# Caisson: A Hardware Description Language for Secure Information Flow

Xun Li Mohit Tiwari Jason K. Oberg* Vineeth Kashyap
Frederic T. Chong Timothy Sherwood Ben Hardekopf

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA
{xun,tiwari,vineeth,chong,sherwood,benh}@cs.ucsb.edu

*Department of Computer Science and Engineering
University of California, San Diego
San Diego, CA
jkoberg@cs.ucsd.edu

## Abstract

Information flow is an important security property that must be incorporated from the ground up, including at hardware design time, to provide a formal basis for a system's root of trust. We incorporate insights and techniques from designing information-flow secure programming languages to provide a new perspective on designing secure hardware. We describe a new hardware description language, Caisson, that combines domain-specific abstractions common to hardware design with insights from type-based techniques used in secure programming languages. The proper combination of these elements allows for an expressive, provably-secure HDL that operates at a familiar level of abstraction to the target audience of the language, hardware architects.

We have implemented a compiler for Caisson that translates designs into Verilog and then synthesizes the designs using existing tools. As an example of Caisson's usefulness we have addressed an open problem in secure hardware by creating the first-ever provably information-flow secure processor with micro-architectural features including pipelining and cache. We synthesize the secure processor and empirically compare it in terms of chip area, power consumption, and clock frequency with both a standard (insecure) commercial processor and also a processor augmented at the gate level to dynamically track information flow. Our processor is competitive with the insecure processor and significantly better than dynamic tracking.

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

***General Terms*** Security, Verification, Languages

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

## 1. Introduction

High-assurance embedded systems such as those used in banks, aircraft and cryptographic devices all demand strong guarantees on information flow. Policies may target confidentiality, so that secret information never leaks to unclassified outputs, or they may target integrity, so that untrusted data can never affect critical system data. The high cost of a policy violation ensures that these systems are evaluated extensively before being deployed; for instance, certifying systems using Common Criteria [2] or FIPS [3] requires a painstaking process at a cost of millions of dollars over multiple years [4].

Information flow policies are expressed using a lattice of security levels [14] such that higher elements in the lattice correspond to information with more restricted flow (i.e., secret information for confidentiality or untrusted information for integrity). A simple example of a security lattice would be the typical military classification levels: `Unclassified` $\sqsubseteq$ `Secret` $\sqsubseteq$ `Top Secret`. An important information flow policy based on such lattices is *non-interference* [18], which requires that no information at a given level in the lattice can flow anywhere except to higher elements in the lattice (e.g., `Secret` information can flow to `Top Secret`, but not vice-versa). High-assurance systems require a static guarantee of non-interference and depend on a *hardware-based root of trust* to enforce this policy. We present a new hardware description language named Caisson that meets this need by extending HDLs like Verilog with language abstractions that enable precise static verification of secure synchronous hardware designs.

### 1.1 Secure Hardware Design

While ciphers provide a sound theoretical primitive for building secure systems, their actual *implementations* have been shown to be a rich source of vulnerabilities. Numerous attacks exploit hardware structures such as shared data caches [35], instruction caches [6], and branch predictors [5, 7] to leak information about private keys. Other studies have found vulnerabilities lurking in obscure and even undocumented corners of hardware designs [41], e.g., when the floating point registers are persistent across context switches. Complete information flow security must begin with a principled approach to designing hardware that accounts for the intricate interaction among different hardware components, analyzes the hardware design in its entirety, and does so efficiently enough to be useful in practice. Note that non-digital side channels such as power analysis [25] are not within the scope of this paper.

Existing work has explored using hardware assistance to dynamically track information flow and prohibit leaks [12, 13, 38, 43]. However, most such systems only track information flow at the ISA level or above, ignoring micro-architectural features such as caches and branch predictors that can leak information. These systems cannot protect against the attacks outlined above.

One existing system, GLIFT [46], dynamically tracks information flow at the gate-level and does take micro-architectural fea-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*PLDI'11,* June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06. . . $10.00

tures into account; however, this technique requires the information tracking logic to be physically instantiated in the synthesized circuit, greatly increasing chip area and power consumption. Also, GLIFT only detects policy violations at runtime; it cannot guarantee statically that no violations will occur. GLIFT is currently the only alternative for enforcing information flow for an entire processor below the ISA level of abstraction, and we use GLIFT as our main comparison point in our evaluation.

### 1.2 Our Approach

In contrast to existing approaches, we take language-level techniques for secure information flow and apply them to domain-specific abstractions for hardware design (specifically, finite state machines) to create a new Hardware Description Language (HDL) named Caisson. Our goal is to enable the creation of synchronous hardware designs that are statically-verifiable as secure. Additional benefits of Caisson are that it allows hardware designers to operate at familiar level of abstraction, enables architects to quickly and easily iterate through potential designs without having to wait for synthesis and simulation to test security properties, and the resulting designs do not suffer from the crippling overhead that comes with dynamic tracking in terms of additional chip area and power consumption

While there are existing HDLs based on finite state machines, none target security as a first-class concern. Caisson employs two novel features on top of finite state machines, *nested states* and *parameterized states* (described in §2) to enable precise and expressive enforcement of security policies. To demonstrate the utility of these features and of Caisson in general, we design an information-flow secure processor in Caisson. Designing secure hardware controllers is an active research area, and specifying a statically-verifiable secure general-purpose processor is an open problem. Such processors have an important application in high-assurance embedded systems such as those found in aircraft and automobiles [27]. Current tools and methodologies for secure hardware design are laborious and expensive (taking millions of dollars and multiple years to complete even simple designs); a general-purpose processor with microarchitectural features such as pipelining and cache is notorious in the hardware community for being too complicated to design in a verifiably secure manner.

Caisson is based on key insights into secure hardware design, and it provides language-level support for these design patterns. We believe, and have found in our own experience, that Caisson promotes *thinking* about secure hardware design in new, useful ways that don't naturally arise in existing languages.

### 1.3 Contributions

This paper makes the following specific contributions:

- We describe Caisson, a hardware description language targeting statically-verifiable information-flow secure hardware design.
- We formally prove that Caisson enforces timing-sensitive non-interference.
- We design and implement a verifiably information-flow secure processor with complex micro-architectural features including pipelining and cache.
- We synthesize our design and empirically compare it with an insecure commercial CPU design as well as a GLIFT CPU design that dynamically tracks information flow. We find that Caisson introduces much less overhead than GLIFT over the baseline processor in terms of chip area (1.35× vs. 3.34×), clock frequency (1.46× vs. 2.63×) and power (1.09× vs. 2.82×).

The rest of the paper is organized as follows. We begin in §2 by informally describing the Caisson language and motivating its features. In §3 we formalize the language description and provide a proof sketch of its security properties. In §4 we describe the design of a secure processor in Caisson and empirically evaluate the characteristics of the synthesized design against a comparable GLIFT hardware design. §5 discusses related work, and §6 concludes.

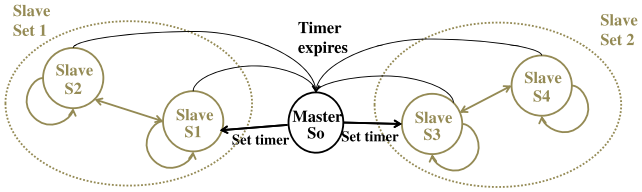



**Figure 1.** State Machine Diagram of Execution Lease Controller

## 2. Overview of Caisson

In this section we provide an overview of the Caisson language and motivate its design via a simple hardware example. For concreteness, we specifically address the issue of integrity using a 2-level lattice `Trusted` ⊑ `Untrusted`[1] (though Caisson can be used for arbitrary lattices). We demonstrate the Caisson language using an Execution Lease secure hardware controller [45]. We first review the concept of execution leases, then demonstrate how Caisson can create a statically-verifiable instantiation of an execution lease controller.

***Execution Leases*** An execution lease is an architectural mechanism used to allow trusted code to grant untrusted code limited access to machine resources. One use-case is to allow a trusted separation kernel [24] to securely multiplex multiple untrusted processes on a CPU. One can think of a lease as a space-time sandbox that allows an untrusted process to take over the CPU and execute code using only a limited range of memory and a limited amount of time; the lease mechanism forces untrusted code to relinquish control back to the trusted code when its time allotment expires. Figure 1 gives a state-machine diagram of the execution lease controller. The trusted master state sets a timer and transfers control to either the untrusted set of slave states (S1 and S2) or the trusted set of slave states (S3 and S4). Each set of slave states can transition among themselves arbitrarily during the lease, but once the timer expires, control is relinquished back to the trusted master state.

