Using Information Flow to Design an ISA that Controls Timing Channels

Drew Zagieboylo
Department of Computer Science
Cornell University
dzag@cs.cornell.edu

G. Edward Suh
Department of Electrical and Computer Engineering
Cornell University
suh@ece.cornell.edu

Andrew C. Myers
Department of Computer Science
Cornell University
andru@cs.cornell.edu

Abstract

Information-flow control (IFC) enforcing languages can provide high assurance that software does not leak information or allow an attacker to influence critical systems. IFC hardware description languages have also been used to design secure circuits that eliminate timing channels. However, there remains a gap between IFC hardware and software; these two components are built independently with no abstraction for how to compose their security guarantees. This paper presents a proposal for an instruction set architecture (ISA) that can provide the appropriate abstraction for joining hardware and software IFC mechanisms. Our ISA describes a RISC-V processor that tracks information-flow labels at run time and uses these labels to eliminate or mitigate timing channels. To make the ISA more practical, it allows constrained downgrading of information; it permits trading off security for performance; and still offers control primitives such as system calls. We prove timing-sensitive noninterference modulo downgrading and nonmalleability for programs executing our ISA. This involves novel restrictions on the mutability of labels beyond previous dynamic IFC systems. Furthermore, we define specific security conditions which correct hardware can implement to provide software-level security and sketch how such hardware may be designed and verified.

1 Introduction

While timing channels have been well known to the security community for decades, recent hardware-based exploits attest that these vulnerabilities remain unsolved problems. For example, the Spectre, Meltdown, and Foreshadow attacks allow unprivileged processes to learn secrets by timing memory accesses [1–3]. The sophisticated security mechanisms provided by these modern processors—privilege rings, memory management units, and software guard extensions [4]—are completely undermined by uncontrolled timing behaviors. Current processors are not timing-safe.

The hardware-security community has investigated how to eliminate timing channels from circuit implementations, but these are not panaceas. Hardware description languages (HDLs) such as SecVerilog [5] and Caisson [6] provide timing-sensitive noninterference. They ensure that the time at which "public" state is updated does not depend on any "secret" state. While they do provide useful primitives for implementing secure processors, these languages are not sufficient for executing timing-safe software in a real-world setting. They can preclude necessary operations (such as modifying security labels at run time) and limit software's ability to specify security policies by baking those policies into the hardware. In practice, software needs the ability to make application-level policy decisions while still benefiting from the timing-sensitive guarantees of security-focused HDLs. On the other hand, more complex instantiations of secure processors lack proofs that their ISAs enforce a meaningful security condition. The Hyperflow processor [7], for instance, allows bounded software modification of the "context label", but no ISA-level security condition gives guidance on how safe this is.

Software attempts to eliminate timing channels have had some success but ultimately are not comprehensive, instead targeting empirically known sources of timing variation. For example, compilers for cryptographic computation [8–10] help to mitigate side channels but are fundamentally incomplete, since they only model well known sources of timing variation such as branching and caching. To fully remove timing channels, a new interface is needed to constrain how hardware state influences timing and which software instructions might leak information [11, 12].

The missing link between these hardware and software approaches is an Instruction Set Architecture (ISA) with an explicit abstraction for the influence of the machine state on timing. With such an ISA, strong timing-sensitive security conditions could be proved about software, relying on the guarantees made by hardware.

As a straw man, a software–hardware contract might ensure that all instructions with secret operands execute in constant time. In fact, existing techniques for securely implementing cryptography implicitly assume such a contract. However, constant time inevitably means worst-case time, in general, so such a contract has daunting implications for the performance of memory operations. We argue that this kind of contract is unnecessarily restrictive. It is not necessary that such instructions take constant time; it is only necessary that the time taken does not leak information.

This paper presents an ISA design that can be the interface connecting high-level timing-sensitive software abstractions to low-level timing-safe processor implementations. Our ISA is based on information flow control (IFC), which means our software–hardware contract is a set of IFC properties, rather than a prescriptive set of implementation behaviors such as forcing certain instructions to take constant time. Because the interface is based on IFC, it is possible to formally prove that only permitted information affects timing.

Our ISA design includes features to avoid being overly restrictive, as IFC systems often are [13]. To this end, it includes downgrading operations that allow software to endorse untrusted inputs and to declassify secret data. We also allow software to specify its own timing security policy, which permits trading off timing-channel protection for performance. Both of these features are limited so that they cannot be abused by attackers to undermine the security guarantees of well-behaved programs. We additionally include security primitives that are required to implement a practical operating system. These instructions are analogous to traditional *system calls*, but they are designed to prevent unexpected information leakage.

The ISA in this paper tackles these goals with novel constructs and stronger formal security assurance:

- The ISA dynamically enforces timing-sensitive nonmalleable information flow [14], while also preventing implicit flows created by checking mutable labels.
- The ISA allows software to control the level of timing-channel protection. The ISA can be used to eliminate timing channels, mitigate timing channels with bounded information leak using predictive mitigation [11], or enforce nonmalleable information flow control without timing channel protection.
- The ISA also includes novel instructions for implementing privilege changes to emulate the functionality of system calls while maintaining nonmalleability.
- The ISA is accompanied by formal, proved security guarantees for programs implemented with it.
- We also formally specify security conditions with which hardware implementations must comply to ensure security of the ISA.

The paper proceeds as follows. Section 2 presents background on security labels and our attacker model. Section 3 sketches our approach to controlling timing channels. Sections 4 and 5 formalize the ISA and discuss its novel features in detail. In Section 6, we discuss the security conditions assumed of the hardware and the practical challenges in realizing those policies with modern HDLs. Section 7 presents the security results for this ISA and brief sketches of their proofs. Section 8 uses example code to demonstrate use of the ISA. In Section 9 we discuss related work and we discuss future work in Section 10.

2 Background

Our ISA both extends the RISC-V ISA[1] [15] with new instructions and modifies the semantics of existing instructions. RISC-V has instructions for computing on data, moving data to and from memory, and for changing program control

[1]Our approach is not specific to RISC-V and could be adapted for use in other instruction sets.

$$
\begin{aligned}
&(i,c)^{\rightarrow} \triangleq c\\
&(i,c)^{\leftarrow} \triangleq i\\
&l_1 \sqsubseteq l_2 \overset{\triangle}{\Leftrightarrow} (l_1^{\leftarrow} \sqsubseteq l_2^{\leftarrow}) \wedge (l_2^{\rightarrow} \sqsubseteq l_1^{\rightarrow})\\
&l_1 \sqcup l_2 \triangleq ((l_2^{\rightarrow} \sqcup l_2^{\rightarrow}), (l_1^{\leftarrow} \sqcap l_2^{\leftarrow}))\\
&l_1 \sqcap l_2 \triangleq ((l_2^{\rightarrow} \sqcap l_2^{\rightarrow}), (l_1^{\leftarrow} \sqcup l_2^{\leftarrow}))\\
&⧖(i,c) \triangleq (c,i)
\end{aligned}
$$

Figure 1: Security lattice operators

flow. *Architectural* state refers to any storage location that is explicitly accessible or modifiable by software, including the 32 general-purpose registers, the program counter and all memory locations. Our extension modifies all architectural state to be associated with a security label. All other hardware state is considered *microarchitectural* and affects only the performance of software but not its functional behavior.

The complete RISC-V ISA has many Control Status Registers (CSRs) which are considered architectural, but for brevity we omit most of them from our formalization. These CSRs should in principle also each have their own security labels.

2.1 Security Labels

As in most IFC systems, our security labels form a lattice that supports a "flows to" relation $\sqsubseteq$, a lattice join $\sqcup$ and a lattice meet $\sqcap$. We use the phrase "more restrictive" to refer to labels higher in the lattice ordering (e.g., $a \sqsubseteq b$ means "b is at least as restrictive as a"). Figure 1 defines useful and mostly standard notation for label reference and manipulation. The label lattice is a product of two other lattices, one for integrity (trustworthiness of data) and one for confidentiality (secrecy of data), so a lattice element is a pair (i, c). For generality, we represent the two component lattices abstractly, but we restrict them to be dual lattices over the same carrier set. That is, the ordering $\sqsubseteq$ is reversed for the integrity and confidentiality components of the label lattice. The *reflection operator* $⧖$, used for controlled downgrading, swaps the two components of a lattice element.

An illustrative instantiation of this lattice is for the component lattice elements to represent principals. For instance, component b could represent both *Bob's* integrity (data written by Bob) and *Bob's* confidentiality (data readable by Bob), where *Bob* is a user of the system. Bob's data can flow to anywhere that has a label at least as confidential and no more trusted than b. Suppose there is a principal $\top$ that is least in the integrity ordering (meaning that it is trusted by everyone) and greatest in the confidentiality ordering; conversely, $\bot$ is highest in the integrity ordering (meaning that it is untrusted) and least in confidentiality. Then data labeled $(\top, b)$ flows to the label $(b, \top)$ because in integrity we have $\top \sqsubseteq b$ and in confidentiality, $b \sqsubseteq \top$.

2.2 Downgrading

Downgrading is the act of lowering the label of data in the lattice, violating the normal direction of information flow expressed by the lattice ordering. While downgrading greatly improves expressibility, it is important to constrain it, so that an attacker cannot leverage the downgrading mechanism to extract more secrets or modify more trusted state than the application developer intended. Our ISA enforces nonmalleability, a form of constrained downgrading, defined by Cecchetti et al. [14]. Nonmalleability guarantees both *robust declassification* and its dual *transparent endorsement*, which respectively constrain the downgrading of confidentiality and integrity.

We define *compromised* labels to represent exactly the set of labels that can never be safely downgraded under nonmalleability.

Definition 1 (Compromised Labels)**.** *A label is compromised if it is not as trusted as it is secret:*

$$l \not\sqsubseteq ⧖(l)$$

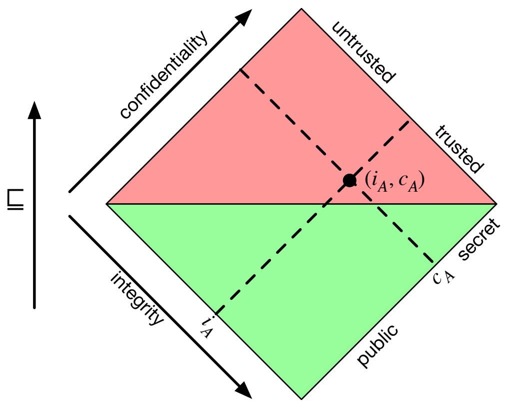

Figure 2: A 2-D slice of the combined confidentiality and integrity lattice. The red section represents all compromised labels. The dotted lines represent valid boundaries specifying a particular attacker model and dividing the lattice into quadrants. The intersection of these lines must be a *compromised* label, but need not be the same in each component lattice.

Intuitively, compromised data contains secret information but has been modified by an attacker or other low-integrity source. Allowing such data to be downgraded opens up the possibility of "confused deputy" style attacks, where trusted code that executes downgrades can be tricked into downgrading arbitrary data.

2.3 Attackers

We represent attackers by the maximal integrity i_A with which they can act and a minimal confidentiality c_A that they cannot observe. This is equivalent to typical attacker definitions which use a *maximal confidentiality* c_M the attacker can observe. Since we assume a finite lattice, we can translate c_M to c_A as follows:

$$L_s = \{l \mid l \not\sqsubseteq c_M\}$$

$$c_A \equiv \bigvee_{l_s \in L_s} l_s$$

c_A represents the disjunction of all labels which c_M is not allowed to read, and therefore defines the minimal confidentiality that they cannot observe.

It is convenient to summarize the attacker as a single label $A = (i_A, c_A)$. As depicted in Figure 2, the components c_A and i_A define upward-closed sets of secret and untrusted labels:

$$\mathcal{S} = \{l \mid c_A \sqsubseteq l\}$$
$$\mathcal{U} = \{l \mid i_A \sqsubseteq l\}$$

The sets of public ($\mathcal{P}$) and trusted ($\mathcal{T}$) labels are simply any labels not in $\mathcal{S}$ or $\mathcal{U}$, respectively. Attackers can only read public data and can only write to untrusted data.

Fair Attacks. Similar to prior work on robust declassification [16], our security guarantees hold against *fair attacks*, where high-secrecy and high-integrity information are only protected from attackers that do not already know those secrets or are not already highly trusted. In this work, *fair attacks* are defined as those where A represents a compromised label:

Definition 2 (Fair Attacker). *Attacker $A = (i_A, c_A)$ is a* fair attacker *if and only if A is a* compromised label.

```
# s0: secret int, a0: public int[], a1: public int
add s1, a0, s0      # s1 = &(a0[s0])
lw s2, 0(s1)        # s2 = *s1
lw a1, 0(a0)        # a1 = a0[0]
```

Figure 3: Meltdown-style timing channel via microarchitectural state

Since a given attacker may be partly trusted with respect to integrity and confidentiality, the label A is not a fixed, known label. Rather, we consider the system to be secure if it is secure against all possible fair attackers A.

Our earlier *Bob* example can illustrate why this definition eliminates unfair attackers. In a security lattice including the orderings $(\top, \bot) \sqsubseteq (b, b) \sqsubseteq (\bot, \top)$, consider the attacker with *Bob's integrity* who is only allowed to read fully public data: $A = (b, b)$.[2] A is *not* a fair attacker: it is as trusted as *Bob* (and can therefore impersonate him) but is not supposed to learn any of *Bob's* secrets. Essentially, this A would model *Bob* attacking himself. Our security condition does not prevent *Bob* from mistakenly releasing his own data to the public; it prevents untrusted attackers from doing so and from manipulating *Bob* into doing so for them.

Other Assumptions. We assume a strong attacker that may observe the wall-clock time at which writes to public locations occur, and not just the ordering of writes. This observational power corresponds to a colocated attacker-controlled process that can race on memory accesses and has access to wall-clock time. Defending against such a strong attacker is preferable since it makes the security assurance correspondingly stronger.

Since our ISA implements a dynamic IFC system, attackers can observe the labels of data through the success or failure of run-time checks [17]. For example, if secret (S) is used (either directly or implicitly through branching) to label another piece of data (D) as secret, then an attacker may learn information about S when their attempt to read D fails. The ISA does not include instructions for explicitly reading labels and therefore we assume attackers cannot directly read label values.

3 Controlling Timing Channels

Here we present high-level examples of where timing channels arise and how we approach mitigating them. Figure 3 contains RISC-V code with a simple microarchitectural timing channel: a secret-dependent load causing cache interference. In this example, `s0` is a secret value; `a0` and `a1` are public information. In modern processors, `lw` ("load word") is not a constant-time operation; its duration depends primarily on the address being accessed and other microarchitectural state (notably the cache). In this case, the address depends on `s0`, a secret offset into array `a0`. Loading the data at address `s1` also causes some region of the `a0` array to be placed in cache. If this region happens to be close to the beginning of the array, the second `lw` experiences a cache hit and executes quickly. In this way, an attacker who can observe how long it takes to load public information learns some secret information. This vulnerability reflects the core information transfer mechanism of the Meltdown attack [2].

