be checked statically. In combination with software-level information flow control, our hardware design method enables building computing systems in which all forms of information flow are tracked, including implicit flows and timing channels.

Our approach has several advantages over the state of the art in secure hardware design. SecVerilog checks information flows statically while providing formal security assurance and guidance to hardware designers; security assurance is obtained directly from the design process. The language is also expressive enough to prove security of a design even when hardware resources are shared among multiple security levels that are changed at per-cycle granularity. The novel dependent type system of the language integrates with external program analyses, avoiding both the duplication of hardware resources and run-time tracking of information flow. Consequently, run-time overhead is lower than in prior work. Our prototype secure pipelined MIPS processor with a cache adds area and clock cycle overheads of about 1%.

In summary, our work makes multiple contributions:

- SecVerilog, a new hardware description language that extends Verilog with fine-grained tracking of information flows within hardware,

- expressive static annotations incorporating dependent security types, enabling flexible, fine-grained reuse and sharing of hardware across security levels,
- a formal proof that the HDL type system soundly controls information flow, and
- the design of a secure microprocessor using SecVerilog, demonstrating the practicality and the power of this methodology. We show that overheads in delay, area, power, performance and designer effort are all low.

The paper is structured as follows. Section 2 gives an overview of our approach for controlling hardware-level information flow, including timing channels. Sections 3 and 4 describe SecVerilog and its security type system. The formal proof that SecVerilog enforces security is sketched in Section 5. An evaluation of using SecVerilog in building a pipelined MIPS processor is in Section 6. Section 7 covers related work, and Section 8 concludes.

2. Background and Approach

2.1 Information flow control

Information flow control aims to ensure that all information flows in a system respect a security policy. For this purpose, information in the system is associated with a *security level* drawn from a lattice $\mathcal{L}$ whose partial ordering $\sqsubseteq$ specifies which information flows are allowed. For example, a lattice with two security levels L (low, public) and H (high, secret) can be used to forbid information labeled as H from flowing into L (H $\not\sqsubseteq$ L) while allowing the other direction (L $\sqsubseteq$ H).

The goal of SecVerilog is to enforce fine-grained information flow control for hardware designs in a statically verifiable fashion. With SecVerilog, hardware designers specify hardware-level information flow policies by annotating wires and registers with security labels and specifying a security lattice. Then, the SecVerilog type system statically checks and verifies timing-sensitive information flow properties within hardware at design time. While we use a simple lattice with two security levels (L and H) in our examples, the approach applies to an arbitrary security lattice.

2.2 Threat model

We target synchronous circuits driven by a fixed-frequency clock. We assume a *software-level* adversary, who can observe *all* information at or below a certain security level that we will call low (L). The adversary can also measure timing of hardware operations at the granularity of a clock cycle. Hence, both storage and timing channels [21] are considered. However, we assume the adversary has no physical access to the hardware, and we do not consider physical attacks such as directly tapping internal circuits or side channels that require physical proximity, such as power consumption analysis.

2.3 Controlling timing channels

The ability to verifiably control fine-grained information flow in hardware can enhance security in many applications. One notable example, and a focus of this paper, is designing efficient hardware that controls timing channels. These channels are perhaps the most challenging aspect of information flow security, because confidential information can affect timing in various ways: at the software level, a branch or loop conditioned on secret values creates timing channels [20]; at the hardware level, sharing hardware resources such as the data cache also creates timing channels [8, 15, 30, 32].

```
1 if (h1)[L]
2   h2:=l1;[H]
3 else
4   h2:=l2;[H]
5 l3:=l1;[L]
```

Restrictions on secure hardware designs:

1) The high partition cannot affect the timing of instructions with label L,
2) the low partition cannot be modified when the timing label is H, and
3) the contents of the high partition cannot affect those of the low partition.

Figure 1. An example of full-system timing channel control. The well-typed program on the left is secure if the hardware enforces the constraints on the right.

Our goal is an efficient hardware design that enforces the complex security policy required by the full-system timing channel control mechanism proposed by Zhang et al. [49]. In this approach, the security of the whole system rests on a concise contract between the software and hardware, provably controlling timing channels if both meet their requirements.

The contract treats the hardware implementation as an abstract *machine environment* whose state is partitioned by security level. For example, with two levels L and H, hardware resources such as caches are conceptually partitioned into a low part and a high part. At the software level, the contract is manifested as one *timing label* for each statement in a source program[1]. With this abstraction, a type system generates *timing labels* at the software layer that should be communicated to hardware. For example, the code fragment in Figure 1 illustrates a well-typed program, in which timing labels are shown in brackets; `h1` and `h2` are confidential, and other variables are public. Since the existence of `h1` in data cache, rather than the value of `h1`, can affect the execution time of line 1, line 1 has a timing label of L. The benefit of these timing labels is that only the timing of instructions with H timing labels needs to be controlled and mitigated at software level, as long as the security policy in Figure 1 is enforced on hardware.

2.4 Example: secure cache design

Designing hardware to meet the complex security policy in Figure 1 is challenging. As an illustration, we consider designing a secure cache, statically partitioned between security levels L (low) and H (high) as proposed in prior work [31,

[1] The general contract in [49] uses two timing labels, called the *read* and *write* labels. SecVerilog is expressive enough to verify the general case, which is implemented in our verified MIPS processor (Section 6). For simplicity, we assume these two labels are equal in most of examples.

```
reg[18:0] {L} tag0[256],tag1[256];
reg[18:0] {H} tag2[256],tag3[256];
wire[7:0] {L} index;
//Par(0)=Par(1)=L  Par(2)=Par(3)=H
wire[1:0] {Par(way)} way;
wire[18:0] {Par(way)} tag_in;
wire {Par(way)} write_enable;

always @(posedge clock) begin
  if (write_enable) begin
    case (way)
    0: begin tag0[index]=tag_in; end
    1: begin tag1[index]=tag_in; end
    2: begin tag2[index]=tag_in; end
    3: begin tag3[index]=tag_in; end
    endcase
  end
end
```

(a) SecVerilog code for cache tags

```
1  wire {L} isLoad,isStore;
2  wire {L} hit0,hit1; // hitX: 1 iff way X gets a cache hit
3  wire {H} hit2,hit3;
4  //LH(0)=L  LH(1)=H
5  wire {LH(timingLabel)} stall, hit, timingLabel;
6  reg[2:0] {LH(timingLabel)} dFsmState;
7
8  assign stall = ((isLoad | isStore) &
9    (~hit | (dFsmState != DFSM_IDLE)));
10 assign hit = (timingLabel == 0) ?
11   ((hit0|hit1)?1:0) : ((hit0|hit1|hit2|hit3)?1:0);
12 ...
13 case (dFsmState)
14   DFSM_IDLE: begin
15     // load hit
16     if (isLoad && hit) begin
17       dFsmState <= DFSM_IDLE; // nonblocking assignment
18   ...
19 endcase
```

(b) SecVerilog code for a cache controller

Figure 2. SecVerilog extends Verilog with security label annotations (shaded in gray).

45]. The L and H partition correspond to the L and H machine environment respectively.

Figure 2(a) presents a simplified fragment of SecVerilog code to update cache tags. For now, ignore the shaded annotations. This design logically partitions a 4-way set-associative cache so that ways 0 and 1 (`tag0` and `tag1`) are used as the L partition, and the other ways (`tag2` and `tag3`) are used as the H partition. The code writes a new cache tag to a way specified by `way` when `write_enable` is asserted.

This simple example shows the intricacy of correctly enforcing the aforementioned security policy in hardware. First, `tag_in` must not contain high information when `way` is 0 or 1, to prevent the H partition from affecting the state of the L partition (`tag0` and `tag1`). Second, `write_enable`, which controls whether a write occurs, cannot be influenced by high information when `way` is 0 or 1 (an instance of implicit flows [35]). Verifying these restrictions is tricky since the cache partition that `tag_in` and `write_enable` belong to can change at run time.

