SpecVerilog: Adapting Information Flow Control for Secure Speculation

Drew Zagieboylo
Cornell University
Ithaca, New York, USA
dzag@cs.cornell.edu

Charles Sherk
Cornell University
Ithaca, New York, USA
cs897@cornell.edu

Andrew Myers
Cornell University
Ithaca, New York, USA
andru@cs.cornell.edu

Gookwon Edward Suh
Cornell University
Ithaca, New York, USA
suh@csl.cornell.edu

ABSTRACT

To address transient execution vulnerabilities, processor architects have proposed both defensive designs and formal descriptions of the security they provide. However, these designs are not typically formally proven to enforce the claimed guarantees; more importantly, there are few tools to automatically ensure that Register Transfer Level (RTL) descriptions are faithful to high-level designs.

In this paper, we demonstrate how to extend an existing security-typed hardware description language to express speculative security conditions and to verify the security of synthesizable implementations. Our tool can statically verify that an RTL hardware design is free of transient execution vulnerabilities without manual proof effort. Our key insight is that *erasure labels* can be adapted both to be statically checkable and to represent transiently accessed or modified data and its mandatory erasure under misspeculation. Further, we show how to use erasure labels to defend a strong formal definition of speculative security. To validate our approach, we implement several components that are critical to speculative, out-of-order processors and are also common vectors for transient execution vulnerabilities. We show that the security of existing defenses can be correctly validated and that the absence of necessary defenses is detected as a potential vulnerability.

CCS CONCEPTS

• **Hardware** → **Hardware description languages and compilation**; • **Security and privacy** → *Hardware security implementation*; **Information flow control**; *Formal methods and theory of security*.

KEYWORDS

Information Flow Control, Hardware security, Transient execution attacks, Hardware description languages, Side channels

ACM Reference Format:
Drew Zagieboylo, Charles Sherk, Andrew Myers, and Gookwon Edward Suh. 2023. SpecVerilog: Adapting Information Flow Control for Secure Speculation . In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623074

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '23, November 26–30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623074

1 INTRODUCTION

Spectre and Meltdown [31, 40] have exposed significant timing channel vulnerabilities ingrained in the designs of modern processor microarchitectures. In response, the software security community has developed a number of mitigations, formal security conditions, and principled defenses [7, 21, 22, 35, 47]; however, most of this work requires assuming certain microarchitectural behaviors or models. To make the job of software easier, architects have proposed many hardware defenses [1, 2, 32, 37, 39, 49–51, 53] that provide stronger speculative security guarantees, including invisible speculation [51], transient non-observability [32], and strictness ordering [1].

Implementations defending these conditions can still allow most speculative execution to minimize run-time overhead, while providing reasonable semantics on which secure software can be built. Despite this plethora of designs, few efforts have tried to ensure that synthesizable implementations of these defenses actually uphold their claimed security guarantees.

One established technique for verifying security properties of Register Transfer Level (RTL) hardware descriptions is Information Flow Control (IFC) [15, 59]. In fact, several aforementioned defenses claim to provide IFC-inspired security conditions [9, 21, 22, 32, 54]. We propose using an IFC type system for an RTL Hardware Description Language (HDL) to statically verify that the hardware synthesized from the design is guaranteed to be free of transient execution vulnerabilities. However, this task is not as simple as implementing a speculative processor in an IFC-typed HDL [19]; limitations of the existing languages make it a challenge to defend speculative noninterference and related security conditions.

In this work, we address these limitations and show how an IFC-typed HDL can be extended to guarantee speculative security. Specifically, we adapt *erasure labels* [10], which express a limited form of temporal IFC policies, to an RTL language. Erasure labels allow us to prevent misspeculated data from persisting and influencing non-transient execution, without needing an explicit functional specification for "misspeculation". We also incorporate a novel form of permissive dynamic label checking to enable dynamic scheduling of concurrent instructions. This is one of the first efforts to statically check RTL hardware designs for speculative security guarantees, and the first that does not require significant manual proof effort by the designer. This paper describes our contributions:

- Section 2 provides background on transient execution vulnerabilities and also the capabilities and limitations of some existing IFC tools for RTL design.

- Section 3 describes SpecVerilog, our extension to an existing IFC HDL. We also demonstrate that erasure labels in SpecVerilog can be employed to detect transient execution vulnerabilities and verify secure mitigation mechanisms.
- Sections 4 and 5 provide formal descriptions of SpecVerilog's security guarantees and how they can be instantiated to prevent transient execution vulnerabilities in an out-of-order processor.
- Sections 6 and 7 describe the SpecVerilog implementation and several case studies implemented in SpecVerilog.
- The remainder of the paper relates prior work to SpecVerilog and discusses its design, benefits, and future opportunities.

The SpecVerilog compiler is publicly available for download [55].

2 BACKGROUND

2.1 Transient Execution Vulnerabilities

Speculation is a critical performance feature in modern processors, which introduces *transient instructions* into the processor pipeline. Transient instructions are not part of the intended execution and thus are not allowed to influence any architectural state. Nevertheless, transient execution vulnerabilities such as Spectre, Meltdown, and a host of others [31, 40, 45, 46], exploit the presence of transient instructions to access otherwise-protected data. Such data can then be leaked through well known microarchitectural side channels. To avoid performance degradation, most defenses to these vulnerabilities seek to allow as much speculation as possible while limiting their effects on microarchitectural state. As understanding of these attacks has progressed, researchers have found an ever-increasing number of side channels for transiently accessed data. Some are very subtle, such as *speculative interference attacks* [5], which leave little to no lasting trace within microarchitectural state. In turn, new attacks prompt a new wave of defenses or other modifications to close the newly discovered holes.

We seek to end the cycle of attack discovery and defense development by comprehensively preventing transient execution vulnerabilities in synthesizable RTL designs. We extend existing IFC techniques for HDLs to safely reason about the potential influence of speculative execution. In this way, any design accepted by our tool is guaranteed to be free of transient execution vulnerabilities—even previously undiscovered ones.

2.2 Information Flow Control HDL

Information flow control is a technique for enforcing policies that govern the flow of information, especially policies about confidentiality and integrity [15, 28, 58, 59]. These policies are often formalized as some form of *noninterference*, which states that *high* system state does not influence *low* state. In the case of confidentiality, high corresponds to secret and low to public. When applied to RTL languages, in which the passage of time is explicit, IFC provides timing-sensitive security guarantees. For instance, IFC has been used to implement timing channel resilient hardware modules, from encryption units that protect keys [25, 43] to processors [19, 42, 43] that provide architecture-level security guarantees.

Static IFC tools rely on designer-provided annotations to capture intended security policies. In the case of confidentiality, the designer annotates the secrecy of the system state. The tools then check that the described hardware design obeys the implied policy, and reject unsafe designs. In this work, we build upon and extend one such tool, SecVerilog [59], which uses a type system to check security annotations (labels) on hardware at compile time. We discuss alternative tools for checking hardware IFC properties in Section 8.

The following code is a simple example of the checks made by SecVerilog's IFC type system.

```
1  input d1  { SECRET };
2  input d2  { PUBLIC };
3  reg o1,o2  { PUBLIC };
4  always@( * ) begin
5    o1 = d1;    //FAIL! SECRET->PUBLIC
6    if (d1) o2 = d2; //FAIL! Implicit Flow from d1
7    else    o2 = 0;  //FAIL! Implicit Flow from d1
8  end
```

The code uses two security labels `PUBLIC` and `SECRET`, where secret information is not allowed to leak to public locations. SecVerilog prevents direct illegal information flows, such as at line 5, by ensuring that the operands on the right-hand side of an assignment are permitted to flow to the destination on the left-hand side. Additionally, SecVerilog prevents *implicit flows*, as in lines 6–7, by ensuring that expressions used in branch conditions may flow to conditionally assigned destinations.

2.3 Dynamic Labels

SecVerilog is a static tool, but it allows policies that depend on run-time behavior. It uses *dynamic labels* [60]: security annotations that are determined by run-time values. In a security-typed HDL like SecVerilog, dynamic labels effectively allow the same physical register to store data from different security levels over time. In the following snippet, register `mode` describes the secrecy of the contents of the register `data`.

```
1  // L(0) = PUBLIC; L(1) = SECRET
2  input new_mode  { PUBLIC };
3  reg       mode  { PUBLIC };
4  reg       data  { L(mode) };
```

The function `L(x)` is a dynamic label that maps run-time values onto security levels as described on line 1. For instance, whenever `mode` stores the value `1`, we know that `data` stores secret information. This kind of dynamic label can model an architecture like ARM TrustZone [33], where the processor can switch between secure and insecure worlds [18].

Since a given register's security level can change when the clock ticks, to check non-blocking assignments, SecVerilog considers the destination's *next-cycle* label. Using the types above, we can consider how SecVerilog checks a `mode` change:

```
1  reg       data  { L(mode) };
2  always@(posedge clk) begin
3    mode <= new_mode;
4    data <= (new_mode < mode) ? 0 : data;
5  end
```

SecVerilog requires that values being written into register `data` are allowed to flow to `L(new_mode)`, since that will be the next-cycle label of `data`. For instance, if `mode` is currently `1` (and thus `data` is secret), but `new_mode` is 0 (and thus `data` will become public), SecVerilog only accepts the design if the contents of `data` are overwritten with public information. Line 4 includes the dynamic check necessary for SecVerilog to conclude that the design is secure.

Although powerful, dynamic labels can also introduce subtle security vulnerabilities. One such problem is the "label of labels"

consideration [27]; in our prior example, `mode` itself is a run-time signal and thus has its own label. Therefore, comparisons on dynamic labels can cause implicit flows. SecVerilog avoids these issues with well-formedness assumptions.

Nevertheless, SecVerilog's dynamic labels cannot sufficiently express and defend speculative security conditions. When speculation is involved, the true security level is only determined in the future when the transient state is either invalidated or affirmed.

2.4 Speculative Noninterference Conditions

Prior work has established a variety of noninterference conditions intended to prevent transient execution vulnerabilities [21, 32, 54]. At a high level, these conditions guarantee that speculatively accessed data does not influence attacker-visible state. The definitions of "attacker-visible" and the scope of which speculatively accessed data is protected vary slightly in each, leading to stronger or weaker security guarantees. For this work, we consider a strong, timing-sensitive noninterference condition similar to transient non-observability [32], we call Transient Noninterference. We define it formally in Section 5. Informally, transient non-observability states that transiently accessed instruction operands should not influence the time at which non-transient instructions commit.