In our ISA, software specifies a timing label, an upper bound on what information may influence instruction completion timing. If the program in Figure 3 executed with a secret timing label, then it would have the same unsatisfactory timing guarantees as current software. However, if the timing label were set to public, then only public information could influence how long any instruction took and the latency of the second `lw` will not reveal any information about `s0`. Obviously, software running at a low timing label may not benefit from all possible performance optimizations, but it does not necessarily require hardware to take worst-case time.

Figure 4 represents a different kind of timing channel, where an attacker can determine information about secrets by observing how long secret-dependent operations take. In this example, the attacker primes the cache by loading a public value, `l1`. Then, by observing when `l0` is updated, they can infer whether or not the memory read operation in between was a cache hit or miss. A hit implies that the `true` branch was taken, since `l1` was already cached.

[2]Note that this label is *not compromised* since $(b, b) \sqsubseteq (b, b)$

```
# l0,l1,l2: public int
# h1,h2:    secret-trusted int
# secret:   secret-trusted boolean
l0 = l1
if (secret): h1 = l1; else: h1 = l2;
l0 = 1
```

Figure 4: Untrusted inputs causing secrets to leak via timing

Table 1: Modified Semantics for Standard RISC-V Instructions

Insn Type	**Restrictions**	**Behavior**
COMPUTE	$pc_l \sqcup L(r_{s1}) \sqcup L(r_{s2}) \sqsubseteq L(r_d)$	$M' = M[r_d \mapsto R_{s1} \otimes R_{s2}]$
LOAD	$pc_l \sqcup L(r_{s1}) \sqcup L(M(R_{s1})) \sqsubseteq L(r_d)$	$M' = M[r_d \mapsto M(R_{s1})]$
STORE	$pc_l \sqcup L(r_{s1}) \sqcup L(r_d) \sqsubseteq L(M(R_{s1}))$	$M' = M[M(R_{s1}) \mapsto R_d]$
BRANCH	$L(r_{s1}) \sqcup L(r_{s2}) \sqsubseteq pc_l$	$pc' = (R_{s1} \otimes R_{s2}) ? imm : pc + 4$
JUMP	$L(r_{s1}) \sqsubseteq pc_l$	$pc' = R_{s1}$
ALL_PC	$L(M(pc_v)) \sqsubseteq pc_l \wedge pc_l \sqsubseteq \bowtie(pc_l)$	applies to all instructions
ALL_T	$t_l \sqsubseteq \bowtie(t_l) \wedge pc_l \sqsubseteq t_l$	applies to all instructions

The problem here is related to the interaction of low-integrity state with high-confidentiality computation; a cache that has been tainted with attacker-influenced state should not be allowed to influence the duration of secret operations. We incorporate this idea into our `upcall` instruction, which allows software to execute in a secret context for a predetermined amount of time. Critically, low-integrity attackers cannot `upcall` their way into learning secrets nor can they influence how trusted code execute their `upcalls`. By considering the relationship between *integrity* and *confidentiality*, we can allow programs similar to Figure 4 to execute safely, while disallowing variants that might leak information through timing.

4 Formalizing The ISA

4.1 Definitions and Model

In this section we present an abridged semantics for our ISA. First, we introduce the model for our semantics and some notational definitions. We represent our ISA as a small-step operational semantics on configurations.

Definition 3 (Configurations)**.** *A processor* ***configuration*** *represents the current state of the processor, encompassing both architecturally visible state and microarchitectural state.*

SW registers/memory	$M : Int \rightarrow Int$
SW label mappings	$L : Int \rightarrow Lbl$
opaque HW state	$\mu : Name \rightarrow Lbl$
program counter and label	$pc : PC = Int \times Lbl$
cycle counter and label	$t : T = Int \times Lbl$
call stack	$CS : List(PC \times T)$
processor configuration	$C : \langle CS, M, L, \mu, pc, t \rangle$

For simplicity, we represent both registers and DRAM as a single mapping M, in which registers are located at special addresses. Addresses are drawn from *Int*, a set of finite-size integers.[3] *Name* is a set of variable names, which

[3]The size of this range (for example, 32 or 64 bits) is architecture-specific.

$$\boxed{GR \vdash \langle CS, M, L, \mu, pc, t\rangle \longrightarrow \langle CS', M', L', \mu', pc', t'\rangle}$$

$$\textsc{execute}\ \frac{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS', M', L', pc', t_l'\rangle \qquad GR \vdash \langle CS, M, L, \mu, pc, t\rangle \longrightarrow_{\mu} \langle \mu', t_v'\rangle}{GR \vdash \langle CS, M, L, \mu, pc, t\rangle \longrightarrow \langle CS', M', L', \mu', pc', t'\rangle}$$

$$\textsc{stall}\ \frac{\langle CS, M, L, \mu, pc, t\rangle \longrightarrow_{\mu} \langle \mu', t_v'\rangle}{GR \vdash \langle CS, M, L, \mu, pc, t\rangle \longrightarrow \langle CS, M, L, \mu', pc, (t_v', t_l)\rangle}$$

Figure 5: Complete CPU operational semantics. These rules defer to semantics which describe how architectural state is modified ($\longrightarrow_{\mathcal{A}}$) and which describe how microarchitectural state is modified ($\longrightarrow_{\mu}$).

can refer to locations but are not directly representable as values. *Lbl* is the set of labels representable in our lattice. For clarity, we abbreviate full configurations as C_i, where subscript i on elements disambiguates between source configurations (e.g., M_1 is the software memory of configuration C_1). Additionally, we use pc_v to refer to the value of the *pc* and pc_l to refer to its label. The same convention is used for t.

In order to reason about the security label of a given piece of state in the processor, we define various conventions for looking up label values and converting integers to labels.

Definition 4 (Label lookup)**.** *Both architectural state and microarchitectural state are tagged with security labels. These functions describe how to determine the value of a location's label, where $i \in Int$, and $n \in Name$.*

Interpret i as a Lbl value	$\gamma(i)$
Label of location i	$L(i)$
Label of n	$\Gamma(C)(n)$

Γ is a function parameterized on processor state. This function is defined statically for a given implementation of the hardware at design time. This parameterization allows the label of any location to depend on software-specified values and/or other run-time microarchitectural state.

4.2 Operational Semantics

We present this ISA as a small-step operational semantics, factored into two semantics: a partial semantics specified by software instructions and an opaque hardware semantics that describes the behavior of microarchitectural state. Figure 5 shows the complete operational semantics for a CPU and how, in any given time step, the CPU can update architectural state (by taking a $\longrightarrow_{\mathcal{A}}$ transition) or "stall" (from the perspective of software) by updating only microarchitectural state. While we provide the explicit semantics for $\longrightarrow_{\mathcal{A}}$ (see Figure 7), the semantics for $\longrightarrow_{\mu}$ are intentionally left unspecified because they are implementation-dependent. The architectural semantics ($\longrightarrow_{\mathcal{A}}$) do not depend upon the current state of μ since μ should not, by definition, influence the behavior of software (beyond timing). Instead, we define a set of properties that the transition function $\longrightarrow_{\mu}$ must satisfy. It is these properties that allows the ISA to offer security guarantees that current architectures lack.

Table 1 provides an abridged definition of instruction restrictions (also referred to as "label checks") and behavior for pre-existing RISC-V instructions. For abbreviation purposes, the notation r_x represents the index of a register specified by an instruction. To refer to the contents of the register, we write R_x, a shorthand for $M(r_x)$, the contents of the special memory location which holds that register. The symbol $\otimes$ represents some arithmetic or relational operator appropriate to the instruction in question.

In general, the restrictions on instructions prevent state with high-security labels from influencing state with low security labels. If the restrictions for a given rule cannot be met, the instruction becomes a "no-op" that increments pc_v but has no other effects. No-ops avoid leaking information through the enforcement of label checks. However, for certain errors, it is safe to jump to a special program counter, `errorpc`, while retaining the current pc_l and t_l. One such error is violation of the ALL_PC rule, which can safely cause the program to jump to `errorpc` without breaking noninterference. The full list of these errors is specified in the technical report [18]. At this point, any error-handling

```
if (s):
  upgrade(ts, UNTRUSTED)
else:
  skip
ts2 := ts
```

Figure 6: Leaking secrets via an integrity upgrade. Execution is successful exactly when `s` is false.

program may execute (for example, to signal termination), as long as it obeys the restrictions on normal execution. To a public observer, a program that produces an error with a secret *pc* label therefore appears equivalent to a correctly operating program.

Appendix A lists specific rules for which label-checking operations can raise explicit errors and which require squashing via no-op. We include in our proof that error handling does not violate our security conditions.

The COMPUTE, LOAD/STORE and BRANCH restrictions are straightforward; they ensure that instruction operands and the *pc* must flow to the destination register. The BRANCH restrictions prevent implicit flows.

The ALL_PC restriction ensures that the instruction being executed is at least as trusted and public as the *pc* itself. This constraint prevents a trusted or public program from reading instructions from secret or untrusted memory. Additionally, ALL_PC maintains the invariant that a program may execute only if it has an uncompromised *pc*. We note in Section 5 that keeping the *pc* uncompromised is required to prevent call gates from breaking nonmalleability.

The ALL_T restriction ensures that the timing label is uncompromised and is *at least* as restrictive as the *pc* label. We summarize these restrictions as a validity condition:

$$ISVALID(pc_l, t_l) \triangleq (pc_l \sqsubseteq t_l) \wedge (pc_l \sqsubseteq \bigtimes(pc_l)) \wedge (t_l \sqsubseteq \bigtimes(t_l))$$

Intuitively, it would be difficult to implement any reasonable hardware that did not guarantee this condition. In any case where the *pc* label was more restrictive, the duration of the instruction would have to be independent of the instruction performed! This is obviously impractical for real systems, and the restriction allows us to mostly reason about pc_l when proving security conditions (see Appendix B).

4.3 Label Mutation

Figure 7 gives the operational semantics for instructions that modify label state or that raise or lower privilege.[4] Label-mutation instructions modify the labels of memory locations. It is well known that *flow-sensitive* monitors, including this ISA[5], can leak information by modifying labels if mutation is not appropriately limited [17, 19]. Since our approach involves no extra static information about the executing software, we implement the *no-sensitive-upgrade* (NSU) policy [20]. The NSU policy dynamically prevents leaks by requiring that the pc_l can flow to both the original label and the final label of the data.

However, this restriction does not eliminate all information leakage caused by label mutation. Consider the example in Figure 6. In this case, the label change is inside a secret context, which requires that the *pc* is secret and trusted. Register `ts` is secret and trusted and the upgrade makes it secret and *untrusted*. The label pc_l flows to both the original and final labels of `ts`, so the aforementioned rule is satisfied. Nevertheless, the final assignment (which occurs in a public context) to `ts2` will succeed in the case where `s` is false and fail otherwise since `ts` now represents untrustworthy information.

Additionally, since label arguments themselves are labeled memory locations, we require that the label of those arguments flows to pc_l. For example, the instruction `dwnlbl x3, x6` means: "Downgrade the label of register `x3` to the label represented by the value stored in register `x6`". If the label of `x6` itself were secret, using it to change the label of `x3` in a public context could allow an observer to learn about the content of `x6`. If the label of a location whose content is used as a label does not flow to pc_l, then the instruction becomes a no-op to prevent this kind of leakage.

[4]The R_{sn} notation refers to RISC-V style register addresses; instruction-size limitations require that the real encoding differ slightly from this notation, but it is semantically equivalent.

[5]Although this ISA is flow-sensitive, it does not have floating labels [19], and therefore labels must be explicitly changed by software instructions.

$$\boxed{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M', L', pc', t_l\rangle}$$

$$\frac{l = L(r_d) \qquad l' = \gamma(R_{s1}) \qquad RELBL(pc_l, l, l') \qquad L(r_{s1}) \sqsubseteq pc_l \qquad L' = L[r_d \mapsto l']}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L', (pc_v + 4, pc_l), t_l\rangle}\ \textsc{dwnlbl}$$

$$\frac{l = L(r_d) \qquad l' = \gamma(R_{s1}) \qquad UPLBL(pc_l, l, l') \qquad L(r_{s1}) \sqsubseteq pc_l \qquad L' = L[r_d \mapsto l']}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L', (pc_v + 4, pc_l), t_l\rangle}\ \textsc{uplbl}$$

$$\frac{\begin{array}{c} \neg INUPCALL \qquad pc'_l = \gamma(R_{s1}) \\ t'_l = \gamma(R_{s2}) \qquad ISVALID(pc'_l, t'_l) \qquad L(r_{s1}) \sqcup L(r_{s2}) \sqcup L(r_{s3}) \sqcup L(r_d) \sqsubseteq pc_l \qquad pc_l \sqcup t_l \sqsubseteq pc'_l \sqsubseteq t'_l \\ endpc = R_{s3} \qquad endt = R_d + t_v \qquad CS'[head] = ((endpc, pc_l), (endt, t_l)) \qquad CS'[tail] = CS \end{array}}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS', M, L, (pc_v + 4, pc'_l), t'_l\rangle}\ \textsc{upcall}$$

$$\frac{INUPCALL \qquad ((endpc, pc'_l), (endt, t'_l)) = CS[head] \qquad t_v \neq endt}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L, pc, t_l\rangle}\ \textsc{upret-nop}$$

$$\frac{INUPCALL \qquad ((endpc, pc'_l), (endt, t'_l)) = CS[head] \qquad CS' = CS[tail] \qquad t_v = endt}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS', M, L, (endpc, pc'_l), t'_l\rangle}\ \textsc{upret-done}$$

$$\frac{\begin{array}{c} \varnothing = CS[head] \qquad endpc = pc_v + 4 \qquad CS'[head] = ((endpc, pc_l), (null, t_l)) \\ CS'[tail] = CS \qquad (pc', t'_l) = GR(R_{s1}) \qquad ISVALID(pc'_l, t'_l) \qquad L(r_{s1}) \sqsubseteq pc_l \qquad pc'_l \sqcup t'_l \sqsubset pc_l \end{array}}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS', M, L, pc', t'_l\rangle}\ \textsc{dwncall}$$

$$\frac{((pc'_v, pc'_l), (null, t'_l)) = CS[head] \qquad pc_l \sqcup t_l \sqsubset pc'_l \sqcap t'_l \qquad CS' = CS[tail]}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS', M, L, (pc'_v, pc'_l), t'_l\rangle}\ \textsc{dwnret}$$

$$\frac{\begin{array}{c} pc'_l = \gamma(R_{s1}) \\ t'_l = \gamma(R_{s2}) \qquad pc_l \sqsubseteq pc'_l \qquad t_l \sqsubseteq t'_l \qquad ISVALID(pc'_l, t'_l) \qquad L(r_{s1}) \sqcup L(r_{s2}) \sqsubseteq pc_l \qquad \varnothing \neq CS[head] \end{array}}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L, (pc_v + 4, pc'_l), t'_l\rangle}\ \textsc{raiselbl}$$

$$\frac{\neg INUPCALL}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L, (pc_v + 4, pc_l), t_l\rangle}\ \textsc{other\_error}$$

$$\frac{INUPCALL}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L, pc, t_l\rangle}\ \textsc{upret\_error}$$

Figure 7: Operational semantics for downgrading and label-mutating instructions given a call-gate registry GR.

```
public_val = 0
while (secret_1 < secret_2):
  # do some slow computation
  secret_1++
public_val = 1
```

Figure 8: Secrets may be learned from the timing of the write to `public_val`.