More challenging is to enforce secure timing: the H partition cannot affect the timing of instructions with timing label L. A simplified fragment of the SecVerilog code for the cache controller is shown in Figure 2(b), where `timingLabel` represents the timing label of a cache access, propagated from the software level, hit$i$ $(0 \leq i \leq 3)$ indicates if way i gets a cache hit, and the `stall` signal indicates when a cache access completes.

Since the `stall` signal affects the execution time of an instruction, a secure design must ensure that only the L partition can affect it when `timingLabel` is 0 (encoding L). Verifying this property is difficult, since the cache controller may access H data even when `timingLabel` is 0 (e.g., to execute line 1 of the example in Figure 1). Perhaps counterintuitively, this access is secure: timing may be affected by the *existence* of H data in the cache but not by the value of the data. Moreover, the `hit` and `dFsmState` signals, which affect `stall` (line 8), are shared across both cache partitions. A secure design must ensure that no information leaks through these shared variables, which is difficult since their uses are spread across multiple statements (lines 13–19 only show a snippet).

2.5 The SecVerilog approach

SecVerilog extends Verilog with the ability to give each variable a *label* that specifies the security level of the variable. In Figure 2, these labels are the shaded annotations, which indicate, e.g., that variables `tag0` and `tag1` are labeled L whereas `tag2` and `tag3` are labeled H. Using these annotations, the SecVerilog type system automatically verifies information flow properties of Verilog code at compile time.

Programming languages that provide the ability to label variables have been developed before [6, 25, 36], but their labels are not expressive enough to handle practical hardware designs where resources need to be shared across security levels. In effect, the security levels change at run time. We use dependent types to address this challenge.

Consider the example in Figure 2(a). The labels of `way`, `write_enable`, and `tag_in` depend on which cache way is being accessed. In fact, we observe that a precise dependent label can be assigned to these variables without any change to the Verilog code. The proper label is `Par(way)`, where the name `Par` denotes a type-level function that maps 0 and 1 to level L, and 2 and 3 to level H (concisely, `Par` $= \{0 \mapsto \mathrm{L}, 1 \mapsto \mathrm{L}, 2 \mapsto \mathrm{H}, 3 \mapsto \mathrm{H}\}$). Intuitively, these dependent labels express a lightweight invariant on variables (e.g., when `way` is 0, `write_enable` must have level L).

For the example in Figure 2(b), `stall`, `hit` and `dFsmState` can be labeled with `LH(timingLabel)` where `LH` $= \{0 \mapsto \mathrm{L}, 1 \mapsto \mathrm{H}\}$ to ensure that they can be affected only by the low partition when `timingLabel` is 0.

Program	Prog	$::= B_1 \ldots B_n$
Thread	B	$::=$ `always` $@(\gamma)\ c$
Trigger	γ	$::=$ `posedge clock` $\mid$ `negedge clock` $\mid \vec{v}$
Cmds	c	$::= \texttt{skip}_\eta \mid \texttt{begin}\ c_1; \ldots; c_n;\ \texttt{end}$
		$\mid v =_\eta e \mid v \Leftarrow_\eta e \mid \texttt{if}_\eta\ (e)\ c_1\ \texttt{else}\ c_2$
Expr	e	$::= v \mid n \mid \texttt{uop}\ e \mid e\ \texttt{bop}\ e$
Vars	x, y, v	$\in$ **Vars**

Figure 3. Syntax of SecVerilog.

Such invariants can be maintained by the type system described in Section 4. For instance, to ensure that the explicit flow from `tag_in` to `tag0` at line 12 in Figure 2(a) is secure, the type system generates a proof obligation ($\texttt{way} = 0 \Rightarrow \texttt{Par}(\texttt{way}) \sqsubseteq L$), meaning that when `way` is 0, information flow from `tag_in` (with label `Par(way)`) to `tag0` (with label L) is permissible. This proof obligation can easily be discharged by an external solver.

The soundness of our type system (Section 5) guarantees that all security violations are detected at compile time. For example, consider the case when `timingLabel` is 0 in line 11 in Figure 2(b). If the H partition, such as variable `hit2`, were accessed in that case, an error would be reported because the type system would generate an invalid proof obligation: $(\texttt{timingLabel} = 0) \Rightarrow H \sqsubseteq \texttt{LH}(\texttt{timingLabel})$.

2.6 Benefits over previous approaches

Our approach enjoys several benefits compared with prior efforts with verifiable information-flow security for hardware [22, 23, 42]. First, verification is done at compile time, avoiding run-time overhead and detecting errors at an early design stage. This is not possible with GLIFT [42] and Sapper [22]. Second, variables and logic can be shared across multiple security levels (e.g., `way` and `hit` are shared with various timing labels), which is not possible with Caisson [23]. Moreover, SecVerilog adds little programming effort: Verilog code can be verified almost as-is, with annotations (security labels) required only for variable declarations.

3. SecVerilog: Syntax and semantics

Except for added annotations, SecVerilog has essentially the same syntax and semantics as Verilog [13]. It builds on the synthesizable subset of the Verilog language. We restrict to synthesizable code because unsynthesizable Verilog code is used only for testing purposes: it has no effect on the final hardware. The target language of our compiler is synthesizable Verilog from which hardware can be generated using existing tools.

A core subset of SecVerilog is shown in Figure 3. We choose this subset because it includes all interesting features, and the omitted features (e.g., `case`, `assign` and the ternary conditional) can be translated into the core language.

Level	$\ell \in \mathcal{L}$
Family	$f \in \mathbb{Z}_n \to \mathcal{L}$
Label	$\tau ::= \ \ell \mid f(v) \mid \tau_1 \sqcup \tau_2 \mid \tau_1 \sqcap \tau_2$

Figure 4. Syntax of security labels.

A SecVerilog program (`Prog`) consists of a set of variable declarations and a set of thread definitions that use these variables. Variable v can represent either a register or a wire. The difference is that wires are stateless, and must be driven by other signals. We do not distinguish them in the syntax.

"Always blocks" (B) in (Sec)Verilog are similar to *threads* from the software perspective. Each `always` block translates into a hardware module that operates in parallel to other modules.

Threads are activated by triggers. A trigger γ can either be a change to the clock signal (`posedge`/`negedge` means the rising/falling edge of the clock signal), or a change to a variable in a variable list $\vec{v}$. For example, commands in the `always` block at line 9 in Figure 2(a) are activated at every rising edge of the clock signal.

Commands c are similar to those in software languages. Symbols η are unique identifiers for program points and can be ignored for now. A feature of Verilog not found in most programming languages is the distinction between blocking assignment $v =_\eta e$ and nonblocking assignment $v \Leftarrow_\eta e$. The effects of blocking assignments are visible immediately, but those of nonblocking assignments are delayed until the end of the current time unit. For example, consider the two code fragments $x = 1; y \Leftarrow x$ and $x \Leftarrow 1; y \Leftarrow x$. If the value of x is initially 0, then y becomes 1 in the first piece of code, but 0 in the second.

We provide a formal operational semantics for SecVerilog in the supplementary material [50].

4. SecVerilog: Type system

The SecVerilog type system statically controls information flow in a rigorous and verifiable way. The most novel features of the type system include: 1) mutable, dependent security labels, 2) a permissive yet sound way of controlling label channels, and 3) a modular design that decouples the program analyses required for precision from the type system. These novel features are essential for statically verifying highly efficient, practical hardware designs.

4.1 Type syntax

Types in SecVerilog are simply Verilog types extended with security label expressions, whose syntax is shown in Figure 4. The simplest form of label τ is a concrete security level ℓ drawn from the security lattice $\mathcal{L}$.