To defend Transient Noninterference with an IFC system, we need to annotate hardware state with security labels that describe its speculative status. Labels reference an underlying lattice of security levels that are used to define allowed influence between labels. The lattice $\sqsubseteq$ relation (read: "flows to") dictates the direction information is allowed to flow. A first attempt at such a lattice might be the following:

$$\text{COMMIT} \sqsubseteq \text{SPEC} \sqsubseteq \text{MISS}$$

This set of levels allows committed data to influence everything; misspeculated data cannot influence anything; and unresolved speculative data is in the middle. Unfortunately, this set of levels requires violating noninterference to promote data from `SPECULATIVE` to `COMMITTED` once we learn the speculation was correct. For instance, the following example implements logic to relabel data upon discovering misspeculation or commitment, but fails to type-check:

```
1  //S(0) = COMMIT; S(1) = SPEC; S(2) = S(3) = MISS
2  wire      specCorrect, specMiss { COMMIT };
3  reg [1:0] isSpec { COMMIT };
4  reg       specData { S(isSpec) };
5  always@(posedge clk) begin
6    if (specCorrect)
7      isSpec <= 0; //FAILS: Downgrades SPEC to COMMIT
8    else if (specMiss)
9      isSpec <= 2; //OK: Upgrades data to MISS
10 end
```

Line 7 fails to type-check in SecVerilog. The value of `specData` stays the same, but its label `S(isSpec)` changes since the value of the `isSpec` register *does* change. Since its new label is lower in the lattice, this assignment appears to SecVerilog to violate noninterference. In the current cycle, `specData` may be speculative, and in the next cycle the same data will be treated as committed. This is exactly the designer's intention, but it is not captured by the labels used and cannot be verified as safe by SecVerilog without voiding the language's security guarantees.

Other attempts to map speculative security conditions onto SecVerilog's labels are equally fraught. For instance, consider another dynamic label that only upgrades data upon discovering misspeculation:

```
1  //S(0) = COMMIT; S(1) = MISS;
2  wire isMiss { COMMIT };
3  reg commData { COMMIT }; reg specData { S(isMiss) };
4  always@(posedge clk) begin
5    //If not speculative RIGHT NOW,
6    //we can forget about the dynamic label
7    commData <= (!isMiss) ? specData : commData;
8  end
```

Line 7 illustrates the issue with this labeling scheme: the semantics of `S(x)` allow `specData` to influence committed state on *any cycle* where `isMiss` is false. However, `isMiss` may become true in the future, and thus `commData` may contain misspeculated state.

This problem cannot be solved with conventional IFC labels because they cannot reason about *future* events. SecVerilog's dynamic labels must immediately resolve to a specific security level since they are functions of the current state. Therefore, in order to precisely encode the concept of misspeculation, one would need to label data with a function that computes the correctness of a speculative prediction from the current circuit state. Defining such a function is unwieldy and infeasible in practice. For typical instances of speculation, it would require dynamic labels to depend on the future contents of arbitrary memory locations, which cannot be predicted in general. And full-blown formal verification would be needed to reason about allowed influence using such labels, losing the scalability of SecVerilog's lightweight symbolic reasoning.

3 ERASURE POLICIES & SECURE SPECULATION

Our key insight is that we can enable tractable reasoning about speculative correctness in SecVerilog by extending it with *erasure policies* [11, 12]. An erasure policy is a form of information-flow policy that allows specifying *when* data must be removed or deleted. For example, a software web app might enforce the erasure policy that a user's session data must be deleted after their session expires. In the context of processor development, erasure policies can be used to specify that transiently accessed data (and anything derived from it) must be deleted after misspeculation is discovered. In order to express speculative security conditions, we incorporate erasure policies into an extension of SecVerilog that we call SpecVerilog.

SpecVerilog supports erasure policies through *erasure labels*, security annotations that can express erasure policies in an IFC system. We adapt prior work on software erasure labels [11] to RTL hardware design. Our novel contributions include *dynamic* erasure labels and enforcing erasure policies *fully statically* (i.e., without a run-time monitor).

3.1 Erasure Labels

In SpecVerilog, erasure labels take the following form:

$$b_1 \overset{c(\vec{x})}{\nearrow} b_2$$

Here, b_1 and b_2 are (potentially dynamic) security labels, and $c(\vec{x})$ is an *erasure condition*: a function from a set of program variables to a boolean. Erasure labels guarantee that label b_1 is enforced, *until* the current system state implies the erasure condition is true.

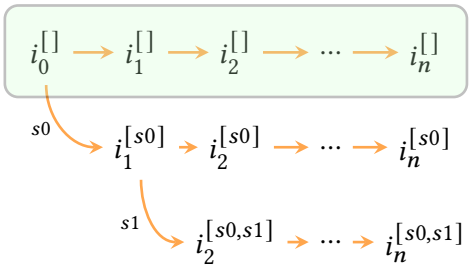

Figure 1: Visualization of Temporal Ordering. The green box highlights all i for which `commit`(i) is true. Annotated arrows represent misspeculation, and an arrow from i_n to i_m indicates that i_m directly follows i_n in program order.

After that point, b_2, the stricter label, is enforced. This mandatory enforcement of a stricter label is called *erasure*. As in prior work, erasure is end-to-end, meaning that the erasure of some data implies the erasure of all data derived from it as well. End-to-end erasure is specified formally in Section 5.

Let us consider an example of erasure in SpecVerilog. For brevity, from here on we use $\bot$ to represent the least restrictive label (e.g., `COMMIT`) and $\top$ to represent the most restrictive (e.g., `MISS`).

```
1  input doErase  { ⊥ };
2  reg   data     { ⊥ doErase↗⊤ };
3  reg   top      { ⊤ };
4  always@(posedge clk) begin
5    data <= data; //FAILS! doErase may be true
6    top  <= data; //OK! top has high label
7  end
```

Since SpecVerilog does not rely on a run-time erasure mechanism, it must verify that whenever an erasure condition might be fulfilled, the relevant data is erased and only written to more restrictive locations. In the above snippet, line 6 is safe because the destination register has a higher label than the upper bound of `data`'s erasure label. However, line 5 is *unsafe* because it is possible that, on any given cycle, `doErase` is true and thus `data` must be erased.

To satisfy the erasure policy and SpecVerilog's type checker, we can change line 5 as follows:

```
data <= (doErase) ? 0 : data;
```

Effectively, SpecVerilog requires the designer to insert run-time checks which enforce the desired erasure policy.

3.2 Ensuring Secure Speculation

Erasure labels are a generic, design-agnostic tool for tracking information flow, but they can be used to prove that hardware designs satisfy speculative security guarantees like Transient Noninterference. Depending on how we apply erasure labels to a processor design, we can achieve different levels of precision or enforce different speculative security conditions.

We illustrate one possible approach that provides comprehensive security with a reasonable level of precision. We show how to defend Temporal Ordering, an approximation of Transient Noninterference introduced by Ainsworth [1]. Ainsworth defines Temporal Ordering as a relation between instructions x and y:

$$x \overset{T}{\Longrightarrow} y \iff \texttt{commit}(x) \vee \texttt{seq}(x, y)$$

The predicate `commit`(x) is true when instruction x has either already completed or is guaranteed to complete. The predicate `seq`(x, y) is true if x comes before y in (a potentially speculative) program order. Figure 1 visualizes a formal definition of these predicates. $i_n^{[s0...sm]}$ is the n^{th} instruction to execute in the program order produced by the sequence of *incorrect* speculative predictions $s0$ through sm. Therefore, only instructions with an empty misspeculation history represent the ISA-defined program order; these are exactly the instructions for which `commit`$(i_n^{[s0...sm]})$ is true.

The `seq` relation (i.e., speculative program order) is equivalent to the arrows in Figure 1. Instruction i_n is only directly connected to i_m by an arrow if i_m is the next instruction in program order or is predicted to be the next instruction by a misspeculation sn. The transitive closure of these arrows is the relation `seq`. In this way, two instructions are only part of the same speculative program order if the misspeculation history of the earlier instruction is a prefix of the later instruction's.

If all value and timing influences between instructions respect Temporal Ordering, the processor also exhibits Transient Noninterference. We formalize this property in Section 5.

To enforce Temporal Ordering with SpecVerilog labels, we only need two underlying lattice elements to represent the security of data: $\bot$ (the least restrictive element) for committed instructions and $\top$ (the most restrictive element) for misspeculated instructions. If we could precisely know, at all times, whether or not an instruction would misspeculate in the future, these simple labels would be sufficient to ensure that misspeculated instructions never influence the time at which other instructions commit. The value of erasure labels is that they track the influence of instructions even without knowing a priori whether they will misspeculate.

3.3 Incorporating Erasure

At a high level, we enforce Temporal Ordering by associating an index x with each instruction which corresponds to its place in the speculative program order. For this design, we assume that the only source of speculation is predicting which instruction to execute next. Later, we discuss how to generalize to other forms of speculation.

We define an erasure label $\text{SL}(x)$ for state associated with instruction x, using an erasure condition we call $\text{INV}(x)$ (for "invalid"):

$$\text{INV}(x) \equiv \texttt{isMiss} \;\&\&\; \texttt{missId} < x$$
$$\text{SL}(x) \equiv \bot \overset{\text{INV}(x)}{\nearrow} \top$$

We assume that `isMiss` and `missId` are control signals in the processor; on any cycle when `isMiss` is non-zero, it indicates that the instruction *after* `missId` was the result of misspeculation (i.e., instruction `missId` made an incorrect prediction).

To illustrate how the SL label can be used to track speculation, consider the following update logic for the program counter (`pc`) register, which stores the address of the next instruction to execute.

```
1  reg pc, pctag, spec_npc  { SL(pctag) };
2  wire isMiss, missId, realnpc  { SL(missId) };
3  always@(posedge clk) begin
4    pc    <= (isMiss && missId < pctag) ?
```

```
5              realnpc : spec_npc;
6    pctag <= (isMiss && missId < pctag) ?
7              missId : pctag + 1;
8  end
```

As the pc changes, we need to keep track of how speculative it is; an obvious way is to also store a tag that increments as it changes. This example has a few interesting components. First, in line 1, we use the SL label for *both* the program counter and its tag. We cannot use $\bot$ for the tag label since the tag itself (i.e., how speculative the next instruction is) is influenced by speculation. Next, in line 2, we introduce the control signals for misspeculation and the "correct" pc value realnpc that is meant to fix the misspeculation. All these signals, including missId itself, are labeled with SL(missId), since intuitively, they must be coming from some instruction *before* the misspeculated instruction in program order. Lastly, the lines to update pc and pctag include cleanup logic for misspeculation; whenever misspeculation is detected, SpecVerilog requires that pc and pctag are overwritten with less-speculative information.

In the above example, the pctag is incremented with each new (speculative) instruction. In this way, the INV erasure condition is consistent with the seq relation from Temporal Ordering:

$$\texttt{seq}(x, y) \iff \text{INV}(x) \implies \text{INV}(y)$$

Next, we generalize the INV erasure condition to properly track instruction order in modern processor designs.