We introduce additional restrictions on both upgrade and downgrade rules to prevent similar kinds of information leakage; these rules differ from each other in order to be more permissive.

Upgrading. The predicate $UPLBL(pc_l, l, l')$ expresses the NSU check for upgrading label l to label l' in the context pc_l:

$$UPLBL(pc_l, l, l') \triangleq (pc_l \sqsubseteq l \sqsubseteq l') \wedge (l' \sqsubseteq \bigtimes(pc_l))$$

The intuition here is that we need an upper bound for the final label to prevent it from moving to a new quadrant in the lattice. $UPLBL$ deviates from the original NSU definition by adding the constraint $l' \sqsubseteq \bigtimes(pc_l)$. This prevents programs from creating untrustworthy information in secret contexts and vice versa. For the program in Figure 6, the `uplbl` instruction fails the $UPLBL$ test, preventing the offending label modification. Unfortunately, this *still* leaks the value of `s` since the program only fails when `s` is true. The key insight for handling this case is that the failure happens while the *pc* is still in a high context, so measures can be taken to prevent a low context from observing the failure. We discuss this leakage in further detail below (Section 4.4).

Downgrading. There are two different cases to consider when downgrading label l to l': $l' \sqsubseteq l$ and $l' \not\sqsubseteq l$. For the first case, the predicate $DWNLBL(pc_l, l, l')$ expresses the existing nonmalleable information flow restrictions when downgrading label l to label l' in the context pc_l.

$$DWNLBL(pc_l, l, l') \triangleq (pc_l \sqsubseteq l') \wedge (l' \sqsubseteq l) \wedge (l \sqsubseteq \bigtimes(l))$$

The other case is the general form of downgrading, which we model as first executing a downgrade from l to $l \sqcap l'$, followed by an upgrade to l'. As one might expect, this essentially combines the restrictions from those other cases:

$$RELBL(pc_l, l, l') \triangleq (pc_l \sqsubseteq l \sqcap l') \wedge (l \sqsubseteq \bigtimes(l)) \wedge (l' \sqsubseteq \bigtimes(pc_l))$$

This check implies the original nonmalleability restrictions,[6] which means it is no more permissive. An alternative for modeling general downgrades would be to simulate first an upgrade to the join and then a downgrade. That requirement implies the one we've just described and is therefore also safe. However, it may be overly restrictive. It is unclear if the two are equally permissive or if downgrade-then-upgrade permits more safe programs for our lattice definition. This question lies outside the scope of this paper.

4.4 Raising context labels

The `upcall`/`upret` instruction pair introduces primitives for controlling timing channels while branching on secret or untrusted values. The `upcall` instruction allows a process to enter a more restricted context with a higher pc_l and t_l, while pushing the current pc_l and t_l to a call stack. In the new context, the program cannot write to low outputs, but its execution timing can be influenced by high hardware state. However, returning from this context reveals timing information about the duration of the subprogram. This problem can be seen in the higher-level program shown in Figure 8. The low adversary is allowed to observe the time of completion for the `while` block, since it can observe the timing of the writes to `public_val`. However, the duration of this block depends upon secret values. This example shows a more general version of the label-checking termination channel from Figure 6.

[6]In our setting, their requirement would roughly translate to the conditions: $l \sqsubseteq l' \sqcup \bigtimes(pc_l \sqcup l)$ and $pc_l \sqsubseteq l'$.

To control timing channels, upcall instructions are given an absolute end time and an ending program counter as arguments. Once the end time is reached, the processor steps to the end pc_v. The instruction arguments are saved onto a hardware call stack along with the caller's pc_l and t_l. Intuitively, this semantics preserves noninterference because the subprogram cannot modify memory locations or labels in a way that changes low observations. Since the completion of the upcall is determined purely from information of at most the level pc_l, no termination channel influences subsequent program steps.

In general, this simple approach will be difficult to use in practice because it requires programmers or compilers to know impractically cycle-accurate durations of program segments. However, it does have a use case for running untrusted functions. The upcall instruction can be used to create a low-integrity sandbox that executes until the provided timeout expires.

Using upcalls for timing mitigation. To support a more flexible programming model, we also expose a generic interface for handling returns from high contexts via an exception. When the timer completes, if the current instruction is not an upret, the configuration steps to a known exception handler pc_v[7]. Furthermore, when a label check fails inside of an upcall, the program simply stalls (i.e., steps to a new configuration where no architectural state has changed). Whichever of these conditions causes the exception is recorded in a status register (implemented as a CSR), with the high label of the upcall. In Figure 7, we use the $INUPCALL$ check to specify whether or not a configuration is inside of an upcall by inspecting the head of the call stack. If $INUPCALL$ is true, then the error can be handled normally, otherwise it should be squashed and the program should stall.

$$\begin{aligned} &INUPCALL \triangleq \\ &\quad (((endpc, pc_l'), (endt, t_l')) = CS[head]) \\ &\wedge (pc_l' \sqsubseteq pc_l \wedge t_l' \sqsubseteq t_l) \end{aligned}$$

With this primitive, the timing mitigation algorithms described in prior work [11, 21] can be implemented, enforcing bounded leakage on information from the high context. We note that this information release is still nonmalleable; both robust declassification and transparent endorsement are maintained under these mitigation mechanisms. Importantly, our restrictions prevent attackers from exploiting mitigation to exfiltrate arbitrary data.

Checking whether or not a high context subprogram failed due to violating the label check restrictions also represents a nonmalleable information release. The data in the status register can be declassified or endorsed to reveal whether or not a label check caused the subprogram to fail. Revealing this information violates the termination sensitivity of the subprogram noninterference. Although the subprogram cannot modify any low state, information is transferred via termination.

Further upcall restrictions. upcall and dwncall instructions may not be executed inside an upcall. Intuitively, a dwncall (which lowers pc_l) would allow a process to produce public outputs while still inside the upcall, leaking information about its timing and progress. As mentioned, the arguments to the upcall instruction must also themselves be labeled so that they flow to the current pc_l. Without this requirement, secret or untrusted information could still influence the duration of the subprogram.

The nesting restriction could be relaxed to allow for multiple upcall instructions so that the context could be raised repeatedly. However, we do not include it in this formalism since it would complicate the requirements for hardware (call stacks would no longer have finite depth). In reality, nesting would be useful for implementing the process of control transfer from user space to operating system privileges and from there to the hypervisor level.

Permanently raising context labels. In addition to the upcall instruction, the pc_l and t_l can be raised by simply writing to them (they are implemented as CSRs). In order to preserve noninterference, the labels can only be raised in this way. Once raised, a program can only lower its context labels by executing a dwncall instruction. This limits the possible leakages caused by the program to outputs produced by the set of trusted functions which it is allowed to call. We discuss this further in the next section.

[7]Termination behavior can be configured on a per-program basis; it is only required that the configuration is completed using only information that is low relative to the program's original pc_l.

4.5 Lowering context labels

The `dwncall`/`dwnret` instructions allow programs to call into more-public and more-trusted contexts via *call gates*. Call gates are essentially labeled functions that have been pre-registered by a public–trusted entity. The call-gate registry is effectively a read-only function lookup table.[8] A call gate registration contains a *pc* and t_l; using a `dwncall` instruction sets the current *pc* and t_l to the gate's values while pushing the prior values onto a call stack. These instructions provide hardware support for the privilege escalation features described in prior work on security and information flow. In particular, they closely resemble the primitives required to implement *gates* from the Multics and HiStar operating systems [22, 23]. In those systems, gates were used respectively to call known functions with higher privileges than the caller, and to implement synchronous RPC.

4.6 Exceptions and Asynchrony

We do not include exception configuration or handling in our ISA formalism or formal security proof. In this section, we describe how one could incorporate these features into our ISA without compromising its security conditions. All exceptions have a triggering condition and an *exception program counter* (`epc`) that points to the interrupt service routine (ISR)[9].

Trigger conditions can be specific to an ISA-extension or architecture and are often defined by the hardware. The `epc` is programmed by software and stored in a CSR. There are additional exception-masking CSRs which software can use to suppress the trigger conditions. In general, in order for an exception to fire, the security label of all trigger conditions (including masks) must flow to the current pc_l; otherwise, an attacker process may learn that an exception fired and deduce some secret related to its cause. For arithmetic exceptions such as integer overflow or divide-by-zero, this implies that the instruction operands flow to the current pc_l; if they don't, the exception must be suppressed. The label of the *pc* while the ISR is actually handling the exception must also be lower bounded by all trigger inputs and the label of the `epc` register itself. In this way, if an exception trigger condition is secret, its handler must be executing in a secret context and cannot produce public outputs.

We believe the primary complications involved in integrating exception handling into such an ISA are as follows. First, it is not always clear how to label exception triggers. For example, should an incoming network packet signal be labeled public or could the timing of packet arrival give an attacker information about co-resident processes? Likely, this choice should be programmable by software depending on the threat model. Secondly, depending upon how hardware state is labeled, asynchronous exceptions (such as timers and incoming network packets) may be frequently dropped or delayed. In order to account for this, the processor and ISA may need to be modified to support batched handling of exceptions along predetermined schedules within the CPU itself. Additionally, it may be difficult to limit the number of actual hardware signals that contribute to exception trigger conditions in real implementations. For example, Van Bulck et al. [24] found that Intel SGX implementations allowed the currently executing instruction to complete before handling certain exceptions. Waiting for instruction completion means that most control signals in the CPU would influence the exception trigger conditions. It is not always possible to immediately transfer control to the ISR without waiting for some state to clear in the CPU, and thus it may be challenging to implement practical exceptions that execute in contexts that have low confidentiality or high integrity.

5 ISA Design Discussion

Here we highlight some salient points of our design and compare and contrast with other language-based IFC systems.

Compromised contexts and data undermine nonmalleability The original nonmalleability paper [14] identified restrictions on downgrading that are equivalent to our observation that compromised labels cannot be downgraded to public or trusted status. We additionally notice that executing in a compromised context can unsafely leak information through timing. Specifically, this can violate the *non-occlusion* principle of declassification described by Sabelfeld and Sands [25]. Consider the scenario where `upcall` operations implement predictive mitigation, and therefore enforce

[8]Using rules similar to the `uplbl` instruction, call gate entries can also be made more secret or less trusted without violating noninterference.

[9] This is not the same as the RISC-V epc CSR, we are paraphrasing the exception handling mechanism for clarity.

nonmalleability (rather than noninterference). Allowing a process to raise its pc_l and/or t_l to a compromised level is unsound because it implicitly allows that process to declassify arbitrary data. With our restrictions, observing the duration of this subprogram leaks only the caller's secrets and is therefore robust; otherwise *any* information could be implicitly declassified via this channel.

Software can control how much information it leaks through timing channels Our ISA provides strong guarantees with respect to timing. As long as a program keeps its timing label low and executes fully low-deterministic upcalls, it leaks no information through its timing behavior. However, programs are not strictly bound by these restrictions. By explicitly exposing the pc_l, t_l and `upcall` timing to software, we grant programs the ability to weaken these restrictions gracefully to suit their needs. This provides important flexibility for situations where our threat model is overly strong or when application-specific data may only require probabilistic guarantees about timing consistency.

Limitations of Our ISA While our ISA has strong security guarantees and important security primitives, there is much room for future research. First of all, our timing label mechanism does provide a bound on which information may be implicitly leaked through timing channels. However, this coarse-grained approach could potentially leak *any* information below the timing label. This behavior is unlike the `dwnlbl` instruction, which explicitly denotes the memory location to be downgraded. Our ISA also does not incorporate explicit timing into any instructions other than `upcall`. While this lack of explicitness is beneficial for remaining implementation-agnostic, it does not give guidance on how to implement secure *and efficient* hardware. Yu et al. [26] describe an ISA which focuses on this performance aspect, by exposing more microarchitectural information in their ISA. Future secure ISAs and ISA extensions must be designed with both of these goals in mind, potentially leading to new semantics or completely novel timing-aware instructions.

Finally, our work only targets the single core subset of the RISC-V ISA and does not provide guidance on how to address multicore communication and interference. This realm of interconnected computing devices communicating via shared memory and coherence networks introduces many more opportunities for timing interference and side channel communication. Investigating this problem requires a significant further effort in analyzing the semantics of existing memory models, microarchitectural coherency guarantees and how to efficiently incorporate IFC labels into these protocols.

6 Hardware Semantics and Properties

As mentioned earlier, an actual hardware implementation of this ISA will be a circuit that not only implements the software-visible semantics but also refines the full CPU semantics. We now discuss properties of a hardware implementation that are sufficient to guarantee the ISA-level security conditions. Additionally, we discuss the implications of these properties on hardware implementations and comment on what techniques may be utilized to verifiably construct hardware with said properties.

Property 1 (Deterministic Execution)**.** *For any configuration C, and for all $i \in \{1, 2\}$*

$$\begin{array}{c} C \longrightarrow_{\mu} \langle \mu_i, t_{vi} \rangle \implies ((\mu_1 = \mu_2) \wedge (t_{v1} = t_{v2})) \\ \wedge \\ C \longrightarrow C_i \implies C_1 = C_2 \end{array}$$

The operational semantics for the transition function on microarchitectural states must be deterministic. Furthermore, we assume that the full semantics which determines when to stall the processor is also deterministic.

We believe that this property can also be relaxed to allow for sources of nondeterminism (such as changes in clock frequencies, random number generators, etc.) as long as this nondeterminism is truly generated by noise or other public/trusted factors. Defining exactly what factors are public/trusted is a complex decision related to particular threat models and is out of scope for this paper.

Property 2 (Single-Step Machine Noninterference)**.** *Given a set of low labels in the security lattice, $\mathcal{L}$,*

$$\begin{aligned}&\forall C, i \in \{1,2\}.\\ &\quad (C_1 =_{\mathcal{L}} C_2) \wedge (C_i \longrightarrow C_i')\\ &\qquad \implies ((\mu_1' =_{\mathcal{L}} \mu_2') \wedge (t_{v1}' =_{\mathcal{L}} t_{v2}')).\end{aligned}$$

The hardware implementation must enforce a timing-sensitive noninterference condition for microarchitectural state for all transitions. With this definition, the label of t effectively bounds which hardware state may affect the timing of operations (including the decision to stall or not stall computation). The above property also implies that $\longrightarrow_{\mu}$ enforces timing-sensitive noninterference on μ and t. Note that this noninterference condition only applies for microarchitectural state, not architectural state. The architectural state may be downgraded using the downgrade instructions in our ISA.

The above definition of timing-sensitive machine noninterference is actually overly strong and we can substitute a slightly weaker property. t is interpreted as a global clock; however, this requirement enforces that hardware end instructions at exactly the same *real time* whenever $t_l' \in \mathcal{L}$. For most cases this isn't a problem, since $t_l \in \mathcal{L}$ and therefore both configurations start executing the instruction at the same time. It is not unreasonable for hardware to therefore ensure that they end at the same time by using only low-labeled state to influence their duration.