Unlike in most previous work on language-based security, SecVerilog supports *dynamic labels*: labels that can change

$$\frac{}{\Gamma, pc, \mathcal{M} \vdash \texttt{skip}_\eta}\ \textsc{T-Skip} \qquad \frac{\Gamma, pc, \mathcal{M} \vdash c_1 \quad \Gamma, pc, \mathcal{M} \vdash c_2}{\Gamma, pc, \mathcal{M} \vdash c_1; c_2}\ \textsc{T-Seq} \qquad \frac{\Gamma \vdash e : \tau \quad v \notin \texttt{FV}(\Gamma(v)) \quad \models P(\bullet\eta) \Rightarrow \tau \sqcup pc \sqsubseteq \Gamma(v)}{\Gamma, pc, \mathcal{M} \vdash v =_\eta e \quad \Gamma, pc, \mathcal{M} \vdash v \Leftarrow_\eta e}\ \textsc{T-Assign}$$

$$\frac{\Gamma \vdash e : \tau \quad \begin{array}{c} v \in \texttt{FV}(\Gamma(v)) \\ v' \notin \Gamma \end{array} \quad \begin{array}{c} \models P(\bullet\eta) \Rightarrow pc \sqsubseteq \Gamma(v)\ ^{\text{if } v \notin \mathcal{M}} \\ \models P(\bullet\eta), v' = \lfloor e \rfloor_a \Rightarrow \tau \sqcup pc \sqsubseteq \Gamma(v)\ \{v'/v\} \end{array}}{\Gamma, pc, \mathcal{M} \vdash v =_\eta e \qquad \Gamma, pc, \mathcal{M} \vdash v \Leftarrow_\eta e}\ \textsc{T-Assign-Rec} \qquad \frac{\Gamma \vdash e : \tau \quad \begin{array}{c} \Gamma, pc \sqcup \tau, \mathcal{M} \cap \texttt{DA}(\eta) \vdash c_1 \\ \Gamma, pc \sqcup \tau, \mathcal{M} \cap \texttt{DA}(\eta) \vdash c_2 \end{array}}{\Gamma, pc, \mathcal{M} \vdash \texttt{if}_\eta\ (e)\ c_1\ \texttt{else}\ c_2}\ \textsc{T-If}$$

Figure 5. Typing rules: commands.

at run time. A dynamic label $f(v)$ is constructed using a type-valued function f applied to a variable v. Type-valued functions are needed in order to decode the simple values that the hardware can convey into labels from the lattice $\mathcal{L}$.

Dynamic labels are needed to accurately describe information flows in complex hardware designs, where hardware resources can be used by multiple security levels. One example is the label `Par(way)` used in Figure 2(a). Note that all security labels in SecVerilog, including dynamic labels and label-decoding functions, only exist for compile-time type checking; they have no run-time manifestation.

Because security labels can mention terms (in particular, variables such as `way`), the type system has dependent types. Dependent security types have been explored in some prior work on security type systems that track information flow (e.g., [25, 43, 51]), where they provide valuable expressive power. However, in order to support analysis of hardware security, the type system for SecVerilog includes some unique features: first, the use of type-valued functions for label decoding, and second, even more unusual, the presence of mutable variables in types—that is, types may depend on variables whose value can change at run time.

The design philosophy of SecVerilog is to offer an expressive language with a low annotation burden, along with fast, automatic type checking. Following this philosophy, the only kind of term to which a label decoding function can be applied is a variable. This restriction ameliorates two problems: first, the undecidability of type equality involving general program expressions, and second, side effects changing the meaning of types.

Despite this restriction, dependent types in SecVerilog nevertheless turn out to be expressive enough for the intended use in hardware design. Restricting dependent types allows type checking to be fast (e.g., two seconds to verify a complete MIPS CPU in Section 6.1) and fully automatic. The syntax also alleviates the resulting limitations on expressiveness by allowing joins ($\sqcup$) and meets ($\sqcap$) of labels.

4.2 Typing rules

Typing rules for expressions have the form $\Gamma \vdash e : \tau$ where Γ is a typing environment that maps variables to security labels, e is the expression, and τ is its label. Since these rules are mostly standard [35], we leave the details in the supplementary material [50].

The typing rules for commands are shown in Figure 5. The typing judgment has the form $\Gamma, pc, \mathcal{M} \vdash c$. Similar to the usual program-counter label [35] for software languages, pc is used to control implicit flows. More interesting is $\mathcal{M}$, which tracks a set of variables that *must* be modified in all alternative executions. The type system uses $\mathcal{M}$ to improve its precision, as we see shortly.

In the next three sections, we explore the challenges of designing the SecVerilog type system and along the way explain the rules of Figure 5 in more depth.

4.3 Mutable dependent security labels

Dependent types need to mention mutable variables in practical hardware designs. For example, variable `way` in Figure 2(a) can be modified whenever a new read request comes to the data cache, updating which cache way to use. Mutability creates some challenges for the soundness of the type system. We begin by illustrating these challenges.

Implicit declassification. Whenever a variable changes, the meaning of any security label that depends on it also changes. To be secure, SecVerilog needs to prevent such changes from implicitly declassifying information. Consider the example in Figure 6. This code is clearly insecure since it copies `secret` into `public` when `x` changes from 1 to 0 (not shown for brevity).

At the assignment to `y` in the first branch, its level is H, but at the assignment to `public`, the level of `y` has become L. The insecurity arises from the change to the label of `y` during the execution, while its content remains the same. In other words, if `x` changes from 1 to 0, the label of `y` cannot protect its content.

We rely on a dynamic mechanism to erase register contents when the old label is not bounded by the new one. Code to dynamically zero out registers is automatically inserted as part of the translation to Verilog. Note that wires in hardware are stateless, so dynamic erasure only applies to registers.

```
reg[7:0] {H} secret, {L} public, {L} x;
reg[7:0] {LH(x)} y; // LH(0)=L LH(1)=H
always@(posedge clock) begin
  if (x==1) begin y ⇐ secret; end
  else      begin public ⇐ y; end
  end
end
```

Figure 6. An example of implicit declassification.

```
1   reg{H} high;
2   reg{L} low, low';
3   reg{LH(x)} x;//LH(0)=L LH(1)=H
4   ...
5   if (high) begin
6     x <= 1;
7   end
8   if (x==0 && low==1) begin
9     low' <= 0;
10  end
11  low <= 1;
12  ...
```

(a) Insecure program with a label channel.

```
1   reg{H} hit2, hit3;
2   reg[1:0]{Par(way)} way;
3   // Par(0)=Par(1)=L
4   // Par(2)=Par(3)=H
5   ...
6   if (hit2 || hit3) begin
7     way <= (hit2 ? 2'b10 : 2'b11);
8   end
9   else begin
10    way <= 2'b10;
11  end
12  ...
```

(b) No-sensitive-upgrade rejects secure code.

```
1   reg{H} high;
2   reg{L} low, low';
3   reg{Par(x)} x;
4   // Par(0)=Par(1)=L
5   // Par(2)=Par(3)=H
6   ...
7   if (x==0) begin
8     low <= 1;
9   end
10  else begin
11    high <= 1;
12  end
13  low' <= low;
14  ...
```

(c) Flow-sensitive systems reject secure code.

Figure 7. Examples illustrating the challenges of controlling label channels.

While this dynamic mechanism may affect the functionality of the original hardware design, we believe that it is not a major issue in practice for the following reasons:

1. Dynamic erasure happens very rarely in our design experience. Most variables with dynamic labels are wires in our prototype processor design (e.g., `way`, `tag_in` and `write_enable` in Figure 2(a)). So the dynamic mechanism has no effect on these variables.
2. For registers with dynamic labels, this clearing is indeed necessary for security; hardware designers need to explicitly implement it anyway. Consider `dFsmState` in Figure 2(b), the state of the cache controller. It is reset anyway in a secure design, when the pipeline is flushed in the case that the timing label changes from H to L.
3. Further, the compiler can notify a designer when automatic clearing is generated, and ask the designer to explicitly approve such changes.

