# Sapper: A Language for Hardware-Level Security Policy Enforcement

Xun Li∗

Facebook
xun@fb.com

Vineeth Kashyap

University of California, Santa
Barbara
vineeth@cs.ucsb.edu

Jason K. Oberg

University of California, San Diego
jkoberg@cs.ucsd.edu

Mohit Tiwari∗

University of Texas, Austin
tiwari@austin.utexas.edu

Vasanth Ram Rajarathinam∗

AMD
vasanthram.rajarathinam@amd.com

Ryan Kastner

University of California, San Diego
kastner@cs.ucsd.edu

Timothy Sherwood

University of California, Santa
Barbara
sherwood@cs.ucsb.edu

Ben Hardekopf

University of California, Santa
Barbara
benh@cs.ucsb.edu

Frederic T. Chong

University of California, Santa
Barbara
chong@cs.ucsb.edu

## Abstract

Privacy and integrity are important security concerns. These concerns are addressed by controlling *information flow*, i.e., restricting how information can flow through a system. Most proposed systems that restrict information flow make the implicit assumption that the hardware used by the system is fully "correct" and that the hardware's instruction set accurately describes its behavior in all circumstances. The truth is more complicated: modern hardware designs defy complete verification; many aspects of the timing and ordering of events are left totally unspecified; and implementation bugs present themselves with surprising frequency. In this work we describe Sapper, a novel hardware description language for designing security-critical hardware components. Sapper seeks to address these problems by using static analysis at compile-time to automatically insert dynamic checks in the resulting hardware that provably enforce a given information flow policy at execution time. We present Sapper's design and formal semantics along with a proof sketch of its security. In addition, we have implemented a compiler for Sapper

and used it to create a non-trivial secure embedded processor with many modern microarchitectural features. We empirically evaluate the resulting hardware's area and energy overhead and compare them with alternative designs.

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

***Keywords*** Hardware Description Language, Non-interference

## 1. Introduction

Security has become a first-order priority in many system designs. High assurance and life-critical systems, such as aircraft control systems and implantable cardioverter-defibrillators and also systems used in banking and the military, all require strong guarantees about their security properties. However, designing systems with provably strong security properties can be extremely challenging and costly; just assessing the assurance of the resulting system can cost upwards of $10,000 per line of code [3]. Our goal is to enable the design of provably-secure hardware systems such that (1) the designs are easily expressed by hardware designers, and (2) the resulting secure systems have low overhead in terms of chip area, performance, and power consumption.

Information flow security [10] is an important category of security properties that encompasses *confidentiality* (confidential information cannot leak into unclassified chan-

---

∗ The work is done when Xun Li, Mohit Tiwari and Vasanth Ram Rajarathinam are graduate students at University of California, Santa Barbara.

nels) and *integrity* (critical system components cannot be affected or tampered with by untrusted parties). Information flow control mechanisms associate *security labels* with the resources and data contained in a system. A *security policy* is specified by ordering the security labels in a lattice, such that data with labels higher in the lattice are more restricted (i.e., can flow to fewer places) than data with labels lower in the lattice. For example, secret data would be labeled with a high label, and an unsecured output port would be labeled with a low label—this policy specifies that the secret data (or any data directly or indirectly derived from the secret data) should not be sent on the publically-visible port. The goal is to ensure that a principal who can observe all data at some security level $\ell$ cannot deduce any information about data at a security level that is higher than or noncomparable to $\ell$. A system that meets this goal is said to enforce *non-interference* [11]. Many approaches have been previously attempted to enforce non-interference at the software or ISA level, yet the resulting systems are still left vulnerable to timing channels and bugs in the hardware implementation.

The core of our approach to this problem is a novel security-aware hardware design synthesis language called Sapper. The Sapper language is an extension of a synthesizable subset of Verilog; the Sapper compiler translates Sapper code into synthesizable Verilog that is guaranteed to meet a specified security policy. It does so by automatically deriving and inserting dynamic security checks into the Verilog hardware design. The semantics of Sapper ensures that the security properties of the resulting system are *statically guaranteed*, even though these security checks are executed dynamically. During the generated system's execution, these dynamic checks intercept any information flow that violates the security policy and replaces the offending operation with one that is guaranteed to be secure (though potentially changing the intended functionality of the system).

The intended design process makes use of the dynamic checks in two phases. First, during testing but before fabrication[2], these dynamic checks will ensure that security violations are revealed to the hardware designers as functionality bugs, which the modern design process already has many techniques for detecting and dealing with. In other words, Sapper transforms the problem of detecting security violations during testing (for which there is little support and experience in the modern design process) into the problem of detecting functionality bugs during testing (for which there is a great deal of support and experience). Through careful design iteration, the hardware designers can detect and fix security problems to ensure that the system will operate as intended in the vast majority of cases, including all common cases, even with these dynamic checks in place.

Once testing is complete and the hardware is deployed, the second function of these checks comes into play, because

the checks will remain in the final hardware design. The checks serve as the last line of defense against run-time violations in conditions never encountered during testing and verification. Both undiscovered hardware bugs and rarely occurring combinations of events may provide opportunity to attack an unprotected system. In contrast to this, hardware designed with Sapper will automatically capture and prevent any runtime violations.

We describe the design of Sapper and provide a formal semantics and proof sketch of soundness. We demonstrate the expressiveness of Sapper by implementing a fully-featured secure embedded processor with a MIPS ISA and an array of modern microarchitectural features; the processor is complete enough to run real-world benchmarks. We empirically evaluate the overhead of the system generated by Sapper and show that it has only a 4% hardware overhead and no performance loss compared to an insecure baseline version of the same processor.

## 2. Background

There are many ways in which a hardware/software system might be compromised. We begin by describing our specific threat model and assumptions, and then present related work.

### 2.1 Threat Model

Sapper focuses specifically on the information-flow policy of noninterference.[3] In this work we do not consider other security properties of hardware systems.

For our purposes, a *system* consists of a set of input ports, a set of output ports, a hardware implementation, and an initial configuration. The input and output ports are assumed to have a set of security labels (for example, a set of high input "pins" and a set of low input "pins", which may be separate physical components or time multiplexed on the same physical component). An information flow policy is specified as a lattice over those labels. Sapper can enforce policies specified by any finite security lattice. For clarity, the discussion in the following sections assumes a simple security lattice with two labels *high* and *low* such that $low < high$. We evaluate the effect of a more complex security lattice in Section 4. We assume that the initial configuration of the machine is labeled conservatively (i.e., no high state is labeled as low). This initial configuration may include both high and low bits, and those bits may represent anything from executable code, to initial memory states, to the start states for various microarchitectural state machines.

The attacker is assumed to have *complete control* over (1) all low inputs to the device; and (2) all of the bits in the initial state labeled as low. The first assumption models an

---

active attacker. The second assumption models any way in which an attacker might take advantage of access to low data used by the system, including but not limited to malicious or compromised software running on the system. We do not assign intent to the hardware; it simply transforms the state of the system as directed by the hardware design and (after our modifications) subject to the security policy. We assume that hardware is fabricated as specified by our tools, and we do not attempt to address the trusted fabrication problem [5, 13, 35, 36], although we do *not* assume that the hardware has been *designed* correctly or securely.

Using Sapper, we create hardware that ensures the data flowing to any output port conforms to the information flow policy specified by a security lattice given at design time, e.g., that no untrusted information contaminates a trusted port, and no secret information leaks to a non-secret port. This threat model includes *both explicit and implicit* information flows, timing and storage channels, and any other digital form of information flow, but does not include the use of physical or analog phenomena such as EM emission, temperature, or power draw. A system is said to be strictly enforcing a policy if it can be shown that the policy can never be violated regardless of the actions of the attacker subject to this model.

## 2.2   Related Work

Denning and Denning are one of the first to show how programming language techniques and static analysis can be used to enforce information flow policies [10]. This approach was later formalized by Volpano [34] and subsequently implemented as various language extensions [20, 27]. A more comprehensive study of programming language techniques related to information flow security can be found in the survey by Sabelfeld and Myers [25].

While language level techniques provide strong guarantees inside application implementations, security enforcement between applications relies on an underlying operating system. Here too there are many related approaches [14, 16, 17, 24, 28, 39]. Security mechanisms at the OS level cannot provide full hardware/software system security guarantees in the face of adversaries that take advantage of information leakage in the underlying hardware implementation, such as through caches [23] and branch predictors [4]. Specific secure hardware component designs have been proposed to defend against existing covert channel attacks [32, 37, 38]. More systematic approaches have also been proposed to control hardware timing channels through software/hardware contracts [40], quantitative measurements [9], or fuzzing mechanisms [19]. Designing a hardware Trusted Computing Base (TCB) with minimum complexity while providing strong security guarantees is also an active research area [22, 33, 41].

Towards this end, various approaches have been proposed in previous work towards analyzing and enforcing information flow security in hardware designs, including Gate Level Information Flow Tracking (GLIFT) [31] (and its extension Star Logic [30]) and Caisson [18]. While these past approaches represent a first generation of secure hardware design languages, both the expressiveness of those languages (the class of hardware systems that could be shown to be secure) and the efficiency of their implementations (the amount of extra logic required to perform checks) can be prohibitively poor.