However, some instructions can lower t_l thereby creating a scenario where $t_l \in \mathcal{H}$ and $t_l' \in \mathcal{L}$. In our ISA, `dwncall` can create this scenario and would theoretically require that two executions always enter the call gate at the same time, even when they previously had high timing labels. Since $t_{v1} \neq t_{v2}$ there is no way for a CPU to ensure $t_{v1}' = t_{v2}'$. Luckily, *real time* equivalence is not really the guarantee we need. We just need the *duration* of the instructions to be equal in both configurations, if $t' \in \mathcal{L}$. For all of the instructions in our ISA, this results in exactly the same security guarantees that we have claimed in Section 7. Below is the amended Single-Step Machine Noninterference property:

$$\begin{aligned}&\forall C, i \in \{1,2\}.\\ &\quad (C_1 =_{\mathcal{L}} C_2) \wedge (C_i \longrightarrow C_i')\\ &\qquad \implies ((\mu_1' =_{\mathcal{L}} \mu_2') \wedge (t_l' \in \mathcal{L} \implies t_{v1}' - t_{v1} = t_{v2}' - t_{v2}))).\end{aligned}$$

Property 3 (Computability of Label Lookups)**.**

$$\exists \Gamma,\ \forall C,\ n \in \mathrm{dom}(\mu), \Gamma(C)(n) \textit{ is computable}$$

Property 3 has so far been an implicit assumption. The function Γ is parameterized on all of the configuration state; it represents a function that must be computed at run time and therefore must be implemented in the microarchitecture. In combination with Property 2, this implies that the process of looking up microarchitectural labels does not violate noninterference [27]. It also implies that, after a configuration step $C \longrightarrow C'$, Γ determines low equivalence by evaluating labels of μ using C', not C (we formalize low equivalence further in Section 7).

Intuitively, the above properties suggest that there is no hardware-level information flow which violates timing-sensitive noninterference except for flows that are explicitly induced by software instructions. For instance, declassifying a secret memory location, *loc*, with a `dwnlbl` instruction can only declassify microarchitectural state that specifically represents *loc*'s data. Section 7 discusses the ISA-level security properties that we can obtain, given these hardware properties, in more detail.

6.1 Implications for Hardware Implementations

Property 1 can be easily satisfied, for the most part, as processors are typically implemented as deterministic digital circuits. While some features require a notion of nondeterminism (such as random number generators or external sensor inputs), these can be modeled as the I/O to a deterministic digital circuit. In the design, one must label and build deterministic circuitry used to process these values (e.g., a buffer containing input packets from the network) but the non-determinism of the outside system has no direct impact on the security of the processor itself. As discussed in Section 4.6, this may lead to different low-level behaviors and performance characteristics in real implementations.

Furthermore, even features with somewhat unpredictable behavior can be modeled deterministically as long as their inputs are deterministic. For example, DVFS [28] modulates clock frequency during execution and can change the wall-clock time of code execution. However, if those modulation decisions are made via a digital circuit and their inputs are deterministic, we can model DVFS as software-visible architectural state and guarantee that its use does not violate our security conditions.

Property 2 requires a processor to be designed to remove timing channels through its microarchitecture. A recent publication [7] shows that such a tagged processor with strong control for microarchitectural timing channels and potentially reasonable overheads is feasible. Yu et al. [26] have also shown recently that it is feasible to build a modern CPU with speculation, out-of-order execution and other microarchitectural optimizations while enforcing probabilistic-noninterference [29]. These results provide evidence that it is possible to build efficient secure hardware, with the appropriate ISA abstractions.

Property 3 suggests that processor microarchitecture needs to be designed in a way that allows the security label of microarchitectural state to be determined. This property can be achieved by either statically labeling hardware modules at design time or by adding hardware tags to track runtime labels. Recursively, these tags are also microarchitectural state and their labels must also be computable. Therefore, real implementations will use both of these techniques (static vs. dynamic labels) since Γ is only computable if it eventually reaches a fixed point.

Our ISA provides hardware designers with the flexibility to choose how to realize timing-sensitive noninterference. For example, in order to remove cache timing channels, a processor designer may: statically partition a cache; dedicate a cache to one security level and flush it when the security level is lowered; bypass the cache; or even introduce scratchpad memory with a fixed latency, etc.

6.2 Enforcing Timing-Sensitive Noninterference in Hardware

For strong security assurance, we ideally want to formally enforce the properties needed for a secure hardware implementation. There exist several efforts to develop security-annotated Hardware Description Languages (HDL) that can provide timing-sensitive noninterference guarantees, similar to the one we specify here [5, 30, 31]. Previous studies show that these security-annotated HDLs can be used to express realistic security policies and implement complex circuits that satisfy them [6, 7, 32, 33].

The primary challenge with proving Property 2 by using secure HDLs is that these languages do not have separate notions of "architectural" and "microarchitectural" state; the entire circuit is represented as a single state machine. Phrased another way, hardware and software are concerned with different definitions of observability; in the hardware description, all state is considered observable, even though software can only directly observe architectural state. This disconnect makes proving a hardware implementation correct challenging for a few specific reasons.

First, it is impossible to prove that an implementation that supports ISA-level downgrading provides microarchitectural noninterference. Any implementation of our ISA must contain downgrades at the HDL level, which correspond to those required to implement downgrading instructions. However, the noninterference guarantees provided by these HDLs are completely obviated by including downgrades; they cannot ensure that the information being downgraded is limited only to architectural state.

A second issue with proving hardware implementations secure is the difference in label equivalence models. We assume that an attacker cannot read the value of a secret label, but can observe the fact that the label is secret. In the hardware, any location which stores a label value must itself be labeled. Given the attacker model above, it is unclear how to write down the label of this location. If we label it as public, then the HDL will allow us to define hardware that leaks the values of secret labels to attackers. If we label it as secret, then the HDL will conservatively disallow some safe label checking operations.

We believe that these problems may be solved by applying prior techniques for verifying CPU correctness (such as Pipecheck and RTLCheck [34, 35]). Moreover, these approaches could be augmented with formal verification tools specifically designed for IFC. For instance, Nickel [36] is a framework for proving noninterference that uses application specific definitions of observational equivalence. Investigating how to utilize these approaches to prove microarchitectural noninterference while supporting software-level downgrading and notions of observability is an interesting open research question.

$$
\begin{aligned}
pc_1 =_{\mathcal{L}} pc_2 &\iff ((pc_{l1} \wedge pc_{l2}) \notin \mathcal{L}) \vee (pc_1 = pc_2) \\
t_1 =_{\mathcal{L}} t_2 &\iff ((t_{l1} \wedge t_{l2}) \notin \mathcal{L}) \vee (t_1 = t_2) \\
L_1 =_{\mathcal{L}} L_2 &\iff (L_1 \approx L_2) \wedge (\forall j \in \text{dom}(L).\quad L(j) \in \mathcal{L} \implies L_1(j) = L_2(j)) \\
M_1 =_{\mathcal{L}} M_2 &\iff (L_1 \approx L_2) \wedge (\forall j \in \text{dom}(M).\quad L(j) \in \mathcal{L} \implies M_1(j) = M_2(j)) \\
\mu_1 =_{\mathcal{L}} \mu_2 &\iff (\Gamma(C_1) \approx \Gamma(C_2)) \wedge (\forall n \in \text{dom}(\mu).\quad \Gamma(C)(n) \in \mathcal{L} \implies \mu_1(n) = \mu_2(n)) \\
CS_1 =_{\mathcal{L}} CS_2 &\iff CS_1 \approxeq_{\mathcal{L}} CS_2 \\
C_1 =_{\mathcal{L}} C_2 &\iff (pc_1 =_{\mathcal{L}} pc_2) \wedge (t_1 =_{\mathcal{L}} t_2) \wedge (M_1 =_{\mathcal{L}} M_2) \wedge (\mu_1 =_{\mathcal{L}} \mu_2) \wedge (CS_1 =_{\mathcal{L}} CS_2)
\end{aligned}
$$

Figure 9: Low Equivalence of Configuration Components, relative to "low" labels, $\mathcal{L}$.

7 ISA Security Properties

This section describes some of the security properties of this ISA and their performance and usability tradeoffs.

Low Equivalence. We start by formalizing the low equivalence of configurations, relative to a set of low labels, $\mathcal{L}$. This models the ability of an observer who can only differentiate between low states; two low-equivalent configurations appear identical to a "low observer". First, we define an equivalence operator on label mappings to formalize our notion that attackers cannot observe exact label values.

Definition 5 (Label Lookup Domain Equivalence)**.** For an attacker inducing label sets $\mathcal{P}, \mathcal{S}, \mathcal{U}$, and $\mathcal{T}$

$$
\begin{aligned}
&L_1 \approx L_2 \iff \forall n \in \text{dom}(L). \\
&\quad (L_1(n) \in \mathcal{P} \iff L_2(n) \in \mathcal{P}) \wedge \\
&\quad (L_1(n) \in \mathcal{T} \iff L_2(n) \in \mathcal{T})
\end{aligned}
$$

We define the $\approx$ relation on the labels of microarchitecture similarly.

Figure 9 shows the definition of low equivalence for all configuration components. We assume that L, M, μ and Γ are total functions so that domain equality is implicit. The requirements of low equivalence explicitly require that "label lookups" for both architectural and microarchitectural state return equivalent but not equal values for high labels. Call stack low equivalence requires that all entries with low pc_l are in the same position in the stack and are themselves low-equivalent. By construction, all low entries must be at the head of the stack[10] so it is sufficient to check that the low prefixes of each call stack are equivalent.

Definition 6 (Call Stack Prefix Low Equivalence)**.**

$$
\begin{aligned}
&CS_1 \approxeq_{\mathcal{L}} CS_2 \iff \\
&\qquad (1) \quad CS_1 = \varnothing \wedge \forall (pc^i, t^i) \in CS_2, pc^i \in \mathcal{H} \\
&\qquad\quad\ \textit{or} \\
&\qquad (2) \quad CS_2 = \varnothing \wedge \forall (pc^i, t^i) \in CS_1, pc^i \in \mathcal{H} \\
&\qquad\quad\ \textit{or} \\
&\qquad (3) \quad CS_1[\textit{head}] = (pc_1, t_1) =_{\mathcal{L}} (pc_2, t_2) = CS_2[\textit{head}] \\
&\qquad\qquad \wedge\ CS_1[\textit{tail}] \approxeq_{\mathcal{L}} CS_2[\textit{tail}]
\end{aligned}
$$

[10]This is enforced by preventing `dwncalls` while inside of an `upcall`.

Security Guarantees. All of the theorems in this section have full proofs, which can be found in Appendix B. First, we show that executing programs that do not contain downgrade or call-gate instructions preserve noninterference.

We use the term *valid* configurations to refer to configurations that were initialized with reasonable values. Specifically, the configurations satisfy the ALL_PC and ALL_T requirements and the initial call stacks are empty.

Theorem 1 (Noninterference Modulo Downgrading and Call Gates)**.**
For any two valid configurations, C_1 and C_2 and any low set of labels, $\mathcal{L}$, where no instruction is a `dwnlbl`*,* `upcall`*,* `upret-done`*,* `dwncall` *or* `dwnret`*:*

$$(C_i \longrightarrow^* C_i^{'}) \wedge (C_1 =_{\mathcal{L}} C_2) \implies C_1^{'} =_{\mathcal{L}} C_2^{'}$$

where $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$.

The proof is a straightforward structural induction on the operational semantics of the processor. By assuming Property 2, essentially all of the work in this proof requires proving noninterference of the $\longrightarrow_{\mathcal{A}}$ semantics.

We next extend Theorem 1 to prove noninterference even when using `upcall` instructions.

Theorem 2 (Noninterference Modulo Downgrading)**.**
For any two valid configurations, C_1 and C_2, and any low set of labels, $\mathcal{L}$, where no instruction is a `dwnlbl`*,* `dwncall` *or* `dwnret`.

$$(C_i \longrightarrow^* C_i') \wedge (C_1 =_{\mathcal{L}} C_2) \implies C_1' =_{\mathcal{L}} C_2'$$

In the scenario covered by Theorem 1, once the pc_l was high, it could never be lowered again. That makes the noninterference proof trivial but also limits functionality. To prove Theorem 2, we show that all operational steps taken while an `upcall` is on the call stack can be modeled as a single operational step to low-equivalent configurations. We can show this since the end configuration of the upcall is predetermined by low-equivalent state and high pcs are noninterfering (i.e., programs executing with a high pc cannot modify any low visible state).

Note that while this theorem is termination-sensitive , it is not timing-sensitive. In the case where $t_l \not\sqsubseteq pc_l$, attackers may make observations about high state based on the timing of writes to low state. We present a corollary that provides timing sensitivity.

Corollary 1 (Timing-Sensitive Noninterference Modulo Downgrading)**.**
If ($pc_l \in \mathcal{L} \implies t_l \in \mathcal{L}$) for all intermediate configurations and `upcall` *regions have fixed durations, then Theorem 2 provides timing sensitivity.*

This corollary ensures that any time that low writes are possible, the attacker will observe them occurring at the same time. Furthermore, the duration of high call gates will be determined by low information.

As defined in Section 2, nonmalleability is essentially defined as maintaining both robust declassification and transparent endorsement. Even with no syntactic restrictions (unlike the prior theorems) our ISA enforces nonmalleability.

Theorem 3 (Nonmalleable Information Flow)**.** *For attacker induced high label sets $\mathcal{S}$ and $\mathcal{U}$ and their respective complements, $\mathcal{P}$ and $\mathcal{T}$ and valid configurations, $\forall\{s,u\} \in \{1,2\}, C_{su}$*

$$\begin{gathered}((C_{su} \longrightarrow C_{su}') \ \wedge (C_{1u} =_{\mathcal{P}} C_{2u}) \ \wedge (C_{s1} =_{\mathcal{T}} C_{s2})) \\ \implies \\ ((C_{11}' =_{\mathcal{P}} C_{21}' \implies C_{12}' =_{\mathcal{P}} C_{22}') \\ \wedge \\ (C_{11}' =_{\mathcal{T}} C_{12}' \implies C_{21}' =_{\mathcal{T}} C_{22}'))\end{gathered}$$

Assuming Theorem 2, we only need to reason about instructions which violate information flow: `dwncall` and `uplbl`. The key restrictions which provide nonmalleability are those that prevent the pc_l or t_l from becoming compromised and the restriction that compromised data is never downgraded.

```
# PCLBL = TLBL = (TRUSTED, PUBLIC)
# L(key) = L(s0) = (TRUSTED, SECRET)
# L(in0) = (TRUSTED, PUBLIC)
upcall est, ST, ST, enc_end
----------------------------
# PCLBL = (T,S), TLBL = (T,S)
andi in0, in0, MASK
xor s0, key, in0
lw s0, 0(s0)          # (a)
andi s0, in0, mask
lw s0, 0(s0)          # (b)
declreg s0, PUBLIC
upret
----------------------------
enc_end:
```

Figure 10: Mitigated AES.

8 Program Examples

We now describe examples of how to use our ISA features in practical scenarios.

AES is a well known encryption algorithm which does not require the program to branch on any secrets [37]. Instead, AES uses a public lookup table indexed by computation involving both the secret key and public input. This behavior of executing secret-dependent memory accesses makes it susceptible to a number of timing-channel attacks [38–42], some of which are similar to the vulnerability in Figure 3.

Figure 10 is a toy version if this AES-style lookup table access in our ISA. Without mitigation techniques, the execution of the second load *(b)* could be faster if it accesses the same cache line from *(a)*. Similarly, another program may also infer the value of the secret through cache contention.