***Caisson Language*** Finite-state representations of hardware control systems, such as in Figure 1, are popular in hardware design. Existing tools such as Altera Quartus, Xilinx ISE, Statecharts [20], and Esterel [44] are widely used to model systems explicitly as state machines. In designing Caisson we wish to capitalize on this trend and allow hardware designers to operate at a familiar level of abstraction, allowing hardware designs to be easily and transparently modeled using Caisson. For this reason, we base the Caisson language on *finite-state machines*. To illustrate this concept, Figure 2**(a)** shows a Caisson implementation of the lease controller. This implementation is secure (i.e., does not leak `Untrusted` information), but it is *not* typable in the Caisson type system we introduce in Section 3. We will use this version of the implementation to motivate and introduce two features of Caisson: *nested states* and *parameterized states*. First, though, we give a quick illustration of the Caisson language using Figure 2**(a)**.

The name of the entire program is `lease`, and it uses four hardware registers: `timer`, `data1`, `data2`, and `mode`. Each register has

[1] This ordering can be confusing, but is correct: `Untrusted` is *high* in the lattice because the flow of untrusted information should be more restricted than the flow of trusted information.

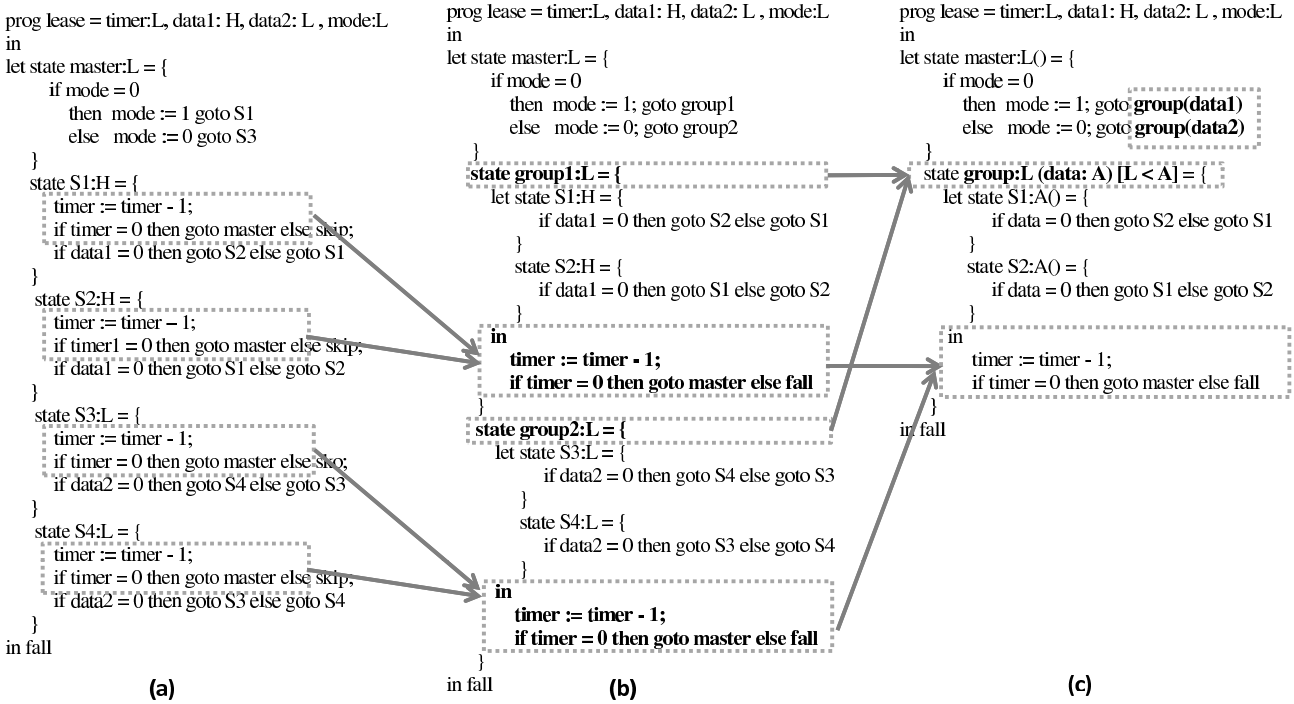



**Figure 2.** Implementation of the Lease Controller in Caisson language. (a) Implementation only with the ability to explicitly define each individual state (b) Implementation with nested states (c) Implementation with parameterized states.

type L (*low*, i.e. `Trusted`) except for `data1`, which is H (*high*, i.e. `Untrusted`). There are five states corresponding to the five states in Figure 1; `master`, `S3`, and `S4` are `Trusted`, while `S1` and `S2` are `Untrusted`. The `master` state uses `mode` to alternately transfer control (using the **goto** command) to either `S1` or `S3`. Each of `S1`—`S4` are similar: they decrement `timer`, check whether `timer` is 0, and if so transition back to the `master` state. Otherwise, depending on the value of `data1` (`data2`), they transition to themselves or their fellow slave state. Each state corresponds to a combinational hardware logic circuit and takes exactly one cycle to execute.

The reason that Figure 2**(a)** is not typable is that `timer` is decremented in states `S1` and `S2`. These states are `Untrusted`, yet they manipulate `Trusted` information (i.e., `timer`). This manipulation can create an implicit information leak from the high security level (`Untrusted`) to the low security level (`Trusted`)—if the `Untrusted` states modify the `Trusted` register `timer` in different ways, then the value of `timer` would depend on which `Untrusted` states are executed. Intuitively, however, we can see that the design actually *is* secure: since *every* state is guaranteed to decrement `timer` in exactly the same way, in reality there is no information leakage—nothing about the `Untrusted` states or transitions between those states can be inferred from the value of `timer`.

***Nested States*** This observation motivates the first Caisson language feature, *nested states*. Nested states allow the designer to factor out shared code among states to identify exactly such situations. Figure 2**(b)** gives a valid (i.e., typable) Caisson implementation of the same design but using nested states. This design nests states `S1` and `S2` into the same *group state* `group1`, and similarly nests `S3` and `S4` into `group2`. In each group, the code common to the nested (or *child*) states has been factored out and associated with the group state containing those child states. The semantics of nested states effectively treats the command of a group state as if it were inlined into the command of each child state, so the code in Figure 2**(b)** has the same behavior as the code in Figure 2**(a)**. For example, each time the code transitions from `S1` to `S2` using **goto**, the command for `group1` that decrements and checks `timer` executes before the command for `S2`. The **fall** command signals that the group state's command is complete and to begin executing the appropriate child state's command. When transitioning to a group state (as in the `master` state's command "**goto** group1" in Figure 2**(b)**), the default fall-through state is the first listed state (e.g., `S1` for group state `group1`).

The benefit of nested states is that Caisson is allowed to type a group state separately from the its child states. In Figure 2**(b)** state `group1` is typed L (*low*, or `Trusted`) while its child states `S1` and `S2` are typed H (*high*, or `Untrusted`). As explained above, this is safe because the semantics of Caisson guarantees that `group1`'s command executes identically in each child state, and so no information is leaked even though `group1`'s command modifies `Trusted` information.

Note that nested states are distinct from the related concept of *hierarchical states* in languages like Statecharts [20]. In Statecharts, child states *specialize* parent states. If an event is not handled by a child state, then the parent states are checked to determine if they can handle the event (somewhat like OOP virtual methods). Caisson's nested states have different semantics, as explained informally above and formally in §3. Nested states can also be seen as a concept dual to the notion of *linear continuations* [16, 55]. Whereas linear continuations identify code that is guaranteed to be executed *afterwards* by factoring the code out into a continuation, nested states identify code that is guaranteed to be executed *beforehand* by factoring it out into a group state.

***Parameterized States*** While Figure 2**(b)** is a valid Caisson program, it is not as efficient as it could be. Note that `group1` and `group2` have identical logic; the only difference is that `group1` operates on `Untrusted` data (`data1`) while `group2` operates on `Trusted` data (`data2`). Therefore the two groups must be kept separate. When synthesizing this design, each group would be com-

piled down to its own separate logic circuit. It would be more efficient in terms of chip area to synthesize the *same* logic circuit for the two groups and reuse that circuit by securely multiplexing the different data (`data1` vs `data2`) onto that circuit. This observation motivates the second Caisson language feature, *parameterized states*. Figure 2**(c)** shows the same program as Figure 2**(b)** except using parameterized states.

This new implementation has a single `group` state that now has a *parameter*: a variable that represents some register on which the state will operate. Since the exact register that the state will operate on can vary each time the state executes, Caisson uses a type variable A to represent the parameter's type and specifies a set of type constraints that the type variable must obey in order to guarantee security (in this example, the only requirement is that A must be no less than L). The Caisson implementation assumes the given type constraints are valid when it type-checks `group`.

When transitioning to a parameterized state, the **goto** command must specify a particular register to pass as the target state's parameter. In Figure 2**(b)**, the `master` state transitions to `group` with two different arguments depending on `mode`: either `data1`, replicating the behavior of the original `group1`, or `data2`, replicating the behavior of the original `group2`. The Caisson implementation statically verifies that any arguments given to a parameterized state must necessarily satisfy the given type constraints, thereby statically guaranteeing the security of the design.