GLIFT tracks every single bit of information in the system through each logic gate. Every bit in the system is associated with a shadow bit to represent its security label, and for every logic gate, a shadow logic circuit is used to calculate the output's security level. The values of the gate's inputs are used to achieve precise tracking, e.g. when a low input of an AND gate is known to be 0, the output should be labeled as low even the other input can be high. This feature is extremely important for implementing a practical system in which trusted and untrusted components can be securely multiplexed [29]. Despite GLIFT's pure dynamic nature, the tracking technique is guaranteed to be complete, i.e. it covers both implicit flows and timing channels, since all forms of information flow become explicit at the gate level. Note that GLIFT itself *does not* provide any enforcement mechanism, but rather works as a foundation for information flow tracking. Later work on Execution Leases [29] builds upon GLIFT and enforces non-interference (*by construction*) through memory and timing boundaries.   To reduce the substantial overhead of GLIFT, the authors reworked their method as a static analysis in the form of *Star Logic* [30]. As a verification tool, Star Logic takes a given hardware design, augments it with GLIFT tracking logic, loads a given piece of trusted computing base (e.g., the system kernel) and uses abstract interpretation to explore the execution space and detect potential violations. The tracking logic is removed from the final design before fabrication. It is important to see that Star Logic does not provide assistance to or early feedback for hardware designers attempting to create secure hardware; instead it allows for the *after-the-fact static verification of a coordinated processor and kernel design*, which is not the same problem that Sapper is solving. Hence throughout this work, references to GLIFT are regarding the dynamic tracking technique originally shown by Tiwari et al. [29, 31] and not with reference to Star Logic.

Caisson is another attempt to use programming language techniques to enable secure hardware design [18]. Caisson takes techniques from information flow security type systems at the programming language level and applies them at the hardware level. The Caisson language syntax is very similar to that of Sapper, and in fact Sapper borrows the concept of nested states from Caisson. However, since Caisson uses a purely static type system, all registers must be duplicated for different security levels and multiplexers are used to choose the corresponding register based on the current

security context. The advantage of this approach is that (1) there is no runtime overhead due to the storage or manipulation of labels as labels are never tracked during execution; and (2) the programmer need never worry about the effects of a security violation because, by construction, there can never be one. It does come with two big problems however: (1) statically typing everything requires that resources be hard-partitioned or even duplicated, and then multiplexed at runtime, resulting in large area overheads; and (2) there is no way for the system to ever examine, react to, or affect the flow of program metadata (a.k.a. labels). Labels are strictly a concept used for analysis, and have no physical manifestation in the final design.

## 3. The Sapper Language

We propose Sapper, a hardware description language that enforces noninterference security policies through statically-inserted logic for dynamic tracking and enforcement. Instead of enforcing security policies completely statically via a type system as in our previous work [18], Sapper dynamically tracks security tags runtime, increasing the expressiveness and decreasing the overhead required for security. Importantly, designers do not need to manually insert dynamic tracking or enforcement logic. Instead, the Sapper compiler automatically does so in a provably sound manner. The tracking and enforcement logic is generated based on static analysis of Sapper programs to cover explicit, implicit, and timing based information flows.

Furthermore, Sapper (unlike any prior formally sound hardware approach) enables hardware systems to be aware of, and react to, the security tags of the data that they operate upon. Hardware designers have complete freedom to choose how the system responds to runtime security violations. Despite this flexibility, Sapper does all of this in a way that allows the security properties of the system to be *statically verified* by the Sapper compiler.

### 3.1 Language Overview

Sapper extends upon a core subset of Verilog and requires minimal changes to existing Verilog source code. A typical Verilog program consists of three parts: *signal declarations* that define registers and wires, a *synchronous block*, and a *combinational logic block*. We refer registers here as flip-flops that store data, and wires as communication channels between flip-flops and logic gates. The synchronous block is responsible for writing data back to flip-flops (i.e. registers) at clock edges. The combinational logic block contains computation that will always be executed within one clock cycle. Commands in combinational logic block are similar to those in software programming languages, including assignments, branches and switch/cases. Although in practice hardware designs can be more complicated, most of them can be reduced to such a simple model. In order to make hardware designs less ambiguous, Sapper simplifies

$$
\begin{array}{ll}
r \in Register & n \in \mathbb{Z} \quad \oplus \in Operator \\
t \in SecurityLabel & \ell \in ProgramLabel
\end{array}
$$

$$
\begin{array}{rcl}
prog \in Prog & ::= & \textbf{prog } \ell = \vec{r}, \vec{a} \textbf{ in } d \\
d \in Def & ::= & \textbf{let } \vec{s} \textbf{ in } c \mid c \\
s \in State & ::= & \textbf{state } \ell = d \\
e \in Exp & ::= & n \mid r \mid a[e] \mid e \oplus e \\
a \in RegisterArray & ::= & n \mapsto r \\
te \in TagExp & ::= & t \mid \textbf{tag}(u) \\
u \in TaggedEntity & ::= & r \mid \ell \mid a[e] \\
c \in Cmd & ::= & \textbf{skip} \mid x := e \mid c \, ; c \mid \textbf{fall}_\ell \\
& & \mid \textbf{goto}_{\ell_1} \ell_2 \mid \textbf{setTag}(u, te) \\
& & \mid \textbf{if}_\ell \, e \textbf{ then } c \textbf{ else } c; \textbf{endif} \\
& & \mid c \textbf{ otherwise } c \\
p \in Phrase & ::= & prog \mid d \mid s \mid e \mid c
\end{array}
$$

*Figure 1:* Abstract Syntax of Sapper

the syntax by having designers to only write a single combinational logic block. All writes to regsiters will be automatically compiled into a synchronous block by the Sapper compiler, while computation will remain in the combinational logic block.

Figure 1 describes the abstract syntax of Sapper. We have chosen the core subset such that all the interesting features of the full language are covered. Other language clauses are omitted either because their formal properties are trivial to reason about (e.g., unary operations such as $\sim x$) or they can be expressed equivalently using the syntax discussed in this section (e.g., `case`/`switch`).

In the rest of the section, we start with a list of basic definitions and assumptions; then describe how Sapper implements tracking and checking of security tags; how state machines are an integral Sapper abstraction for precise tracking and enforcement; and finally how Sapper enables runtime security tag manipulation and how Sapper enables configurable yet secure reactions to runtime security violations. The formal semantics of Sapper will be presented in the end of the setion.

### 3.2 Definitions and Assumptions

We define some terms used in the remainder of the section:

- **Variables:** We refer to the set of registers, wires, inputs and outputs in the hardware design as variables.
- **Join:** Given a security lattice, the operation $\sqcup$ takes two security levels as operands and calculates their least upper bound in the lattice, e.g., $high \sqcup low = high$
- **Security Context:** The security context of a statement in a Sapper program is informally defined as the maximum security level among all implicit information flowing into that statement. Sapper models the hardware design as a state machine, and the system can only be in one state at any given cycle. Hence the security context of
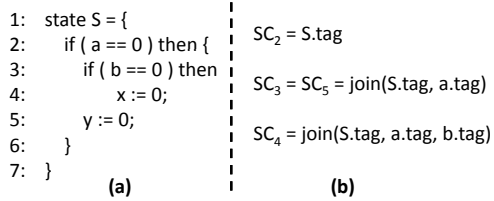
```
1:  state S = {
2:    if ( a == 0 ) then {          SC₂ = S.tag
3:      if ( b == 0 ) then
4:        x := 0;                    SC₃ = SC₅ = join(S.tag, a.tag)
5:      y := 0;
6:    }                             SC₄ = join(S.tag, a.tag, b.tag)
7:  }
          (a)                            (b)
```

*Figure 2:* Example showing how to compute security context.

any statement is determined by the security context of the current state as well as any conditionals guarding the statement. The default security context of any statement inside a state is the same as the security context of the state itself. The security context of a statement inside a branch is calculated by taking the join of the default security context and the security level of the branch condition. Figure 2 illustrates how the security context is determined.

We assume that the security tags of data are public information, i.e., that only knowing the security level of data (but not the data's value) will not leak any information. This assumption is commonly used in previous work for enforcing information flow security. Without this assumption, the system would not be able to react to any runtime security violations, and hence could not enforce the security policy. It is important to note that, Sapper does ensure that security tags cannot be changed based upon data's value in a way that might violate security policies. More details are given in Section 3.5.

We further assume that the set of security tags are based on a statically-known (i.e., defined at design time) security lattice of arbitrary but finite size. Extending Sapper with dynamically-defined lattices rather than using a statically-known lattice would allow different processes in a system to define their own lattices for the hardware to enforce. This can be done through efficient protocols to encode and decode security policies in the hardware. The Raksha architecture proposed by Dalton et al. [8] demonstrates one way to achieve this. Supporting programmable lattices in Sapper is left to future work.

## 3.3   Security Tags

Variables (i.e., signals, wires, etc.) in Sapper are associated with security tags that are tracked and checked for security policy violations at runtime. Checking *every* data movement in hardware for violations of noninterference would be extremely expensive, both in terms of additional hardware and performance overhead. We observe that in most hardware designs only certain outputs are exposed and observable by software/programmers, and thus only these outputs require strict enforcement. Many variables, such as internal pipeline registers and wires used to hold intermediate results, are only used for temporary storage and are not directly observable. These non-observable variables only require security tags to be tracked dynamically so that their secu-

rity level is correctly reflected at runtime; no enforcement is required. As such, Sapper allows designers to declare data variables as one of the following two categories: **Enforced Tagged** variables of which information flow will always be checked for non-interference, and **Dynamic Tagged** variables whose security tags will be automatically tracked and propagated at runtime. By default, a variable declared in Sapper is dynamic tagged. Designers can explciitly declare a enforced tagged variable by giving it a initial security type.