One existing software-based mitigation technique for preventing this cache timing channel is to preload the entire lookup table ahead of time [43]. Preloading allows a cache implementation to fill its entries with useful data based only on public addresses. However, this approach is not guaranteed to be secure on normal hardware; if a cache were too small to contain the entire table (or evicted entries for any other reason), it is possible that some lookups would trigger misses, thereby leaking information with an unexpectedly *slow* duration for certain keys. Other efforts to eliminate these problems with AES still rely on the assumption that certain instructions are constant-time [44].

Our ISA enables software to control microarchitectural timing channels in a principled manner. On hardware implementing our ISA, the secret-dependent loads in Figure 10 cannot affect public microarchitectural state and therefore cannot leak secret information through memory contention. Additionally, the strategy of preloading the cache can still improve performance on some implementations. One potential CPU implementation might maintain private and public cache partitions. During the preload phase, public and trusted code fills up the public cache partition with some or all of the AES table. During the encryption phase, secret code can read those entries but cannot modify them, instead making updates only to the private cache partition. This implementation would allow for a more secure and efficient AES execution. Nevertheless, the duration of the entire execution could leak some information about the secret key; this example also shows how software can use an `upcall` instruction to obscure that duration by providing an explicit end time (via the `est` argument in the example's upcall).

8.1 Password Checker

In this example, we show how to implement a nonmalleable password checker which can be called by untrusted users with the `dwncall` instruction. The code for this checker is shown in Figure 11. This program starts in a public and untrusted context, which would be typical for an unauthenticated user. The untrusted user generates their guess and puts it into the register called `guess`. Then they use the `dwncall` instruction to call the `check_pass` function and gain

```
# PCLBL = TLBL = (PUBLIC, UNTRUSTED)
# L(guess) = (PUBLIC, UNTRUSTED)
# L(pass) = (SECRET, TRUSTED)
dwncall check_pass
==============================
# PCLBL = TLBL = (PUBLIC, TRUSTED)
check_pass:
  endoreg guess, TRUSTED
  upcall est, ST, ST, end_check
------------------------------
# PCLBL = TLBL = (SECRET, TRUSTED)
  beq guess, pass, success
  li res 0
  upret
success:
  li res 1
  upret
------------------------------
end_check:
  declreg res, PUBLIC
  dwnret
```

Figure 11: Password checking in the proposed ISA.

high integrity. This is analogous to executing a system call in a typical operating system, where the user program is linked with trusted libraries and jumps into that code.

Once the `check_pass` function has started, it must endorse the user's guess, since a trusted *pc* cannot branch on low-integrity data. In order to compare the secret password value with the guess, the program executes an `upcall` instruction to enter a timing-mitigated region. Inside that region, the program computes either a 1 or 0 based on whether or not the guess was right or wrong, and then returns. Finally, at the end of the `check_pass` function, the result is declassified to public and the call gate exits back to the untrusted context.

If an untrusted user were to execute the `check_pass` function like a normal function call, their attempts to endorse their own guess and `upcall` into a secret and trusted state would both fail. This example illustrates the nonmalleability guarantees and how trusted system code can be resident in the system but only accessible via call gates.

9 Related Work

Software Information Flow Control. Software-based IFC has been applied in many settings with the goal of eliminating timing channels [11, 17, 45–50]. Kashyap et al. [48] discuss various software strategies for enforcing timing-sensitive noninterference. In particular, they focus on using lattice scheduling to ensure that the ordering of visible events does not leak secret information. Parsec [46] is a language for concurrent programming which, given a race-freedom analysis, ensures observational determinism, a noninterference condition for concurrent programs. Bedford et al. [17] have also shown how a hybrid IFC system can provide progress-sensitive noninterference, a weaker condition than timing sensitivity; it does not leak information based on which sets of outputs a program successfully produces. Secure multi-execution, where a program is executed multiple times at varying security levels, has also been used to prove timing-sensitive noninterference [50]. LIO [47] is a Haskell-based language extension for mitigating both external and internal channels through the use of monadic computation and IFC. Of the aforementioned systems, only LIO handles external timing channels. Like our ISA, LIO provides a dynamic semantics for enforcing noninterference but lacks features such as downgrading and integrity tracking.[11] Additionally, it is a high-level language which requires a software runtime for its security, making it unsuitable as an ISA description.

[11] Follow-up work (e.g., [19, 51]) addresses some of these features.

This paper describes a *dynamic label*[12] model where all data is labeled, including data used to represent labels. This is reminiscent of systems such as JFlow/Jif [52, 53] and the dynamic security labels formalism [27]. These languages rely on static annotations and dynamic label-checking operations to guarantee noninterference while still permitting labels to be used in types. Since our labels have a hardware representation, label-checking operations can be implemented efficiently in hardware. Our platform appears well suited to accelerating such languages, allowing them to execute outside of a software runtime. We considered explicitly modeling precise "labels of labels" as software-accessible state, but it was unclear what further security this provided, and it significantly complicated the run-time checks. Other dynamic IFC languages, such as LIO [47], treat the label of label values as visible to the current context but do not allow for their precise manipulation. This choice mirrors our rule that the label of label arguments must flow to pc_l.

Hardware-level information flow control. IFC techniques have also been used to build timing-safe hardware. While not focused on timing, Suh et al. [54] showed that processors could implement efficient information flow tracking. Caisson and Sapper [6, 33] provided a nested state machine abstraction for circuit design and proved that hardware built using those tools enforced timing-sensitive noninterference. More expressive HDLs that provide similar security guarantees have also been developed using dependent types [30, 55]. The Hyperflow processor [7] is a fully-featured implementation of a RISC-V CPU developed using these techniques.

Secure ISAs. While many of the above HW IFC systems presented CPUs and ISAs, they were focused on security guarantees about the circuits. None of them have proved security results for programs executing on top of their example abstractions. Ge et al. [12] have defined a set of properties they argue post-Spectre ISAs (called aISAs) must enforce to provide efficient, timing-sensitive security. These properties primarily focus on prescribing how an operating system can interact with the hardware to provide timing security. They refer to concrete mechanisms such as hardware partitioning and time multiplexing rather than the security properties that these mechanisms should aim to enforce. Our ISA provides more fundamental guarantees than those suggested in their work, but real implementations of our ISA would likely exhibit many of the properties they list.

Yu et al. [26] have built an ISA extension for "oblivious computing" and have proved probabilistic noninterference results. They have also built and measured the performance of a speculative, out-of-order processor using this ISA and demonstrated its performance improvements over more conservative techniques. Their ISA treats security as an optional component which software may opt-in to by labeling instruction operands as public or secret. This is promising evidence of the practicality of efficient microarchitectures for secure ISAs.

The work of Zhang et al. [11] on language-based timing mitigation defines a software–hardware contract based on "write labels" and "read labels" that almost directly parallel our pc_l and t_l. However, that contract requires well-typed programs that correctly specify write and read labels; the hardware itself is not assumed to enforce any restrictions on how these labels change over time. Furthermore, our ISA considers both confidentiality and integrity while enforcing nonmalleable downgrading. We do not require a fully trusted entity to perform timing mitigation: any `upcall` caller can implement their own mitigation algorithm in their own context.

OS-level information flow control. Asbestos [56] and HiStar [23] are two well known IFC operating systems. They do not assure timing safety. However, HiStar's notion of *gates* informed our call gate mechanism, but the restrictions on gates and the security guarantees differ from ours. NickelOS [36] has been recently developed using *intransitive noninterference*, which allows more flexible security policies than traditional IFC. However, NickelOS is not timing-sensitive and focuses on information flow exposed through OS APIs.

10 Conclusion and Future Work

In this paper, we have proposed an ISA that defines a contract between software and hardware that defines how information may or may not affect the timing of instructions. Importantly, it provides timing safety without requiring

[12]The word dynamic is unfortunately overloaded. Here, it refers to labels whose values are explicitly visible or comparable at run time. Dynamic IFC systems are those that enforce security via run-time checks. Our ISA both has dynamic labels *and* is a dynamic IFC system.

that instructions explicitly execute in worst-case time. As a byproduct, our proofs delineate conditions that hardware should satisfy, thus providing guidance to hardware designers.

We foresee many avenues for further research in the domain of timing secure ISAs. Modeling more ISA features such as exceptions, memory models, and other concurrency mechanisms can provide evidence toward the practicality of this approach to ISA design. Furthermore, it will help expose more potential side channels that exist throughout the complex environment of multicore processors.

Given this foundation, we can develop new instructions or instruction semantics that expose different timing characteristics, such as fixed-latency scratchpad memory [57] or other "oblivious" computation [26]. Experimenting with these new ideas in the context of a nonmalleable ISA can also ensure that the security guarantees hold end to end.

The largest open question is how to formally verify that hardware implementations satisfy the properties defined in Section 6, allowing us to connect security guarantees of high-level languages and verified operating systems to the actual behavior of the underlying hardware. We think there are many opportunities to improve existing secure HDLs for finer grained downgrading (of both data and time), and to adapt hardware functional verification techniques to prove IFC properties of processors.

Acknowledgments

We would like to thank Adrian Sampson, Ryan Doenges, Ethan Cecchetti, Josh Gancher, Rolph Recto, Tom Magrino, Sungbo Park and our reviewers for their constructive feedback. This work was partly sponsored by NSF grant CNS-1513797 and DARPA contract HR0011-18-C-0014. Opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[3] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symp.*, 2018, pp. 991–1008.

[4] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[5] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 503–516, 2015.

[6] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 109–120.

[7] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. Suh, "Hyperflow: A processor architecture for nonmalleable, timing-safe information flow security," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[8] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, “Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC,” Cryptology ePrint Archive, Report 2015/1241, 2015, https://eprint.iacr.org/2015/1241.

[9] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, “System-level non-interference for constant-time cryptography,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’14. New York, NY, USA: ACM, 2014, pp. 1267–1279. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660283

[10] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, “CT-wasm: Type-driven secure cryptography for the web ecosystem,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 77:1–77:29, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290390

[11] D. Zhang, A. Askarov, and A. C. Myers, “Language-based control and mitigation of timing channels,” in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’12. New York, NY, USA: ACM, 2012, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254078

[12] Q. Ge, Y. Yarom, and G. Heiser, “No security without time protection: We need a new hardware-software contract,” in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, ser. APSys ’18. New York, NY, USA: ACM, 2018, pp. 1:1–1:9. [Online]. Available: http://doi.acm.org/10.1145/3265723.3265724

[13] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[14] E. Cecchetti, A. C. Myers, and O. Arden, “Nonmalleable information flow control,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1875–1891.

[15] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, “The RISC-V instruction set manual. volume 1: User-level ISA, version 2.0,” CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, Tech. Rep., 2014.

[16] A. C. Myers, A. Sabelfeld, and S. Zdancewic, “Enforcing robust declassification and qualified robustness,” *Journal of Computer Security*, vol. 14, no. 2, pp. 157–196, 2006.

[17] A. Bedford, S. Chong, J. Desharnais, E. Kozyri, and N. Tawbi, “A progress-sensitive flow-sensitive inlined information-flow control monitor (extended version),” *Computers & Security*, vol. 71, pp. 114–131, 2017.

[18] D. Zagieboylo, G. E. Suh, and A. C. Myers, “Using information flow to design an isa that controls timing channels,” Cornell University, Tech. Rep., 2019.

[19] P. Buiras, D. Stefan, and A. Russo, “On dynamic flow-sensitive floating-label systems,” in *2014 IEEE 27th Computer Security Foundations Symposium*. IEEE, 2014, pp. 65–79.

[20] T. H. Austin and C. Flanagan, “Efficient purely-dynamic information flow analysis,” in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, ser. PLAS ’09. New York, NY, USA: ACM, 2009, pp. 113–124. [Online]. Available: http://doi.acm.org/10.1145/1554339.1554353

[21] A. Askarov, D. Zhang, and A. C. Myers, “Predictive black-box mitigation of timing channels,” in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 297–307.

[22] J. H. Saltzer, “Protection and the control of information sharing in Multics,” *Communications of the ACM*, vol. 17, no. 7, pp. 388–402, 1974.

[23] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, “Making information flow explicit in HiStar,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI ’06. Berkeley, CA, USA: USENIX Association, 2006, pp. 263–278. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298481

[24] J. Van Bulck, F. Piessens, and R. Strackx, “Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic.”

[25] A. Sabelfeld and D. Sands, “Dimensions and principles of declassification,” in *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. IEEE, 2005, pp. 255–269.

[26] J. Yu, L. Hsiung, M. El Hajj, and C. W. Fletcher, “Data oblivious ISA extensions for side channel-resistant and high performance computing.”

[27] L. Zheng and A. C. Myers, “Dynamic security labels and noninterference,” in *Formal Aspects in Security and Trust*. Springer, 2005, pp. 27–40.

[28] S. Herbert and D. Marculescu, “Analysis of dynamic voltage/frequency scaling in chip-multiprocessors,” in *Int'l Symp. on Low Power Electronics and Design*, 2007.

[29] M. Backes and B. Pfitzmann, “Computational probabilistic noninterference,” *International Journal of Information Security*, vol. 3, no. 1, pp. 42–60, 2004.

[30] A. Ferraiuolo, “Security results for SIRRTL, a hardware description language for information flow security,” 2017.

[31] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, “Chisel: constructing hardware in a scala embedded language,” in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 2012, pp. 1212–1221.

[32] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, “Verification of a practical hardware security architecture through static information flow analysis,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 555–568. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037739

[33] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, “Sapper: A language for hardware-level security policy enforcement,” in *ACM SIGPLAN Notices*, vol. 49, no. 4. ACM, 2014, pp. 97–112.

[34] D. Lustig, M. Pellauer, and M. Martonosi, “PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models,” in *47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 635–646. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.38

[35] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer, “RTLcheck: Verifying the memory consistency of RTL designs,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 463–476. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124536

[36] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang, “Nickel: a framework for design and verification of information flow control systems,” in *13th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 287–305.

[37] J. Daemen and V. Rijmen, “AES proposal: Rijndael,” 1999.

[38] D. J. Bernstein, “Cache-timing attacks on AES.”

[39] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, “Wait a minute! A fast, cross-VM attack on AES,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 299–319.

[40] R. Spreitzer and T. Plos, “Cache-access pattern attack on disaligned AES T-tables,” in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2013, pp. 200–214.

[41] B. Gülmezoğlu, M. S. Inci, G. Irazoqui, T. Eisenbarth, and B. Sunar, "A faster and more realistic flush+ reload attack on AES," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2015, pp. 111–126.

[42] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20.

[43] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities." *IACR Cryptology ePrint Archive*, vol. 2006, p. 52, 2006.

[44] E. Käsper and P. Schwabe, "Faster and timing-attack resistant AES-GCM," in *Cryptographic Hardware and Embedded Systems-CHES 2009*. Springer, 2009, pp. 1–17.

[45] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *Computer Security Foundations Workshop, IEEE(CSFW)*, vol. 00, 06 1997, p. 156. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CSFW.1997.596807

[46] S. Zdancewic and A. C. Myers, "Observational determinism for concurrent program security," in *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE.* IEEE, 2003, pp. 29–43.