The benefit of parameterized states is that Caisson can synthesize a single logic circuit that can safely be used at multiple security levels. In other words, the data being operated on (the Caisson registers) must have distinct types, but the logic operating on the data (the Caisson states) can be parameterized over the types of the data, making the synthesized circuit much more efficient.

## 3. Formal Description of Caisson

In this section we formally describe a core subset of Caisson and its type system and prove that Caisson enforces timing-sensitive noninterference. The actual implementation of Caisson incorporates a large subset of Verilog, allowing existing Verilog designs to be easily refactored into Caisson programs. This subset of Verilog does not add anything of interest to the formal presentation and so we omit the full details from this section.

Figure 3 describes Caisson's abstract syntax (which uses types as described in Figure 5—we defer discussion of types to §3.2). Registers in the language correspond to registers in hardware and hold integer values. Variables range over registers rather than values (i.e., a variable maps to a register) and act as state parameters to abstract a state's behavior from specific registers.

A Caisson program consists of a list of registers followed by a list of nested state definitions. The nested state definitions form a hierarchy of states with a single *root* state. We define a *leaf state* as a state at the bottom of the state hierarchy; these states each specify a single command. A *group state* is any non-leaf state and specifies both (1) a nested list of states and (2) a command. The **goto** command triggers a state transition. Caisson describes synchronous hardware designs, hence the language implementation enforces that the length of time between any two state transitions (i.e., **goto**s) is exactly one cycle. Within each cycle, Caisson enforces the following invariant: before executing the command of any state $S$, Caisson executes the commands of all of $S$'s ancestor states (the intended semantics of nested states).

The **fall** command forces execution to fall-through from the current state to one of its child states: it ends evaluation of the current state's command and begins the evaluation of the child state's command. Falling from a state to its child does not count as a state transition (i.e., a **fall** command does not end a cycle, only a **goto** command can do that). By default the target child state of a **fall** command is the first child state in the list of nested state definitions; this state is called the *default child state* of that group state. The target child state for a **fall** may change during program execution as described in the next subsection.

$$r \in \mathit{Register} \qquad v \in \mathit{Variable} \qquad x \in \mathit{Register} \cup \mathit{Variable}$$
$$n \in \mathbb{Z} \qquad \oplus \in \mathit{Operator} \qquad l \in \mathit{Program\ Label}$$

$$\begin{aligned}
\mathit{prog} \in \mathit{Prog} &::= \textbf{prog}\ l = \vec{r_\ell}\ \textbf{in}\ d \\
d \in \mathit{Def} &::= \textbf{let}\ \vec{s}\ \textbf{in}\ c \mid c \\
s \in \mathit{State} &::= \textbf{state}\ l_\tau\ (\vec{v_\alpha})\ \kappa = d \\
e \in \mathit{Exp} &::= n \mid x \mid e \oplus e \\
c \in \mathit{Cmd} &::= \textbf{skip} \mid x := e \mid c\ \textbf{;}\ c \mid \textbf{fall}_l \\
&\quad \mid \textbf{goto}\ l(\vec{x}) \mid \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c \\
p \in \mathit{Phrase} &::= \mathit{prog} \mid d \mid s \mid e \mid c
\end{aligned}$$

**Figure 3.** Abstract syntax. Type annotations $\ell$, $\alpha$, $\tau$, and $\kappa$ are described in Figure 5

To simplify the formal presentation of the language we make the following assumptions without explicitly enforcing them (though it is simple to do so in practice):

- All the variables and type variables have distinct names.
- A default child state can not take any parameters.
- Every **goto** targets a defined label and can only target a state in the same group and at the same nested depth[2].
- For each $\textbf{fall}_l$, the subscript label $l$ must be the label of the state containing that **fall** command (the label is a syntactic convenience for the semantics and type system; it can be left out of the concrete syntax). A leaf state can not contain a **fall**.
- 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**.

The structure of a program defines a tree of state definitions (the state hierarchy) with **prog** being at the root. From this structure we derive different versions of a function $\mathcal{F} : l \mapsto (\vec{v} \cup p \cup l)$ that for each program label $l$ gives the following mappings:

- $\mathcal{F}_{pnt}(l)$ maps to the label of state $l$'s parent state.
- $\mathcal{F}_{def}(l)$ maps to the label of state $l$'s default child state.
- $\mathcal{F}_{cmd}(l)$ maps to state $l$'s command.
- $\mathcal{F}_{prm}(l)$ maps to state $l$'s parameters.

In addition, $\mathcal{F}_{root}$ maps to the root command of the **prog** program phrase.

### 3.1 Semantics

Figure 4 shows the small-step operational semantics of Caisson. This figure gives the abstract machine configuration, defines the evaluation context, and gives the small-step transition rules.

[2] This requirement is due to the semantics of nested states: the target of a **goto** influences which ancestor states' commands execute, which could leak information without this restriction. This restriction does not greatly impact expressiveness: we can still get from any state to any other state by constructing a sequence of **goto**s, though we won't necessarily get there in a single cycle.

$$\gamma \in \mathit{Env} : (v \mapsto r) \cup (l \mapsto l) \qquad \sigma \in \mathit{Store} : r \mapsto n$$
$$\delta \in \mathit{Time} : \mathbb{N} \qquad \mathbb{C} \in \mathit{Config} : \langle p, \gamma, \sigma, \delta \rangle$$

$$E ::= \Box \mid E \oplus e \mid n \oplus E \mid x := E \mid E \texttt{ ; } c \mid \textbf{prog } l = \vec{r_\ell} \textbf{ in } E \mid \textbf{let } \vec{s} \textbf{ in } E$$

$$\langle E[r], \gamma, \sigma, \delta \rangle \rightsquigarrow \langle E[\sigma(r)], \gamma, \sigma, \delta \rangle \quad \text{(REG)}$$

$$\langle E[v], \gamma, \sigma, \delta \rangle \rightsquigarrow \langle E[\sigma(\gamma(v))], \gamma, \sigma, \delta \rangle \quad \text{(VAR)}$$

$$\langle E[n_1 \oplus n_2], \gamma, \sigma, \delta \rangle \rightsquigarrow \langle E[n_1 \; [\![\oplus]\!] \; n_2], \gamma, \sigma, \delta \rangle \quad \text{(OP)}$$

$$\frac{\langle e, \gamma, \sigma, \delta \rangle \rightsquigarrow^* \langle n, \gamma, \sigma, \delta \rangle}{\langle E[r := e], \gamma, \sigma, \delta \rangle \rightsquigarrow \langle E[\textbf{skip}], \gamma, \sigma[r \mapsto n], \delta \rangle} \quad \text{(ASSIGN-R)}$$

$$\frac{\langle e, \gamma, \sigma, \delta \rangle \rightsquigarrow^* \langle n, \gamma, \sigma, \delta \rangle}{\langle E[v := e], \gamma, \sigma, \delta \rangle \rightsquigarrow \langle E[\textbf{skip}], \gamma, \sigma[\gamma(v) \mapsto n], \delta \rangle} \quad \text{(ASSIGN-V)}$$

$$\frac{\langle e, \gamma, \sigma, \delta \rangle \rightsquigarrow^* \langle n, \gamma, \sigma, \delta \rangle \qquad c' = \begin{cases} c_1 & : n = 0 \\ c_2 & : n \neq 0 \end{cases}}{\langle E[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], \gamma, \sigma, \delta \rangle \rightsquigarrow \langle E[c'], \gamma, \sigma, \delta \rangle} \quad \text{(IF)}$$

$$\langle E[\textbf{skip ; } c], \gamma, \sigma, \delta \rangle \rightsquigarrow \langle E[c], \gamma, \sigma, \delta \rangle \quad \text{(SEQ)}$$

$$\langle E[\textbf{fall}_l], \gamma, \sigma, \delta \rangle \rightsquigarrow \langle \mathcal{F}_{\mathit{cmd}}(\gamma(l)), \gamma, \sigma, \delta \rangle \quad \text{(FALL)}$$

$$\frac{\begin{array}{c}\gamma_1 = \mathit{Reset}(\gamma, l) \\ \gamma_2 = \gamma_1[\mathcal{F}_{\mathit{pnt}}(l) \mapsto l] \\ \gamma_3 = \gamma_2[\mathcal{F}_{\mathit{prm}}(l) \mapsto \gamma(\vec{x})]\end{array}}{\langle E[\textbf{goto } l(\vec{x})], \gamma, \sigma, \delta \rangle \rightsquigarrow \langle \mathcal{F}_{\mathit{root}}, \gamma_3, \sigma, \delta{+}1 \rangle} \quad \text{(GOTO)}$$

**Figure 4.** Small-step semantic rules for Caisson ($\rightsquigarrow^*$is the reflexive transitive closure of $\rightsquigarrow$).