This dichotomy requires designers to make decisions on what data should be tracked versus enforced, but it is often an easy decision to make since typical architectures only consist of a small portion of components exposed to users or central to data movement. In many architectures, selecting enforced tags for all the bus output ports, the memory and the cache will be sufficient. Note that as long as the I/O ports are enforced, not enforcing policies on some of the other components does not lead to unsoundness, but rather makes the system less precise and thus potentially harder to use. The Sapper compiler is responsible for generating dynamic tracking logic and inserting dynamic checks for enforcement depending on the tag of the target variable. Below we describe the details of tracking and enforcement.

### 3.3.1   Tracking Tags

Assignments to dynamic tagged variables will trigger the propagation of security tags: the maximum security level over all information that may affect the assigned value (directly or indirectly) shall be propagated to the target variable's tag (rule *ASSIGN-DYN-REG* in Figure 6(a)). Without careful consideration, simply tracking information at a fine granularity can lead to significant overhead, as in previous work. In general, tracking overhead consists of two parts: extra hardware bits needed to store security labels and extra hardware logic needed to perform tag propagations. Sapper aims at achieving the flexibility of dynamic tracking with minimum overhead. Instead of generating tracking logic for every single logic gate as in some previous work [31], Sapper takes advantage of static analysis on the HDL code and inserts tracking logic aggregately at the granularity of expressions and code blocks. Potential implicit flows (i.e., information flows arising from conditionals) are also derived by the compiler, which inserts logic to ensure sound security tag propagation. Purely dynamic tracking cannot handle implicit flows, and thus the static analysis is required to make this possible.

Sapper also uses simple logic to compute security levels: the security level of the output is the least upper bound of the security levels of the inputs. Sapper tracks security labels and tags data appropriately at the register level; [4] each variable has an $n$-bit tag independent of that variable's width,

---
[4] Note that we do not mean only architectural registers here (like %eax), we mean register-transfer-level registers, which is any set of bits used as a group in the hardware description language.

where $n$ depends on the size of the security lattice. The cache and main memory are each treated as a continuous array of $m$-bit registers (where $m$ is the width of the hardware being designed), with a $n$-bit label for each $m$ bits.

In theory, Sapper may be less precise (but still sound) compared to bit- and gate-level tracking due to the coarser tracking granularity and relaxed tag propagation. However we observe that the major purpose of using precise fine-grained tracking in previous work is to avoid label creep[5] and allow a secure switch from a high to low security context. In the next section we will describe how the "nested states" feature we use in Sapper provides exactly what is needed to satisfy this requirement. In fact, there is nothing that prohibits bit-level tracking in Sapper, but we believe this is not necessary because the state transforms can be expressed in the language itself rather than needing to be inferred from the generated logic. Hence Sapper achieves sufficient precision for security enforcement with significantly less overhead while retaining a high degree of flexibility.

### 3.3.2 Enforcing Policy

Any assignment to a variable with enforced tags needs to be checked for noninterference and violations must be dealt with in a secure way (rule *ASSIGN-ENF-REG* in Figure 6(a)). Specifically, the security level of the target variable must be higher than or equal to the maximum security level of information that may affect the assigned value. The necessary enforcement conditions will be derived by the compiler and the security checks will be automatically inserted into the resulting logic. Therefore, these assignments will take effect only when they are guaranteed to be secure. Sapper also provides flexibility for designers to specify how the system should handle violations, which will be described in Section 3.6. Figure 3 shows the generated Verilog code for an *8-bit-and* design written in Sapper. There are two different cases shown in the figure, one with enforcement (CHECK) while another with tracking (TRACK) only. Note that both the tracking and enforcement logic are automatically generated by the compiler and there is no need for designers to manually specify anything except the initial enforced tags.

### 3.4 A State Machine-based Language

Timing in synchronous hardware designs is strictly aligned to clock edges; for example, registers are only updated at clock edges. To capture the notion of hardware timing, the Sapper language explicitly models hardware designs as *state machines*. During a clock cycle the hardware can only be in one of the state machine's logical states,[6] and all of the program logic from that state will be executed within that clock

---

[5] Label creep happens when a large portion of the system has to be conservatively marked as tainted due to inability of a more precise analysis

[6] In the following discussion, *state* will always mean a state of the finite state machine specified by a Sapper program.

| | Sapper | Verilog |
|---|---|---|
| **CHECK** | reg[7:0] a : L,<br>reg[7:0] b, c;<br>a <= b & c; | reg[7:0] a,b,c;<br>reg a_tag,b_tag,c_tag;<br>if (a_tag>=(b_tag\|c_tag))<br>    a <= b & c; |
| **TRACK** | reg[7:0] a, b, c;<br>a <= b & c; | reg[7:0] a,b,c;<br>reg a_tag,b_tag,c_tag;<br>a <= b & c;<br>a_tag <= b_tag \| c_tag; |

*Figure 3:* An 8-bit adder written in Sapper along with the generated Verilog code. There are two cases: in the first case register $a$ is enforced tagged hence the assignment needs to be checked for noninterference; in the second case $a$ is dynamic tagged hence only tracking is needed.

cycle. State transitions (indicated by **goto** in the Sapper program) always take effect at clock edges. Another important motivation behind modeling hardware as state machines is that state machines are a common pattern used by hardware designers, and most hardware designs are already written as or can be easily transformed to state machines.

Because state transitions can be conditional, they open up the possibility of implicit leaks, i.e., information flow due to conditional execution. Therefore states must also have security tags, and these tags must be correctly propagated or checked during state transitions just like tags for variable. In the same manner as variables, states can be declared with enforced or dynamic tags. The security level of states with dynamic tags will be tracked dynamically at runtime, while states with enforced tags will be enforced for noninterference and their security level will not change unless it is explicitly modified. An immediate advantage of Sapper is that a single dynamic tagged state can safely act at different security levels at runtime, and hence that state can be reused between different security levels instead of requiring the design to duplicate states in order to have one per security level.

To properly enforce noninterference in the presence of conditional execution, a transition from some state $A$ to some state $B$ should only occur if A's tag is lower than B's tag. In the case of a state machine diagram that is strongly connected (i.e., every state can reach every other state), the existence of any high state will eventually require all states to be high. This problem is known as label creep. Sapper uses the concept of *nested states* proposed in Caisson [18] to solve this problem. States can be organized hierarchically as a tree structure. Within each clock cycle, before executing the logic of some state $S$, the logic of $S$'s parent state must be executed first; this rule is recursively applied until the root of the tree. To give parent states complete control over the execution of child states, **fall** commands are used to explicitly indicate transfer of control from the parent state to the child state. By having parent states with low security levels and child states with high security levels, low states

have the freedom to decide when to terminate high states without violating security.

Figure 4 shows an example of a state machine diagram for a secure hardware design based on TDMA (Time Division Multiple Access), which is a common design pattern used by secure systems. A trusted timer (low) is used to control the execution of untrusted components. In particular, the Master state (trusted, labeled with low, enforced tagged) sets up a timer and transits to the Slave state (also trusted, low), which falls into its child state (potentially untrusted, dynamic tagged) and executes the computation logic. At the beginning of every cycle, the Slave State is always executed first and the timer is checked. If the timer expires then control will transfer back to the Master State. The security level of the child state (i.e., Pipeline State in the diagram) can be either high or low at runtime depending on the data it is dealing with at the time. No matter what level it is, it will never affect the behavior of the parent states, thus enforcing noninterference. The corresponding implementation in Sapper is also shown on the side. When the code is compiled down to Verilog, tracking and checking logic will be generated based on the Sapper formal semantics. Although the runtime security level of the Pipeline State is dynamically changing, the generated checking logic will guarantee that the Master State is always trusted.

To ensure that state transitions and falls are securely performed, Sapper provides the following rules for **goto** commands and **fall** commands:

- **goto** *to enforced tagged state (GOTO-ENFORCED):* For a state transition **goto** $S$ in which $S$ is a enforced tagged state, it is required that the security context of the **goto** command be lower than the security level of $S$, otherwise information can leak implicitly.

- **goto** *to dynamic tagged state (GOTO-DYNAMIC):* For a state transition **goto** $S$ in which $S$ is a dynamic tagged state, no enforcement is required. Instead, the security tag of the target state $S$ will be updated to the security context of the **goto** command. Furthermore, if a **goto** command is guarded by conditionals (i.e., the system can transit to different states based on the value of conditions), implicit flows exist from the current state to all reachable states via the conditional **goto** commands. To precisely capture such implicit flows statically, the security tags of all dynamic registers that are assigned in all **goto**-reachable states need to be updated to the security context of the **goto**. This rule is the major cause of label creep in most designs, and Sapper provides nested states to contain states with higher security level in the child group, leaving parent states unaffected by the **goto**s from child states.