[47] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres, "Addressing covert termination and timing channels in concurrent information flow systems," in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '12. New York, NY, USA: ACM, 2012, pp. 201–214. [Online]. Available: http://doi.acm.org/10.1145/2364527.2364557

[48] V. Kashyap, B. Wiedermann, and B. Hardekopf, "Timing-and termination-sensitive secure information flow: Exploring a new approach," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 413–428.

[49] J. Agat, "Transforming out timing leaks," in *27th ACM Symp. on Principles of Programming Languages (POPL)*, Jan. 2000, pp. 40–53. [Online]. Available: http://dl.acm.org/citation.cfm?id=325694.325702

[50] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 109–124.

[51] P. Buiras, D. Vytiniotis, and A. Russo, "HLIO: Mixing static and dynamic typing for information-flow control in Haskell," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015. New York, NY, USA: ACM, 2015, pp. 289–301. [Online]. Available: http://doi.acm.org/10.1145/2784731.2784758

[52] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif: Java information flow," *Software release. Located at http://www. cs. cornell. edu/jif*, vol. 2005, 2001.

[53] A. C. Myers and A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1999, pp. 228–241.

[54] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *ACM Sigplan Notices*, vol. 39, no. 11. ACM, 2004, pp. 85–96.

[55] A. Ferraiuolo, W. Hua, A. C. Myers, and G. E. Suh, "Secure information flow verification with mutable dependent types," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: ACM, 2017, pp. 6:1–6:6. [Online]. Available: http://doi.acm.org/10.1145/3061639.3062316

[56] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris, "Labels and event processes in the Asbestos operating system," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 17–30.

[57] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, “Scratchpad memory: A design alternative for cache on-chip memory in embedded systems,” in *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*. IEEE, 2002, pp. 73–78.

A Definitions

$$\boxed{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M', L', pc', t_l\rangle}$$

$$\dfrac{\neg \mathit{INUPCALL} \qquad L(M(pc_v)) \not\sqsubseteq pc_l \vee pc_l \not\sqsubseteq \text{⧖}(pc_l)}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L, (errorpc, pc_l), t_l\rangle}\ \text{ALL\_PC\_ERROR}$$

$$\dfrac{\neg \mathit{INUPCALL} \qquad L(r_{s1}) \sqcup L(r_{s2}) \not\sqsubseteq pc_l}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L, (errorpc, pc_l), t_l\rangle}\ \text{BRANCH\_ERROR}$$

$$\dfrac{\neg \mathit{INUPCALL} \qquad l' = \gamma(R_{s1}) \qquad (L(r_{s1}) \not\sqsubseteq pc_l) \vee (l' \not\sqsubseteq \text{⧖}(pc_l))}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L, (errorpc, pc_l), t_l\rangle}\ \text{UPLBL\_ARG\_ERROR}$$

$$\dfrac{\neg \mathit{INUPCALL} \qquad l' = \gamma(R_{s1}) \qquad (L(r_{s1}) \not\sqsubseteq pc_l) \vee (l \not\sqsubseteq \text{⧖}(pc_l))}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L, (errorpc, pc_l), t_l\rangle}\ \text{RELBL\_ARG\_ERROR}$$

$$\dfrac{\neg \mathit{INUPCALL} \qquad L(r_{s1}) \sqcup L(r_{s2}) \not\sqsubseteq pc_l}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L, (errorpc, pc_l), t_l\rangle}\ \text{RAISELBL\_ARG\_ERROR}$$

$$\dfrac{\neg \mathit{INUPCALL} \qquad L(r_{s1}) \sqcup L(r_{s2}) \sqcup L(r_{s3}) \sqcup L(r_d) \not\sqsubseteq pc_l}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L, (errorpc, pc_l), t_l\rangle}\ \text{UPCALL\_ARG\_ERROR}$$

$$\dfrac{\begin{gathered}(GR(R_{s1}) = \varnothing) \qquad \vee \\ (\varnothing \neq CS[head]) \vee (L(r_{s1}) \not\sqsubseteq pc_l) \vee (pc'_l \sqcup t'_l \not\sqsubset pc_l)\end{gathered}}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L, (errorpc, pc_l), t_l\rangle}\ \text{DWNCALL\_ARG\_ERROR}$$

$$\dfrac{\neg \mathit{INUPCALL}}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L, (pc_v + 4, pc_l), t_l\rangle}\ \text{OTHER\_ERROR}$$

$$\dfrac{\mathit{INUPCALL}}{GR \vdash \langle CS, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CS, M, L, pc, t_l\rangle}\ \text{UPRET\_ERROR}$$

Figure 12: Operational semantics for error handling rules given a call-gate registry GR.

We begin by restating some definitions from Section 2. Here, l_i, i and c represent elements of *Lbl*.

$$
\begin{aligned}
&(i,c)^{\rightarrow} \triangleq c \\
&(i,c)^{\leftarrow} \triangleq i \\
&l_1 \sqsubseteq l_2 \overset{\triangle}{\Leftrightarrow} (l_1^{\leftarrow} \sqsubseteq l_2^{\leftarrow}) \wedge (l_2^{\rightarrow} \sqsubseteq l_1^{\rightarrow}) \\
&l_1 \sqcup l_2 \triangleq ((l_2^{\rightarrow} \sqcup l_2^{\rightarrow}), (l_1^{\leftarrow} \sqcap l_2^{\leftarrow})) \\
&l_1 \sqcap l_2 \triangleq ((l_2^{\rightarrow} \sqcap l_2^{\rightarrow}), (l_1^{\leftarrow} \sqcup l_2^{\leftarrow})) \\
&\bowtie(i,c) \triangleq (c,i)
\end{aligned}
$$

Call stack validity can be formalized as:

$$
\begin{aligned}
&CS = ((pc_{l0}, pc_{v0}, t_{l0}, t_{v0})...(pc_{ln}, pc_{vn}, t_{ln}, t_{vn})) \\
&ISVALID(CS) \triangleq \forall i \in (0, n-1).(pc_{li} \sqsubseteq pc_{li+1} \texttt{ or } pc_{li+1} \sqsubseteq pc_{li}) \\
&ISVALID(CS, pc_l) \triangleq ISVALID(CS) \texttt{ and } (pc_l \sqsubseteq pc_{l0} \texttt{ or } pc_{l0} \sqsubseteq pc_l \texttt{ or } CS = \varnothing)
\end{aligned}
$$

$ISVALID$ $(CS$,$pc_l)$ can be read as "*CS is valid for pc_l*", meaning that the call stack itself has ordered entries and the pc_l is ordered with respect to the head of the call stack. This restriction maintains the idea that call gates cannot be freely mixed with moving the pc_l around via `raiselbl`; call gates need to reflect the actual sequence of control transfer agreed upon when the gates are established via `dwncall` or `upcall`.

Definition 7 (Call Stack Validity)**.**
A call stack, CS, is valid with respect to the current pc_l if it represents an uninterrupted sequence of call gate calls and returns.

A configuration is *valid* if it has a valid pc_l, t_l and CS:

Definition 8 (Configuration Validity)**.**
A configuration, C, is valid iff: $pc_l \sqsubseteq \bowtie(pc_l)$, $pc_l \sqsubseteq t_l$, $t_l \sqsubseteq \bowtie(t_l)$, and CS is valid for pc_l.

This validity condition captures the notion that the pc_l and t_l remain *uncompromised*, in addition to call stack validity.

Notation. Two low-equivalent configurations C_1 and C_2 contain state such as *pc* or t_l which may vary between them or be the same. When the values must be equivalent in both configurations we omit subscripts. When they may differ, we use subscripts to denote to which configuration they belong.

B Proofs

Based on our attacker definition, all secret and untrusted labels are compromised.

Lemma 1. *For $\mathcal{S}$ and $\mathcal{U}$ sets induced by an attacker:*

$$\forall l \in \textit{Lbl}.\, l \in \mathcal{S} \cap \mathcal{U} \implies l \not\sqsubseteq \bowtie(l)$$

Proof. Recall that attacker-induced sets are defined as upward-closed sets with a minimum confidentiality c_A and a maximum integrity i_A. The label (i_A, c_A) is itself compromised and by Definition 1, we have $(i_A, c_A) \not\sqsubseteq (c_A, i_A)$.

The upward closure property implies $\forall l \in \mathcal{S} \cap \mathcal{U}.(i_A, c_A) \sqsubseteq l$. We show that $l \sqsubseteq \bowtie(l) \implies (i_A, c_A) \sqsubseteq (c_A, i_A)$ and by contrapositive, l must be compromised. We represent l explicitly as (l_i, l_c).

$$\begin{aligned}
& l \sqsubseteq \bowtie(l) \equiv (l_i, l_c) \sqsubseteq (l_c, l_i) && \\
& \equiv (l_i \sqsubseteq l_c) \wedge (l_i \sqsubseteq l_c) && \textit{(By definition of} \sqsubseteq \textit{)} \\
& \implies i_A \sqsubseteq l_c && \textit{(By } (i_A, c_A) \sqsubseteq l \implies (i_A \sqsubseteq l_i)\textit{)} \\
& \implies i_A \sqsubseteq c_A && \textit{(By } (i_A, c_A) \sqsubseteq l \implies (l_c \sqsubseteq c_A)\textit{)} \\
& \equiv (i_A \sqsubseteq c_A) \wedge (i_A \sqsubseteq c_A) && \\
& \equiv (i_A, c_A) \sqsubseteq (c_A, i_A) && \textit{(By definition of} \sqsubseteq \textit{)}
\end{aligned}$$

□

Lemma 2 (No Compromised Call Stack Entries)**.** *No valid call stack will contain any entries whose pc_l is compromised.*

Proof. Any instruction may only execute successfully if the pc_l itself is *valid*. The pc_l validity condition requires that it is an uncompromised label. Therefore, any time an `upcall` or `dwncall` succeeds and places an entry onto the stack, the label of that entry (the current pc_l) must be valid and uncompromised. □

Configuration validity is preserved under our instruction semantics.

Lemma 3 (Validity of Configurations)**.** *If C is a valid configuration and $C \longrightarrow^* C'$, configuration C' is also valid.*

Proof. We show that no instruction will step to an *invalid* configuration and by induction, this lemma holds. Instructions `upcall`, `dwncall`, and `raiselbl` check that the $\mathit{ISVALID}$ (pc_l,t_l) condition holds for the new pc_l and t_l. If the new labels would be invalid, then the current (and valid) labels are retained. Furthermore, when the `upret-done` or `dwnret` instructions execute, by Lemma 2, the resulting pc_l and t_l will be uncompromised (and a similar argument fulfills the remainder of the $\mathit{ISVALID}$ (pc_l,t_l) condition).

Therefore, the only validity condition we must check is call stack validity. The only instructions which change CS are `upcall`/`upret-done` and `dwncall`/`dwnret`. Instructions `upcall` and `dwncall` explicitly require that the current pc_l is ordered with respect to the new pc (pc'_l). For `upcall`, $pc_l \sqsubseteq pc'_l$ and for `dwncall` $pc'_l \sqsubseteq pc_l$. Since the new "head" of the call stack will have an entry label of pc_l, validity is preserved. Inductively, any `dwnret` or `upret-done` instruction will also preserve call stack validity since the pc_l of CS's first entry must either be ordered with respect to CS's second entry or CS has only one entry.

The `raiselbl` instruction is prohibited from changing pc_l or t_l whenever CS is nonempty; therefore the only other instruction which changes the pc_l cannot violate the ordering relationship between CS [head] and pc_l.

In particular, the `dwncall` and `upcall` restrictions require that pc_l' and pc_l are ordered, and then push pc_l onto the call stack. The `raiselbl` restriction ensures that pc_l can only be raised when already in an upcall *or* when the call stack is empty. In the latter case, validity is trivially preserved. In the former, the `upcall` restrictions ensure that the label of $CS[head] \sqsubseteq pc_l$, which means raising pc_l will maintain that ordering. □

Lemma 4 (Call Stack Upcall History)**.** *For any two valid configurations C_1 and C_2, If $C_1 =_{\mathcal{L}} C_2$ and $pc_l \in \mathcal{L}$, then either both configurations are in an* `upcall` *region, or neither is.*

Proof. If either one of the call stacks is empty, then, by Definition 6, the other must also be empty or all of its entries have a high pc label (which we'll denote pc_{cs}). If both are empty, then neither is in an `upcall` region.

If only one is empty, a `dwncall` from high to low must have generated the head entry of that stack. An `upcall` could not have executed successfully since `upcall` requires that the pc label of the executing context flows to the new pc label. Since pc_l is low and pc_{cs} is high, $pc_{cs} \not\sqsubseteq pc_l$ (by upward closure of $\mathcal{H}$). Furthermore, since a `dwncall` can only be made when the call stack is empty, there could have been no prior `upcall` executed without a corresponding return. In this case, one configuration is inside a `dwncall` region and the other is inside no call-gate region.

If both call stacks are nonempty, let $CS_i[head] = ((pc_{vi}, pc_{li}), (t_{vi}, t_{li}))$. If $pc_{li} \in \mathcal{H}$ then, by the same argument as before, both configurations must be inside a `dwncall` region but not inside an `upcall` region. If $pc_{li} \in \mathcal{L}$, then $pc_{l1} = pc_{l2}$. This means that the most recent call gate instruction was either an `upcall` or a `dwncall`, but it must be the same for both configurations. If it was a `dwncall`, then $CS_i[tail] = \varnothing$ and neither configuration is in an `upcall` region. ☐

Lemma 5 (Low Equivalence of Error Rules in Low Contexts)**.** *For any set of low labels, $\mathcal{L}$, and any two valid configurations C_1, C_2 and $C_1 \longrightarrow C_1'$ by applying an* ERROR *rule, and* **insn** $\not\equiv$ DWNLBL,DWNCALL,DWNRET

$$(C_1 =_{\mathcal{L}} C_2) \wedge (pc_l \in \mathcal{L}) \implies (C_1' =_{\mathcal{L}} C_2')$$

Proof. First, we show that this lemma holds for all of the error rules in Figure 12. For any check in the form of $L(var) \sqsubseteq pc_l$[13], we can guarantee that the result of that check is the same for low-equivalent configurations when $pc_l \in \mathcal{L}$.

If $L_1(var) \in \mathcal{L}$, then by the low equivalence of configurations, $L_2(var) = L_1(var)$. The evaluation of $L(var) \sqsubseteq pc_l$ therefore results in the same outcome in both configurations. If $L_1(var) \in \mathcal{H}$, then by the low equivalence of configurations, $L_2(var) \in \mathcal{H}$. In both cases, the check will fail since $\mathcal{H}$ is upward closed and $pc_l \in \mathcal{L}$. By Lemma 4, either both configurations are in an upcall region, or neither is. Therefore the `INUPCALL` check has the same result for C_1 and C_2. All of the rules in Figure 12 contain only checks of the form $L(var) \sqsubseteq pc_l$[14] and `INUPCALL` checks. Since the outcome of the label checking must be the same in both configurations, updating the pc_v to **errorpc** does not violate low equivalence in the resulting configurations.

We now consider the label checks specific to various instructions.

Branch/Jump:

These label checks only contains checks of the form $L(var) \sqsubseteq pc_l$. By our earlier argument, the BRANCH_ERROR rule from Figure 12 always results in low-equivalent configurations.