The abstract machine configuration consists of the current program phrase $p$, an environment $\gamma$, a store $\sigma$, and a time value $\delta$. The environment $\gamma$ maps variables (state parameters) to registers and maps state labels to other state labels; the store $\sigma$ maps registers to values; and $\delta$ is the current time measured in cycles. The mapping from labels to labels in $\gamma$ records for each state $l$ its target child state (i.e., the command to begin evaluating if $l$ executes a **fall** command—we initialize the mapping to the default child states). The rules are fairly standard except for **fall** and **goto**, which we now describe in detail.

The FALL rule applies when executing a $\textbf{fall}_l$ command. The rule looks up in $\gamma$ the target child state for the current state $l$ (recall that we require $l$ for a command $\textbf{fall}_l$ to be the label of the current state) and begins executing that target child state's command.

The GOTO rule applies when executing a **goto** command. The rule does three things: (1) creates a new environment $\gamma$ as described below; (2) increments the time $\delta$, since a state transition marks the beginning of a new cycle; and (3) begins evaluating the root command at the top of the state hierarchy.

The most complex part of the GOTO rule is the set of premises that create the new environment $\gamma_3$ with which we evaluate the root command. The key is to understand the intended semantics of nested states: upon each state transition to a state $l$, we must maintain the invariant that all of $l$'s ancestor states' commands execute before executing $l$'s command. Hence, the **goto** sets the next command to $\mathcal{F}_{\mathit{root}}$ rather than the command of the target state. When evaluating a state's command it will eventually execute either another **goto** (restarting this whose process) or a **fall**. The environment must map each state's target child state such that a sequence of **fall**s will follow the correct path down the state hierarchy to eventually reach $l$, the original target of the **goto** that began the current cycle.

The first premise uses a helper function *Reset*, which we define here: *Reset* takes an environment $\gamma$ and a state label $l$ and returns a new environment $\gamma'$ identical to $\gamma$ except that label $l$ and the labels of all states that are descendents of $l$ in the state hierarchy are mapped to their default child states (the same as their initial values). This ensures that each time the program transitions to a state group, any prior evaluations of that state group do not affect the current evaluation.

The second premise looks up $l$'s parent state ($\mathcal{F}_{\mathit{pnt}}(l)$) and sets the parent state's target child state to be $l$. This ensures that when evaluating the parent state's command and executing a **fall**, execution will fall-through to the correct child state (i.e., $l$). The last premise maps the target state $l$'s parameters to the arguments of the **goto** command.

### 3.1.1 Example

Consider the code in Figure 2**(b)** and assume `mode` is initialized to 0 and `timer` is initialized to 3. Note that $\mathcal{F}_{\mathit{root}}$ is the outermost **fall** command (at the bottom of the code). Execution will proceed in the following manner,where C⟨X⟩ represents cycle X in the execution:

C0 The outermost **fall** command is executed, which by default falls through to state `master`. Since the **if** guard is true, `mode` is set to 1 and the command `goto group1` is executed. The GOTO rule changes the environment so that the fall-through state is now `group1` instead of `master` and resets the next command to be $\mathcal{F}_{\mathit{root}}$.

C1 The outermost **fall** command is executed. Because of the changed environment, control falls through to state `group1`. `timer` is decremented to 2; since `timer` is not 0 the else branch is taken and the **fall** command is executed. By default, control falls through to state `S1`. Assume that `data1` = 0; then the command `goto S2` is executed. This changes the environment so that the fall-through state for `group1` is now `S2` instead of `S1` and resets the next command to be $\mathcal{F}_{\mathit{root}}$.

C2 The outermost **fall** command is executed. As previously, control falls through to state `group1` and `timer` is decremented to 1. Since `timer` is not 0 the else branch is taken and the **fall** command is executed. Because of the changed environment control falls through to state `S2`. Assume `data1` is now 1; then the command `goto S2` is executed, which leaves the environment unchanged and resets the next command to be $\mathcal{F}_{\mathit{root}}$.

C3 The outermost **fall** command is executed. As previously, control falls through to state `group1` and `timer` is decremented to 0. Since `timer` is now 0, the true branch is taken and the command `goto master` is executed; this changes the environment so `group1`'s fall-through state goes back to its default (i.e., `S1`) and the root fall-through state is now `master` instead of `group1`, and resets the next command to be $\mathcal{F}_{\mathit{root}}$.

C4 The logic from cycle C0 is repeated, except that `mode` = 1 and so the program will transition to `group2` instead of `group1`.

| | |
|---|---|
| $\ell \in \mathcal{L}$ | security labels |
| $\tau ::= \ell \mid \alpha$ | base types, type variables |
| $\kappa ::= \{\tau <: \tau\} \mid \kappa_1 \cup \kappa_2$ | type constraints |
| $\rho ::= \tau \mid \texttt{cmd}_\tau \mid \texttt{st}_\tau(\vec{\alpha}, \kappa)$ | phrase types |

**Figure 5.** Types

### 3.2 Type System

Figure 5 gives the types used by our language. The base types are elements of the security lattice $\mathcal{L}$. Type variables range over base types and are implicitly universally quantified. We assume each state uses a disjoint set of type variables. The type variables are bounded by the type constraints $\kappa$ which specify required subtype relations among the type variables and base types. The type system statically checks that all **goto** commands satisfy these constraints.

Expressions have type $\tau$, commands have type $\texttt{cmd}_\tau$, and states have type $\texttt{st}_\tau(\vec{\alpha}, \kappa)$. We omit the standard subtyping rules: expression types are covariant ($e : \tau$ means that $e$ contains no variables or registers greater than $\tau$); command types are contravariant ($c : \texttt{cmd}_\tau$ means that $c$ does not change the state of any variable or register less than $\tau$); and state types are invariant ($s : \texttt{st}_\tau(\vec{\alpha}, \kappa)$ means that, assuming the state parameters have types $\vec{\alpha}$ that satisfy the constraints in $\kappa$, the state's command is type $\texttt{cmd}_\tau$).

The program syntax explicitly notes the types of each register and state; we use this information to initialize the type environment $\Gamma : (r \mapsto \ell) \cup (v \mapsto \alpha) \cup (l \mapsto \texttt{st}_\tau(\vec{\alpha}, \kappa)) \cup \kappa$. $\Gamma$ maps all registers to a base type, all state parameters to a type variable, and all state labels to a state type, and also records the type constraints in all $\kappa$. The whole program (i.e. the *root* state) is always mapped to $\texttt{st}_\bot(_, _)$. $\Gamma$ also records the subtype relations between base types (i.e., security labels) as given in the security lattice. Since $\Gamma$ remains constant, we factor it out of the individual type rules.

Most of the rules are standard (see, e.g., Volpano et al [51]) except for rules T-FALL and T-GOTO which we now explain in detail. Rule T-FALL states that if the type of the default child state's command is $\texttt{cmd}_\tau$ then so must be the type of the current state's command. This requirement is due to the differing semantics of **fall** and **goto** commands: a **fall** command immediately begins executing the child state's command, whereas a **goto** begins executing the root command $\mathcal{F}_{root}$. Without this rule, if a conditional with a *high* guard has a **goto** in one branch and a **fall** in the other then a *low* observer might be able to observe the outcome of the conditional based on whether execution proceeds from $\mathcal{F}_{root}$ or not. The rule only needs to check the type of the default child state's command ($\mathcal{F}_{cmd}(\mathcal{F}_{def}(l))$) even though at runtime a state can fall-through to any child state. Since we require transitions must be to neighbor states in the state hierarchy, other child states can only be reached via the default child state. Thus, typing consistency between the parent state and all its child states is enforced indirectly by a combination of the T-FALL and T-GOTO rules.

The most complicated type rule is T-GOTO. The **goto** command has a target state $l$ and a list of arguments $\vec{x}$. The type rule must accomplish two things:

- It must verify that the arguments $\vec{x}$ given to the **goto** satisfy the type constraints $\kappa$ of the target state $l$. This requirement is checked by the first premise on the second line of the type rule. State $l$'s type constraints $\kappa$ are modified to map the type of each state parameter to the type of its corresponding argument (via $\kappa[\vec{\alpha} \mapsto \Gamma(\vec{x})]$); then the resulting type constraints are verified to be valid assertions (i.e., that the constraints can be derived using the sub-typing rules and the information in $\Gamma$).