Label channels. Mutable dependent types create *label channels* in which the value of a label becomes an information channel. For instance, consider the code snippet in Figure 7(a). This example appears secure as the assignment to `low'` only occurs when the label of `x` is L (when `x` is 0). When `high` is 1, the label of `x` becomes H, which correctly protects the secrecy of `high`. However, this code is insecure because the *change of label* `x` also leaks information. Suppose that the variables represent flip-flops that are initialized to $(\texttt{x} = 0, \texttt{low} = 0, \texttt{low}' = 1)$ on a reset. The value of `x` in the second clock cycle after a reset is determined by the value of `high` in the first cycle; 1 if `high` is 1, 0 if `high` is 0. Then, `low'` in the third clock cycle reflects the value of `x` in the second cycle, leaking information from `high` to `low'`.

Similar vulnerabilities have also been observed in the literature on flow-sensitive security types, in which security labels of a variable may change dynamically (e.g., [5, 18, 34]). However, prior solutions are all too conservative (i.e., they reject secure programs) for practical hardware designs.

The first approach is *no-sensitive-upgrade* [5], which forbids raising a low label to high in a high context. However, this restriction rules out useful secure code, such as the secure code in Figure 7(b), adapted from our partitioned cache design. This code selects a cache way to write to. Variables `hit2` and `hit3`, representing the existence of a hit in high cache, have label H. No-sensitive-upgrade rejects this program, since `way` might be L before the assignment.

The second approach [18, 34] raises the label of variables modified in any branch to the context label (the label of the branch condition). Returning to the example in Figure 7(a), the label of `x` would become H because of the if-statement at lines 5–7. This over-approximation can be too conservative as well. For example, consider the secure code in Figure 7(c). Here, the label of `x` specifies an invariant: whenever `x` is 0 or 1 (i.e., `Par(x)`=L), nothing is leaked by the value of `x` nor by the time at which its value changes. Hence, `low`'s transition to 1 at line 8 is secure. However, the approach in [18] raises the label of `low` to `Par(x)` after the if-else statement. This conservative label of `low` makes checking at line 13 fail, since there is a flow from H to L when `x` is 2 or 3. Even a more permissive approach rejects this secure code. When `x` is 2 or 3, the dynamic monitor described in [34] tracks a set of variables that may be modified in another branch (`low` in this case), and raises their label to the context label (H). Hence, line 13 is still rejected.

We propose a more permissive mechanism that accepts secure programs in Figure 7(b) and 7(c). Our insight is that no-sensitive-upgrade is needed for security, but only when the modified variable *might not* be assigned in an alternative path. For example, in Figure 7(b), the variable `way` is modified in both branch paths. Here, the label of `way` is checked for both branch paths on the assignments to `way` (line 7 and 10), ensuring that the label of `way` must be higher than the context label (H) at the merge point. In other words, the fact that the label of `way` becomes H leaks no information. Hence, the no-sensitive-upgrade check is unnecessary in this case. This insight is formally justified in our soundness proof in the supplementary material [50].

This insight motivates using a *definite-assignment analysis*, which identifies variables that must be assigned to in any possible execution. Definite assignment analysis is a common static program analysis useful for detecting uninitialized variables. Since SecVerilog, like Verilog, has no aliasing, definite-assignment analysis is simple; we omit the details.

We assume an analysis that returns $\mathtt{DA}(\eta)$, variables that must be assigned to in any possible execution of the command at location η. The type system propagates this information to branches, so that for an assignment to v, the no-sensitive-upgrade check is avoided if v must be assigned to in other paths. For example, the program in Figure 7(b) is well-typed because `way` is modified in both branches, avoiding the limitations of [5]. Moreover, the type system still enables the remaining (necessary) no-sensitive-upgrade checks. So there is no need to raise the label of a variable assigned to in an alternative path. For example, there is no need to check the assignment to `low` at line 8 of Figure 7(c) in a high context (the else branch), avoiding the limitations of [18, 34]. Soundness is preserved despite the extra permissiveness (see Section 5).

4.4 Constraints and hypotheses

The design goal of SecVerilog is to achieve both soundness and precision, with a low annotation burden. The key to precision is to make enough information about the run-time values of variables available to the type system. For instance, consider the assignment to `hit` at line 10 in our cache controller (Figure 2(b)). To rule out an insecure flow from `hit2` and `hit3` (with label H) to `hit` (with label `LH(timingLabel)`), the type system must ensure H $\sqsubseteq$ `LH(timingLabel)`. In other words, in any possible evaluation of the assignment, the label H must be bounded by `LH(timingLabel)`. In fact, this must be true because the condition `timingLabel=1` holds whenever the assignment happens (note that `timingLabel` is a single bit). However, a naive type system without knowledge of run-time values of `timingLabel` has to conservatively reject the program.

We use a modular design to separate the concerns of soundness and precision of our type system. In this design, the type system, along with a race-condition analysis in Section 5.3, ensures soundness (i.e., *observational determinism* in Section 5.2). The precision of the type system is improved further, without harming soundness, by integrating two program analyses: a *predicate transformer* analysis and the definite-assignment analysis already discussed.

Specifically, the type system generates proof obligations: partial orderings that must hold on pairs of security labels, regardless of the run-time values of those labels. To statically check a partial ordering on labels, we might require the partial ordering to hold for *any* possible values of free variables:

$$\tau_1 \sqsubseteq \tau_2 \Leftrightarrow \forall \vec{n} \,.\, \tau_1 \{\vec{n}/\vec{v}\} \sqsubseteq \tau_2 \{\vec{n}/\vec{v}\}$$

where $\vec{v} = \mathtt{FV}(\tau_1) \cup \mathtt{FV}(\tau_2)$, and $\mathtt{FV}(\tau)$ is the free variables in τ. However, this static approximation is too conservative.

To escape this conservatism, the type system uses a more precise approximation of the possible hardware states that can arrive at each program point. We denote the facts that program analysis has derived about the hardware states as predicates indexed by command identifiers η. The predicates $P(\bullet\eta)$ and $P(\eta\bullet)$ respectively denote overapproximations of the hardware states that can exist before and after the execution of the command at location η. Using even simple program analyses to generate these predicates considerably improves the precision of information flow analysis without harming soundness. Returning to our example, supposing that the program analysis can derive $P(\bullet\eta) = (\mathtt{timingLabel} = 1)$. The type system then only needs to know that the flow from H to `LH(timingLabel)` is secure when `timingLabel` is 1. This requirement can be expressed as an (easily verified) constraint:

$$\mathtt{timingLabel} = 1 \Rightarrow \mathrm{H} \sqsubseteq \mathtt{LH}(\mathtt{timingLabel})$$

$$\lfloor n \rfloor_a = n \qquad \lfloor v \rfloor_a = v \qquad \lfloor e \rfloor_a = \top \text{ (otherwise)}$$

$$\lfloor e_1 \;\mathtt{bop}\; e_2 \rfloor_b = \begin{cases} \lfloor e_1 \rfloor_b \;\mathtt{bop}\; \lfloor e_2 \rfloor_b \text{ if } \mathtt{bop} \in \{\wedge, \vee\} \\ \lfloor e_1 \rfloor_a \;\mathtt{bop}\; \lfloor e_2 \rfloor_a \text{ if } \mathtt{bop} \in \{=, \neq\} \\ \top \text{ otherwise} \end{cases}$$

$$\lfloor \mathtt{uop}\; e \rfloor_b = \begin{cases} \neg \lfloor e \rfloor_b \text{ if } \mathtt{uop} \in \{\neg\}\,,\, \lfloor e \rfloor_b \neq \top \\ \top \text{ otherwise} \end{cases}$$

$$\frac{}{\{P\}\; \mathtt{skip}_\eta \;\{P\}} \qquad \frac{\{P\}\; c_1 \;\{Q\} \quad \{Q\}\; c_2 \;\{R\}}{\{P\}\; c_1; c_2 \;\{R\}}$$

$$\frac{Q = \mathtt{remove}(v, P)}{\{P\}\;\; v =_\eta e \;\{Q \wedge (v = \lfloor e \rfloor_a)\}} \qquad \frac{}{\{P\}\;\; v \Leftarrow_\eta e \;\{P\}}$$

$$\frac{\{P \wedge (\lfloor e \rfloor_b)\}\; c_1 \;\{Q\} \quad \{P \wedge (\neg \lfloor e \rfloor_b)\}\; c_2 \;\{R\}}{\{P\}\; \mathtt{if}_\eta\; (e)\; c_1 \;\mathtt{else}\; c_2 \;\{Q \cap R\}}$$

Figure 8. Predicate generation in Hoare logic.