- **fall** *to enforced tagged state (FALL-ENFORCED):* **fall** to a enforced tagged state has the same rule as **goto** to a enforced tagged state.
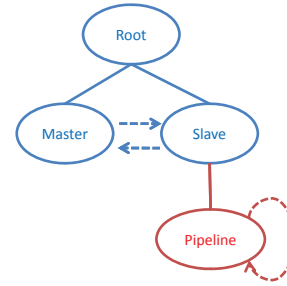


```
reg[31:0] timer : L;

state Master:L = {
    timer = 100;
    goto Slave;
}

state Slave:L = {
    let state pipeline = {
        // Pipeline logic
        goto pipeline;
    }
    in
        if (timer == 0) begin
            goto Master;
        end else begin
            timer <= timer - 1;
            fall;
        end
}
```

*Figure 4:* State Machine Diagram example of a secure hardware controller, along with its corresponding implementation in Sapper. Noninterference is achieved by having a trusted timer controlling the behavior of the computation logic.

- **fall** *to dynamic tagged state (FALL-DYNAMIC):* When there is a **fall** from a state $A$ to its child state $B$, the security tag of $B$ will be updated to the join of the context of the **fall** and the current security level of $B$. The reason we need to take the current security level of $B$ into consideration is as follows: when we have a **goto** from one state to another, their ancestor states will be executed first along with a series of **fall** commands. Hence before a **fall** to state $B$, there can exist a **goto** $B$ which will update $B$'s security level to be the context of the **goto**.

## 3.5 Manipulating Tags

One important advantage of Sapper compared to purely static mechanisms is that security labels can be read, reacted upon, and updated at runtime. As we have defined earlier, the security level of enforced tagged registers will not change until they are explicitly modified through the language provided interface. This feature can be used by system kernels to efficiently and securely share memory among different security levels (e.g., the kernel can allocate memory to a high process, then reclaim it, reset the memory's tag to low, and allocate it to a low process). Although we allow security tags to be modified explicitly, they cannot be modified arbitrarily, otherwise information can be leaked. Sapper provides a pre-defined command **setTag** to allow modification of the security tags of enforced tagged variables and states. Sapper language rules will ensure that no information can be leaked by using this command: (1) the security level of any data can only be changed under a context whose level is not higher than the data's (e.g., low data cannot be hoisted under a high context), thus no information can flow from high to low by manipulating tags; and (2) when data is downgraded (e.g., changed from high to low) the data is automatically *zeroed* instantly to avoid leakage. The logic for checking, changing the tag, and zeroing the data is generated by the Sapper compiler. The formal semantics of this instruction is provided in Figure 6(a) (SET-REG-TAG). Explicit declassification is not supported as Sapper is currently targeted to

information-theoretic rather than cryptographic notions of security. Declassification can be added and is certainly useful in many cases (e.g. a crypto unit). It will be a simple relaxation of our existing mechanism for modifying security levels. Of course one must be careful how declassification is allowed; but these mechanisms have been extensively studied in the literature [21, 26].

### 3.6 Handling Security Violations

As previously described, the Sapper compiler will insert checking logic in the appropriate locations in the design to detect all security policy violations. The natural question, then, is how should the system react if the runtime check does not pass, i.e., the security policy is about to be violated. To handle this situation, the Sapper compiler inserts alternative actions that are executed instead of any violating operation. For each enforced operation, Sapper has a default replacement action that is guaranteed to be secure. These default actions are inserted into the generated Verilog code in the form of branches as shown in Figure 5(a). For example, a default secure action can turn a violating assignment into a noop, or turn a violating state transition into a different transition to a secure state.

To give hardware designers full flexibility to decide how to react to runtime security violations, Sapper provides a language interface for specifying the replacement behavior when violation is *about* to happen. The syntax is shown in Figure 5(b), which specifies that if there exists any security violation in *command*, *secure action* will be executed in replace of *command*. The above code will also be transformed into a branch by the Sapper compiler as shown in the figure.

Note that *command* will never be speculatively executed, instead, only one of *secure action* and *command* will be executed depending on the value of the generated condition. These *otherwise* rules can be defined recursively, meaning that the action in the otherwise branch can itself have potential violations or even another otherwise clause. These nested otherwise clauses are terminated by the default, guaranteed safe action; thus all commands in the program are guaranteed to be secure even if designers provide buggy otherwise clauses. figure 5(c) shows an example when the provided action is also a command that requires runtime enforcement.

### 3.7 Formal Semantics of Sapper

Figure 6(a) provides the formal semantics for Sapper. The semantic domains and evaluation context grammar used in these semantic rules are provided in Figure 6(d). The semantics in Figure 6(a) also makes use of big-step semantic relations for expressions (given by $\Downarrow_e$, which are standard and omitted) and big-step semantic rules for tag expressions provided in Figure 6(b). The definition of $\phi$, which is used in the semantic rules, is provided in Figure 6(c). The abstract machine configuration (provided in Figure 6(d)) for
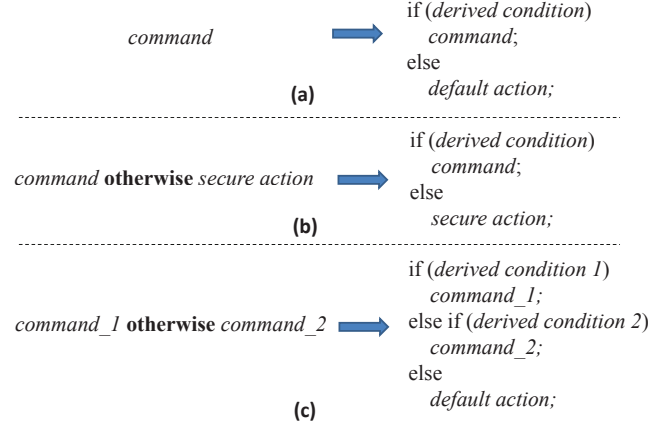


*Figure 5:* Violation handling logic generated by our Sapper compiler: (a) any command that requires enforcement will be guarded by conditions that enforce noninterference. These conditions are generated by static analysis on the context of the command. If the conditions fail, a secure default action (provided by the compiler) will be executed instead; (b) designers can also specify replacement actions of their own, using the *otherwise* clause; (c) When the designer-provided action also requires enforcement, our compiler will generate conditions and default secure actions recursively.

the semantic rules in Figure 6(a) consists of (1) the current program phrase $p$, (2) a fall map $\rho$, (3) a store $\sigma$, (4) a tag map $\theta$, (5) a security context stack $\mathcal{S}$, and (6) current time value $\delta$.

We define the following mappings for each $\ell \in \textit{ProgramLabel}$ that are used in the semantic rules:

- if $\ell$ refers to a state name, $\mathcal{F}_{pnt}(\ell)$ maps to the label of state $\ell$'s parent state.

- if $\ell$ refers to a state name, $\mathcal{F}_{cmd}(\ell)$ maps to state $\ell$'s command.

- If $\ell$ is a label attached to an **if** statement, then $\mathcal{F}_{cd}(\ell)$ maps to the set of all the registers that are targets of assignments control-dependent on the **if** statement, union with the set of all dynamic states whose reachability (via **goto** or **fall**) is control dependent on the **if** statement.

In addition, $\mathcal{F}_{root}$ maps to the root command of the **prog** program phrase. Finally, we also define the following two helper functions:

- *ResetFallMap* takes a *FallMap* $\rho$ and a state label $\ell$ and returns a new *FallMap* $\rho'$ identical to $\rho$ except that label $\ell$ and the labels of all states that are descendents of $\ell$ in the state hierarchy are mapped to their default child states (the same as their initial values).

- *ResetTagMap* takes a *TagMap* $\theta$ and a state label $\ell$ and returns a new *TagMap* $\theta'$ identical to $\theta$ except that (1) labels of all dynamic states that are descendents of $\ell$ in the state hierarchy are mapped to $\perp$ (2) if *isDynamicState*$(\ell)$, $\ell$ is mapped to $\perp$.

A proof of non-interference is given in Appendix A.

$$\frac{isEnforcedRegister(r) \qquad \langle e,\sigma\rangle \Downarrow_e n \qquad \sigma' = \begin{cases} \sigma[r\mapsto n] & : \theta(r)\sqsupseteq \phi(e)\sqcup sc \\ \sigma & : otherwise\end{cases}}{\langle E[r:=e],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \langle E[\mathbf{skip}],\rho,\sigma',\theta,\mathcal{S},\delta\rangle}$$
(ASSIGN-ENF-REG)

$$\frac{\neg isEnforcedRegister(r) \qquad \langle e,\sigma\rangle \Downarrow_e n}{\langle E[r:=e],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \langle E[\mathbf{skip}],\rho,\sigma[r\mapsto n],\theta[r\mapsto \phi(e)\sqcup sc],\mathcal{S},\delta\rangle}$$
(ASSIGN-DYN-REG)

$$\frac{\begin{array}{c}\langle e_1,\sigma\rangle \Downarrow_e n_1 \qquad \langle e_2,\sigma\rangle \Downarrow_e n_2 \\ a(n_1)=r \qquad isEnforcedRegister(r) \\ \sigma' = \begin{cases} \sigma[r\mapsto n_2] & : \theta(r)\sqsupseteq \phi(e_1)\sqcup \phi(e_2)\sqcup sc \\ \sigma & : otherwise\end{cases}\end{array}}{\langle E[a[e_1]:=e_2],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \langle E[\mathbf{skip}],\rho,\sigma',\theta,\mathcal{S},\delta\rangle}$$
(ASSIGN-ENF-REG-ARR)