$$n : \bot \quad \text{(T-CONST)}$$

$$\frac{\Gamma(x) = \tau}{x : \tau} \quad \text{(T-REG/VAR)}$$

$$\frac{e_1 : \tau \qquad e_2 : \tau}{e_1 \oplus e_2 : \tau} \quad \text{(T-OP)}$$

$$\frac{\Gamma(x) = \tau \qquad e : \tau}{x := e : \texttt{cmd}_\tau} \quad \text{(T-ASSIGN)}$$

$$\frac{e : \tau \qquad c_1 : \texttt{cmd}_\tau \qquad c_2 : \texttt{cmd}_\tau}{\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : \texttt{cmd}_\tau} \quad \text{(T-IF)}$$

$$\frac{c_1 : \texttt{cmd}_\tau \qquad c_2 : \texttt{cmd}_\tau}{c_1 \ ; \ c_2 : \texttt{cmd}_\tau} \quad \text{(T-SEQ)}$$

$$\textbf{skip} : \texttt{cmd}_\top \quad \text{(T-SKIP)}$$

$$\frac{\Gamma(l) = \texttt{st}_\tau(\vec{\alpha}, \kappa) \qquad \mathcal{F}_{cmd}(\mathcal{F}_{def}(l)) : \texttt{cmd}_\tau}{\textbf{fall}_l : \texttt{cmd}_\tau} \quad \text{(T-FALL)}$$

$$\frac{\Gamma(l) = \texttt{st}_\tau(\vec{\alpha}, \kappa) \quad \vdash \kappa[\vec{\alpha} \mapsto \Gamma(\vec{x})] \quad \tau' = \tau[\vec{\alpha} \mapsto \Gamma(\vec{x})]}{\textbf{goto } l(\vec{x}) : \texttt{cmd}_{\tau'}} \quad \text{(T-GOTO)}$$

$$\frac{\Gamma(l) = \texttt{st}_\tau(\vec{\alpha}, \kappa) \qquad d : \texttt{cmd}_\tau}{\textbf{state } l_\tau \ (\vec{v_\alpha}) \ \kappa = d : \texttt{st}_\tau(\vec{\alpha}, \kappa)} \quad \text{(T-STATE)}$$

$$\frac{s_i : \texttt{st}_{\tau_i}(\vec{\alpha_i}, \kappa_i) \qquad c : \texttt{cmd}_\tau}{\textbf{let } \vec{s} \textbf{ in } c : \texttt{cmd}_\tau} \quad \text{(T-DEF)}$$

$$\frac{d : \texttt{cmd}_\tau}{\textbf{prog } l = \vec{r_\ell} \textbf{ in } d : \texttt{cmd}_\tau} \quad \text{(T-PROG)}$$

**Figure 6.** Caisson type rules

- It must also confirm the type of the **goto** command in the conclusion of the rule. This is confirmed by the last premise on the second line of the rule. The target state has type $\texttt{st}_\tau(\vec{\alpha}, \kappa)$, meaning that its command has type $\texttt{cmd}_\tau$. However, we cannot simply make the type of the **goto** command be type $\texttt{cmd}_\tau$—$\tau$ may be a type variable, and we assume that the sets of type variables used in any two states are disjoint. Therefore, the rule must translate type $\tau$, valid for the target state, into an appropriate type $\tau'$, valid for the source state. The rule uses the same substitution operator as used earlier to perform this translation.

We sketch a proof that the Caisson type system enforces timing-sensitive noninterference in Appendix A.

#### 3.2.1 Example

Here we use the type rules to show that specific parts of the example program in the previous section are well-typed. We don't show the entire type derivation, but we do show how both the T-GOTO and T-FALL rules are used.

Consider the code in Figure 2**(c)**, and specifically the command `goto group(data1)`. The rule T-GOTO first confirms that the type of the argument (i.e., `data1`, which is type H) satisfies the type constraints of the target `group`, i.e., that L <: H—this is true. The rule then finds a suitable type for the `goto` command based on the type of the target state (`group`, which is type L); hence the type of the `goto` command is $\texttt{cmd}_L$.

Now consider the command `goto group(data2)`. By the same logic this command is also typed $\texttt{cmd}_L$, except that when checking that the argument (i.e., `data2`, which is type L) satisfies the type constraints, the rules confirms that L <: L, which is also true. Since `mode` is type L, the type of the **if** statement is $\texttt{cmd}_L$ which matches the declared type of state `master`.

From the above we can confirm that all transitions to `group` satisfy the type constraints. When typing `group` itself, we assume that the type constraints are met, i.e., that L <: A. Hence, when typing the false branch of the **if** command in `group` the **fall** command, using rule T-FALL, is initially typed as $\texttt{cmd}_A$. However, by contravariance and since L <: A, it can also be typed as $\texttt{cmd}_L$. Since the true branch is typed $\texttt{cmd}_L$ (by the T-GOTO rule), the **if** command is well-typed as $\texttt{cmd}_L$, which matches the declared type of state `group`.

When checking state `S1`, the declared type (i.e., $\texttt{st}_A$(L <: A)) forces the **if** command to be type $\texttt{cmd}_A$; the T-GOTO rules confirms that is true for both branches of the **if** command. The same holds true for state `S2`.

Therefore each state of the program, and the entire program itself, is well-typed.

## 4. Information-Flow Secure Processor

In this section we concretely demonstrate the utility of Caisson by designing and evaluating an information-flow secure processor that safely muliplexes execution of *trusted* and *untrusted* code. Securely executing mixed-trust programs has historically been difficult to get right. For instance, all known cache attacks stem from the basic problem that cache controllers must use both trusted and untrusted information to decide which lines to evict from a cache [35, 54]. Information has been also shown to leak through branch predictor history table [5] or through the instruction cache lines [6]. More generally, any perturbation in the execution of a trusted program based on untrusted data is a vulnerability, and we must prevent all possible sources of such information leaks.

We begin the section with a description of a complete four-stage processor pipeline that securely implements a RISC ISA. We then extend the processor to include a cache hierarchy that is verifiably secure against all known attacks [35, 54]. These designs demonstrate that Caisson's language abstractions allow the CPU to be cleanly specified in a natural fashion and enable a statically-verifiable, efficient design. For this processor design we employ the same two-level lattice `Trusted` $\sqsubseteq$ `Untrusted` as used in §2.

### 4.1 Secure CPU Design

The public interface of a CPU is its Instruction Set Architecture (ISA). Table 1 describes the ISA of our processor; this is a standard ISA (derived from the commercial Altera Nios processor) . The ISA is implemented using a combination of hardware data- and control-flow controlled by the execution pipeline. Figure 7 shows the four-stage pipeline of our processor, with stages Fetch, Decode, Execute, and Commit. Additional microarchitectural features such as caches, prefetchers, and branch predictors can be attached to the pipeline to improve processor performance. Our processor implements a cache to illustrate how these microarchitectural features can be designed for security; the other features can be implemented using similar strategies.

The easiest method to ensure that `Untrusted` data can never affect `Trusted` computation is to physically isolate them, i.e., have two separate instances of the CPU, one for `Trusted` computation and one for `Untrusted` computation. While simple and easy to verify, economic reality means that this is not a practical solution. Even in high assurance systems the hardware components are often shared, e.g., the Boeing 787 trusted aircraft control network shares the physical bus with the untrusted passenger network [15].

| Instruction | Description |
|---|---|
| jump If RegZero | If the value of a register is zero then jump to the target. |
| jump To RegValue | Jump to the address which is stored in specified register. |
| load Immediate | Load an immediate value to a register. |
| load/store Direct | Load/store data from/into an immediate memory address to a register. |
| load PC To Reg | Load the value of PC into a register (for store/restore context purpose). |
| load/store Indirect –global | Load/store data from/into a memory address specified by a based address in register and an offset. The global version operates on data outside the current lease which is a range setup by the leaser while the local version operates on local data inside current memory bound. |
| load/store Indirect –local | |
| directJump | Directly jump to a target address. |
| and, or, not, xor, shl , shr , add, sub, cmpeq, cmpl | Arithmetic instructions. |

**Table 1.** The ISA of our RISC processor

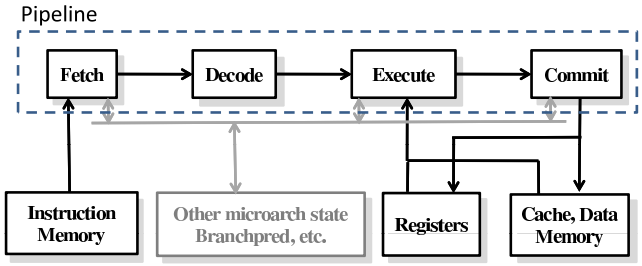



**Figure 7.** A typical CPU pipeline and its interaction with memories, registers and other components. Our CPU design implements all parts in **bold**.

The only alternative solution is to *time multiplex* the `Trusted` and `Untrusted` computation on the same physical hardware. The key to secure hardware design is to guarantee that any state changes due to `Untrusted` computation never affect any `Trusted` computation even when the computations share the same pipeline stages, cache, and other processor features. Caisson's language abstractions and type system collaborate to provide the hardware designer with the tools needed to easily encode and verify a secure design. In the remainder of the section we describe in detail the Caisson implementation of our processor.

### 4.2 Secure Execution Pipeline

The execution pipeline is the backbone of the processor and the foundation of a secure design. Our goal is to take a *trusted context* (i.e., registers and memory) and a separate *untrusted context* and use the same physical processor logic circuits to safely operate on both. The resulting processor is a 32-bit RISC processor with 128KB each of Instruction and Data memory (64KB for the trusted context and 64KB for the untrusted context), two program counters, and 16 general purpose registers (split evenly between the trusted and untrusted contexts). There is a *single* four-stage pipeline shared between the trusted and untrusted contexts (as well as a 2K shared data cache, described in the next subsection). Figure 8 shows a state-machine diagram of the pipeline design.