Compute/Load/Store:

The checks for these instructions ensure that all of the labels of the operands flow to the label of the destination. Wlog. we can analyze a check of the form $L(v_1) \sqsubseteq L(v_2)$.

Let us consider the four possible cases that C_1 can evaluate:

Case 1) $L(v_1), L(v_2) \in \mathcal{L}$

In this case, by the low equivalence of configurations both $L_1(v_1) = L_2(v_1)$ and $L_1(v_2) = L_2(v_2)$ and so the success or failure of the label check is the same for both configurations.

Case 2) $L(v_1) \in \mathcal{L}, L(v_2) \in \mathcal{H}$

In this case, $L_1(v_1) = L_2(v_1)$ and $L(v_2)$ is high in both configurations. It is possible that this label check fails in C_1 but succeeds in C_2 or vice versa.

Case 3) $L(v_1) \in \mathcal{H}, L(v_2) \in \mathcal{L}$

In this case the check will fail in both configurations, since $\mathcal{H}$ is upward closed and $L(v_1)$ is high in both configurations.

Case 4) $L(v_1) \in \mathcal{H}, L(v_2) \in \mathcal{H}$

Like case 2, this label check could fail in only one of the configurations, since not all high values flow to one another.

In cases 1 and 3, both configurations will fail (this lemma considers only the cases where C_1 fails label checking) and therefore neither will update memory or label state, which implies that low equivalence will be preserved. In cases 2 and 4, the $L(v_2)$ is a high label. This means that even if the operation succeeds in one configuration and not the other, the changes to memory happen only on elements with high labels. Therefore, memory low equivalence is preserved.

[13]Or of the form $L(var) \not\sqsubseteq pc_l$

[14]Or $L(var) \sqsubseteq \bigtimes(pc_l)$ which does not change our reasoning.

Uplabel:

$$UPLBL(pc_l, l, l') \triangleq (pc_l \sqsubseteq l \sqsubseteq l') \wedge (l' \sqsubseteq \bigtimes(pc_l))$$

As argued above, if the UPLBL_ARG_ERROR rule is matched, low equivalence is preserved since both configurations will fail this check and step to the error program counter. Let us now consider the case where an UPLBL instruction fails label checking but the UPLBL_ARG_ERROR rule does not apply. In this case, $L_1(r_{s1}) \sqsubseteq pc_l$ and is therefore low. This implies that R_{s1} is equal in both configurations and therefore $l'_1 = l'_2$. The $l' \sqsubseteq \bigtimes(pc_l)$ component of the label check will therefore result in the same value for both configurations. Lastly, we consider the requirement: $pc_l \sqsubseteq l \sqsubseteq l'$.

Again there are 4 cases to analyze:

Case 1) $l, l' \in \mathcal{L}$

In this case, $l_1 = l_2$ by low equivalence, so the label-checking result will be the same in both configurations.

Case 2) $l' \in \mathcal{L}, l \in \mathcal{H}$

In this case, both l_1 and l_2 are in $\mathcal{H}$ by low equivalence and by upward closure of $\mathcal{H}$, $l \sqsubseteq l'$ will fail in both configurations.

Case 3) $l \in \mathcal{L}, l' \in \mathcal{H}$

By low equivalence, we have $l_1 = l_2$ and both configurations compute the same label-check result.

Case 4) $l, l' \in \mathcal{H}$

In this case, l_1 and l_2 are both in the high domain of L. Even if this label check succeeds in one configuration, it will only change the value of a high label to another high label. This does not affect low equivalence so $L'_1 =_{\mathcal{L}} L'_2$.

RaiseLbl:

If the RAISELBL_ARG_ERROR rule is not matched then the pc_l, pc'_l and t'_l must be the same in both configurations. The only two cases to consider are based on the current value of t_l. If $t_l \in \mathcal{L}$, then both configurations will fail the label check and low equivalence is preserved. If $t_l \in \mathcal{H}$, then even if only one configuration succeeds $t'_{l1} =_{\mathcal{L}} t'_{l2}$ since both have high labels.

Upcall:

If the UPCALL_ARG_ERROR rule is not matched, then R_d, $endpc, pc'_l$ and t'_l will be the same in both configurations. Since both configurations are valid we know that $pc_l \sqsubseteq t_l$ for both configurations. By the earlier argument, both configurations will resolve the $pc_l \sqsubseteq pc'_l \sqsubseteq t'_l$ check the same way. If true then that implies the restriction $t_l \sqsubseteq t'_l$ is true for both configurations as well. Therefore, if either configuration passes the label checking process, then both configurations must pass it. By contrapositive, if either configuration fails then both will fail label checking.

Upret:

By Lemma 4, both configurations will either be in an `upcall` region or not, meaning that this instruction either causes an error or does not in both configurations. In the case where an error occurs, no state is updated, and therefore the resulting configurations are low-equivalent.

□

Lemma 6 (High PC Call Stack Noninterference). *For any two valid configurations C_1 and C_2, where neither configuration executes an* UPRET-DONE *step.*

$$pc_l \in \mathcal{H}, C_1 =_{\mathcal{L}} C_2, C_i \longrightarrow C'_i \implies CS'_1 =_{\mathcal{L}} CS'_2$$

Proof. By the definition of call stack equivalence, both call stacks begin with equivalent prefixes of entries with low pc_l, or neither contains any entries with a low pc_l.

In the former case, the heads of these stacks must contain low-equivalent entries, where the pc label of these entries (pc_{lentry}) is $\in \mathcal{L}$. In both configurations, $pc_l \in \mathcal{H}$ and therefore this entry must have been produced by an `upcall` instruction. While in an upcall region, the only stack-modifying instruction that can successfully return is

UPRET-DONE. Since we are excluding that rule in this lemma, the resulting call stacks in this case must always be low-equivalent, because they cannot be modified.

In the latter case, call stacks produced by these configurations are low-equivalent. Popping off entries from the stack cannot introduce entries with low pc_l and pushing on entries uses the current pc_l, which is high. Therefore any call-stack modifying instruction will result in low-equivalent call stacks.

□

This also leads to a useful corollary: if two configurations have a high pc_l and neither is inside of a upcall with a low entry label, their call stacks are always noninterfering. As noted in Lemma 6, if the entries on the call stack all have high pc_l, then no operation will place a low-labeled entry on the stack. Therefore call-stack low equivalence is preserved.

Corollary 2. *For any two valid configurations C_1 and C_2, where $CS_i[head] = \varnothing$ or $CS_i[head] = ((pc_{vi}, pc_{li}), (t_{vi}, t_{li}))$ and $pc_{li} \in \mathcal{H}$.*

$$pc_l \in \mathcal{H}, C_1 =_{\mathcal{L}} C_2, C_i \longrightarrow C_i' \implies CS_1' =_{\mathcal{L}} CS_2'$$

Now we show that configurations with high pc_l cannot make any low-visible changes to state without the use of downgrading instructions or call gates. We then use this result to prove the more general noninterference of high pcs lemma.

Lemma 7 (Noninterference of High PCs Modulo Call Gates)**.** *For any two valid configurations, C_1 and C_2, if* **insn** $\not\equiv$ DWNCALL,DWNRET,UPCALL,UPRET-DONE

$$\begin{aligned}&(pc_l \in \mathcal{H}) \wedge (C_i \longrightarrow C_i') \wedge (C_1 =_{\mathcal{L}} C_2)\\ &\quad \implies \langle CS_1', M_1', L_1', pc_1', t_{l1}'\rangle =_{\mathcal{L}} \langle CS_2', M_2', L_2', pc_2', t_{l2}'\rangle\end{aligned}$$

Proof. First, we note that C_1 and C_2 may or *may not* be executing the same instruction; therefore, this proof must rely on reasoning about the visible effects of any given single configuration.

Label Mappings:

We show that $L_i =_{\mathcal{L}} L_i'$ to prove noninterference of label mappings since $(L_1 =_{\mathcal{L}} L_2) \wedge (L_i =_{\mathcal{L}} L_i') \implies L_1' =_{\mathcal{L}} L_2'$. For each instruction we must show that the low and high domains of L do not change, and that low labels are not modified at all. To show the former, we need to show that any label being changed remains in the same quadrant of the lattice after modification. In general, if any label-modifying instruction fails, low equivalence is preserved since L cannot change.

First, we consider the `uplbl` instruction, which sets some location's label from l to l'.

Recall that the primary restriction for this instruction is:

$$UPLBL(pc_l, l, l') \triangleq (pc_l \sqsubseteq l \sqsubseteq l') \wedge (l' \sqsubseteq \bowtie(pc_l))$$

There are four cases to consider here: $l \in \mathcal{L}$; $\mathcal{H} = \mathcal{S} \wedge l \in \mathcal{S} \cap \mathcal{T}$; $\mathcal{H} = \mathcal{U} \wedge l \in \mathcal{P} \cap \mathcal{U}$; $l \in \mathcal{S} \cap \mathcal{U}$.

Case 1) $l \in \mathcal{L}$

$UPLBL$ requires $pc_l \sqsubseteq l$, so this case cannot succeed.

Case 2) $\mathcal{H} = \mathcal{S} \wedge l \in \mathcal{S} \cap \mathcal{T}$

$UPLBL$ requires $l \sqsubseteq l' \wedge l' \sqsubseteq \bowtie(pc_l)$. We show by contradiction that if $l' \in \mathcal{S} \cap \mathcal{U}$ and $pc_l \in \mathcal{S}$, this instruction cannot succeed. (And consequently, $l' \in \mathcal{S} \cap \mathcal{T}$ when the instruction does succeed.)

$$\begin{aligned}&\textit{Assume: } l' \sqsubseteq \bowtie(pc_l)\\ &\equiv (l'^{\leftarrow} \sqsubseteq pc_l^{\rightarrow}) \wedge (pc_l^{\leftarrow} \sqsubseteq l'^{\rightarrow}) \quad (\textit{By definition})\\ &\implies (A^{\leftarrow} \sqsubseteq pc_l^{\rightarrow}) \qquad (l' \in \mathcal{U} \implies A^{\leftarrow} \sqsubseteq l'^{\leftarrow})\\ &\implies (A^{\leftarrow} \sqsubseteq A^{\rightarrow}) \quad (pc_l \in \mathcal{S} \implies pc_l^{\rightarrow} \sqsubseteq A^{\rightarrow})\\ &\equiv i_A \sqsubseteq c_A \equiv (i_A, c_A) \sqsubseteq \bowtie((i_A, c_A))\end{aligned}$$

This assumption contradicts our definition of attacker, so we have $l' \not\sqsubseteq \bowtie(pc_l)$. In conclusion, if $l' \in \mathcal{S} \cap \mathcal{U}$, the instruction will not succeed.

In the case where the instruction succeeds, $l' \in \mathcal{S} \cap \mathcal{T}$.

Case 3) $\mathcal{H} = \mathcal{U} \wedge l \in \mathcal{P} \cap \mathcal{U}$

This follows the same logic as in case 2: we show by contradiction that, if $l' \in \mathcal{S} \cap \mathcal{U}$, and $pc_l \in \mathcal{U}$, then $l' \not\sqsubseteq \bowtie(pc_l)$. When the instruction succeeds, $l' \in \mathcal{P} \cap \mathcal{U}$.

Case 4) $l \in \mathcal{S} \cap \mathcal{U}$

Since $l \sqsubseteq l'$, $l' \in \mathcal{S} \cap \mathcal{U}$.

Next we consider the `dwnlbl` instruction, which modifies the label of some location ($l = L(r_d)$) to some new value ($l' = \gamma(R_{s1})$). The primary restriction on executing this instruction is:

$$RELBL(pc_l, l, l') \triangleq (pc_l \sqsubseteq l \sqcap l') \wedge (l \sqsubseteq \bowtie(l)) \wedge (l' \sqsubseteq \bowtie(pc_l))$$

Again, there are 4 cases to consider: $l \in \mathcal{L}$; $\mathcal{H} = \mathcal{S}, l \in \mathcal{S} \cap \mathcal{T}$; $\mathcal{H} = \mathcal{U}, l \in \mathcal{P} \cap \mathcal{U}$; $l \in \mathcal{S} \cap \mathcal{U}$.

Case 1) $l \in \mathcal{L}$

$RELBL$ requires $pc_l \sqsubseteq l \sqcap l'$, so this case cannot succeed.

Case 2) $\mathcal{H} = \mathcal{S}, l \in \mathcal{S} \cap \mathcal{T}$

If the instruction succeeds, $l' \notin \mathcal{P}$ since $RELBL$ requires $pc_l \sqsubseteq l \sqcap l'$. Let us assume $l' \in \mathcal{S} \cap \mathcal{U}$. In that case, by the same reasoning as for `uplbl`, $l' \not\sqsubseteq \bowtie(pc_l)$. Therefore, if the label check passes in this case, $l' \in \mathcal{S} \cap \mathcal{T}$.

Case 3) $\mathcal{H} = \mathcal{U}, l \in \mathcal{P} \cap \mathcal{U}$

This case has exactly the same reasoning as case 2. If label checks succeed, then $l' \in \mathcal{P} \cap \mathcal{U}$.

Case 4) $l \in \mathcal{S} \cap \mathcal{U}$

By Lemma 1, $l \not\sqsubseteq \bowtie(l)$, and therefore this case will fail the label check and not modify L.

Memories:
For all of the instructions which write to M, the pc_l must flow to the label of the modified memory location. Since no label modification instructions change the low domains of memory this implies that neither configuration can make low visible modifications to memory: $(pc_l \in \mathcal{H}) \wedge (M_1 =_{\mathcal{L}} M_2) \wedge (L_i \approx L'_i) \implies M'_1 =_{\mathcal{L}} M'_2$
Program Counters and PC/Time Labels:
First we consider the `raiselbl` instruction. It requires $pc_l \sqsubseteq pc'_l$ and $t_l \sqsubseteq t'_l$, ensuring that pc'_l and t'_l are also high labels, and therefore pc'_v will be low-equivalent across the configurations.
If the executing instruction fails for any reason (thus triggering the ERROR rule), the pc_l and t_l remain high and no other state is modified. □

Theorem 1 (Noninterference Modulo Downgrading and Call Gates)**.** *For any two valid configurations, C_1 and C_2 and any low set of labels, $\mathcal{L}$, if insn* $\neq$ DWNLBL,DWNRET,UPCALL,UPRET-DONE,DWNCALL

$$C_i \longrightarrow^* C_i^* \wedge C_1 =_{\mathcal{L}} C_2 \implies C_1^* =_{\mathcal{L}} C_2^*$$

Proof. We prove this by structural induction over the $\longrightarrow$ operator: $C_i \longrightarrow C'_i$. By Property 2, we know that μ and t are noninterfering, such that $\mu'_1 =_{\mathcal{L}} \mu'_2$ and $t'_{v1} =_{\mathcal{L}} t'_{v2}$. When the STALL rule is applied, no architectural state changes and therefore $C'_1 =_{\mathcal{L}} C'_2$.
Therefore, we now only need to consider the $\longrightarrow_{\mathcal{A}}$ function and the configuration it produces.
As a reminder:

$$GR \vdash \langle CS, M, L, pc, t \rangle \longrightarrow_{\mathcal{A}} \langle CS', M', L', pc', t'_l \rangle$$

We show that

$$\langle CS'_1, M'_1, L'_1, pc'_1, t'_{l1}\rangle =_{\mathcal{L}} \langle CS'_2, M'_2, L'_2, pc'_2, t'_{l2}\rangle$$

holds for all possible applications of $\longrightarrow_{\mathcal{A}}$.