$$\frac{\begin{array}{c}\langle e_1,\sigma\rangle \Downarrow_e n_1 \qquad \langle e_2,\sigma\rangle \Downarrow_e n_2 \\ a(n_1)=r \qquad \neg isEnforcedRegister(r) \\ \theta_1 = \theta[r\mapsto \phi(e_1)\sqcup \phi(e_2)\sqcup sc]\end{array}}{\begin{array}{c}\langle E[a[e_1]:=e_2],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \\ \langle E[\mathbf{skip}],\rho,\sigma[r\mapsto n_2],\theta_1,\mathcal{S},\delta\rangle\end{array}}$$
(ASSIGN-DYN-REG-ARR)

$$\frac{\begin{array}{c}\langle e,\sigma\rangle \Downarrow_e n \qquad c' = \begin{cases} c_1 & : n=0 \\ c_2 & : n\neq 0\end{cases} \\ \theta' = \theta[\forall x.x\in \mathcal{F}_{cd}(\ell) : x\mapsto \theta(x)\sqcup \phi(e)\sqcup sc]\end{array}}{\begin{array}{c}\langle E[\mathbf{if}_\ell\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2; \mathbf{endif}],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \\ \langle E[c'; \mathbf{endif}],\rho,\sigma,\theta',(\phi(e)\sqcup sc)::\mathcal{S},\delta\rangle\end{array}}$$
(IF)

$$\frac{}{\begin{array}{c}\langle E[\mathbf{skip}; \mathbf{endif}],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \\ \langle E[\mathbf{skip}],\rho,\sigma,\theta,\Sigma,\delta\rangle\end{array}}$$
(ENDIF)

$$\frac{\begin{array}{c}\langle te,\theta,\sigma\rangle \Downarrow_t t \\ \theta' = \begin{cases} \theta[r\mapsto t] & : \theta(r)\sqsupseteq sc \wedge t\sqsupseteq sc \\ \theta & : otherwise\end{cases} \\ \sigma' = \begin{cases} \sigma[r\mapsto 0] & : \theta(r)\sqsupseteq t \wedge t\sqsupseteq sc \\ \sigma & : otherwise\end{cases}\end{array}}{\langle E[\mathbf{setTag}(r,te)],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \langle E[\mathbf{skip}],\rho,\sigma',\theta',\mathcal{S},\delta\rangle}$$
(SET-REG-TAG)

$$\frac{\langle te,\theta,\sigma\rangle \Downarrow_t t \qquad \theta' = \begin{cases} \theta[l\mapsto t] & : \theta(\ell)\sqsupseteq sc \wedge t\sqsupseteq sc \\ \theta & : otherwise\end{cases}}{\langle E[\mathbf{setTag}(\ell,te)],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \langle E[\mathbf{skip}],\rho,\sigma,\theta',\mathcal{S},\delta\rangle}$$
(SET-STATE-TAG)

$$\frac{\begin{array}{c}\langle te,\theta,\sigma\rangle \Downarrow_t t \qquad \langle e,\sigma\rangle \Downarrow_e n \qquad a(n)=r \\ \theta' = \begin{cases} \theta[r\mapsto t] & : \theta(r)\sqsupseteq sc \wedge t\sqsupseteq sc\sqcup \phi(e) \\ \theta & : otherwise\end{cases} \\ \sigma' = \begin{cases} \sigma[r\mapsto 0] & : \theta(r)\sqsupseteq t \wedge t\sqsupseteq sc\sqcup \phi(e) \\ \sigma & : otherwise\end{cases}\end{array}}{\langle E[\mathbf{setTag}(a[e],te)],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \langle E[\mathbf{skip}],\rho,\sigma',\theta',\mathcal{S},\delta\rangle}$$
(SET-REG-ARR-TAG)

$$\frac{}{\langle E[\mathbf{skip}\ ;\ c],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \langle E[c],\rho,\sigma,\theta,\mathcal{S},\delta\rangle}$$
(SEQ)

$$\frac{\begin{array}{c}isEnforcedState(\ell) \qquad p' = \begin{cases} \mathcal{F}_{cmd}(\ell') & : \theta(\ell')\sqsupseteq sc \\ E[\mathbf{goto}_\ell\ \ell] & : otherwise\end{cases} \\ \ell' = \rho(\ell) \qquad \mathcal{S}' = \begin{cases} \theta(\ell')::Nil & : \theta(\ell')\sqsupseteq sc \\ \mathcal{S} & : otherwise\end{cases}\end{array}}{\langle E[\mathbf{fall}_\ell],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \langle p',\rho,\sigma,\theta,\theta(\ell)::Nil,\delta\rangle}$$
(FALL-ENFORCED)

$$\frac{\begin{array}{c}isDynamicState(\ell) \qquad \theta' = \theta[\ell'\mapsto sc\sqcup \theta(\ell')] \\ \mathcal{S}' = \theta'(\ell')::Nil \qquad \ell' = \rho(\ell)\end{array}}{\langle E[\mathbf{fall}_\ell],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \langle \mathcal{F}_{cmd}(\rho(\ell)),\rho,\sigma,\theta',\mathcal{S}',\delta\rangle}$$
(FALL-DYNAMIC)

$$\frac{\begin{array}{c}isEnforcedState(\ell_2) \qquad \ell = \begin{cases} \ell_2 & : \theta(\ell_2)\sqsupseteq sc \\ \ell_1 & : otherwise\end{cases} \\ \rho_1 = ResetFallMap(\rho,\ell) \qquad \rho_2 = \rho_1[\mathcal{F}_{pnt}(\ell)\mapsto l] \\ \mathcal{S}' = \theta(\mathcal{F}_{root})::Nil\end{array}}{\langle E[\mathbf{goto}_{\ell_1}\ \ell_2],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \langle \mathcal{F}_{root},\rho_2,\sigma,\theta,\mathcal{S}',\delta+1\rangle}$$
(GOTO-ENFORCED)

$$\frac{\begin{array}{c}\rho_1 = ResetFallMap(\rho,\ell_2) \qquad \rho_2 = \rho_1[\mathcal{F}_{pnt}(\ell_2)\mapsto \ell_2] \\ isDynamicState(\ell_2) \qquad \mathcal{S}' = \theta_2(\mathcal{F}_{root})::Nil \\ \theta_1 = \theta[\ell_2\mapsto sc] \qquad \theta_2 = ResetTagMap(\theta_1,\ell_1)\end{array}}{\langle E[\mathbf{goto}_{\ell_1}\ \ell_2],\rho,\sigma,\theta,\mathcal{S},\delta\rangle \rightsquigarrow \langle \mathcal{F}_{root},\rho_2,\sigma,\theta_2,\mathcal{S}',\delta+1\rangle}$$
(GOTO-DYNAMIC)

(a) Semantic rules for Sapper. Throughout the rules, we assume $\mathcal{S} = sc::\Sigma$. Various helper functions are used, which are described in the accompanying text.

$$\frac{\langle t,\theta,\sigma\rangle \Downarrow_t t}{\text{(T-SEC-LBL)}}$$

$$\frac{\langle e,\sigma\rangle \Downarrow_e n \qquad a(n)=r}{\langle \mathbf{tag}(a[e]),\theta,\sigma\rangle \Downarrow_t \theta(r)\sqcup \phi(e)}$$
(T-REG-ARR)

$$\frac{}{\langle \mathbf{tag}(r),\theta,\sigma\rangle \Downarrow_t \theta(r)}$$
(T-REG)

$$\frac{}{\langle \mathbf{tag}(\ell),\theta,\sigma\rangle \Downarrow_t \theta(\ell)}$$
(T-STATE)

(b) Big-step semantics for *TagExp*. $\Downarrow_t\subseteq \langle te,\theta,\sigma\rangle \times t$

$$\phi(n) = \bot \qquad\qquad \phi(e_1\oplus e_2) = \phi(e_1)\sqcup \phi(e_2)$$

$$\phi(r) = \theta(r) \qquad\qquad \frac{\langle e,\sigma\rangle \Downarrow_e n \qquad a(n)=r}{\phi(a[e]) = \phi(e)\sqcup \theta(r)}$$

(c) Definition of $\phi : e \mapsto t$. $\Downarrow_e$ is the standard big step semantics relation.

$$\rho \in FallMap : ProgramLabel \rightarrow ProgramLabel$$

$$\theta \in TagMap : TaggedEntity \rightarrow SecurityLevel$$

$$\sigma \in Store : Register \rightarrow \mathbb{Z} \qquad \mathcal{S} \in List \qquad \delta \in Time : \mathbb{N}$$

$$\mathbb{C} \in Config : \langle p,\rho,\sigma,\theta,\mathcal{S},\delta\rangle$$

$$E ::= \Box \mid x := E \mid E\ ;\ c \mid \mathbf{prog}\ \ell = \vec{r},\vec{a}\ \mathbf{in}\ E$$
$$\mid\ \mathbf{let}\ \vec{s}\ \mathbf{in}\ E \mid \mathbf{setTag}(\ell,E) \mid \mathbf{setTag}(r,E)$$
$$\mid\ \mathbf{setTag}(a[E],te) \mid \mathbf{setTag}(a[n],E)$$

(d) Semantic domains and evaluation context grammar.