This design interweaves the `Trusted` and `Untrusted` computation at a coarse granularity. There is a Reset state that resets the hardware state to its initial conditions and a Master state that controls the pipeline logic. The Master state sets a timer and allows the pipeline to be used in alternate time intervals by the `Trusted` and `Untrusted` computation. In this design, every stage of the pipeline contains data from the same execution context (`Trusted` or `Untrusted`). Figure 9 shows the hardware diagram of the cor-

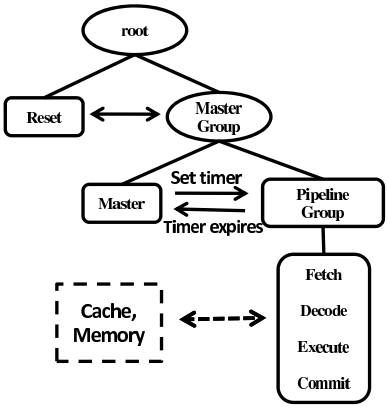
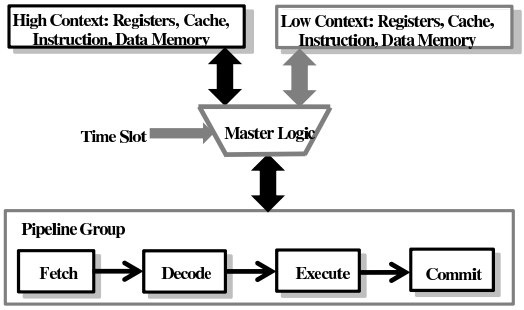



**Figure 8.** State Machine Diagram of the Coarse-Grained Time-multiplexed Pipeline. The memory hierarchy shown in the dash box does not represent any concrete state in the state machine, but it is included in our CPU and accessed by the pipeline.




(b)

**Figure 9.** Synthesized hardware implementation of the Coarse-Grained Time-multiplexed Pipeline CPU logic when the entire pipeline is multiplexed between *high* and *low* code.

responding synthesized circuit. Figure 10 gives a skeleton of the Caisson code that implements the pipeline design (where 'High-Context' and 'LowContext' stand for the untrusted and trusted contexts, respectively). Note that the design uses nested states to allow verification of the design and parameterized states to share the physical logic. In fact, this design is similar to the Execution Lease controller design in §2.

#### 4.2.1 Fine-Grained Sharing of Execution Pipeline

One possible drawback of the previous pipeline design is that the sharing between `Trusted` and `Untrusted` computation is coarse-grained; the time intervals during which they each have control of the pipeline must be sufficiently large to counteract the cost of stalling the pipeline each time the context is switched between them. An alternative design can share the pipeline at a much finer granularity, so that *each pipeline stage* continuously alternates between `Trusted` and `Untrusted` computation at each cycle. This design may be an attractive option when the computation at each security level requires a low latency response.

Figure 12 shows the intuition behind this design, along with a state-machine diagram illustrating the design. Figure 11 gives a skeleton of the Caisson code that implements the design. Each pipeline stage is initialized at a particular security level, so that Fetch is `Trusted`, Decode is `Untrusted`, Execute is `Trusted`, and Commit is `Untrusted`. In each cycle (i.e., at each state transition

```
prog CPU = HighContext: H, LowContext:L, timer:L, mode:L
in let state reset:L() = {
        mode := 0;
        goto mastergroup()
    }
    state mastergroup:L() = {
      let state master:L() = {
            timer = …
            if mode then mode:=1;  goto pipegroup(HighContext)
                    else mode:=0;  goto pipegroup(LowContext)
        }
        state pipegroup:L(context:A) [L<A] = {
          let state pipe:L() = {
                //Fetch, Decode, Execute, Commit
                goto pipe()
            }
          in
            if (timer==0) then goto master()
                          else timer := timer – 1; fall
        }
      in
        fall
    }
```

**Figure 10.** Implementation of the coarse-grained multiplexing pipeline. The timer can be initialized to any number hence we omit the concrete value here.

```
prog CPU = F_init: L, D_init: H, E_init: L, C_init : H
in let state reset:L() = {
            goto pipe(F_init, D_init, E_init, C_init)
    }

    state pipe:L(F_ctxt:A, D_ctxt: B, E_ctxt: A, C_ctxt: B) = {
            //Fetch, Decode, Execute, Commit
            // New context with security level B is obtained for Fetch
            // Each stage calculates and updates their context
            goto pipe(newF_ctxt, F_ctxt, D_ctxt, E_ctxt)
    }
```

**Figure 11.** Implementation of the fine-grained multiplexing pipeline using Caisson

`goto`), the security context at one stage is passed on to the next stage in turn.

### 4.3 Secure Cache Design

A secure execution pipeline prevents information leaks via hardware data- and control-flow, but information can still be leaked via microarchitectural features such as cache memory. For example, there are well-known security attacks that exploit a shared memory to covertly communicate between two supposedly isolated processes by selectively causing page faults and/or cache misses [5, 35]. We implement a secure cache for our processor design to illustrate how to secure microarchitectural features; other features, such as branch predictors, can be securely implemented using similar strategies.

As with the execution pipeline, there are two basic approaches to securing the cache: physical isolation (statically partitioning the cache between `Trusted` and `Untrusted data`) or time multiplexing (sharing the cache, but flushing it each time the processor switches between the `Trusted` and `Untrusted` contexts). In this case, unlike the pipeline, the extreme cost of flushing the cache at each context switch means that partitioning the cache is the pre-

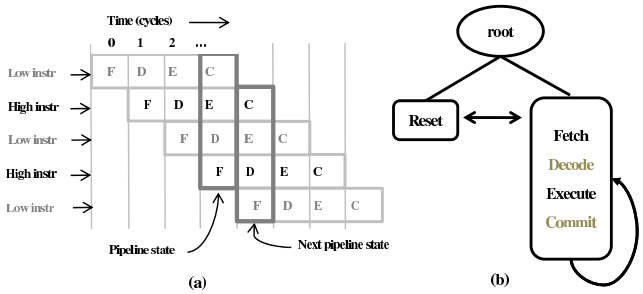



**Figure 12.** Fine-Grained Pipeline Multiplexing. (a) Intuition behind the design (b) State Machine Diagram

ferred solution. Other existing work has come to the same conclusion [54]. Our contribution is not the design of a secure cache itself, but the fact that a secure cache can be easily implemented and statically verified using Caisson, as well as securely integrated into a larger secure system (i.e., our processor). Static verification is important—previous work on secure cache design [54] has been shown to possess subtle flaws that violate security [26].

In Caisson, the implementation of a partitioned cache is simple: the design passes the particular cache partition each context should use as part of the context information for the parameterized states. In Figure 10, the cache partitions would be part of 'HighContext' and 'LowContext'. Equally as important as the cache memory itself is the cache controller—the logic that processes cache hits and misses. Unlike the cache memory, the cache controller can be shared among the different security levels in the same manner as the execution pipeline.

### 4.4 Evaluation

We have proven that Caisson gaurantees the security of our processor design, but an interesting question is how the resulting design performs in comparison to existing processor designs, both traditional insecure processors and other processors designed for security. The relevant performance metrics are: synthesis time (how long it takes to transform a high-level design into an actual circuit); chip area (how large the resulting circuit is); delay (inversely proportional to clock frequency); and power (the power consumption of the circuit).

#### 4.4.1 Synthesis Methodology

To quantify the hardware design overhead introduced by our approach we compare our processor design (**Caisson**) with a non-secured, simplified version of the commercial Nios Processor (**Base**) and the same Nios processor augmented to dynamically track information flow using GLIFT (**GLIFT**) [46]. GLIFT implements full system information flow tracking at the logic gate level: it associates each bit in the system with a taint bit indicating its security level, and augments each gate in the hardware design with additional gates that compute taint propagation.

All CPUs have identical functionality and configuration. However both **Caisson** and **GLIFT** can only utilize half of the cache and memory capacity effectively although they have identical configuration as the **Base** processor. The reason is that in our Caisson design the memory and cache have to be partitioned into two parts with different security levels, while **GLIFT** needs to associate a one-bit tag for each bit in the memory and cache. Increasing the cache and memory utilization efficiency for **Caisson** is part of our future work.