3.4 Leveraging the Reorder Buffer

Speculative, out-of-order (OoO) processors typically maintain a reorder buffer (ROB), which provides the source of truth for program order. The ROB is a first-in-first-out (FIFO) data structure that stores all of the metadata associated with each instruction; entries are inserted in speculative program order and are only removed when they are *committed*: that is, they have updated architectural processor state. If an entry is found to be the result of misspeculation, that entry must be invalidated before it would be committed.

Since ROB order is a proxy for instruction order, we can define a new erasure condition using the ROB ordering:

$$\text{INV}(x, h) \equiv \ \text{isMiss}\ \&\&\ (\text{missId} - h)\,\%\,\text{sz} < (x - h)\,\%\,\text{sz}$$

The design-time constant, sz, describes the size of the ROB, isMiss and missId are the same control signals as before, x is the index of a given ROB entry, and h is the index of the oldest ROB entry. In this way, the ROB can be the source of truth for all of the labels in the processor and is the only component whose labeling we need to trust. For the code examples in this paper, we use the integer erasure conditions that can be compared with $<$ to simplify presentation. Our implementations use the (more realistic) ROB labels.

3.5 Implementing Secure Modules

Hardware modules may handle both speculative and non-speculative state. Securely managing state from multiple security levels is challenging and error-prone; we show how IFC brings some of the potential pitfalls to light, and how to label secure implementations such that they type-check. For this section we use secure caches as our motivating example, but these methods apply to any hardware module that manages state influenced by multiple instructions.

The labels described so far require overwriting the contents of registers that might contain speculative data, a potentially costly implementations for large hardware structures. Instead, real implementations of secure caches [1, 2, 48] use valid bits to mark data as "erased", or even just delay potentially unsafe operations until speculation has been resolved [38]. These optimizations can also be verified as secure in SpecVerilog by using dynamic labels in conjunction with erasure conditions.

Efficient Invalidation. Valid bits can easily be incorporated as dynamic labels to efficiently mark data as unusable. We can modify our erasure labels from the previous example to support this feature:

$$\text{VALID}(b) \equiv \texttt{if}\ (b)\ \bot\ \texttt{else}\ \top$$
$$\text{SLVAL}(b, v) \equiv \text{VALID}(b)\ {}^{\text{INV}(v)}\!\nearrow\!\top$$

The VALID label allows data to flow freely whenever the valid bit, b, is set to 1, else it applies $\top$, meaning the data cannot influence anything. In this way, any data with the SLVAL label can be "erased" upon misspeculation by unsetting its associated valid bit.

```
1  wire isMiss, missId { SL(missId) };
2  reg  sId, sValid       { SLVAL(sValid, sId) };
3  reg  sData             { SLVAL(sValid, sId) };
4  always@(posedge clk) begin
5    sData  <= sData; //OK! erased by marking as invalid
6    sValid <= isMiss && missId < sId ? 0 : sValid;
7  end
```

SpecVerilog correctly accepts the above implementation since the SLVAL label uses VALID as its lower bound; in any cycle where sData must be erased, its valid bit will be set to 0 and so in the next cycle it will be treated as $\top$ (i.e., *above* the required erasure level). If we had removed line 6, then this snippet would not be accepted by SpecVerilog, since there would be no guarantee that sData is invalidated upon misspeculation. Without the use of dynamic labels *in conjunction* with erasure labels, we would not be able to verify this optimization.

Secure Dynamic Scheduling. Secure designs also need to appropriately order or delay operations when they might affect the timing of less speculative ones. Processors that do not correctly schedule operations are potentially vulnerable to SpectreRewind [20] or other speculative interference [5] attacks. To accept implementations that correctly schedule speculative operations, while still rejecting unsound designs, SpecVerilog needs to reason precisely about *label comparisons*. In this context, label comparisons are dynamic checks that determine which of two pieces of data is more speculative.

Consider a latency-insensitive interface that uses *ready* and *valid* bits for making requests to a module. Whenever the following implementation of such a module is *not* currently handling a request it sets *ready* to true and will accept *valid* incoming requests:

```
1  //input and output must have the same label, defined by client
2  input  reqId, reqValid, req  { SLVAL(reqValid, reqId) };
3  output reqReady              { SLVAL(reqValid, reqId) };
4  reg    curId, curValid, cur  { SLVAL(curValid, curId) };
5  always@(posedge clk) begin
6    if (reqValid && !curValid) begin
7      //FAIL! if 0, curValid cannot influence anything
8      curValid <= 1; curId <= reqId; cur <= req;
9    end
10   //FAIL! req might not be allowed to observe cur
11   reqReady = ~curValid;
12 end
```

This logic is, in general, insecure. When the current request is *more speculative* than the incoming one, the incoming request is delayed, leading to a violation of Transient Noninterference. SpecVerilog correctly rejects this design since it cannot prove that SLVAL(`curValid`, `curId`) may influence SLVAL(`reqValid`, `reqId`).

In this scenario, secure designs must allow less speculative requests to preempt others, but must prevent the opposite; this practice is called *leapfrogging* in prior work [1]. SpecVerilog has a permissive label comparison operator that enables implementations of leapfrogging without violating SpecVerilog's security guarantees.

Here we demonstrate how to compute the *ready* bit in a secure SpecVerilog implementation:

```
1  if (L(curValid) ⊑ L(reqReady))
2  //!reqValid || (curValid && curId <= reqId)
3    reqReady = ~curValid;
4  else
5    reqReady = 1;
```

Line 1 demonstrates a label comparison in SpecVerilog, which dynamically computes the labels of `curValid` and `reqReady` and evaluates to 1 only if the comparison holds. The comment describes the actual logic that computes the label comparison. We discuss this operator and its limitations further in Section 4.3. This version safely implements preemption and, with the interface labels from the first example, is accepted by SpecVerilog. Most IFC-based type systems would reject this program because the label comparison can cause an implicit flow. SpecVerilog introduces a novel and more permissive rule that accepts the above code but does not compromise security.

Variable	x		
Level	l	$\in$	$\mathbb{L}$
Function	f	$\in$	$\mathbb{Z}^n \to \mathbb{L}$
Condition	c	$\in$	$\mathbb{Z}^n \to \mathbb{B}$
Basic Types	b	$::=$	$l \mid f(\vec{x})$
Label	τ	$::=$	$b \mid b_1 \overset{c(\vec{x})}{\nearrow} b_2 \mid \tau_1 \sqcup \tau_2 \mid \tau_1 \sqcap \tau_2$
Typing Context	Γ	$\in$	$x \to \tau$

Figure 2: Syntax of security labels. Label functions and policies are specified with variables. Policy conditions may also contain free variables.

4 SPECVERILOG DESIGN

This section briefly presents SpecVerilog's type system, formal security guarantees, and implementation.

4.1 Typing Rules

SpecVerilog extends SecVerilog's existing type system with rules for erasure labels and permissive label comparisons. Our syntactic presentation here differs slightly from prior work [17, 59] but is effectively the same, other than SpecVerilog's novel contributions.

Figure 2 presents the syntax for security labels in SpecVerilog. Other than erasure labels and conditions, this syntax is directly borrowed from SecVerilog. It allows users to define their own underlying security labels, and to define dynamic labels as dependent types (i.e., functions of program state). Erasure labels are formed from two non-erasure labels and an erasure condition[1]. Erasure conditions are functions of run-time state that return true or false; when true, the data labeled with this condition must be erased.

Since SpecVerilog is dependently typed, we also refer to the current system state in some of our rules and definitions. In this presentation we keep the state mostly abstract; this table describes our syntax:

Syntax	Operation
σ	Current system state
$\sigma[x]$	Value of variable x
$\sigma[\vec{x}]$	Value of list of variables $\vec{x}$
$\sigma \to \sigma'$	Clock-tick transition to next state

Well-formedness. We assume several well-formedness conditions about a program's types, which are defined in Figure 3. First, dependent labels must only depend upon variables whose labels are less restrictive; this prevents unwanted information flow channels through label checking. Second, we require that all variables appearing in dependent types are either the same as the variable of the type (i.e., a recursive label) or they must be *sequential* variables (i.e., variables whose values only change on a clock edge). This restriction ensures that any time a dynamic label changes its value, the change is checked for safety by the typing rules. Lastly, we assume that all erasure policies actually represent *upgrades*; after erasure, the policy must be at least as restrictive as before erasure.

(1) $\forall v \in \textbf{Vars}.\forall v' \in FV(\Gamma(v)).\forall \sigma.\text{obs}(\Gamma(v')) \;{}_{\sigma}\!\sqsubseteq \text{obs}(\Gamma(v))$
(2) $\forall v \in \textbf{Vars}.\forall v' \in FV(\Gamma(v)).v' \neq v \implies v' \in \texttt{seq}$
(3) $\forall \tau_1 \overset{c(\vec{x})}{\nearrow} \tau_2.\forall \sigma.\tau_1 \;{}_{\sigma}\!\sqsubseteq \tau_2$

Figure 3: The well-formedness conditions for SpecVerilog type environments. Vars is the set of all variables in the program; $FV(\tau)$ is the set of variables referenced in the type τ; $\text{obs}(\tau)$ is data visibility and is defined in Figure 8.

4.2 Type Checking

SpecVerilog's type checking rules primarily rely on a *may-flow-to* relation, $\sqsubseteq$, which describes allowed influences between security types. As is typical for IFC, this relation is reflexive and transitive and relies on the underlying security lattice ordering. The complete definition of $\sqsubseteq$ can be found in Figure 10 in the appendix.

In Figure 4, we present the most interesting rules for SpecVerilog: those concerning erasure labels. The Erase-Intro rule allows us to add an erasure condition onto an existing policy and the Erase-Elim rule allows us to replace an erasure label with a more restrictive label. The Erase-Weaken rule describes how to replace one erasure label with another. Influence is allowed if the lower and upper bounds both may flow, and if the new erasure condition would evaluate to true any time the original condition would evaluate to true, regardless of the system state.

As in SecVerilog, we use two different typing rules based on whether the destination is updated combinationally, or sequentially.

[1]Erasure labels cannot be nested, but this does not limit expressiveness. Taking the least upper bound ($\sqcup$) of multiple erasure labels can achieve the same effect as placing erasure conditions inside of lower or upper bounds.

$\boxed{\tau \sqsubseteq \tau'}$

$$\textsc{Erase-Elim}\ \frac{\tau_2 \sqsubseteq \tau'}{\tau_1\ {}^{c(\vec{v})}\!\nearrow\!\tau_2 \sqsubseteq \tau'} \qquad \textsc{Erase-Intro}\ \frac{\tau \sqsubseteq \tau_1}{\tau \sqsubseteq \tau_1\ {}^{c(\vec{v})}\!\nearrow\!\tau_2}$$

$$\textsc{Erase-Weaken}\ \frac{\tau_1 \sqsubseteq \tau_1' \qquad \tau_2 \sqsubseteq \tau_2' \qquad \forall \sigma.\sigma \vDash c(\vec{v}) \implies \sigma \vDash c'(\vec{v}')}{\tau_1\ {}^{c(\vec{v})}\!\nearrow\!\tau_2 \sqsubseteq \tau_1'\ {}^{c'(\vec{v}')}\!\nearrow\!\tau_2'}$$

Figure 4: The environment-independent may-flow-to relation for erasure labels.