*Figure 6:* Sapper Semantics

| Instruction Type | Instruction List |
|---|---|
| Additive Arithmetic | add, addu, addiu, sub, subu |
| Binary Arithmetic | and, andi, or, ori, xor, xori, nor, sll, sllv, sra, srav, srl, srlv |
| Multiplicative Arithmetic | mult, multu, div |
| FPU instructions | add.s, sub.s, mul.s, div.s, neg.s, abs.s, mov.s cvt.s.w, cvt.w.s, le.s, lt.s, ge.s, gt.s |
| Branch | beq, bgt, ble, bne, bltz, bgez; beql, bnel, blel, bltzl; bc1t |
| Jump | j, jr, jal, jalr |
| Memory Operation | lb, lbu, lhu, lw, sb, sh, sw; lwl, lwr, swl, swr; swc1, lwc1 |
| Others | slti, sltiu, lui, mflo, mfhi, mtc1, mfc1 |
| Security  Related | set-tag*, set-timer |

*Figure 7:* Complete ISA of our processor.

| Module Name | LOC |
|---|---|
| Fetch | 52 |
| Decode + Register File | 590 |
| Execute + ALU + FPU | 3981 |
| Memory + Cache | 442 |
| Write Back | 29 |
| Control Logic + Forwarding + Stalling | 303 |
| **Total** | **5397** |

*Figure 8:* Lines of Code (LOC) of each component in our processor.

# 4. Processor Design and Evaluation

Sapper is capable of building a wide variety of security critical hardware, e.g., an arbiter, a network-on-chip, or a secure co-processor. In this section, we evaluate Sapper's utility specifically on one of the most general purpose designs one might consider building: a pipelined microprocessor. Our microprocessor design is significantly more complex than any existing processor designs with strong security properties presented in the literature [18, 29, 30]. We describe in detail what kind of processor we have built, how we implement it in Sapper to enforce noninterference, and how we validate its functional correctness and security enforcement. Finally, we demonstrate the benefits of our technique through empirical evaluation of hardware overhead for two security policies: a simple two-level policy, and a more complicated "diamond" policy.

## 4.1 Processor Description

Our processor design is a 5-stage pipelined processor with a number of components that can be found in modern processors, including cache, a division/multiplication unit, and a floating point unit that alone accounts for 3000+ lines of code. Typical pipelining techniques, including hazard detection, stalling, and data forwarding, are all implemented. Figure 8 lists the components of the processor along with their lines of code (LOC). The total length of the implementation is over 5K LOC. A majority of the standard MIPS ISA is implemented as shown in Figure 7. The ability to design a practical processor with strict security properties that has

logic similar to that found in modern CPUs has significant value—not only because more complex systems are simply harder to evaluate for security, but because it also demonstrates the expressiveness of our technique.

We implement an L1 cache, shared by both data and instructions. The main memory is modeled as a single large register array (64MB) that supports byte-level random access with one read port and one write port. This simplified memory system still allows us to exercise the basic functionality of the processor and cache in a non-trivial manner.

## 4.2 Security Enforcement

Along with the standard MIPS instructions, the processor supports two extra security related instructions called *set-tag* and *set-timer*. Neither instruction is part of the Sapper language syntax, rather, they are hardware-provided ISA instructions *implemented* in Sapper. It is worth noting that here *set-tag* is an ISA instruction of the processor, while **setTag** described in the previous section is a command in the Sapper syntax. The *set-tag* instruction allows software to explicitly modify the security tag of a word in memory (the memory is modeled as an array of enforced tagged words). This instruction is implemented through the **setTag** primitive in Sapper. Security checks to ensure the safety of the instruction are automatically generated as we have described in Section 3.5.

The *set-timer* instruction allows software to set the timing boundary of any untrusted program and allows for securely switching from a high to low security context under the control of software running on the processor. It is implemented in the hardware by taking advantage of the nested states feature in Sapper: the parent state controls the timer (an enforced tagged register) which is always labeled as low, and it checks whether the timer expires every cycle while running high code in child states. When the timer expires it always switches back to the low state. Calling the *set-timer* instruction from software will modify the value of the timer register. Because it is an enforced tagged register, the value change operation will always be checked to ensure that a high program can never change the value of timer. The state machine structure of our processor is similar to that shown in our previous example in Figure 4.

Without Sapper, one has to implement a cache very carefully, using sophisticated techniques in order to deliver strong security guarantees. However, even sophisticated designs can still be vulnerable to attacks due to unforeseen considerations [15]. In contrast, a cache implemented with Sapper is always guaranteed to enforce noninterference no matter how it is implemented. However, it is worth mentioning that an *insecure cache design if Sapper is not involved* will become a *buggy cache when Sapper is applied*, since the checking logic will change cache behavior when violations are about to occur. Designers must make certain that the resulting cache design is functionally correct, which fortunately is a requirement even when security is not a design

goal—the processor can be tested without changing existing testing frameworks. In our processor design, we simply implement the cache as two partitions corresponding to the two security levels. We choose this scheme for its simplicity and effectiveness, however Sapper is not limited to this technique. For example, one could choose to flush the cache at every context switch from *low* to *high*. More advanced scheduling schemes can also be applied and their security guarantees can be easily tested.

The advantage of Sapper over previous techniques becomes clear when looking at the implementation of memory. Instead of being forced to partition the memory into independent sections with one per security level, as proposed by all previous approaches, Sapper allows sharing of the same memory among different security levels in a secure manner. The *set-tag* instruction allows the kernel to manage the memory and recall resources securely upon context switches.

### 4.3 Functional Validation

We implement our processor in Sapper and compile it to Verilog using the Sapper compiler. To demonstrate the complexity and functional completeness of our processor design, we use Mentor Graphics ModelSim to simulate our processor. We pick applications from various benchmark suites and load them into our processor. Since we have not developed a fully-featured operating system that supports I/O and dynamic memory allocation, we modify each application so that all I/O operations are in memory and memory resources are statically allocated. We pick benchmarks from two benchmark suites for evaluation, SPEC CPU 2006 [2] and MiBench [12], due to their popularity in evaluating architectural designs. We only choose a representative subset from each benchmark suite due to significant efforts required to modify each application to run on our processor without operating system support. Note that we do not modify the applications in any way that would change the functionality or behaviors that are not related to system calls. As such, we are able to validate our processor by cross-comparing the output of a benchmark on our processor with its output on a real machine.

The evaluated applications include three benchmarks (mcf, specrand and bzip2) from SPEC CPU2006, 2 security benchmarks (sha, rijndael), and one floating point benchmark (FFT) from MiBench. These applications are compiled with the GCC 3.4.4 cross-compiler targeting MIPS binaries, which are then loaded into a dedicated range of memory in our processor. We chose this GCC version due to its simplicity when used as a cross-compiler for generating well-formatted MIPS binaries. Being able to run realistic and diversified benchmark suites on the processor shows the potential and power of Sapper, that it is possible to design and implement a provably information flow secure processor that is comparable to commercial embedded processors. In fact, running real benchmarks forced us to debug all kinds of corner cases, some related to functional correctness and some captured by Sapper as security flaws. We spent roughly five times as long debugging the processor as we spent in its initial development.

### 4.4 Security Validation

We demonstrate that our processor enforces noninterference at runtime by running applications with different security levels on our processor. To do so, we implement a simplified micro-kernel in MIPS assembly. It contains a scheduler that can schedule among multiple processes with different security levels and be responsible for storing/restoring the registers during a context switch. A static memory allocation mechanism is also implemented to provide private stack space for each process. It is important to note that the micro-kernel is not providing any security enforcement: all security enforcement is achieved through Sapper in the processor. The major responsibility of the micro-kernel is to provide a useful interface to applications running on it.

To take advantage of the processor's capability to prevent *timing leaks*, the kernel issues a "set-timer" instruction before every switch to an untrusted application. The processor guarantees that when the timer expires, the control always jumps back to the kernel. The kernel will be responsible for storing all register values and restoring them before the next switch to the same application to ensure it continues from wherever it pauses. To make the system more efficient, the micro-kernel is responsible for reclaiming memory regions for reuse through *set-tag* instructions.

### 4.5 Quantitative Evaluation

For a quantitative analysis, we evaluate the overhead that Sapper imposes on the hardware by synthesizing our processor to a 90nm library provided by Synopsys [1] and compare this to an insecure base processor in terms of area (ASIC chip area), delay (estimates minimum clock period), and power (total power including both leakage power and dynamic power). We also compare Sapper with two existing secure hardware design mechanisms, namely GLIFT [31] and Caisson [18]. Since neither Caisson nor Sapper changes the number of cycles a program will execute, the clock delay determines performance.

We first implement our processor design without any security features using standard Verilog (designated the *Base Processor*). To collect overhead results for our processor design with GLIFT logic, the base processor is first synthesized using Synopsys' Design Compiler targeting its `and_or.db` library which contains only gate primitives (AND, OR, and Inverters) and flip-flops. At this stage, the processor is augmented with GLIFT logic by associating information flow tracking logic with each gate. Once the tracking logic has been added, the GLIFT processor is synthesized again, this time targeting the 90nm library to obtain area, delay and power results. Note that GLIFT only provides tracking logic without enforcement (which can be done through an architectural mechanism named Execution

| Processor | Area (µm²) | | Delay (ns) | | Power (mW) | | Memory |
|---|---|---|---|---|---|---|---|
| Base Processor | 560,687 | | 5.57 | | 2.79 | | |
| GLIFT | 4,277,412 | **7.6X** | 21.62 | **4X** | 13.4 | **5X** | **2X** |
| Caisson | 1,171,448 | **2X** | 6.36 | **1.14X** | 5.54 | **2X** | **2X** |
| Sapper | 582,333 | **1.04X** | 5.57 | **1X** | 2.93 | **1.05X** | **1.03X** |

*Figure 9:* Hardware overhead comparison of processors designed with different techniques: the original non-secure processor (Base Processor), processor designed under GLIFT, Caisson and Sapper, normalized to the Base Processor.