We implemented the **Base** processor (from the Nios design) in Verilog with no additional security features. To get the Caisson implementation we remodeled the **Base** implementation using

| | Base | GLIFT | | Caisson | |
|---|---|---|---|---|---|
| Synthesis Time (min) | 1:50 | 153:56 | **83.96X** | 4:04 | **2.22X** |
| Area ($\mu m^2$) | 38462.86 | 128506.89 | **3.34X** | 52088.78 | **1.35X** |
| CPU delay (ns) | 2.96 | 7.79 | **2.63X** | 4.32 | **1.46X** |
| Power ($\mu W$) | 614.148 | 1730 | **2.82X** | 666.912 | **1.09X** |

**Table 2.** Synthesized results of the different CPU designs: the simplified Nios Processor (Base), the GLIFT-based CPU, and the Caisson-based CPU.

security widgets provided by the Caisson language and statically partitioned all registers, caches, and memories into `Trusted` and `Untrusted`. To get the **GLIFT** implementation, we first synthesized the **Base** design into a gate level netlist and then augmented the netlist with shadow logic to track information flow. We passed the **Base** and **Caisson** designs through Altera's QuartusII v8.0 tool to synthesize the designs onto a Stratix II FPGA for functional testing and verification. We then obtain the area, timing and power results using the Synopsis Design Compiler and the SAED 90nm technology library [1] assuming a switching activity factor of 50% for the circuit.

#### 4.4.2 Results

Almost as important as the quantitative performance results are the *qualitative* results of how easy each design was to implement—this is an important test for the usability of a language. We find anecdotally that Caisson is easily usable by a programmer trained in Verilog. The original **Base** design required 709 lines of Verilog—the corresponding **Caisson** design required only 724 lines and took little additional time to implement. By contrast, **GLIFT** required us to make a hard choice: we could either (1) manually design the gate-level netlist at a structural level (i.e., manually place the logic gates to create the design), which in our experience is infeasible for such a complex design; or (2) generate a gate-level netlist from the behavioral Verilog design using an existing tool, then automatically generate the GLIFT shadow logic using the resulting netlist. We used the latter option, and while it simplifies the process for the programmer the resulting design is intractably difficult to debug and optimize.

Table 2 gives the performance figures for each design. We give the concrete numbers for all three designs as well as normalized numbers for **Caisson** and **GLIFT** (using **Base** as a baseline). The area numbers do not include the memory hierarchy since all three designs use an identical memory configuration. The power numbers include both dynamic and leakage power. The **GLIFT** design comes with a large overhead in all four performance categories due to the shadow logic that GLIFT introduces to the processor. This shadow logic takes a long time to synthesize, requires a large chip area, consumes a great deal of power, and drastically slows the processor cycle frequency.

**Caisson**, in contrast, has a much lower overhead, though it is certainly not free. This overhead mainly comes from two sources: the duplicated state (i.e., registers) and the additional encoders and decoders used to multiplex the partitioned state onto the same logic circuits. We note that the overhead generated by the **Caisson** design does not grow with CPU complexity (e.g., number of functional units)—a more powerful and complex CPU would not require any additional overhead, while the **GLIFT** design's overhead *would* proportionately with the CPU complexity. For perhaps the most important performance metric, power, **Caisson**'s overhead is almost negligible. The synthesis time for the Caisson design includes type-checking, which is sufficient to verify the design's security. The GLIFT synthesis time does *not* include verification—GLIFT only detect security violations at runtime.

These results show that designing a secure processor using Caisson not only provides a strong static guarantee about information flow security, but also (1) allows a more straightforward and natural way of designing a secure processor, and (2) introduces much less area, timing and power overhead than dynamic tracking techniques such as GLIFT.

## 5. Related Work

Secure information flow has been widely studied; a survey by Sabelfeld and Myers [39] gives a comprehensive summary of this work. Here we concentrate on (1) hardware-assisted secure information flow, and (2) analyzing hardware designs for secure information flow, for both secrecy and integrity. Although the concept of integrity has sometimes been generalized to include other properties such as program correctness [17, 30], we deal only with information integrity in this paper.

A number of papers deal with timing-sensitive secure information flow for programming languages [8–10, 21, 37, 42, 52, 56]. These papers enforce timing-sensitivity by restricting computation, e.g., by not allowing any branches or loops on high information. These restrictions wouldn't allow for any useful hardware designs and so these works can't be used in Caisson; instead, we take advantage of the characteristics of sychronous hardware to enforce timing sensitivity (as explained in Appendix A).

### 5.1 Hardware-Assisted Secure Information Flow

Secure information flow has been enforced at a variety of levels of abstraction in computer systems. At the programming language level information flow can be enforced statically (e.g., via a type system [33, 53]) or dynamically [29, 36]. At lower levels of abstraction dynamic enforcement is the norm. Projects such as LoStar [59], HiStar [58] and Flume [28] apply distributed information flow control (DIFC) [57] through general-purpose operating system abstractions. Tag-based tracking at the virtual machine, architecture, or ISA levels is a popular dynamic solution that tracks information flows through all registers and memory [12, 13, 34, 38, 43, 49, 50].

However, even hardware-assisted secure-information flow tracking does not go below the ISA abstraction level to account for microarchitectural features such as pipelining and cache. There are existing dedicated secure cache [54] and memory controller [32] designs, however these designs only enforce information flow policies for specific components in the computer architecture; this existing work does not address the entire processor design, nor does it provide a general methodology for designing secure hardware.

### 5.2 Information Flow Analysis for Hardware Design

The traditional method for checking the security of a hardware design is to simulate and extensively test the design, a laborious and expensive process. While static analysis [19, 22, 23, 40] and model checking [11] are often used for *functional* verification, they are not often used to check security. Tolstrup et al [47, 48] describe a type-system based information flow analysis to verify hardware security policies for VHDL designs. Their work is limited to analyzing simple cryptographic hardware designs, and as pointed out by Li et al [31], directly applying conventional information flow analysis to existing hardware description languages often leads to imprecise results.

Unlike the existing work, Caisson extends HDLs like VHDL and Verilog with language abstractions that specifically target precise static verification of hardware designs.

## 6. Conclusion

Hardware mechanisms for information flow control often form the root of trust in high assurance systems and are used to enforce policies such as non-interference. While programming language techniques have been used extensively for creating secure software, languages to create information-flow secure hardware have not received much attention. We combine insights from traditional type-based secure languages with domain-specfic design patterns used for hardware design and present a new hardware description language, Caisson, for constructing statically-verifiable secure hardware designs. By formalizing certain security design patterns and providing direct language support for enforcing their correct use, Caisson promotes *thinking* about secure hardware design in new, useful ways that don't naturally arise in existing languages. Using Caisson, we are able to express information flow control mechanisms in a natural manner and quickly verify a variety of novel secure hardware designs. In particular, we show how the insights gained from developing Caisson allow us to design the first ever implementation of a pipelined processor that verifiably enforces noninterference.

## A. Proof of Noninterference

We sketch a proof that Caisson enforces timing-sensitive noninterference between security levels. The invariant that we wish to enforce is that an observer at security level $\ell$ cannot distinguish between two runs of the same program that differ only in the information at security levels $\ell' \not\sqsubseteq \ell$. Because Caisson models synchronous hardware designs there are two important implications that we leverage: (1) observers can see the stores only at the end of each cycle, i.e., they cannot see changes to the store that happen during a cycle until that cycle ends; and (2) the length of time between two sequential **goto** commands is *always* exactly one cycle, regardless of the number of semantic steps taken. These two facts are justified by the synchronous nature of the hardware: the flip-flops only read in new values on a clock edge, and the clock frequency is set so that each state (corresponding to a combinational circuit) is guaranteed to have completed within one cycle.

We define "distinguishability" using the *L-equivalence* relation defined below. We then give a set of lemmas and our main noninterference theorem.

### A.1 L-equivalence

First we define the set of security types $L$ that an observer at security level $\ell$ can observe. This includes base types that are subtypes of $\ell$ as well as type variables of state parameters that the current environment $\gamma$ maps to registers whose base types are subtypes of $\ell$. Formally, for a given security level $\ell$ and environment $\gamma$ let $L =$ the minimum fixpoint of $\{\ell' \mid \ell' \sqsubseteq \ell\} \cup \{\alpha \mid \exists v \in dom(\gamma).v : \alpha \wedge \gamma(v) \in L\}$. Then let $H = \{\tau \mid \tau \notin L\}$, i.e., the security types that an observer at level $\ell$ can't distinguish. Types $L$ and $H$ are always with respect to some environment $\gamma$.

We lift the type constructors and typing relation to operate on $L$ and $H$ in the obvious way. We now use $L$ and $H$ to define the *L-equivalence* relation $\sim_L$ on stores, environments, commands, and configurations such that $L$-equivalent items are indistinguishable from each other for an $L$-observer.