4.5 Generating state predicates

Many techniques can be used to generate predicates describing the run-time state, with a tradeoff between precision and complexity. For example, weakest preconditions [10] could be used. However, shallow knowledge of run-time state is enough for our type system to be effective. We use a simple abstract interpretation to propagate predicates forward through each thread definition, starting from the predicate `true` and overapproximating the postcondition at each program point. The rules defining this analysis are given in Figure 8. The algorithm generates predicates in linear time.

Expression results are coarsely approximated by tracking only constant values and variables, and replacing more complex expressions with the "unknown" value $\top$. Operators $\lfloor e \rfloor_a$ and $\lfloor e \rfloor_b$ estimate the arithmetic and boolean values of e, respectively. The result of binary operators is $\top$ if any operand is $\top$. The translation rules on commands are written as admissible weakenings of the rules of Hoare Logic [17]. They should be read as specifying how to compute a postcondition from a precondition. To make reasoning practical, the rules do not derive the strongest possible postcondition—but of course it is sound to weaken postconditions. Consequently, postconditions and preconditions are represented as conjunctions. The rule for assignment weakens the strongest-postcondition rule [11] by discarding all conjuncts that mention the assigned variable, in $\mathtt{remove}(v, P)$. For efficiency, the rule for `if` weakens the obvious postcondition, $Q \vee R$, by syntactically intersecting the sets of conjuncts in Q and R.

4.6 Discussion of typing rules

The most interesting rules in Figure 5 are (T-ASSIGN), (T-ASSIGN-REC) and (T-IF). Proof obligations are generated for assignments $v =_\eta e$ and $v \Leftarrow_\eta e$. These proof obligations are discharged by an external solver; our implementation uses Z3 [9]. The informal invariants the type system maintains are 1) the new label of v is more restrictive than both the context label pc and label of e, 2) the no-sensitive-upgrade check is enabled if there might not be an assignment to v in an alternative branch. Rule (T-ASSIGN-REC) checks these invariants explicitly. To check the invariant after update, the rule generates a fresh variable v' to represent the new value of v. Though pc and the security level of e may also change after the assignment, the rule checks against the old value since semantically, information flows from the old state to variable v. The no-sensitive-upgrade check is enforced with the condition $v \notin \mathcal{M}$ adding precision in the case where the variable is assigned in every branch. The single check in Rule (T-ASSIGN) is sufficient for these invariants since $\Gamma(v)$ remains the same when there is no self-dependency, and the check entails no-upgrade-check (because $pc \sqsubseteq \tau \sqcup pc$). To improve precision of the type system, predicates on states are used only as hypotheses in these proof obligations. Blocking and nonblocking assignments use the same typing rule, differing only on when the assignment takes effect.

Rule (T-IF) propagates the set of variables that must be modified in both branches ($\mathtt{DA}(\eta)$) to c_1 and c_2. Taking the intersection of $\mathcal{M}$ and $\mathtt{DA}(\eta)$ is needed for nested if-statements.

4.7 Scalability of type checking

Queries sent to Z3 are generated by typing rules (T-ASSIGN) and (T-ASSIGN-REC) in Figure 5. Note that these queries are essentially predicates on a (finite) lattice of security labels. In another word, only simple theories (e.g., no quantifiers, no real numbers) of the full-fledged Z3 solver are utilized by the type system. These queries can be efficiently solved by Z3.

Moreover, the static analyses used by SecVerilog to enable precise type checking (definite assignment analysis and predicate generation) are both modular. Race condition analysis may vary depending on the hardware design tool, but is scalable for most tools.

For the complete MIPS CPU in Section 6.1, it takes a total of only two seconds to generate all 1257 constraints by the type system, and solve them with Z3, suggesting that type checking is likely to scale to larger hardware designs.

4.8 Well-formed typing environments

The use of dynamic labels also puts constraints on the typing environment Γ: Γ is *well-formed*, denoted $\vdash \Gamma$, when 1) no variable depends on a more restrictive variable, preventing secrets from flowing into a label, and 2) no dependencies are chained, preventing cyclic dependencies. If $\mathtt{FV}(\tau)$ is the free variables in τ, this can be expressed formally as follows:

DEFINITION 1 (Well-formedness). *Γ is well-formed iff*

$$\forall v \in \mathbf{Vars}\,.\,(\forall v' \in \mathtt{FV}(\Gamma(v))\,.\,\Gamma(v') \sqsubseteq \Gamma(v)) \\ \wedge(\forall v' \in \mathtt{FV}(\Gamma(v))\,.\,v' \neq v \Rightarrow \mathtt{FV}(\Gamma(v')) = \emptyset)$$

5. Soundness

Central to our approach is rigorous enforcement of a strong information security property. We formalize this property and sketch a soundness proof of our type system; the full proof is available in the supplementary material [50].

5.1 Proving hardware properties from HDL code

Our goal is to prove that the actual hardware implementation controls information flow. However, information flow is analyzed at the level of the HDL. The argument that language-level reasoning is accurate has two steps. First, the operational semantics of SecVerilog correspond directly to hardware simulation at the RTL (Register Transfer Level) of abstraction. Second, for a synchronously clocked design, these RTL simulations accurately reflect behavior of synthesized hardware; in fact, functional verification of modern hardware relies mainly on RTL simulation. Thus, HDL-level reasoning suffices to prove hardware-level security properties.

5.2 Observational determinism

Our formal definition of information flow security is based on *observational determinism* [33, 48], a generalization of noninterference [12] that provides a strong end-to-end security guarantee even for nondeterministic systems. Observational determinism requires that in any two executions that receive the same low (adversary-visible) input, the system's low behavior must also be indistinguishable regardless of both high inputs and (possibly adversarial) nondeterministic choices.

Formalizing this property in the presence of dynamic labels presents some challenges, since the security level of a variable may differ in two hardware states. We start by defining a *low-equivalence* relation $\approx_\ell$ on hardware states σ, indexed by a level $\ell \in \mathcal{L}$. Two states are low-equivalent at level ℓ if they cannot be distinguished by an adversary able to observe information only at that level or below.

We assume a typing environment Γ that maps variables to security labels. Given state σ, the security level of a variable x is: $\mathcal{T}(x, \sigma) = \ell'$, where ℓ' is the value of label $\Gamma(x)$ in σ. We formalize the low-equivalence relation as follows:

DEFINITION 2 (Low equivalence at level ℓ). *Two states are low-equivalent at level ℓ iff any variable whose label is below ℓ in one state must have the same label and value in the other:*

$$\forall \sigma_1, \sigma_2 \,.\, \sigma_1 \approx_\ell \sigma_2 \iff \forall x \in \mathbf{Vars}\,.\\ (\mathcal{T}(x,\sigma_1) \sqsubseteq \ell \Leftrightarrow \mathcal{T}(x,\sigma_2) \sqsubseteq \ell)\\ \wedge (\mathcal{T}(x,\sigma_1) \sqsubseteq \ell \Rightarrow \sigma_1(x) = \sigma_2(x))$$

It is straightforward to check that $\approx_\ell$ is an equivalence relation. Note that we require the level of x to be bounded by ℓ in σ_2 whenever $\mathcal{T}(x,\sigma_1) \sqsubseteq \ell$. This definition corresponds to our adversary model: all variables below ℓ are observable to the adversary. For example, consider the case $\Gamma(x) = \texttt{LH}(x), \sigma_1(x) = 0$ and $\sigma_2(x) = 1$. Since x has different labels in the two states, $\sigma_1 \not\approx_L \sigma_2$. This is necessary because the ability to make the observation itself leaks information.