$\boxed{\tau \downarrow_\sigma \tau'}$

$$\textsc{Labels}\ \frac{}{l \downarrow_\sigma l} \qquad \textsc{Functions}\ \frac{\vec{v} = \sigma[\vec{x}] \qquad l = f(\vec{v})}{f(\vec{x}) \downarrow_\sigma l}$$

$$\textsc{Erase}\ \frac{\tau_1 \downarrow_\sigma \tau_1' \qquad \tau_2 \downarrow_\sigma \tau_2' \qquad \vec{v} = \sigma[\vec{x}]}{\tau_1\ {}^{c(\vec{x})}\!\nearrow\!\tau_2 \downarrow_\sigma \tau_1'\ {}^{c(\vec{v})}\!\nearrow\!\tau_2'}$$

Figure 5: The function that resolves variables in labels.

$$\textsc{ComMFT}\ \frac{\tau \downarrow_\sigma \tau' \qquad \tau_1 \downarrow_\sigma \tau_1' \qquad \tau'\ {}_\sigma\!\sqsubseteq \tau_1'}{\tau\ {}_\sigma\!\sqsubseteq \tau_1}$$

$$\textsc{SeqMFT}\ \frac{\neg\mathrm{Erase}(\sigma, \tau, \tau_1) \qquad \sigma \rightarrow \sigma' \qquad \tau \downarrow_\sigma \tau' \qquad \tau_1 \downarrow_{\sigma'} \tau_1' \qquad \tau' \sqsubseteq \tau_1'}{\tau\ {}_\sigma\!\sqsubseteq_{\mathbf{next}}\ \tau_1}$$

Figure 6: May-flow-to relations parameterized on a given system state. Rules ComMFT and SeqMFT type-check combinational and sequential assignments, respectively.

$$\textsc{ComAssign}\ \frac{\Gamma \vdash e \dashv \tau \qquad \Gamma(x) = \tau' \qquad x \notin FV(\tau') \qquad C \implies pc \sqcup \tau\ {}_\sigma\!\sqsubseteq \tau'}{C, \Gamma, pc \vdash x = e}$$

$$\textsc{SeqAssign}\ \frac{\Gamma \vdash e \dashv \tau \qquad \Gamma(x) = \tau' \qquad x \notin FV(\tau') \qquad C \implies pc \sqcup \tau\ {}_\sigma\!\sqsubseteq_{\mathbf{next}}\ \tau'}{C, \Gamma, pc \vdash x <= e}$$

$$\textsc{LabelComp}\ \frac{\Gamma(v_1) = \tau_1 \qquad \Gamma(v_2) = \tau_2 \qquad \forall \sigma.\tau_1\ {}_\sigma\!\sqsubseteq \tau_2 \vee \tau_2\ {}_\sigma\!\sqsubseteq \tau_1 \qquad \tau = \tau_1 \sqcap \tau_2}{\Gamma \vdash v_1 \sqsubseteq v_2 \dashv \tau}$$

Figure 7: Type checking rules for selected statements and expressions in SpecVerilog. C is a set of constraints about the current (σ) and next (σ') states. pc tracks the label of variables read in the current context.

Figure 6 describes how we resolve variables based on the kind of assignment. For combinational assignments (i.e., blocking), all dependent types are resolved in the current context. For sequential assignments (i.e., non-blocking), the label of the destination is evaluated in the next-cycle context instead. Additionally, for sequential assignments, we require that the source does not need to be explicitly erased with the Erase side condition. This condition returns true when the Erase-Weaken rule must be applied to prove that the flow is allowed *and* the left hand erasure condition in that rule is true this cycle . Figure 7 demonstrates the actual typing rules for assignment statements in SecVerilog, which reference the variable resolution rules in Figure 6. We omit the rules for when labels are recursive for both combinational and sequential assignments for brevity; they are nearly identical to those from SecVerilog.

4.3 Erasure Label Design

The evaluation of erasure conditions in Figures 4 and 5 has a peculiar-looking definition, so here we justify its design. Unlike normal dynamic labels, erasure conditions are allowed to contain *free variables*. Consider our misspeculation example from Section 3.3:

$$\mathrm{INV}(x) \equiv \ \mathrm{isMiss}\ \&\&\ \mathrm{missId} < x$$

INV has two free variables, isMiss and missId. When applying the Erase-Weaken rule to check if $\mathrm{INV}(x)$ implies some other condition, the implication must hold for any possible values of isMiss and missId, but only for a concrete value of x.

This design is necessary to make erasure labels usable while still ensuring erasure conditions are checked in the future. Obviously, if we resolve all of the variables to values before checking, then the implication will hold on any cycle where INV is true, which would (inappropriately) allow us to stop monitoring erasure conditions. On the other hand, leaving all variables free would prevent typing clearly safe programs. Effectively, we would be unable to leverage knowledge of current and future system state to weaken the label.

The following snippet demonstrates a safe program which would not type-check under the more restrictive treatment of erasure weakening. In this example, we check an erasure condition and then conditionally copy speculative data into a register.

```
1  input rId, rData {SL(rId)};
2  reg   sId, sData {SL(sId)};
3  always@(posedge clk) begin
4  //assume erase properly checks the erasure condition for rId
5    if (!erase) begin
6      sData <= rData; sId <= rId;
7    end
8  end
```

If we did not resolve variables in the Erase-Weaken rule, we would need to prove the following to type-check the above code:

$$\forall\, \mathrm{isMiss}, \mathrm{missId}, \mathrm{sId}, \mathrm{rId}.$$
$$(\mathrm{isMiss} \ \wedge\ \mathrm{missId} < \mathrm{sId}) \implies (\mathrm{isMiss} \ \wedge\ \mathrm{missId} < \mathrm{rId})$$

Unfortunately, the above statement does not hold; we cannot actually prove the assignment is safe without relating the current value of `sId` to the new value of `rId` after the assignment.

With the Erase-Weaken rule from Figure 4 and resolving variables according to the correct cycle values as in Figure 6, we can

$\boxed{\text{obs}(\tau) = \tau'}$

$$\text{obs}(\tau_1 \sqcap \tau_2) = \text{obs}(\tau_1) \sqcap \text{obs}(\tau_2)$$
$$\text{obs}(\tau_1 \sqcup \tau_2) = \text{obs}(\tau_1) \sqcup \text{obs}(\tau_2)$$
$$\text{obs}(b_1 \overset{c(\vec{x})}{\nearrow} b_2) = \text{obs}(b_1)$$
$$\text{obs}(b) = b$$

Figure 8: The obs **function defines the visibility of data with a given label. Erasure labels use their lower bound.**

correctly type-check the prior example. We need to prove:

$$\forall \text{isMiss}, \text{missId}.\ (\text{isMiss} \ \wedge \ \text{missId} < \text{sId}) \implies (\text{isMiss} \ \wedge \ \text{missId} < \text{next(rId)})$$

This implication can be proved by statically analyzing the program; in the context of the assignment we can prove sId = next(rId). which is sufficient to prove the overall implication.

Dynamic Label Checks. In SecVerilog (and other IFC type systems with dynamic labels), run-time label comparisons use the same rule as any binary operation: the label of the result is the join ($\sqcup$) of the labels of the operands. Since labels are run-time values, they are computed from some information and can also leak information; this rule prevents those leaks from violating security policies. Unfortunately, this rule is too restrictive for efficient processor designs; specifically, it makes it impossible to type-check dynamically scheduled modules such as caches. Caches handle a limited number of concurrent requests and must decide which to delay when they are overburdened. In practice, this involves comparing the provenance (i.e., labels) of requests; the typical label check rule renders the result of this decision too restrictive to be useful.

However, in the special case where the labels are guaranteed to be *ordered* then there *is* a safe implementation (namely the preemption described in Section 3.5). In this case, we introduce a more permissive typing rule in Figure 7: the LabelComp rule, which effectively uses the *lower label* as the label of the result of the comparison and allows SpecVerilog to type-check the safe implementation. This is an *unintuitive* result which relies on the fact that an attacker can be statically sure that one of labels *must* be able to flow to the other. Appendix B contains a proof of the safety of this typing rule.

5 SECURITY GUARANTEES

In this section, we formalize SpecVerilog's security guarantees. First, we describe noninterference and end-to-end erasure, conditions that hold for all well-typed SpecVerilog programs. Then, we define Transient Noninterference with respect to an abstract processor model. Finally, we show how to use erasure labels on a real processor design to statically guarantee that it satisfies Transient Noninterference.

5.1 Noninterference and Erasure

SpecVerilog provides traditional IFC-style security guarantees.

Observational Equivalence. Observers, including attackers, are characterized by their ability to distinguish different executions. An observer is defined by a security level l. It can observe the value of any variable which may influence data with level l. Additionally, since the labels of variables may change during execution, the *set of variables* that may influence l is visible to the observer as well. Figure 8 defines the observation function that translates labels in a given environment into the corresponding observable lattice level. The interesting part of this function is for erasure labels, which defines the *lower bound* of the erasure label to be the observable level. Data marked by an erasure label is considered visible at the lower bound until it must be erased.

Given this observation function, we define *observational equivalence*. When two program states are observationally equivalent with respect to some level, l, any attacker that can observe up to l *cannot* distinguish the two states by direct observation.

Definition 1 (Observational Equivalence). *Two states are observationally equivalent w.r.t level l ($\sigma_1 \approx_l \sigma_2$) when:*

$$\forall x \in \mathit{VARS}.\ o = \text{obs}(\Gamma(x)) \ \wedge o_{\sigma_1} \sqsubseteq l \iff o_{\sigma_2} \sqsubseteq l \ \wedge \ o_{\sigma_1} \sqsubseteq l \implies \sigma_1[x] = \sigma_2[x]$$

Well-typed SpecVerilog programs exhibit noninterference. Simply put, noninterference ensures that, if an attacker cannot distinguish two states, then the attacker will not be able to distinguish the result of executing those states.

Definition 2 (Noninterference). *A program is noninterfering if for an attacker defined by an observation level l, observational equivalence is preserved during execution:*

$$\forall l \in \mathcal{L}, i \in \{1,2\}.\sigma_i \rightarrow \sigma_i' \wedge \sigma_1 \approx_l \sigma_2 \implies \sigma_1' \approx_l \sigma_2'$$

These definitions of observational equivalence and noninterference are standard for IFC systems with dynamic labels and mirror those of SecVerilog.