Lease [29]), however adding Execution Lease support would make the overhead even larger than GLIFT alone [29].

We then migrate our base processor (as a Verilog program) into both Caisson and Sapper, both enforcing the same noninterference guarantee. We also add the two new security-related instructions to the ISA in both processors to support an interface for software. Both designs are then compiled back into Verilog—the design in Caisson is type checked, while the design in Sapper is augmented with tracking and checking logic. Once both designs are in Verilog, the designs are synthesized using the same 90nm library to obtain results. Memory itself cannot be synthesized directly using this methodology, and depending on the technology, size, and type of memory, varying results would be obtained. As a result, we only synthesize the datapath and control logic of the processor. This does, however, include the logic associated with the memory subsystem, including cache control logic.

We now have four versions of the processor design, with almost identical functionality (because the GLIFT version does not provide enforcement). Figure 9 shows the overhead for the Base, GLIFT, Caisson, and Sapper processors. The GLIFT and Caisson processors incur a $7.6\times$ and $2\times$ area overhead, respectively. The Sapper processor, on the other hand, imposes an overhead of only $1.04\times$. Similar observations also apply to delay and power. We also observe that Caisson's intensive use of multiplexers to select the right security level for each register/wire use leads to significant overhead when the processor design complexity increases. Elimination of those multiplexers is a big gain for Sapper. Furthermore, since both GLIFT and Caisson require duplication of resources, memory itself must be duplicated, while in Sapper no duplication is necessary (only extra tag store is required, taking 3% overhead). This provides further incentive for using Sapper in systems which are heavily memory dominated and have little space for extra functional logic.

### 4.6 Diamond Lattice

To demonstrate the scalability of Sapper, we further augment our processor with a more complex security lattice that contains four security levels ($L$, $H$, $M1$, and $M2$). This "diamond lattice" has $L$ as the lowest element, $H$ as the highest element, and $M1$ and $M2$ both being higher than $L$ and less than $H$ (and noncomparable with each other). This lattice can be used to demonstrate the enforcement of both secrecy and integrity in the same system, a big step towards implementing a practical provably secure system. We found that supporting a more complex lattice is as natural as implementing a two-level lattice, since all states will be traced and checked in similar ways. We did not need to modify the Sapper design except for the lattice specification. Major changes to the generated design (introduced by the Sapper compiler) include one more bit for each tag and relatively more complicated checking logic to compare among four different security levels. We do need to modify the micro-kernel to handle the scheduling among four different security levels, however this was straightforward. With the same evaluation method as used above, a secure processor enforcing noninterference on a diamond lattice using Sapper incurs only slightly more overhead (3% more) compared to a two-level lattice counterpart. While supporting such kind of lattice in Caisson would require duplicate all resource into four pieces.

## 5. Conclusions

This paper is a step towards a new class of tools that help inform hardware designers about the security ramifications of their design choices and that assist them in guarding against unforeseen exploits. We explore provable security properties in hardware designs through Sapper, an extension of the Verilog hardware description language that automatically augments a hardware design with appropriate security checks so that it is impossible to violate an information flow security policy. While there is still more work to do, we have proven the security of Sapper's constructions with respect to specified policies; we have shown Sapper's expressive power by designing a complex processor core that includes non-trivial microarchitectural features complete enough to execute a number of real-world programs; and we have performed a quantitative evaluation that shows we can provide security at a reasonably low cost while remaining expressive enough to describe complex and interesting designs.

### Acknowledgments

### References

[1] 90nm generic CMOS library, Synopsys University program, Synopsys Inc.

[2] SPEC CPU 2006. http://www.spec.org/cpu2006/.

[3] What does CC EAL6+ mean? http://www.ok-labs.com/blog/entry/what-does-cc-eal6-mean/.

[4] O. Accigmez, J. pierre Seifert, and C. K. Koc. Predicting secret keys via branch prediction. In *The Cryptographers' Track at the RSA Conference*, 2007.

[5] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan detection using IC fingerprinting. *Security and Privacy*, 2007.

[6] T. Alves. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4), 2004.

[7] R. Benadjila, O. Billet, S. Gueron, and M. J. Robshaw. The Intel AES instructions set and the SHA-3 candidates. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, 2009.

[8] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 482–493, New York, NY, USA, 2007. ACM.

[9] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 106–117, Washington, DC, USA, 2012. IEEE Computer Society.

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

[11] J. A. Goguen and J. Meseguer. Security policies and security models. In *Security and Privacy*, 1982.

[12] 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 *WWC*, 2001.

[13] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and implementing malicious hardware. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, LEET'08, pages 5:1–5:8, Berkeley, CA, USA, 2008. USENIX Association.

[14] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[15] J. Kong, O. Aciiçmez, J.-P. Seifert, and H. Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proc. of the 2nd ACM workshop on Computer security architectures*, 2008.

[16] M. Krohn and E. Tromer. Noninterference for a practical difc-based operating system. In *Security and Privacy*, 2009.

[17] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM.

[18] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf. Caisson: A hardware description language for secure information flow. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 109–120, New York, NY, USA, 2011. ACM.

[19] R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 118–129, Washington, DC, USA, 2012. IEEE Computer Society.

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

[21] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Comput. Secur.*, 14(2):157–196, Apr. 2006.

[22] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

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

[24] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 63–74, New York, NY, USA, 2009. ACM.

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

[26] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE workshop on Computer Security Foundations*, 2005.

[27] V. Simonet. Flow Caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*, 2003.

[28] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 85–96, New York, NY, USA, 2004. ACM.

[29] 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 *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 493–504, New York, NY, USA, 2009. ACM.

[30] M. Tiwari, J. K. Oberg, X. Li, J. 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 *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 189–200, New York, NY, USA, 2011. ACM.

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

*Languages and Operating Systems*, ASPLOS XIV, pages 109–120, New York, NY, USA, 2009. ACM.

[32] J. Valamehr, M. Chase, S. Kamara, A. Putnam, D. Shumow, V. Vaikuntanathan, and T. Sherwood. Inspection resistant memory: Architectural support for security from physical examination. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 130–141, Washington, DC, USA, 2012. IEEE Computer Society.

[33] A. Vasudevan, J. McCune, J. Newsome, A. Perrig, and L. van Doorn. CARMA: A hardware tamper-resistant isolated execution environment on commodity x86 platforms. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2012.

[34] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4, 1996.

[35] A. Waksman and S. Sethumadhavan. Tamper evident microprocessors. In *Security and Privacy*, 2010.

[36] A. Waksman and S. Sethumadhavan. Silencing hardware backdoors. In *Security and Privacy*, 2011.

[37] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.

[38] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 83–93, Washington, DC, USA, 2008. IEEE Computer Society.

[39] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[40] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 99–110, New York, NY, USA, 2012. ACM.

[41] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

# A. Proof of Noninterference

## A.1 Assumptions

We make the following assumptions on Sapper programs (each of these are easily enforced by a syntactic pass over the program):

- For each **fall**$_\ell$, the subscript $\ell$ must be the label of the state containing that **fall** command. A leaf state cannot contain a **fall**.

- For each **goto**$_{\ell_1}$ $\ell_2$, subscript $\ell_1$ must indicate the state containing that **goto** command, and $\ell_2$ must a state in the same group and at the same depth.

- Either both branches of an **if** command must execute a **goto** or **fall** or neither of them do. All paths through a state end in either a **goto** or a **fall**.

- Each **if** statement is given a unqiue label.

- The root state is fixed.

## A.2 L-equivalence

Our noninterference theorem uses the notion of *L-equivalence* between configurations, which we define in this section. For a given security level $t$, let $L = \{t' \mid t' \sqsubseteq t\}$ and $H = \{t' \mid t' \notin L\}$. We use this definition in the rest of the section. We then have the following L-equivalence definitions:

- **Store:** Two stores are L-equivalent if all L-tagged registers have the same value in both stores, i.e., $\sigma_1 \sim_L \sigma_2 \iff \forall x. \ x$ is a register $\wedge x \in dom(\sigma_{i=1,2}) \wedge \theta(x) \in L \Rightarrow \sigma_1(x) = \sigma_2(x)$

- **Stack:** Two stacks are L-equivalent, i.e.., $sc_1 :: \Sigma_1 \sim_L sc_2 :: \Sigma_2$, if both the following conditions hold: (1) $sc_1 \in L \vee sc_2 \in L \Rightarrow sc_1 = sc_2 \wedge \Sigma_1 = \Sigma_2$, (2) Let $\zeta_L(\mathcal{S})$ return a new stack obtained by popping out elements from $\mathcal{S}$ until its top element $\in L$, then $sc_1 \in H \wedge sc_2 \in H \Rightarrow \zeta_L(\Sigma_1) = \zeta_L(\Sigma_2)$.