An *event* is a pair (t, σ), meaning that state σ occurred at clock cycle t. Assuming synchronous logic, events are produced only when a clock tick occurs. A *trace* T is a countably infinite sequence of events. We write $\langle \sigma, \texttt{Prog}\rangle \hookrightarrow T$ if executing Prog with initial states σ produces a trace T. Since the semantics is nondeterministic, there can be multiple traces T such that $\langle \sigma, \texttt{Prog}\rangle \hookrightarrow T$. Two traces are low-equivalent when the states in traces are clockwise low-equivalent.

We formalize observational determinism as follows:

DEFINITION 3 (Observational Determinism). *Program* Prog *obeys observational determinism if for any low-equivalent states σ_1 and σ_2, execution from those states always produces low-equivalent traces:*

$$\sigma_1 \approx_L \sigma_2 \;\wedge\; \langle \sigma_1, \texttt{Prog}\rangle \hookrightarrow T_1 \wedge \langle \sigma_2, \texttt{Prog}\rangle \hookrightarrow T_2 \\ \implies T_1 \approx_L T_2$$

Note that traces include the clock-cycle counter, so this definition is timing-sensitive, controlling timing channels.

5.3 Soundness of SecVerilog

The type system in Section 4 along with a race-condition analysis ensures that well-typed SecVerilog programs satisfy observational determinism.

Race freedom. Today's synchronous hardware design methods disallow race conditions in order to produce deterministic systems. Existing synthesis tools prevent races by ensuring that only one thread updates each variable once per clock cycle. Intuitively, a program is race-free if the sequence of thread executions does not affect the synchronized state.

Soundness proof. We use the notation $\langle c, \sigma\rangle \Downarrow \sigma'$ to denote a big step: fully evaluating command c in state σ results in state σ'. To simplify notation, $\mathcal{V}(\tau, \sigma)$ represents the security level resulting from evaluating type τ in σ.

The first lemma states that any variable assigned to in a high context has a high label in the final state.

LEMMA 1 (Confinement). *Let $\langle \sigma, c\rangle \Downarrow \sigma'$. If c can be typed under a given program counter label pc and well-formed typing environment Γ, then for every variable v assigned in command c, we have*

$$\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma')$$

Proof. By induction on the structure of c. The correctness of the predicate transformer analysis is used for assignments. □

The next theorem states that running a command atomically to finish enforces noninterference.

THEOREM 1 (Single-command noninterference). *If the states σ_1, σ_2 are low-equivalent at the beginning of a clock cycle, running any well-typed command c in σ_1 and σ_2 produces low-equivalent states at the beginning of next cycle as well:*

$$(\vdash \Gamma) \wedge (\Gamma \vdash c) \wedge (\sigma_1 \approx_L \sigma_2) \wedge \langle \sigma_1, c\rangle \Downarrow \sigma_1' \wedge \langle \sigma_2, c\rangle \Downarrow \sigma_2' \\ \implies \sigma_1' \approx_L \sigma_2'$$

Proof. By induction on the structure of c. The interesting case is an if-statement that takes different branches in σ_1 and σ_2. When both branches assign to v, the label of v must be higher than L in both σ_1' and σ_2' by Lemma 1. When v is assigned in only one branch, the correctness of definite-assignment analysis implies $v \notin \mathcal{M}$. Lemma 1 and the no-sensitive-upgrade check (T-ASSIGN-REC) together ensure the label of v is higher than L in σ_1' and σ_2'. □

Finally, any well-typed SecVerilog program obeys observational determinism and is therefore secure:

THEOREM 2 (Soundness of the type system). *If a SecVerilog program is well-typed under any well-formed typing environment, the program obeys observational determinism:*

$$(\vdash \Gamma) \wedge (\Gamma \vdash \texttt{Prog}) \wedge (\sigma_1 \approx_L \sigma_2) \wedge \\ \langle \sigma_1, \texttt{Prog}\rangle \hookrightarrow T_1 \wedge \langle \sigma_2, \texttt{Prog}\rangle \hookrightarrow T_2 \\ \implies T_1 \approx_L T_2$$

Proof. By induction on the length of T_1. For the inductive step, consider two atomic runs in σ_1 and σ_2 where all threads run to finish without being preempted, producing σ_1' and σ_2'. By Theorem 1, states σ_1' and σ_2' are low-equivalent. By the correctness of the race-freedom analysis, all possible runs from σ_1 and σ_2 produce the same states σ_1' and σ_2'. Hence, the final states must be low-equivalent in all runs. □

6. Evaluation

We used SecVerilog to design and verify a secure MIPS processor. We sketch the processor design, and show how SecVerilog helped avoid security vulnerabilities, including some not identified in prior work. We then provide results on the overhead of SecVerilog and timing channel protection. Overall, we found that the capability to statically control information flow at a fine granularity enables efficient secure hardware designs, and that the SecVerilog type system only requires a small number of changes to the Verilog code with no added run-time overhead.

6.1 A secure MIPS processor design

We implemented a SecVerilog compiler based on Icarus Verilog [1]. The constraints generated by the type system are

Module Name	LOC
Fetch	60
Decode + Register File	465
Execute + ALU	218
FPU	N/A
Memory + Cache	537
Write Back	20
Control Logic + Forwarding + Stalling	419
Total w/o FPU	1719

Table 1. Lines of Code (LOC) for each processor component.

Instruction type	Instructions
Additive Arithmetic	add, addi, addiu, addu, sub, subu
Binary Arithmetic	and, or, xor, nor, srl, sra, sll, sllv srlv, srav, slt, sltu, slti, sltiu, andi, ori, xori
Multiply/divide	mult, multu, div, divu
Floating point	add.s, sub.s, mul.s, div.s, neg.s, abs.s mov.s, cvt.s.w, cvt.w.s, c.lt.s, c.le.s
Branch and jump	bne, beq, blez, bgtz, jr, jalr, j, jal
Memory operation	lw, lhu, lh, lbu, lb, sw, sh, sb, swc1, lwc1
Others	mfhi, mflo, lui, mtc1, mfc1 syscall, break
Security-related	setr, setw

Table 2. Complete ISA of our MIPS processor.

solved by Z3 [9]. Using this implementation, we designed a complete MIPS processor that enforces the timing label contract discussed in Section 2.3. Our processor is based on a classic 5-stage in-order pipeline with separate instruction and data caches, both of which are 32kB and 4-way associative. The processor also includes typical pipelining techniques, such as data hazard detection, stalling and data bypassing, as well as a floating point unit (FPU) that we constructed using the Synopsys DesignWare library.

The Verilog code for our processor has more than 1700 LOC excluding the FPU, as shown in Table 1. LOC for the FPU is not reported because the source for the DesignWare library component is not available. Table 2 summarizes the processor's ISA, which is rich enough that we can compile a recent OpenSSL release with an off-the-shelf GCC compiler. The ISA is at least comparable to the ISAs of prior processors with formally verified security (e.g., [22]). New instructions `setr` and `setw` are used to set timing labels.

Our secure processor design supports fine-grained sharing of hardware resources between different security levels. For example, the design allows both high and low cache partitions to be securely used by a single program. This effectively increases the cache size and improves performance for applications with multiple security levels.

To implement such a rich policy, we divide a 4-way cache into a low partition and a high partition. When the timing label is H, both low and high partitions can be used securely. When the timing label is L, both the low and high partitions are still searched. However, to ensure that timing can be affected only by the low cache partition, a cache access is treated as a miss even when there is a hit in the high partition. To avoid data duplication, the cache line moves from the high partition to the low partition when the data arrives from memory, achieving functional correctness without violating the timing constraint. Since cache states have static labels, they are not zeroed out when timing label changes.

The pipeline, on the other hand, is dynamically partitioned using the timing label. When the timing label changes, the pipeline is flushed to avoid leaking information. A pipeline that interleaves high and low instructions without flushing is indeed insecure, since high instructions may stall low ones.