End-To-End Erasure. All well-typed SpecVerilog programs also enforce *end-to-end erasure*. Intuitively, erasure ensures that, once an erasure policy's condition is fulfilled, the labeled data must only have influenced state that can be observed at or above the upper label. Our definition of erasure is inspired by Hunt and Sands [24], but we modify it to account for our definition of observability, and also to incorporate dynamic labels and semantic erasure conditions.

While the following specification of end-to-end erasure is dense, it can be summarized concisely. If, at any point, some variable will eventually need to be erased, then replacing that variable with an uninterpreted value and continuing execution results in a post-erasure state that is indistinguishable (for a low observer) from an execution that uses the original value.

Definition 3 (End-To-End Erasure). *Given an infinite trace of system states: $\sigma_0 \rightarrow \sigma_1 ... \rightarrow \sigma_n ...$, if for all variables x, the erasure policy of x in state σ_i is ever satisfied in some future state, σ_j, then replacing x in σ_i with an uninterpreted value ($\perp$) results in a future state that is l-equivalent to σ_{j+1} for any l that the upper bound on x's erasure policy cannot observe.*

$$\forall i, j, x, c(\vec{y}), b, l.\quad i \leq j \ \wedge \ (c(\vec{y}), b) \in \text{eraseTo}(\Gamma(x)) \ \wedge \ \vec{v} = \sigma_i[\vec{y}] \ \wedge \ \sigma_j \vDash c(\vec{v}) \ \wedge \ \sigma_i' = \sigma_i[x \mapsto \perp] \ \wedge \ \sigma_i' \rightarrow^{j-i+1} \sigma_{j+1}' \ \wedge \ b_{\sigma_i} \not\sqsubseteq l \implies \sigma_{j+1}' \approx_l \sigma_{j+1}$$

This definition relies on the eraseTo function, which returns a set of pairs of erasure conditions and levels. When the condition

evaluates to true, the variable must be erased to the given level. For simple erasure labels, eraseTo is specified in the obvious way, returning the erasure condition and the upper bound label. Definition 7 in the appendix describes the complete eraseTo function.

5.2 Speculative Security

Here we formalize a strong and usable definition of speculative security and sketch how SpecVerilog can enforce this condition by applying Temporal Ordering-based labels.

Attacker Model. Most prior models of speculative security [21, 22, 54] are made with respect to an attacker that can make direct observations of microarchitectural state or actions (such as caches, branch predictors, or speculatively loaded addresses). However, these models can both lead to unsoundness by overlooking potential attacks or, conversely, overestimate the attacker's power by leaking state that may not actually influence timing.

Instead, we model software-level timing side channel attackers more realistically: attackers can observe the time at which each instruction completes but they may not observe intermediate microarchitectural states. Our model faithfully reflects an attacker that can execute code on the processor and make deductions from the timing of its execution, but does not reflect the power of attackers with physical access to the hardware (who might exploit other side channels such as power or EM radiation).

Transient Noninterference. As mentioned in Section 2.4, Transient Noninterference effectively enforces a security condition defined by prior work, transient non-observability. However, in order to formally show that Transient Noninterference is safe with respect to our strong attacker model, we use a definition more similar to Unique Program Execution (UPE) [16], a baseline confidentiality condition for processors. UPE says that if secret architectural state[2] (register or memory contents) can be leaked to an attacker via timing channels, then that state must also affect the values of attacker-visible architectural state.

So far, we have argued that erasure labels can be used to enforce Temporal Ordering. However, Temporal Ordering is not sufficient to enforce UPE (and Transient Noninterference) on its own. Processors may pathologically leak information about arbitrary architectural state via timing channels even *without* speculative execution. For instance, a processor satisfying Temporal Ordering could use an arbitrary value in memory to prefetch cache lines, or to otherwise delay instruction commit; implementations exhibiting these behaviors would violate UPE. While these sorts of leaks do represent potential bugs, they are of the kind that architects and functional verification tools are likely to find and eliminate.

Therefore, we prove a slightly weaker theorem by restricting our guarantee to processors that only read architectural state associated with some transiently executed instruction[3]. We formalize this assumption by defining an abstract OoO processor semantics, which Figure 9 in the appendix depicts. This model is more general than those used by prior work (e.g., [8, 22]) so that our guarantees rely on few microarchitecural assumptions or attack vectors.

The processor consists of the following components:

Syntax	Description
rob	Reorder buffer for in-flight instruction metadata
A	Architectural state (registers, memory, and pc)
μ	Microarchitectural state

The processor can: speculatively *fetch* instructions, placing them into the rob; *execute* instructions, updating μ based on the ISA–defined semantics that instruction; *commit* instructions, removing them from the rob and updating A; and rollback state upon discovering *mispredictions*. The processor has an abstract scheduler that determines which operations to run each cycle and is a function of μ and any architectural state read by instructions in the rob. We assume that the scheduler respects registered hardware semantics (i.e., persistent state can only be written once per clock cycle).

The $[\![\cdot]\!]$ notation is used to extract the (infinite) trace of architectural states (A_i) from executing a given initial processor configuration ($\mathcal{P}$):

$$[\![\mathcal{P}]\!] = A_0\, A_1 \ldots A_n \ldots$$

This trace contains the entire architectural state on every clock cycle during the processor's execution.

We also define observation functions to express both architectural and timing-sensitive attackers.

Definition 4 (Low Architectural Observer). *A low architectural observer (O_l) is defined with respect to an arbitrary subset of the architectural state ($A_l \subseteq A$). This observer can view the time-independent sequence of low architectural states.*

$$O_l(A_0\, A_1 \ldots) = \text{ if } (A_{l0} = A_{l1}) \text{ then } O_l(A_1 \ldots) \text{ else } A_{l0}\, O_l(A_1 \ldots)$$

Definition 5 (Timing-Sensitive Observer). *A timing-sensitive low observer (T_l) can observe the A_l on every clock cycle.*

$$T_l(A_0\, A_1 \ldots) = A_{l0}\, A_{l1} \ldots$$

Lastly, we formally define Transient Noninterference, which can be enforced via Temporal Ordering and SpecVerilog-checked erasure labels.

Definition 6 (Transient Noninterference). *Processor $\mathcal{P}$ exhibits Transient Noninterference if, for any partitioning of A, all executions indistinguishable to a low-architectural observer are also indistinguishable to a timing-sensitive observer.*

$$\forall i \in 1, 2.\ \mathcal{P}_i = \langle rob_i, A_i, \mu_i \rangle,$$
$$A_i = \langle A_{li}, A_{hi} \rangle,\ A_{l1} = A_{l2}, \mu_1 = \mu_2, rob_i = pc_i$$
$$O_l([\![\mathcal{P}_1]\!]) = O_l([\![\mathcal{P}_2]\!]) \implies T_l([\![\mathcal{P}_1]\!]) = T_l([\![\mathcal{P}_2]\!])$$

5.3 Enforcing Transient Noninterference

In this section, we briefly justify both why a processor that satisfies Temporal Ordering must also satisfy Transient Noninterference, and also how properly applied erasure labels enforce Temporal Ordering using the processor semantics from Figure 9. Note that this model assumes that the only source of speculation is next-instruction prediction; this assumption is simplifying for presentation but not necessary. We discuss how to extend these results to more general speculation and other processor models in Section 9.

[2]Here, "secret" is defined with respect to an arbitrary architecture–level security policy, so architectural state may be split arbitrarily into secret and public data.

[3]Note that most defenses to transient execution attacks provide similar guarantees, as it is generally assumed processors do not exhibit these kinds of pathological vulnerabilities.

Temporal Ordering enforces Transient Noninterference. The timing behavior of any processor that refines the semantics in Figure 9 is only a function of three things: data read by instructions that commit; data read by those that *only transiently* execute; and the initial microarchitectural state.

Temporal Ordering restricts influence so that different paths of speculative execution cannot influence each other; therefore, varying transiently read data has no impact on timing. Furthermore, by assumption, execution results in the same set of low-architectural observations. Since timing is therefore only a function of low-architectural state, time-sensitive observations of low-architectural state are also independent of architectural secrets.

Erasure labels enforce Temporal Ordering. We use the ROB labels described in Section 3.4 to label our abstract processor. The ROB always contains a reference to the *oldest* instruction that is currently executing, which is guaranteed to eventually commit. We call the index of this instruction in the ROB the *head*. We label each entry in the ROB recursively based on its index, i, with the label: SL(i, *head*). All committed state has the same label as *head*; effectively we stop precisely tracking which instruction influenced architectural state once that instruction is guaranteed to commit. The rest of the processor is labeled such that it type-checks in SpecVerilog (and thus satisfies noninterference and end-to-end erasure).

Here, we argue that noninterference and end-to-end erasure when using these labels on an abstract OoO processor guarantees Temporal Ordering. All state in the processor is a function of the architectural state accessed by *some* set of instructions, therefore so is the time at which each instruction commits. We write $\mathcal{I}(x)$ to denote the set of instructions that have influenced register x at some point in the execution. We use the notation from Figure 1 to denote instructions across speculative program orders. $i_j^{[s_1,\ldots s_n]}$ denotes instruction j that was the result of the (incorrect) speculative predictions s_1 through s_n.

Theorem 1 (Enforcing Temporal Ordering). *For any two registers in a well-typed $\mathcal{P}$, x and y, if x may influence y according to SpecVerilog, then that influence obeys Temporal Ordering.*

$$\forall x.y.i_j^{[s_1,\ldots,s_n]} \in \mathcal{I}(x), i_k^{[p_1,\ldots,p_n]} \in \mathcal{I}(y), \sigma.$$
$$\Gamma(x) \,{}_\sigma\!\sqsubseteq \Gamma(y) \implies i_j^{[s_1,\ldots,s_n]} \overset{T}{\Longrightarrow} i_k^{[p_1,\ldots,p_n]}$$

The proof is relatively straightforward and relies on one main lemma; any given register is only influenced by instructions on a single speculative program order.

Lemma 1. *For any register, x, in a well-typed $\mathcal{P}$,*

$$\forall i_j^{[s_1,\ldots,s_n]}, i_k^{[p_1,\ldots,p_n]} \in \mathcal{I}(x),$$
$$j \le k \implies [s_1,\ldots s_n] \preceq [p_1,\ldots,p_n]$$

where $\preceq$ is prefix order.

Intuitively, Lemma 1 means that, following a misspeculation event, there are no remnants of the misspeculated instructions; the processor is always executing down only one speculative road at a time. Lemma 1 follows from end-to-end erasure, and induction on the OoO processor semantics. We include a proof sketch in Appendix B. Here we sketch the proof for Theorem 1.