- **FallMap, TagMap:** Two pairs of *FallMap* and *TagMap* are L-equivalent, i.e.., $\langle \rho_1, \theta_1 \rangle \sim_L \langle \rho_2, \theta_2 \rangle$, if both the following conditions hold: (1) If one *FallMap* maps to a L-tagged state, then the other *FallMap* maps to the same state, i.e.., $\forall l. \ \theta_1(\rho_1(\ell)) \in L \vee \theta_2(\rho_2(\ell)) \in L \Rightarrow \rho_1(\ell) = \rho_2(\ell)$ (2) If in one *TagMap*, a *TaggedEntity* is tagged L, then it is tagged L in the other *TagMap* too, i.e.., $\forall u. \ \theta_1(u) \in L \iff \theta_2(u) \in L$.

- **Configuration:** Let $\mathbb{C}_1 = \langle p_1, \rho_1, \sigma_1, \theta_1, sc_1 :: \Sigma_1, \delta_1 \rangle$ and $\mathbb{C}_2 = \langle p_2, \rho_2, \sigma_2, \theta_2, sc_2 :: \Sigma_2, \delta_2 \rangle$. Two configurations $\mathbb{C}_1$ and $\mathbb{C}_2$ are L-equivalent, i.e., $\mathbb{C}_1 \sim_L \mathbb{C}_2$, if the following conditions hold: (1) $p_1$ and $p_2$ are either syntactically equivalent or $sc_1 \in H \wedge sc_2 \in H$, (2) $\langle \rho_1, \theta_1 \rangle \sim_L \langle \rho_2, \theta_2 \rangle$, (3) $\sigma_1 \sim_L \sigma_2$, (4) $sc_1 :: \Sigma_1 \sim_L sc_2 :: \Sigma_2$, (5) $\delta_1 = \delta_2$

In effect, two L-equivalent configurations are indistinguishable to an observer at a security level in L.

## A.3 Lemmas

We provide two Lemmas in this subsection, which are used in the proof of noninterference in the next subsection. The Simple Security Lemma states that L-tagged expressions are made up of L-tagged subexpressions only. The Confinement Lemma states that program steps taken with security context in $H$ preserves L-equivalence.

**Lemma 1** (Simple Security). $\phi(e) \in L \Rightarrow (\forall e'.\ e'$ is a subexpression of $e \Rightarrow \phi(e') \in L)$

*Proof.* By induction on definition of $\phi$ in Figure 6(c). $\qquad\square$

**Lemma 2** (Confinement). *Let* $\mathbb{C}_1 = \langle p_1, \rho_1, \sigma_1, \theta_1, sc_1 :: \Sigma_1, \delta_1 \rangle$ *and* $\mathbb{C}_2 = \langle p_2, \rho_2, \sigma_2, \theta_2, sc_2 :: \Sigma_2, \delta_2 \rangle$. *Then if* $\mathbb{C}_1 \rightsquigarrow \mathbb{C}_2 \wedge sc_1 \in H \wedge sc_2 \in H \Rightarrow \sigma_1 \sim_L \sigma_2 \wedge \langle \rho_1, \theta_1 \rangle \sim_L \langle \rho_2, \theta_2 \rangle$.

*Proof.* By induction on $p_1$ $\qquad\square$

### A.4 Noninterference

The noninterference theorem states that if an L-observer cannot distinguish between two configurations at the beginning of a cycle, then she cannot distinguish between them at the beginning of the next cycle either. We assume that an L-observer can observe changes to L-tagged registers only at the beginning of each cycle, *not* during the cycle itself. This assumption is valid because we are designing synchronous hardware—changes to register values come into effect only at the end of a cycle. Two computations can take a different number of semantic steps within a single cycle, but the hardware is timed such that the two computations still complete at exactly the same time. We measure time in number of cycles and thus the theorem of noninterference we provide is timing-sensitive. The formal statement of noninterference is as follows:

**Theorem 1** (Noninterference). *Let*
$\mathbb{C}_1 = \langle \mathcal{F}_{root}, \rho_1, \sigma_1, \theta_1, \Sigma_1, \delta \rangle, \mathbb{C}'_1 = \langle \mathcal{F}_{root}, \rho'_1, \sigma'_1, \theta'_1, \Sigma'_1, \delta{+}1 \rangle,$
$\mathbb{C}_2 = \langle \mathcal{F}_{root}, \rho_2, \sigma_2, \theta_2, \Sigma_2, \delta \rangle, \mathbb{C}'_2 = \langle \mathcal{F}_{root}, \rho'_2, \sigma'_2, \theta'_2, \Sigma'_2, \delta{+}1 \rangle$
*and $\rightsquigarrow^*$ be the reflexive transitive closure of $\rightsquigarrow$.*
*Then* $\mathbb{C}_1 \rightsquigarrow^* \mathbb{C}'_1 \wedge \mathbb{C}_2 \rightsquigarrow^* \mathbb{C}'_2 \wedge \mathbb{C}_1 \sim_L \mathbb{C}_2 \Rightarrow \mathbb{C}'_1 \sim_L \mathbb{C}'_2$

*Proof.* **Basis**: $\mathbb{C}_1 \sim_L \mathbb{C}_2$, because the $\rightsquigarrow^*$ is reflexive, the case holds trivially.
**Inductive Step**: Let

- $\mathbb{C}_a = \langle p_a, \rho_a, \sigma_a, \theta_a, sc_a :: \Sigma_a, \delta \rangle$
- $\mathbb{C}_b = \langle p_b, \rho_b, \sigma_b, \theta_b, sc_b :: \Sigma_b, \delta \rangle$
- $\mathbb{C}'_a = \langle p'_a, \rho'_a, \sigma'_a, \theta'_a, sc'_a :: \Sigma'_a, \delta \rangle$
- $\mathbb{C}'_b = \langle p'_b, \rho'_b, \sigma'_b, \theta'_b, sc'_b :: \Sigma'_b, \delta \rangle$

Also, let $\mathbb{C}_a \sim_L \mathbb{C}_b$, $\mathbb{C}_a \rightsquigarrow \mathbb{C}'_a$ and $\mathbb{C}_b \rightsquigarrow \mathbb{C}'_b$. Then we need to prove $\mathbb{C}'_a \sim_L \mathbb{C}'_b$. We divide this proof sketch into 2 cases:
(1) When $p_a = p_b$. We prove this by induction on $p_a$. Due to the lack of space, we only sketch the proof for a subset of what $p_a$ can stand for:

- When $p_a = r := e$ and $r$ is a enforced register (refer rule ASSIGN-ENF-REG): The register $r$ has the same tag in both $\theta_a$ and $\theta_b$. If the tag $\in L$, then the register $r$ is mapped to $n$ in both the stores (when the dynamic check, that the tag of the register is $\sqsupseteq$ tag of $e$ join the current security context, succeeds in both cases), or both

the stores remain the same (if dynamic check in one fails, it fails in the other), using Lemma 1. If the tag $\in H$, then taking this step does not modify L-observable store. Hence this case holds.

- When $p_a = \mathbf{if}_\ell\ e\ \mathbf{then}\ c\ \mathbf{else}\ c\mathbf{;}\ \mathbf{endif}$ (refer rule IF): If $sc_a, sc_b \in L$, then executing the **if** statement will push the same tag onto the security context stack (hence $sc'_a = sc'_b$). Also, both $\theta_a, \theta_b$ are modified in the same way, because $\mathcal{F}_{cd}$ is a constant function (note that we update all registers that are control dependent on the **if** statement to handle implicit flows due to branches not taken). The tag of $e$ is $\ell$ in both cases (in which case $p'_a = p'_b$), or $H$ in both cases (in which case $sc'_a, sc'_b \in H$). If $sc_a, sc_b \in H$, then from Lemma 2, L-equivalence is preserved by taking this step. Hence this case holds.

- When $p_a = \mathbf{skip;}\ \mathbf{endif}$ (refer rule ENDIF): This statement pops the top of the security context stack. Based on the definition of the L-equivalence of stacks, and our assumption that if one branch of an **if** does not have a **goto** or **fall**, the other branch does not either, this case holds.

- When $p_a = \mathbf{goto}_{\ell_1}\ \ell_2$ and $\ell_2$ is a enforced state (refer rule GOTO-ENFORCED): In the step taken in both configurations, the security context stack is set to the tag of the root state. If $sc_a, sc_b \in L$, either the dynamic check (that tag of state $\ell_2 \sqsupseteq$ the current security context) succeeds in both cases or fails in both cases, modifying the fall map in the same way. If $sc_a, sc_b \in H$, then they do not modify the fall map for any states with tag $\in L$. Thus this case holds.

(2) When $sc_a, sc_b \in H$: for the cases where neither $p_a, p_b$ is **endif** or **goto**, proof follows from Confinement Lemma. Hence this case holds. Consider the case where one command is a **goto** and the other is not: since all program paths must end in **goto**, the other command will eventually be **goto**. Since the cycle length is enforced, both **goto**s are executed only at the end of each cycle. Hence even if they take different number of semantic steps to reach there, they take the same time. For the case where one command is a **endif** and other is not, because of the assumption that either both branches contain **fall** or **goto**, or neither of them do, the other command will eventually become **endif** (during which L-equivalence is preserved throughout, using Confinement Lemma)
This completes the proof sketch. $\qquad\square$