We found that implementing such a complex policy securely would be difficult without using SecVerilog. For example, the SecVerilog type checker caught a security flaw not foreseen by the authors: the dirty bit copied from the high partition to the low partition created a potential timing channel. Our solution is to set the dirty bit for every cache line immediately after it is fetched. This change still allows store hits to write directly to cache line without writing to memory.

Another security issue caught by the SecVerilog type checker is a stall at the instruction fetch stage affecting the memory stage: in our pipeline implementation, a load miss in an instruction could stall instructions in later pipeline stages. Thus, when the timing label changes, an instruction with timing label H can stall another instruction with label L, breaking the timing-label contract. To make the design type-check, the pipeline is flushed at every timing-label switch.

6.2 Overhead of SecVerilog

SecVerilog may require designers to add additional branches to establish invariants needed to convince the type system of the security of the design. For instance, the type system fails to infer in our cache design that variable `way` can only be 2 or 3 at a particular point in the code. In this case, the design needs to include an if-statement establishing this fact. These added branches represents the overhead of using SecVerilog.

To measure this overhead, we compare our secure MIPS processor written in SecVerilog ("Verified") with another secure design written in Verilog ("Unverified"). The Unverified design is essentially the same as Verified, including the same timing channel protections. Because it eliminates the if-statements necessary for type checking, it cannot be verified.

Designer effort and verification time. The Unverified MIPS processor comprises 1692 lines of Verilog code. Converting this design to the Verified processor in SecVerilog required adding only 27 lines of extra code to the cache module, in order to establish necessary invariants to convince the type checker, suggesting that very little overhead is imposed by imprecision of the type system.

The current implementation of SecVerilog requires a programmer to explicitly write down one security label for each variable declaration, unless the variable has the default label L. However, most security labels can be automatically inferred via adding type inference (e.g., as in [7, 26, 46]) to the SecVerilog compiler, which we leave as future work.

The verification process is also fast. For our processor design, it takes a total of two seconds to both generate all obligations and then discharge them with Z3.

Delay, area and power. We synthesized the processor designs using the Synopsys synthesis flow, using the 90nm `saed90nm_max` digital standard cell library. For all designs, we increased the frequency of the processors to the maximum achievable to see what overhead the Verified design adds to the critical path. The synthesis results are shown in Table 3. Here "Insecure" represents the baseline, unmodified MIPS processor without timing channel protection. We discuss the baseline result in the next subsection.

The Verified design only adds 27 lines of code to Unverified, so we found that the delay, area, and power consumption of the two designs are almost identical. For example, area overhead is only 0.16% even without including cache SRAM, which is identical for all designs. This overhead is much lower than that of other secure design techniques, as reported in [22]: GLIFT, 660%; Caisson, 100%; and Sapper, 4%. Power consumption of the two designs is identical. Critical path delay is slightly lower for Verified, likely due to randomness in synthesis. The results show the benefit of sharing hardware across security levels and of controlling information flow at design time, without run-time checks.

Performance. The Verified design does not add any performance overhead over the Unverified design because the added logic does not change cycle-by-cycle behavior.

6.3 Overhead of timing channel protection

The timing channel protection mechanisms in our processor ("Verified") adds overheads compared to the unmodified baseline ("Baseline") without any protection.

Delay, area and power. When an FPU is included, we found that the critical path delay is identical for both Verified and Baseline, as shown in Table 3. This is because the critical path of the processor lies in the FPU, which is largely unmodified for secure designs. To more meaningfully evaluate the impact of secure design, we also measured the maximum achievable frequency without an FPU. Nevertheless, the delay overhead is still only 1.22%. The area overhead of 0.67% is also quite low, and power overhead is almost negligible. Because SecVerilog allows hardware resources to be shared across security levels while properly restricting their allocations, timing channel protection mostly does not require duplicating or adding hardware.

Performance. The timing channel protection in our secure processor design imposes restrictions on cache usage and results in additional pipeline flushes and cache write-backs. We measured the performance overhead of the Verified processor and tested its correctness on two security benchmarks.

	Baseline	Unverified	Verified
Delay w/ FPU (ns)	4.20	4.20	4.20
Delay w/o FPU (ns)	1.64	1.67	1.66
Area (μm^2)	399400	401420	402079
Power (mW)	575.5	575.6	575.6

Table 3. Comparing processor designs.

Our benchmarks include three security programs (blowfish, rijndael, SHA-1) from MiBench, a popular embedded benchmark suite for architectural designs[2] [16], as well as ciphers and hash functions in a recent release (version 1.0.1g) of OpenSSL, a widely used open-source SSL library.

Thanks to the rich ISA of our MIPS processor, compiling and running these benchmarks requires only modest effort. We use an off-the-shelf GCC compiler to cross-compile the benchmarks to the MIPS 1 platform. We use Cadence NCVerilog to simulate our processor design running these binaries. Because we lack an operating system on the processor, system calls (e.g., open, read, close, time) are emulated by Programming Language Interface (PLI) routines. Dynamic memory allocation is implemented by simple code using preallocated static memory.

Most test programs in these benchmarks were used as is. The only exceptions are a few tests in OpenSSL that take a long time to simulate. To make evaluation feasible on these tests, we replace long inputs with shorter ones.

We evaluate two security policies: "nomix", a coarse-grained policy where the entire program is labeled H, corresponding to the security policy targeted by previous secure hardware design methods, and "mixed", a fine-grained policy allowing mixed H and L instructions, enabled by the new features of SecVerilog. In the latter case, we use a simple policy to decide timing labels: for ciphers (e.g., AES, RSA), the encryption and decryption functions are marked as H; for secure hash functions (e.g., MD4, SHA512), we pretend part of the input is secret, and mark the hash functions on these inputs as H. Performance results for a single run of each test are shown in Figure 9. Multiple runs are unnecessary for our evaluation since the simulation is deterministic.

From the MiBench suite, only rijndael shows noticeable performance overhead, at 19.6%. Overhead is reduced to 12.2% when the fine-grained model with mixed labels is used. Overhead on OpenSSL ranges from 0.3% (Blowfish) to 34.9% (SHA-0), with an average of 21.0%. For the fine-grained model, the overhead on OpenSSL ranges from −3.9% (CAST5) to 21.7% (DES), with an average of 8.8%. CAST5 runs faster with the partitioned cache because H instructions cannot evict frequently used data in the L partition.

The results clearly show the benefit of fine-grained information flow control within a single application. Most slowdown comes from the restriction that H instructions cannot write to the low cache partition. Allowing mixed H and L in-

[2] The only benchmark omitted is PGP, which requires a full-featured OS.

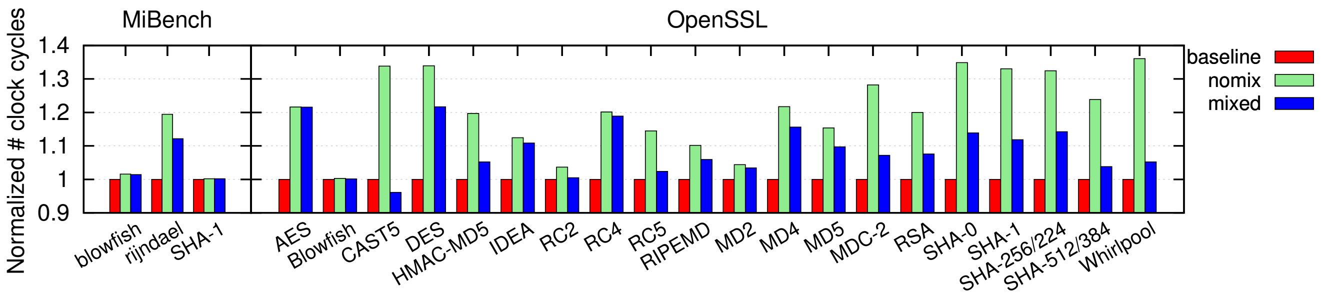

Figure 9. Performance overhead of timing channel protection.

structions in a single program improves performance because the restrictions only apply to a subset of program instructions.