Proof Sketch. In any given state, if $\Gamma(x) \,{}_\sigma\!\sqsubseteq \Gamma(y)$, then x is ordered before y in *some* speculative program order (by the processor semantics and ROB labels). By Lemma 1, $\mathcal{I}(x)$ and $\mathcal{I}(y)$ must each only contain instructions from a single speculation path; it must in fact be the same speculation path since they are ordered. Therefore, we have $\mathcal{I}(x) \subseteq \mathcal{I}(y)$, which directly implies Theorem 1. □

6 IMPLEMENTATION

SpecVerilog extends the existing SecVerilog type checker by adding erasure labels[4]. To support dependent types, SecVerilog relies on the Z3 SMT solver [13] for type checking. Dynamic labels are specified by the user as Z3 functions, which are referenced by the constraints that the SecVerilog type checker generates.

In SpecVerilog, users also supply erasure conditions as Z3 functions. We support new syntax for erasure labels directly in Verilog source code that can reference these erasure conditions. SpecVerilog generates constraints based on the may-flow-to relation and typing rules described in Figures 4 and 7 and discharges these constraints to the Z3 SMT solver to correctly type-check designs.

We made several other modifications and improvements to the SecVerilog compiler to incorporate features from other research efforts on IFC type systems for hardware [17, 18, 59]. The modifications improved the efficiency of the compiler (i.e., simplified the generated Z3 constraints), enabled us to precisely type-check our most complicated examples, and fixed some bugs in the publicly available SecVerilog implementation.

Permissive Label Comparisons. The permissive LabelComp rule is not implemented in the current SpecVerilog prototype. The difficulty in implementing this rule is that it requires automatically generating Verilog code *from label definitions* that correctly define the may-flow-to relation, as opposed to generating Z3 constraints *from Verilog code*. It is certainly viable to implement such a translation for a limited yet sufficiently expressive subset of Verilog expressions, but we leave it as future work. In our example processor implementations, we manually translate label comparison operations into Verilog expressions and use a *declassify* statement to explicitly label the result according to the LabelComp rule.

7 CASE STUDIES

In addition to our theoretical results, we used our implementation to empirically evaluate the utility of SpecVerilog through case studies. Table 1 provides a high-level summary of our case studies, the rewrite effort required to satisfy the type checker, and the mitigation techniques demonstrated by the example. Each of our case studies targets a component that is critical to the design of speculative, OoO processors. Some are known to contribute to transient execution vulnerabilities, while others were chosen to illustrate that SpecVerilog can still accept complex, yet safe, designs. For each of these case studies, we used the ROB-based labels described in Section 3.4 to label inputs and outputs associated with instructions.

[4]The implementation is a fork of Icarus Verilog and can be found at https://github.com/dz333/secverilog.

Case Study	Annotation Burden (Lines)			Register	Solver Time	Mitigation Strategy			SpecVerilog Features
	Labeled	Changed	Original	Overhead	(sec)	Delay	Rollback	Partition	Used
Reorder Buffer	22	5	86	None	0.247	–	✓	–	**LC**, **IA**, **VL**
Cache + GM	250	9	1527	None	1.43	✓	✓	✓	**LC**, **VL**
Predictor	16	0	68	None	0.104	✓	–	–	**LC** or **IA**
Predictor + GM	38	6	157	None	0.993	✓	✓	✓	**LC**, **VL**
Renaming RF	117	26	366	1 label/replica	36.9	✓	✓	–	**LC**, **IA**, **VL**

Table 1: A qualitative summary of secure hardware module case studies. Annotation burden is relative to a secure design in plain Verilog. Labeled counts the lines that needed explicit labeling. Changed includes modifications and additions needed to satisfy the type-checker. Original is the total lines of the original Verilog description. LC are label comparisons, IA are input assertions, VL are labels that leverage valid bit or validity logic to erase contents.

7.1 Case Study Modules

Reorder Buffer. We implement a reorder buffer skeleton to illustrate that our source of truth for labels can be implemented securely and with minimal assumptions about functional correctness. Our ROB supports insertion, misspeculation and instruction commitment; we leave the ROB entry format abstract for this example.

Discussion. In this module (and all of our case studies), in order to type-check, we had to assert that the oldest entry in the ROB would never be misspeculated; this follows directly from functional correctness and is essentially a minimal assumption. To implement erasure we used a dynamic label that only marked entries between the head and tail of the ROB as valid; this enabled us to type-check a normal ROB since re-setting the tail pointer upon misspeculation effectively "erased" misspeculated entries.

Cache. We implemented a blocking, direct-mapped L1 cache in SpecVerilog and labeled its interface so that all requests and responses were associated with some instruction. Responding to requests requires multiple cycles and so the cache maintains state associated with the current request. We also built a GhostMinion-like [1] module to store cache lines from speculative requests which were promoted to the original cache on instruction commit.

Discussion. Even with such a simple design, SpecVerilog forced us to implement essentially all of the mitigation mechanisms described in the original GhostMinion work: free-slotting, time-guarding, and leapfrogging. We did not need to add extra state or dynamic checks beyond those needed for the above mitigation techniques.

Branch Predictor. We built a standard 2-bit history predictor. To ensure this design was safe we had to insert a dynamic check that ignored speculative updates to its state. Alternatively, this check can be established as an assumed precondition on valid requests. In addition, we made a second version applying the GhostMinion methodology to allow speculative updates to predictor state.

Discussion. Branch predictors can be the targets of speculative fetch redirect vulnerabilities [32]; the usual method of defending against such attacks is delaying updates to predictor state. Our design represents this delay mitigation by forcing some dynamic check, either in the predictor or in an instantiating module.

Renaming Register File. Renaming register files are not typically vulnerable to transient execution vulnerabilities; however, they do mix speculative and committed state and their safety relies on invariants established by functional correctness. We modify an existing implementation so that it type-checks in SpecVerilog.

Discussion. This was the only module where SpecVerilog forced us to add unnecessary state and/or dynamic checks. We needed to add explicit labels for name file replicas (that are used to reset the architectural-to-physical name mappings upon misspeculation) which would require only hundreds of bits for a realistically sized implementation. Additionally, we had to change some of the logic that updated the list of free names; the original logic was safe due to invariants that could only be established with gate-level reasoning (e.g., a tool like GLIFT [43]). Lastly, we encoded some invariants about valid usage of the rename file as input assumptions to avoid inserting extra unnecessary dynamic checks.

7.2 Experience Report

While developing these case studies, we learned several key takeaways about designing secure hardware in SpecVerilog:

- SpecVerilog frequently forced us to fix potential vulnerabilities that we had missed. These bugs included both forgetting to invalidate misspeculated data or metadata and also incorrectly handling interactions between different speculative requests.
- We did usually have to syntactically alter designs for them to be accepted by SpecVerilog, although this often did not change their functionality or require extra state.
- Some designs are only secure under certain assumptions. These assumptions often follow from functional correctness, and could be verified separately, either using formal verification or by establishing the necessary invariants via another hardware module.

At a high level, many vulnerabilities we encountered were subtle, and it was not immediately obvious whether the preexisting code was insecure or whether SpecVerilog was incorrectly rejecting a design due to imprecision. Usually, there was a real security problem, which we often discovered by rewriting the relevant logic from a blank slate, guided by the SpecVerilog type checker; it was much easier to build a secure design than fix an insecure one.

Annotation Burden. Most of the required design effort is from explicitly defining and annotating labels. We wrote 21 lines of Z3 constraints to specify the definitions of dynamic labels and erasure

conditions across all example combined; these label definitions represent a one-time effort and can be reused in other designs. Neither SpecVerilog nor its predecessor, SecVerilog, have label inference, and therefore all registers and wires in the design must have their labels annotated by the programmer. While standard IFC inference algorithms [36] could be used to ease some of this burden, erasure labels and per-element labels [18] complicate this problem and would require further investigation.

The other primary change made to assist type checking was to translate some dynamic checks to use the "flows to" operator instead of an equivalent logical formulation. Typically, these changes did not alter functionality but allowed SpecVerilog to prove the safety of the design.

Register and Logic Overhead. We almost never needed to add extra registers to verify secure designs since the valid bits or instruction identifiers necessary to represent dynamic labels were usually necessary anyway to implement a secure design. However, in a few instances we did need to add redundant dynamic checks that could add overhead to the final designs. This redundancy was caused by imprecision in the static analysis used by SpecVerilog to prove relationships between dynamic labels; improving the precision of this analysis could enable removing these redundant checks.

Compile-Time Overhead. SpecVerilog imposed little compile-time overhead. All of our examples compile and type-check in less than one minute and most complete in less than one second, despite relying on an SMT solver. The vast majority of the compile time was spent in the solver and Table 1 shows those times for each case study. Even the most complex individual queries that relied on the Z3 theory of arrays took no more than ten seconds. When a design is insecure, Z3 provides a counterexample that violates the type checking constraints.

8 RELATED WORK

8.1 Architectural Mitigations

Since the discovery of Spectre and Meltdown, dozens of microarchitectural mitigation mechanisms have been proposed (e.g., [1, 2, 32, 37, 51, 53]). Xiong and Szefer [50] survey microarchitectural mitigation techniques. While some designs come with formal security conditions [32, 53], or even security proofs [54], none of these designs (to our knowledge) are formally checked for correctness at the RTL level. Most are implemented in the architecture simulator gem5 [6], which does not accurately capture timing behavior and does not describe synthesizable circuits.

SpecVerilog is an RTL-level language that can be used to implement and verify the security of many of these mitigation mechanisms. Most of these mitigations rely on a combination of delaying potentially leaky operations, rolling back speculative modifications, partitioning state, and taint tracking. SpecVerilog's information flow type system with erasure labels can be used to validate defenses using all of these techniques. However, some defenses leverage randomness [26, 48, 49]; SpecVerilog would likely consider them insecure since it cannot reason about probabilistic security.

Orthogonally, some security-centric architectures [19, 52, 57] include annotations that enable software to specify fine-grained protection of specific processor data. SpecVerilog could be used to check the security of speculative implementations of such ISAs.

8.2 Secure Hardware Design Tools

In this work, we extend SecVerilog [17, 18, 59], one of a few information flow type systems for secure RTL design [14, 19, 29, 30]. While some of the above allow dependent security labels, none of them have been used to defend speculative security conditions. In addition to type systems, there are a number of other static analysis tools for secure hardware design.

GLIFT [23, 43] tracks information flow at the gate level and leverages properties of boolean logic for high precision. RTLLIFT [4] applies similar techniques but improves verification performance by working at the RTL level. The high precision comes at the cost of scalability; these tools are not designed for or appropriate for verifying complex dynamic processor security properties, as Transient Noninterference is. Their taint-tracking approach might be used in conjunction with an IFC type system to improve precision when verifying low-level bit manipulations.

Clepsydra [3] and Xenon [44] are designed to check RTL designs for timing side channels. Clepsydra verifies coarse-grained timing security polices such as constant-time execution and timing isolation. Xenon, an interactive tool for verifying constant-time execution, scales to more complex circuits, including in-order processors. Neither tool has been applied to security of transient execution.