- **Environment:** Two environments are equivalent ($\gamma_1 \sim_L \gamma_2$) if they cannot be used to create effects distinguishable by an $L$-observer. This means that (1) any writable $L$-typed state parameter (i.e., not from an $H$-typed state) must be mapped to the same register in both environments; and (2) if $\gamma_1$ maps a label $l$ to an $L$-typed state then so must $\gamma_2$ and vice-versa:

- For all parameters $v$ of any state $l : \mathtt{st}_L(\vec{\alpha}, \kappa)$, $v : L$ w.r.t. either $\gamma_1$ or $\gamma_2 \Rightarrow \gamma_1(v) = \gamma_2(v)$, **and**
- $\forall l \,.\, \gamma_1(l) : \mathtt{st}_L(\vec{\alpha}, \kappa) \vee \gamma_2(l) : \mathtt{st}_L(\vec{\alpha}, \kappa) \Rightarrow \gamma_1(l) = \gamma_2(l)$

- **Store:** Two stores are equivalent if they agree on all values that can be seen by an $L$-observer. Let the $\downarrow_L$ operator project out all registers with type $H$ from a store (since registers always have a base type we don't need $\gamma$ to determine $H$); then $\sigma_1 \sim_L \sigma_2$ if $(\sigma_1 \downarrow_L) = (\sigma_2 \downarrow_L)$
- **Command:** Two commands are equivalent w.r.t. an environment $\gamma$ ($c_1 \sim_L^{\gamma} c_2$) if (1) they are the same command and both typed $\mathtt{cmd}_L$ w.r.t. $\gamma$, hence will modify state in the same way; or (2) both commands are typed $\mathtt{cmd}_H$ w.r.t $\gamma$, hence cannot modify state that is $L$-observable:
  - $c_1 = c_2 \wedge c_1 : \mathtt{cmd}_L \wedge c_2 : \mathtt{cmd}_L$, **or**
  - $(c_1 : \mathtt{cmd}_H \wedge c_2 : \mathtt{cmd}_H)$
- **Configuration:** Two configurations are equivalent ($\mathbb{C}_1 \sim_L \mathbb{C}_2$) if their stores, environments, and commands are equivalent *and* they have the same time: $\exists \gamma$ s.t. $\gamma_1 \sim_L \gamma \wedge \gamma_2 \sim_L \gamma \wedge c_1 \sim_L^{\gamma} c_2 \wedge \sigma_1 \sim_L \sigma_2 \wedge \delta_1 = \delta_2$

### A.2 Lemmas

This section introduces a set of lemmas that we use for the noninterference proof. Simple security states that an expression $e : L$ contains only low sub-expressions, while confinement states that $c : \mathtt{cmd}_H$ cannot create any effects involving $L$-typed variables. The types $L$ and $H$ are with respect to a well-formed environment $\gamma$.

**Lemma 1** (Simple Security). *$e : L \Rightarrow$ no sub-expression of $e$ has type $H$.*

*Proof.* By induction on the structure of $e$. □

**Lemma 2** (Confinement). *$c : \mathtt{cmd}_H \Rightarrow$ evaluating $c$ using environment $\gamma$ and some store $\sigma$ does not modify the value of any variable or register with type $L$.*

*Proof.* By induction on the structure of $c$. □

**Lemma 3** (Subject Reduction). *$\langle c, \gamma, \sigma, \delta\rangle \rightsquigarrow \langle c', \gamma', \sigma', \delta\rangle$ and $c : \mathtt{cmd}_H$ w.r.t. $\gamma \Rightarrow c' : \mathtt{cmd}_H$ w.r.t. $\gamma'$*

*Proof.* By induction on the structure of $c$. □

### A.3 Noninterference

We now come to the noninterference theorem itself, which holds for all well-typed Caisson programs. Recall that because the language models synchronous hardware, we assume that changes to the store are only visible upon a state transition and that time only increments at state transitions (regardless of the number of semantic steps taken). With that in mind, we give the noninterference theorem below. It states that given two configurations $\mathbb{C}_A$ and $\mathbb{C}_B$, both of which are $L$-equivalent and start at the root command, when the resulting evaluations each reach a state transition (i.e., a **goto**) the resulting new configurations must also be $L$-equivalent.

**Theorem 1** (Noninterference).
*Let*

- $\mathbb{C}_A = \langle \mathcal{F}_{root}, \gamma_A, \sigma_A, \delta_A\rangle$
- $\mathbb{C}'_A = \langle \mathcal{F}_{root}, \gamma'_A, \sigma'_A, \delta_A + 1\rangle$
- $\mathbb{C}_B = \langle \mathcal{F}_{root}, \gamma_B, \sigma_B, \delta_B\rangle$
- $\mathbb{C}'_B = \langle \mathcal{F}_{root}, \gamma'_B, \sigma'_B, \delta_B + 1\rangle$

*Then*

$$\mathbb{C}_A \rightsquigarrow^* \mathbb{C}'_A \wedge \mathbb{C}_B \rightsquigarrow^* \mathbb{C}'_B \wedge \mathbb{C}_A \sim_L \mathbb{C}_B \Rightarrow \mathbb{C}'_A \sim_L \mathbb{C}'_B$$

*Proof.* By induction on the steps of the computation and Lemmas 1, 2, and 3. □

The proof itself is straightfoward and omitted for space. Note that the noninterference theorem is timing-sensitive: it guarantees that given two $L$-equivalent configurations, a Caisson program must make identical $L$-observable changes *at the same times* under both configurations (where time is measured as number of cycles).

## Acknowledgments

This research was supported by the US Department of Defense under AFOSR MURI grant FA9550-07-1-0532.
The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring agencies.

## References

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

[2] Common critera evaluation and validation scheme. http://www.niap-ccevs.org/cc-scheme/cc_docs/.

[3] Validated FIPS 140-1 and FIPS 140-2 cryptographic modules. http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140val-all.htm.

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

[5] O. Accigmez, J. pierre Seifert, and C. K. Koc. Predicting secret keys via branch prediction. In *The Cryptographers' Track at the RSA Conference*, pages 225–242. Springer-Verlag, 2007.

[6] O. Aciiçmez. Yet another microarchitectural attack: Exploiting i-cache. In *CCS Computer Security Architecture Workshop*, 2007.

[7] O. Aciiçmez, J.-P. Seifert, and C. K. Koc. Micro-architectural cryptanalysis. *IEEE Security and Privacy*, 5:62–64, July 2007.

[8] J. Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 40–53, New York, NY, USA, 2000. ACM.

[9] G. Barthe, T. Rezk, and M. Warnier. Preventing Timing Leaks Through Transactional Branching Instructions. *Electronic Notes Theoretical Computer Science*, 153:33–55, May 2006.

[10] G. Boudol and I. Castellani. Noninterference for concurrent programs. pages 382–395, 2001.

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

[12] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Micro*, pages 221 – 232, 2004.

[13] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *ISCA*, pages 482–493, 2007.

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

[15] F. A. A. (FAA). Boeing model 787-8 airplane; systems and data networks security-isolation or protection from unauthorized passenger domain systems access. http://cryptome.info/faa010208.htm.

[16] A. Filinski. Linear continuations. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 27–38, New York, NY, USA, 1992. ACM.

[17] C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *POPL*, pages 323–335, 2008.

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

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

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

[21] D. Hedin and D. Sands. Timing aware information flow security for a javacard-like bytecode. 141(1):163–182, 2005.

[22] C. Hymans. Checking safety properties of behavioral VHDL descriptions by abstract interpretation. In *International Static Analysis Symposium*, pages 444–460. Springer, 2002.

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

[24] 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 *SOSP*, pages 207–220, 2009.

[25] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397, 1999.

[26] 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*, pages 25–34, 2008.

[27] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. *IEEE Symposium on Security and Privacy*, pages 447–462, 2010.

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

[29] L. C. Lam and T.-c. Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 463–472, 2006.

[30] P. Li, Y. Mao, and S. Zdancewic. Information integrity policies. In *Proceedings of the Workshop on Formal Aspects in Security and Trust*, 2003.

[31] X. Li, M. Tiwari, B. Hardekopf, T. Sherwood, and F. T. Chong. Secure information flow analysis for hardware design: Using the right abstraction for the job. *The Fifth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security(PLAS)*, June 2010.

[32] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Micro*, pages 146–160, 2007.

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

[34] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

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

[36] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Micro*, pages 135–148, 2006.

[37] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. pages 120–135, 2007.

[38] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing dynamic information flow tracking. In *SPAA*, pages 35–45, 2008.

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

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

[41] O. Sibert, P. A. Porras, and R. Lindell. An analysis of the intel 80x86 security architecture and implement ations. *IEEE Transactions on Software Engineering*, 22(5):283–293, 1996.

[42] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. pages 355–364, 1998.

[43] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, pages 85–96, 2004.

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

[45] M. Tiwari, X. Li, H. Wassel, F. Chong, and T. Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong noninterference. In *Micro*, 2009.

[46] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *ASPLOS*, March 2009.

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

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

[49] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *Micro*, pages 243–254, 2004.

[50] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *HPCA*, pages 196–206, 2008.

[51] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.

[52] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. page 156, 1997.

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

[54] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, pages 494–505, New York, NY, USA, 2007. ACM.

[55] S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. 15(2-3):209–234, 2002.

[56] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. pages 29–43, 2003.

[57] N. Zeldovich, S. Boyd-Wickizer, and D.Mazieres. Security distributed systems with information flow control. In *NSDI*, pages 293–308, Apr. 2008.

[58] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

[59] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *OSDI*, Dec. 2008.