We could not compare our performance overhead with prior work [22, 23, 42] because they do not report the performance overhead over a baseline with unpartitioned cache[3].

7. Related work

Verifiable secure hardware. Dynamic information flow tracking is applied at the logic-gate level in GLIFT [28, 29, 40–42]. Dynamic checks in the initial GLIFT design [42] add high overheads in area, power, and performance. Subsequent work [28, 29, 41] checks designs before fabrication, but enumerates all possible states through gate-level simulation, an approach unlikely to scale to large designs without rigid time-and-space multiplexing. SecVerilog allows more flexible resource sharing and identifies security issues early in the design process.

Sapper [22] also adds logic for tracking information flows, incurring run-time overhead. Sapper cannot capture the dependencies between types and values needed for complex security policies. For example, it would not be possible to use the label `LH(timingLabel)` for variable `stall`, as shown in Figure 2(b), to capture the policy that the label of `stall` must be L when `timingLabel` is 0.

Caisson [23] supports static analysis but with purely static security levels that prevent fine-grained sharing of hardware resources across security levels. E.g., `write_enable`, `tag_in` and `stall` in Figure 2 would require duplication (per security level) since their labels cannot be determined at compile time. Duplicated resources must be controlled by extra encoders and decoders, adding run-time overhead.

Dynamic security labels. Some prior type systems for information flow also support limited forms of dynamic labels [14, 19, 24, 25, 39, 43, 51]. The type-valued functions needed to express the communication of security levels at the hardware level are absent in most of these, and none permit dynamic labels to depend on mutable variables, a feature key to allowing SecVerilog to verify practical hardware designs. The modular design of the SecVerilog type system makes it more amenable to future extension. Fine [37] and F* [38] can verify stateful information flow policies, modeling state changes with affine types. Affine types suffice for functional programming, but HDLs need SecVerilog's new feature of dependence on mutable variables.

Flow-sensitive information flow control. Flow-sensitive information flow control [5, 18, 34], where security labels may change during execution, encounters label channels similar to those observed in our type system. Our type system controls these channels more permissively (Section 4.3) because it captures the dependency between types and values.

Dependent type systems. Dependent types have been widely studied and have been applied to practical programming languages (e.g., [4, 7, 25, 46, 47]). Information flow adds new challenges, such as precise, sound handling of label channels. RHTT [27] supports rich information flow policies with dependent types, but has much more complex specifications and verification is not automatic.

8. Conclusion

We have shown SecVerilog makes it possible to efficiently design complex hardware with strong security assurance. This is enabled by novel features such as type-valued functions for dependent labels, the ability to soundly and precisely use mutable variables within labels, and the modular incorporation of program analyses to improve precision. Using SecVerilog, a reasonably complex processor can be designed in such a way that it satisfies a software–hardware contract for comprehensive control of information flows, including timing channels. The added overhead and effort required to design hardware in this way were both small. Our experience with SecVerilog suggests it will be a useful tool for designing complex hardware with strong security assurance.

Acknowledgments

We thank Tao Chen, Chinawat Isradisaikul, Jed Liu, Dan Lo, Derek Lockhart, Stephen Longfield, Tom Magrino, Matthew Milano and the anonymous reviewers for their helpful suggestions.

This work was partially supported by Office of Naval Research grant N00014-13-1-0089, MURI grant FA9550-12-1-0400, Air Force grant FA8750-11-2-0025, and by National Science Foundation grants CNS-0746913, CCF-0905208, and CCF-0964409.

[3] The previous method [22] calls a secure but unverified design "insecure", and reports the overhead of verified vs. unverified as we do in Section 6.2.

References

[1] Icarus Verilog. http://iverilog.icarus.com/.

[2] O. Acıiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *Proc. ACM Workshop on Computer Security Architecture (CSAW '07)*, pages 11–18, 2007.

[3] O. Acıiçmez, C. Koç, and J. Seifert. On the power of simple branch prediction analysis. In *ASIACCS*, pages 312–320, 2007.

[4] L. Augustsson. Cayenne—a language with dependent types. In *Proc. 3rd ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP)*, pages 239–250, 1998.

[5] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 113–124, 2009.

[6] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, Apr. 2003. ISBN 0321136160.

[7] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *Proc. European Symposium on Programming (ESOP)*, pages 520–535, 2007.

[8] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. *IEEE Symposium on Security and Privacy*, pages 45–60, 2009.

[9] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[10] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, Aug. 1975.

[11] R. W. Floyd. Assigning meanings to programs. In *Proc. Sympos. Appl. Math.*, volume XIX, pages 19–32, 1967.

[12] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[13] M. Gordon. The semantic challenge of Verilog HDL. In *LICS*, pages 136–145, 1995.

[14] R. Grabowski and L. Beringer. Noninterference with dynamic security domains and policies. In *Advances in Computer Science – ASIAN 2009. Information Security and Privacy*, pages 54–68, 2009. LNCS 5913.

[15] D. Gullasch, E. Bangerter, and S. Krenn. Cache games—bringing access-based cache attacks on AES to practice. In *IEEE Symposium on Security and Privacy*, pages 490–505, 2011.

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, 2001.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, Oct. 1969.

[18] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL 33*, pages 79–90, 2006.

[19] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. In *Proc. 13th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP)*, pages 27–38, 2008.

[20] P. Kocher. Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO'96*, Aug. 1996.

[21] B. W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, Oct. 1973.

[22] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong. Sapper: A language for hardware-level security policy enforcement. In *Proc. 19th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 97–112, 2014.

[23] X. Li, M. Tiwari, J. Oberg, V. Kashyap, F. Chong, T. Sherwood, and B. Hardekopf. Caisson: a hardware description language for secure information flow. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 109–120, 2011.

[24] L. Lourenço and L. Caires. Dependent information flow types. *FCT/UNL Technical Report*, Oct. 2013.

[25] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL 26*, pages 228–241, Jan. 1999.

[26] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif 3.0: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2006.

[27] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *Proc. IEEE Symp. on Security and Privacy*, pages 165–179, 2011.

[28] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. Theoretical analysis of gate level information flow tracking. In *Proc. 47th Design Automation Conference*, pages 244–247, 2010.

[29] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. Information flow isolation in I2C and USB. In *Proc. 48th Design Automation Conference*, pages 254–259, 2011.

[30] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. *Topics in Cryptology–CT-RSA 2006*, Jan. 2006.

[31] D. Page. Partitioned cache architecture as a side-channel defense mechanism. In *Cryptology ePrint Archive, Report 2005/280*, 2005.

[32] C. Percival. Cache missing for fun and profit. In *Proc. BSDCan*, 2005.

[33] A. W. Roscoe. CSP and determinism in security modeling. In *Proc. IEEE Symp. on Security and Privacy*, 1995.

[34] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF)*, CSF '10, pages 186–199, 2010.

[35] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[36] V. Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

[37] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *Proc. European Symposium on Programming (ESOP)*, pages 529–549, 2010.

[38] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proc. 16th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP)*, pages 266–278, 2011.

[39] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 369–383, 2008.

[40] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Micro '09*, Dec. 2009.

[41] M. Tiwari, J. Oberg, X. Li, J. K. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proc. International Symposium on Computer Architecture (ISCA)*, June 2011.

[42] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *ASPLOS XIV*, pages 109–120, 2009.

[43] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(1):6, 2007.

[44] Z. Wang and R. Lee. Covert and side channels due to processor architecture. In *ACSAC '06*, pages 473–482, 2006.

[45] Z. Wang and R. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proc. International Symposium on Computer Architecture (ISCA)*, pages 494–505, 2007.

[46] H. Xi. Imperative programming with dependent types. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 375–387, 2000.

[47] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.

[48] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. 16th IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.

[49] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 99–110, 2012.

[50] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A hardware design language for efficient control of timing channels. Technical report, Cornell University, Apr. 2014. `http://hdl.handle.net/1813/36274`.

[51] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *Intl' J. of Information Security*, 6(2–3), Mar. 2007.