The only static analysis tool (to our knowledge) that can soundly verify processor speculation security properties is that of Fadiheh et al. [16], based on Unique Program Execution Checking. Unlike SpecVerilog, their tool can require significant manual proof effort from the user and also does not check these proofs for correctness, creating a large surface area for bugs.

8.3 Information Flow Erasure Policies

Information flow erasure policies [10] were originally defined as part of a type system for software. However, Chong and Myers assume that a run-time monitor enforces erasure policies via dynamic clearing. Hunt and Sands [24] describe a type system that statically enforces erasure, although when to erase data is defined syntactically via scope rather than semantically. Stewart et al. [41] use dependent types to support erasure within data structures.

Our erasure labels expand upon these systems; they incorporate semantic erasure conditions that specify *when* data should no longer be used, are statically enforceable, and leverage dependent types to mix and reuse state across security levels.

8.4 Speculation-Secure Software

Other efforts focus on verifying the security of *software* given various speculative hardware semantics [8, 9, 21, 22, 34, 47]. These all adopt abstract processor semantics and leakage models. However, unlike the semantics we utilize in Section 5.2, they often rely on specific assumptions about processor behavior and do not incorporate time explicitly into their attacker models. This is a reasonable choice for these tools since the timing and speculative behavior of processors is unspecified by the ISA. However, explicitly timing-sensitive guarantees like Transient Noninterference and UPE provide more complete security and we believe should be the gold standard for

secure processor implementations. Guarnieri et al. [22] have proposed hardware–software contracts to bridge this gap; Transient Noninterference prevents leakage of architecturally accessed state, corresponding to their $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ contract.

9 DISCUSSION & CONCLUSION

SpecVerilog enables the verification of speculative security guarantees of RTL designs via the incorporation of erasure labels into an IFC type system. Here, we present a final discussion of some of the benefits, potential, and limitations of SpecVerilog with respect to secure processor design and verification.

Processor Verification. Verification of Transient Noninterference or similar conditions fundamentally requires reasoning about functional correctness, since speculation is defined relative to the ISA-specified behavior. SpecVerilog provides a clean divide between functional verification and security analysis via erasure conditions. Erasure conditions abstract *when* misspeculation occurs without having to reason about *why* it occurs. In this way, traditional processor verification techniques can be used to prove the assumptions needed by erasure labels (such as "the oldest instruction always commits") while SpecVerilog handles vulnerability checking. Alternatively, high-level HDLs, such as PDL [56], could be used in tandem with SpecVerilog to provide an end-to-end guarantee of both functional correctness and speculative security.

Generalizing Speculation. We have described a single methodology for OoO processor labeling and speculation, but SpecVerilog is not limited to next-instruction prediction or to the ROB labels we chose. SpecVerilog can be used for any microarchitecture as long as each discrete speculation site is labeled with a corresponding erasure condition. For example, SpecVerilog can be used to check processors that incorporate multiple concurrent sources of speculation, such as both branch and value prediction. However, we have yet to implement such a design. Furthermore, labels could be assigned *per branch* rather than *per instruction* to achieve more precise reasoning about potential vulnerabilities. One key challenge of this approach is reasoning about the separation between "front-end" speculation that applies to every instruction fetch, and branch speculation that only applies to some.

Erasure Expressivity. Modern processors do not necessarily propagate control signals globally in a single cycle, due to latency and power constraints. Additionally, some misspeculation clean-up implementations take multiple cycles and thus do not satisfy our definition of end-to-end erasure. SpecVerilog cannot represent the propagation or resolution of delayed misspeculation: erasure must happen synchronously and immediately. To support these feature, we believe SpecVerilog could incorporate explicit temporal logic operators (e.g., "next") into erasure conditions.

10 ACKNOWLEDGMENTS

This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] Sam Ainsworth. 2021. GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. Association for Computing Machinery. https://doi.org/10.1145/3466752.3480074

[2] Sam Ainsworth and Timothy M Jones. 2020. Muontrap: Preventing cross-domain Spectre-like attacks by capturing speculative state. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*.

[3] Armaiti Ardeshiricham, Wei Hu, and Ryan Kastner. 2017. Clepsydra: Modeling timing flows in hardware designs. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. https://doi.org/10.1109/ICCAD.2017.8203772

[4] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE.

[5] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, et al. 2021. Speculative interference attacks: Breaking invisible speculation schemes. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* (2011).

[7] Chandler Carruth. 2018. RFC: Speculative load hardening (a Spectre variant 1 mitigation). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html.

[8] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new Spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[9] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. https://doi.org/10.1109/CSF.2019.00027

[10] S. Chong and A.C. Myers. 2005. Language-based information erasure. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. https://doi.org/10.1109/CSFW.2005.19

[11] Stephen Chong and Andrew C. Myers. 2005. Language-Based Information Erasure. In *18th IEEE Computer Security Foundations Workshop (CSFW)*. 241–254. http://www.cs.cornell.edu/andru/papers/erasure.pdf

[12] Stephen Chong and Andrew C. Myers. 2008. End-to-End Enforcement of Erasure and Declassification. In *IEEE Computer Security Foundations Symp. (CSF)*. 98–111. http://www.cs.cornell.edu/andru/papers/enferasure-csf08.pdf

[13] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary). Springer-Verlag, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[14] Shuwen Deng, Doğuhan Gümüşoğlu, Wenjie Xiong, Sercan Sari, Y Serhan Gener, Corine Lu, Onur Demir, and Jakub Szefer. 2019. SecChisel framework for security verification of secure processor architectures. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*.

[15] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Comm. of the ACM* 19, 5 (1976), 236–243. https://dl.acm.org/citation.cfm?id=360056

[16] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Müller, Jörg Bormann, Sayak Ray, Jason M. Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2023. An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors. *IEEE Trans. Comput.* (2023). https://doi.org/10.1109/TC.2022.3152666

[17] Andrew Ferraiuolo, Weizhe Hua, Andrew C. Myers, and G. Edward Suh. 2017. Secure Information Flow Verification with Mutable Dependent Types. In *Design Automation Conference (DAC)*.

[18] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2017. Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[19] Andrew Ferraiuolo, Mark Zhao, Andrew C Myers, and G Edward Suh. 2018. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[20] Jacob Fustos, Michael Bechtel, and Heechul Yun. 2020. SpectreRewind: Leaking secrets to past instructions. In *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*.

[21] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[22] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1868–1883. https://doi.org/10.1109/SP40001.2021.00036

[23] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. 2012. On the complexity of generating gate level information flow tracking logic. *Transactions on Information Forensics and Security* (2012).

[24] Sebastian Hunt and David Sands. 2008. Just forget it—the semantics and enforcement of information erasure. In *European Symposium on Programming*.

[25] Zhenghong Jiang, Hanchen Jin, G Edward Suh, and Zhiru Zhang. 2019. Designing secure cryptographic accelerators with information flow enforcement: A case study on AES. In *Proceedings of the 56th Annual Design Automation Conference*.

[26] Georgios Keramidas, Alexandros Antonopoulos, Dimitrios N Serpanos, and Stefanos Kaxiras. 2008. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems* 12 (2008).

[27] Elisavet Kozyri, Fred B Schneider, Andrew Bedford, Josée Desharnais, and Nadia Tawbi. 2019. Beyond labels: Permissiveness for dynamic information flow enforcement. In *32nd Computer Security Foundations Symposium (CSF)*. IEEE.

[28] Peng Li, Yun Mao, and Steve Zdancewic. 2003. Information Integrity Policies. In *Workshop on Formal Aspects in Security and Trust (FAST)*.

[29] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. 2014. Sapper: A Language for Hardware-level Security Policy Enforcement. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[30] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: A Hardware Description Language for Secure Information Flowx. In *Conference on Programming Language Design and Implementation (PLDI)*.

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[32] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. DOLMA: Securing Speculation with the Principle of Transient Non-Observability. In *30th USENIX Security Symposium (USENIX Security 21)*.

[33] ARM Ltd. 2009. ARM Security Technology: Building a Secure System using TrustZone Technology.

[34] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. 2022. Axiomatic hardware-software contracts for security. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*.

[35] Lars Müller. 2018. KPTI a Mitigation Method Against Meltdown. *Advanced Microkernel Operating Systems* (2018).

[36] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *26th ACM Symp. on Principles of Programming Languages (POPL)*. 228–241. https://doi.org/10.1145/292540.292561

[37] Gururaj Saileshwar and Moinuddin K Qureshi. 2019. Cleanupspec: An "undo" approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*.

[38] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction. In *Proceedings of the 46th International Symposium on Computer Architecture*.

[39] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. ConTExT: A Generic Approach for Mitigating Spectre. In *NDSS*.

[40] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. 2016. SPECTRE: A Fast and Scalable Cryptocurrency Protocol. Cryptology ePrint Archive, Report 2016/1159. https://eprint.iacr.org/2016/1159

[41] Gordon Stewart, Anindya Banerjee, and Aleksandar Nanevski. 2013. Dependent Types for Enforcement of Information Flow and Erasure Policies in Heterogeneous Data Structures. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*. https://doi.org/10.1145/2505879.2505895

[42] Mohit Tiwari, Jason K Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. 2011. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. *ACM SIGARCH Computer Architecture News* (2011).

[43] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. 2009. Complete information flow tracking from the gates up. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*.

[44] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. 2021. Solver-Aided Constant-Time Hardware Verification. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*.

[45] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings for the 27th USENIX Security Symposium*.

[46] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[47] Marco Vassena, Craig Disselkoen, Klaus V Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2020. Automatically eliminating speculative leaks from cryptographic code with Blade. *arXiv preprint arXiv:2005.00294* (2020).

[48] Zhenghong Wang and Ruby B Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*.

[49] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization.. In *USENIX Security Symposium*.

[50] Wenjie Xiong and Jakub Szefer. 2021. Survey of Transient Execution Attacks and Their Mitigations. *ACM Comput. Surv.* (2021). https://doi.org/10.1145/3442479

[51] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[52] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. 2018. Data oblivious ISA extensions for side channel-resistant and high performance computing. *Cryptology ePrint Archive* (2018).

[53] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. 2019. Speculative taint tracking (STT): a comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*.

[54] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. 2019. Speculative taint tracking (STT): A formal analysis. *Univ. of Illinois at Urbana-Champaign and Tel Aviv Univ., Tech. Rep* (2019).

[55] Drew Zagieboylo. 2023. https://github.com/dz333/secverilog.

[56] Drew Zagieboylo, Charles Sherk, G. Edward Suh, and Andrew C. Myers. 2022. PDL: A High-level Hardware Design Language for Pipelined Processors. In *43rd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*.

[57] Drew Zagieboylo, G Edward Suh, and Andrew C Myers. 2019. Using information flow to design an ISA that controls timing channels. In *IEEE 32nd Computer Security Foundations Symposium (CSF)*.

[58] Steve Zdancewic and Andrew C. Myers. 2001. Robust Declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW)* (Cape Breton, Nova Scotia, Canada). 15–23. https://doi.org/10.1109/CSFW.2001.930133

[59] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 503–516. http://www.cs.cornell.edu/andru/papers/asplos15

[60] Lantian Zheng and Andrew C. Myers. 2007. Dynamic Security Labels and Static Information Flow Control. *International Journal of Information Security* 6, 2–3 (March 2007). http://www.cs.cornell.edu/andru/papers/dynlabel-ijis.pdf

A DEFINITIONS

Definition 7 (Erase-To Function). *Each label implies an erasure policy. Each erasure policy is a list of condition and label pairs that denote under what circumstances data must be erased and to what level. This function defines the erasure policy for each label.*

$$\boxed{\text{eraseTo}(\tau) \dashv L}$$

$$\textit{LABEL}\ \frac{}{\text{eraseTo}(l) \dashv [\,]}$$

$$\textit{FUNCTION}\ \frac{}{\text{eraseTo}(f(x)) \dashv [\,]}$$

$$\textit{ERASE}\ \frac{}{\text{eraseTo}(b_1\ {}^{c(\vec{x})}\!\nearrow b_2) \dashv [(c(\vec{x}), b_2)]}$$

$$\textit{JOIN}\ \frac{\forall (c_l, l_l) \in \text{eraseTo}(\tau_1).\ \forall (c_r, l_r) \in \text{eraseTo}(\tau_2).\quad (c_l, l_l \sqcup \tau_2) \in L \qquad (c_r, l_r \sqcup \tau_1) \in L \qquad (c_l \wedge c_r, l_l \sqcup l_r) \in L}{\text{eraseTo}(\tau_1 \sqcup \tau_2) \dashv L}$$

$$\textit{MEET}\ \frac{\forall (c_l, l_l) \in \text{eraseTo}(\tau_1).\ \forall (c_r, l_r) \in \text{eraseTo}(\tau_2).\quad (c_l, l_l \sqcap \tau_2) \in L \quad (c_r, l_r \sqcap \tau_1) \in L \quad (c_l \wedge c_r, l_l \sqcap l_r) \in L}{\text{eraseTo}(\tau_1 \sqcap \tau_2) \dashv L}$$

$$\boxed{\mathcal{P} = \langle rob, A, \mu \rangle}$$

$$\textsc{Clock}\ \frac{}{\mathcal{P} \xrightarrow[\text{clock}]{A} \mathcal{P}} \qquad \textsc{Fetch}\ \frac{rob = \_ :: x \qquad rob' = rob :: x' \quad x' = \text{predict}(\text{arch}(x, A), \mu) \quad \mu' = \text{update}(\text{arch}(x, A), \mu)}{\mathcal{P} \xrightarrow[\text{fetch } x]{} \mathcal{P}'}$$

$$\textsc{Exec}\ \frac{rob = \_ :: x :: \_ \quad \mu' = \text{compute}(\text{arch}(x, A), \mu)}{\mathcal{P} \xrightarrow[\text{exec } x]{} \mathcal{P}'}$$

$$\textsc{Commit}\ \frac{rob = x :: rob' \quad A' = \text{commit}(\text{arch}(x, A), \mu)}{\mathcal{P} \xrightarrow[\text{commit } x]{} \mathcal{P}'}$$

$$\textsc{Miss}\ \frac{rob = rob'' :: x :: \_ \quad rob' = rob'' :: x :: x' \qquad x' = \text{real-npc}(\text{arch}(x, A), \mu) \quad \mu' = \text{miss-upd}(\text{arch}(x, A), \mu)}{\mathcal{P} \xrightarrow[\text{miss } x]{} \mathcal{P}'}$$

$$\textsc{Schedule}\ \frac{d = \text{sched}(\text{arch}(rob, A), \mu) \qquad \mathcal{P} \xrightarrow[d]{} \mathcal{P}'}{\mathcal{P} \to \mathcal{P}'}$$

Figure 9: Semantics for Abstract Out-of-Order Processors

$$\boxed{\tau \sqsubseteq \tau'} \qquad \textsc{Reflexive}\ \frac{}{\tau \sqsubseteq \tau}$$

$$\textsc{Labels}\ \frac{l_1 \sqsubseteq_l l_2}{l_1 \sqsubseteq l_2} \qquad \textsc{Transitive}\ \frac{\tau \sqsubseteq \tau'' \qquad \tau'' \sqsubseteq \tau'}{\tau \sqsubseteq \tau'}$$

$$\textsc{Join-Intro}\ \frac{\tau \sqsubseteq \tau_1 \vee \tau \sqsubseteq \tau_2}{\tau \sqsubseteq \tau_1 \sqcup \tau_2} \qquad \textsc{Join-Elim}\ \frac{\tau_1 \sqsubseteq \tau \qquad \tau_2 \sqsubseteq \tau}{\tau_1 \sqcup \tau_2 \sqsubseteq \tau}$$

$$\textsc{Meet-Intro}\ \frac{\tau \sqsubseteq \tau_1 \qquad \tau \sqsubseteq \tau_2}{\tau \sqsubseteq \tau_1 \sqcap \tau_2} \qquad \textsc{Meet-Elim}\ \frac{\tau_1 \sqsubseteq \tau \vee \tau_2 \sqsubseteq \tau}{\tau_1 \sqcap \tau_2 \sqsubseteq \tau}$$

$$\textsc{Erase-Elim}\ \frac{\tau_2 \sqsubseteq \tau'}{\tau_1\ {}^{c(\vec{v})}\!\not\nearrow \tau_2 \sqsubseteq \tau'} \qquad \textsc{Erase-Intro}\ \frac{\tau \sqsubseteq \tau_1}{\tau \sqsubseteq \tau_1\ {}^{c(\vec{v})}\!\not\nearrow \tau_2}$$

$$\textsc{Erase-Weaken}\ \frac{\tau_1 \sqsubseteq \tau_1' \quad \tau_2 \sqsubseteq \tau_2' \qquad \forall \sigma. \sigma \vDash c(\vec{v}) \implies \sigma \vDash c'(\vec{v}')}{\tau_1\ {}^{c(\vec{v})}\!\not\nearrow \tau_2 \sqsubseteq \tau_1'\ {}^{c'(\vec{v}')}\!\not\nearrow \tau_2'}$$

Figure 10: The complete environment-independent may-flow-to relation. The ordering relation of the lattice of basic security levels is $\sqsubseteq_l$.

B PROOFS

Theorem 2 (Safety of Ordered Label Comparisons). *The result of any label comparison (on ordered labels) is guaranteed to be low-equivalent at or above the meet of the labels of the labels.*

$$\forall l. x, y \in \mathit{VARS}.$$
$$\sigma_1 \approx_l \sigma_2 \ \wedge\ (\forall \sigma. \Gamma(x)\ {}_\sigma\!\sqsubseteq \Gamma(y) \vee \Gamma(y)\ {}_\sigma\!\sqsubseteq \Gamma(x)) \ \wedge\ \Gamma(x) \sqcap \Gamma(y)\ {}_{\sigma_1}\!\sqsubseteq l \implies \sigma_1[x] \sqsubseteq \sigma_1[y] = \sigma_2[x] \sqsubseteq \sigma_2[y]$$

Proof. By the definition of observational equivalence, both σ_1 and σ_2 agree on whether $\Gamma(x)$ and $\Gamma(y)$ are in the high ($\mathcal{H}$) or low ($\mathcal{L}$) sets: any label which flows to l is in the low set, everything else is in the high set. There are four possible cases which correspond to the sets that contain $\Gamma(x)$ and $\Gamma(y)$ respectively:

$\mathcal{L}, \mathcal{L}$. In this case, by observational equivalence, $\sigma_1[x] = \sigma_2[x]$ and $\sigma_1[y] = \sigma_2[y]$, so the result of computing on them is equal.

$\mathcal{L}, \mathcal{H}$. In this case, $\Gamma(y)\ {}_\sigma\!\not\sqsubseteq \Gamma(x)$ by the definition of $\mathcal{L}$. Therefore, the expression must return true in both σ_i since either $\Gamma(y)\ {}_\sigma\!\sqsubseteq \Gamma(x)$ or $\Gamma(x)\ {}_\sigma\!\sqsubseteq \Gamma(y)$ by the ordering assumption.

$\mathcal{H}, \mathcal{L}$. This is symmetric to the prior case, except the expression must return false.

$\mathcal{H}, \mathcal{H}$. In this case, we cannot be sure that the result is equivalent in both σ_i since there are no equality constraints on x or y. However, $\Gamma(x) \sqcap \Gamma(y)$ must be exactly equal to either $\Gamma(x)$ or $\Gamma(y)$ since they are ordered; therefore $\Gamma(x) \sqcap \Gamma(y) \in \mathcal{H}$ (i.e., $\Gamma(x) \sqcap \Gamma(y)\ {}_\sigma\!\not\sqsubseteq l$). □

Lemma 1. *For any register x, in a well-typed $\mathcal{P}$ ($\preceq$ is prefix order),*

$$\forall i_j^{[s_1,\ldots,s_n]}, i_k^{[p_1,\ldots,p_n]} \in \mathcal{I}(x), j \le k \implies [s_1, \ldots s_n] \preceq [p_1, \ldots, p_n]$$

Proof.

Base Case: At first, this vacuously holds since no state is yet influenced by an instruction.

Fetch: Each newly fetched instruction has a larger instruction index than any prior instruction, and either keeps the same speculative path or adds a new prediction:

$$\frac{pc \approx i_j^{[s_1,\ldots,s_n]} \qquad \mathcal{I}(pc') = \mathcal{I}(pc) + i_{j+1}^{[s_1,\ldots,s_n]} \ \vee\ \mathcal{I}(pc') = \mathcal{I}(pc) + i_{j+1}^{[s_1,\ldots,s_n,s_{n+1}]}}{\text{fetch } pc}$$

In the above, $pc \approx i_j^{[s_1,\ldots,s_n]}$ denotes the logical instruction associated with the current instruction address pc.

Exec & Commit: Interim execution cannot introduce *new* speculative paths or instructions and thus cannot affect this invariant.

Miss: The miss case is similar to the *Fetch* case: the new instruction x' has the same influence set as instruction x, with a higher instruction index and *does not extend* x's speculative path.

In this case, μ and the ROB may now contain registers influenced by x' and instructions along the misspeculated path, violating the invariant. However, end-to-end erasure effectively allows us to remove the influence of erased instructions from influence sets (since their values have no impact on future execution).

All instructions ordered after x in ROB order (i.e., $j \le k$) are erased since they must all have labels l such that $SL(x, head)\ {}_\sigma\!\sqsubseteq l$: exactly the set of instructions whose speculation paths are *not* prefixes of instruction x's. After this influence removal, all register influence sets contain only instructions that satisfy the invariant. □