By Lemma 7, when $pc_l \in \mathcal{H}$

$$\langle CS'_1, M'_1, L'_1, pc'_1, t'_{l1}\rangle =_{\mathcal{L}} \langle CS'_2, M'_2, L'_2, pc'_2, t'_{l2}\rangle.$$

Now we consider the case when $pc_l \in \mathcal{L}$. By Lemma 5, we know that in situations where one or more of the configurations fails label checking, low equivalence of configurations is preserved. We now consider only scenarios where both configurations pass label checking.

Both configurations must be executing the same instruction (by the ALL_PC rule which ensures that the instruction itself is located in low-labeled memory).

Label Mappings:

Unlike in Lemma 7 we can allow labels to change their current quadrant, as long as the change results in low-equivalent label maps. Again, UPLBL is the only instruction which modifies L. This instruction updates label $l \equiv L(r_d)$ to $l' \equiv \gamma(R_{s1})$ and requires $l \sqsubseteq l'$.

r_d and r_{s1} must both have the same respective value across configurations since they are part of the instruction.

$l'_1 = l'_2$ since $L(R_{s1}) \sqsubseteq pc_l \implies R_{s1}$ has the same value in both configurations.

The three possible cases are: $l, l' \in \mathcal{L}$; $l \in \mathcal{L}$ and $l' \in \mathcal{H}$; and $l, l' \in \mathcal{H}$. Since there is only 1 case where $l \in \mathcal{H}$, $l'_1 = l'_2$ and $L_1 =_{\mathcal{L}} L_2$, both configurations must execute the same case.

Case 1) $l, l' \in \mathcal{L}$

Since the only mapping updated in L' is $r_d \mapsto l'$ and it is changed equivalently in L_1 and L_2, $L'_1 =_{\mathcal{L}} L'_2$.

Case 2) $l \in \mathcal{L}, l' \in \mathcal{H}$

In this case, the low and high domains of L_1 and L_2 change, but they will still both change in the same way ($L(r_d)$ goes from $\mathcal{L}$ to $\mathcal{H}$ in both configurations).

Case 3) $l, l' \in High$

The domains of L_1 and L_2 do not change and none of the labels in the low domains change, so this does not change the low equivalence of L_1 and L_2.

Memories:

All of the compute instructions require that the labels of the operands flow to the labels of the destination. Furthermore, the location in memory to be updated is part of the instruction and therefore is equivalent in both configurations. Therefore, if the label of the destination is low, the operands will have low labels and be equivalent. This ensures that changes to low memory happen the same way in each configuration. Additionally, the proof that $L'_1 =_{\mathcal{L}} L'_2$ implies that the new low domains of memory will be equivalent as well ($L'_1 \approx L'_2$).

Program Counters and PC/Time Labels:

All branch/jump instructions require that their operands flow to pc_l. Therefore, with a low pc_l all of the operands will have low labels and be equivalent in successful configurations: $pc'_{v1} = pc'_{v2}$.

The `raiselbl` instruction can raise the pc_l and t_l. This is analogous to the UPLBL rule for memory locations and for similar reasons ensures that $pc'_1 =_{\mathcal{L}} pc'_2$ and $t'_{l1} =_{\mathcal{L}} t'_{l2}$.

□

Corollary 3 (Timing-Sensitive Noninterference Modulo Downgrading and Call Gates). *For any two valid configurations, C_1 and C_2 and any low set of labels, $\mathcal{L}$, if insn $\not\equiv$ DWNLBL,UPCALL,UPRET,DWNCALL,DWNRET and, for all configuration steps $pc_l \in \mathcal{L} \implies t_l \in \mathcal{L}$,*

$$C_i \longrightarrow^* C_i^* \wedge C_1 =_{\mathcal{L}} C_2 \implies C_1^* =_{\mathcal{L}} C_2^*$$

Proof. This follows directly from the previous theorem and Property 2. If $t_l \in \mathcal{L}$, then low equivalence of t ensures that their values will be equal after each transition ($t_{v1} = t_{v2}$ and $t'_{v1} - t_{v1} = t'_{v2} - t_{v2}$ implies $t'_{v1} = t'_{v2}$). Essentially, the two configurations must execute in lock step. In this scenario the only instruction which can raise t_l is RAISELBL, so as long as $t_l \in \mathcal{L}$ and no RAISELBL instruction raises the t_l above pc_l, timing sensitivity is preserved. □

Executing a complete upcall region from a low context will preserve low equivalence. Since the end state of the upcall is determined a priori by low-equivalent configurations, it ensures that both configurations will reach a state where they are again low-equivalent and have exactly the same call stack state as they did initially.

Lemma 8 (High Upcall Noninterference)**.** *For any two valid configurations, C_1 and C_2, if $pc_l \in \mathcal{L}$, $C_1 =_{\mathcal{L}} C_2$, insn* $\equiv$ `upcall`, *$C_i \longrightarrow C'_i$ and $pc'_l \in \mathcal{H}$, then:*

$$(C_i \longrightarrow^* C''_i \equiv \langle CS_i, M''_i, L''_i, \mu''_i, (endpc, pc_{li}), (t''_{vi}, t_{li})\rangle) \wedge (C''_1 =_{\mathcal{L}} C''_2)$$

Proof. $C_i \longrightarrow C'_i$ and $C_1 =_{\mathcal{L}} C_2$.

By Lemma 5 either both configurations execute the UPCALL or neither does.

If they execute the UPCALL, the labels of r_{s1}, r_{s2}, r_{s3} and r_d flow to pc_l, which implies that $pc'_l, t'_l, endpc$ and R_d have equal respective values in both configurations. Additionally, both configurations will add a new call stack entry with the following form: $((endpc, pc_l), (endt_i, t_{li}))$. These call stack entries are low-equivalent: pc_l and $endpc$ are the same in both traces. If $t_l \in \mathcal{H}$, then $endt_1 \neq endt_2$ but their values need not be equal for low equivalence. If $t_l \in \mathcal{L}$, then $t_{v1} = t_{v2}$ and by low equivalence of R_d, $endt_1 = endt_2$.

Pushing low-equivalent entries onto low-equivalent call stacks preserves low equivalence: $CS'_1 =_{\mathcal{L}} CS'_2$.

Additionally, since pc'_l and t'_l are low-equivalent across configurations, $pc'_1 =_{\mathcal{L}} pc'_2$ Therefore, $C'_1 =_{\mathcal{L}} C'_2$.

Once the upcall is executing, no `dwncall` or `upcall` instructions can execute. Therefore, by Lemma 7, we can conclude that these configurations will only step to other low-equivalent configurations. Additionally, their call stacks will not be modified until they return from the upcall region via the UPRET-DONE rule.

At some point (when $t_{vi} = endt_i$) each configuration executes the UPRET-DONE rule. This may not happen after the same number of transition steps, and if $t_{li} \in \mathcal{H}$, it may not even happen at the same wall-clock time for each configuration.

When each configuration executes the UPRET-DONE rule, it pops the head of its call stack. We know that the original call stacks were low-equivalent, so popping off the heads results in the original call stacks.

Additionally, they will restore the original pc_l and t_l values, which were low-equivalent in the original configuration and both will set $pc_v = endpc$. Therefore, the `upcall` region executes without making any changes to state that violate low equivalence and the call stacks, pc and t_l restored by the UPRET-DONE are also low-equivalent.

Note that, if $t_l \in \mathcal{L}$, then end_t is also low-equivalent and these two configurations must execute the UPRET-DONE instruction at the same time. They may have executed a different number of instructions while in the call-gate region, but the wall-clock time will be identical. □

Theorem 2 (Noninterference Modulo Downgrading and Dwncalls)**.** *For any two valid configurations, C_1 and C_2, and any low set of labels, $\mathcal{L}$, where no instruction is a* `dwnlbl`, `dwncall` *or* `dwnret`.

$$(C_i \longrightarrow^* C'_i) \wedge (C_1 =_{\mathcal{L}} C_2) \implies C'_1 =_{\mathcal{L}} C'_2$$

Proof. This follows naturally from Theorem 1 and Lemma 8.

As long as the current instruction is not an `upcall` or `upret-done`, both configurations will step ($\longrightarrow$) to low-equivalent configurations, by Theorem 1.

If the current instruction is an `upcall` or the UPRET-DONE rule applies, then we must consider both the high and low pc_l cases.

***$pc_l \in \mathcal{L}$*.** By Lemma 5, `upcall` will fail or succeed in both configurations. Let us first consider what happens when `upcall` executes successfully.

If $pc'_l \in \mathcal{H}$ then by Lemma 8, both configurations will only step to low-equivalent states while in the upcall and will exit the upcall in low-equivalent states.

If $pc'_l \in \mathcal{L}$, then by Theorem 1 as long as no more `upcall` instructions execute, low equivalence is preserved. It is important to note that, by the UPCALL rule, $pc_l \sqcup t_l \sqsubseteq pc'_l$. In this case, it implies that $t_l \in \mathcal{L}$ and therefore the end time of the call gate will be exactly the same time in both configurations (same start and end, not just duration). If another UPCALL is executed while the upcall is running, it will fail in both configurations since they cannot be nested.

When the UPRET-DONE rule is applied, both configurations will step to low-equivalent configurations with equal pc_v and t_v. Since the implementation is deterministic (Property 1), the initial configurations are low-equivalent and the t_l in the call gate is exactly the same between both configurations, we are guaranteed that the same number of $\longrightarrow_{\mathcal{A}}$ steps execute in both configurations before the call gate expires at time $endt$. Without this guarantee, it would be possible for one upcall to exit having completed fewer instructions than the other and in that case their end states would not necessarily be low-equivalent.

$pc_l \in \mathcal{H}$. Either both configurations are inside a low-pc-originating upcall or neither is (by the same argument as in Lemma 6, regarding call stack low-equivalent prefixes). If they are executing inside an `upcall` region, then this reduces to a case in Lemma 8, which ensures they will eventually return outside of the upcall and will maintain low equivalence.

If neither one is in a low-pc-originating upcall, then their call stacks may differ: one of them may be in a upcall while the other configuration is not. However, we can apply both the corollary to Lemma 6 and Lemma 7 here to show that these configurations will step to low-equivalent configurations. The corollary states exactly the condition we have: if neither configuration is in a low-pc-originating upcall, and $pc_l \in \mathcal{H}$, then the resulting call stacks are low-equivalent. Lemma 7 ensures that all other state is noninterfering for high pcs. Therefore, high pcs are always noninterfering as long as the configuration is valid.

□

Corollary 4 (Timing-Sensitive-Noninterference Modulo Downgrading and Dwncalls). *For any two valid configurations, C_1 and C_2, and any low set of labels, $\mathcal{L}$, where no instruction is a* `dwnlbl`*,* `dwncall` *or* `dwnret` *and for all pc, $pc_l \in \mathcal{L} \implies t_l \in \mathcal{L}$.*

$$(C_i \longrightarrow^* C'_i) \wedge (C_1 =_{\mathcal{L}} C_2) \implies C'_1 =_{\mathcal{L}} C'_2$$

Proof. This is a direct corollary to Theorem 2 and is strictly stronger than Corollary 3 since it also allows the use of low-deterministic UPCALL instructions (i.e. no timing mitigation). □

Theorem 3 (Nonmalleable Information Flow). *For attacker induced high label sets $\mathcal{S}$ and $\mathcal{U}$ and their respective complements, $\mathcal{P}$ and $\mathcal{T}$, and valid configurations, $\forall \{s,u\} \in \{1,2\}, C_{su}$*

$$C_{su} \longrightarrow C'_{su} \;\wedge\; \forall u \in \{1,2\}.\, C_{1u} =_{\mathcal{P}} C_{2u} \;\wedge\; \forall s \in \{1,2\}.\, C_{s1} =_{\mathcal{T}} C_{s2}$$
$$\implies$$
$$C'_{11} =_{\mathcal{P}} C'_{21} \implies C'_{12} =_{\mathcal{P}} C'_{22} \quad\wedge\quad C'_{11} =_{\mathcal{T}} C'_{12} \implies C'_{21} =_{\mathcal{T}} C'_{22}$$

Proof. For all of the instructions other than `dwncall`/`dwnret` and `dwnlbl`, Theorem 2 implies the nonmalleability condition. Therefore, we only need to consider how the new instructions affect processor state.

Additionally, since the conditions are exactly dual we only prove the first of the two requirements:

$$C'_{11} =_{\mathcal{P}} C'_{21} \implies C'_{12} =_{\mathcal{P}} C'_{22}$$

First we consider the `dwnlbl` instruction. We have already proven that this instruction results in low-equivalent configurations for high pcs in Lemma 7. Therefore, if $pc_l \in \mathcal{S} \cup \mathcal{U}$, `dwnlbl` results in low equivalent configurations for both $\mathcal{L} = \mathcal{P}$ and $\mathcal{L} = \mathcal{T}$.

Therefore, we now consider the case where $pc_l \in \mathcal{P} \cap \mathcal{T}$. The `dwnlbl` instruction modifies the label of r_d ($l = L(r_d)$) to a new label value ($l' = \gamma(R_{s1})$). In the case where RELBL_ARG_ERROR rule matches, all configurations will step to the error pc and not modify any other state (by the same argument as for UPLBL_ARG_ERROR in Lemma 5). If that error rule does not match, l' is equivalent in all four configurations since $L(r_{s1})$ flows to pc_l and $pc_l \in \mathcal{P} \cap \mathcal{T}$.

Since l' is equivalent in all configurations, then the label checks: $pc_l \sqsubseteq l'$ and $l' \sqsubseteq \bigtimes(pc_l)$ will both succeed or fail in all four configurations. If they fail, then all four new configurations still maintain low equivalence since $L_{\mathbf{su}}$ is not modified in any of them.

Next let us consider the label check $l \sqsubseteq \bigtimes(l)$.

$L_1 =_{\mathcal{L}} L_2 \implies L_1 \approx L_2$ for any set of low labels, $\mathcal{L}$. Additionally, domain equivalence ($\approx$) is transitive. Therefore, all four starting configurations agree on the domain of all locations ($L_{\mathbf{11}} \approx L_{\mathbf{21}} \approx L_{\mathbf{12}} \approx L_{\mathbf{22}}$). We will now consider the four possible quadrants in which l can reside:

Case 1) $l \in \mathcal{P} \cap \mathcal{T}$

In this case, l is exactly the same in all four configurations. Therefore, label checking will succeed or fail in all four configurations and result in the same modifications to L (l' and r_d are also equal in all four configurations). The low equivalence relations between configurations will be maintained.

Case 2) $l \in \mathcal{P} \cap \mathcal{U}$

For any two public equivalent ($=_{\mathcal{P}}$) configurations then this will also result in the same label checking result and the same modifications to L. Any configurations which were public equivalent before this instruction are still public equivalent.

$$C_{\mathbf{1u}} =_{\mathcal{P}} C_{\mathbf{2u}} \implies C'_{\mathbf{1u}} =_{\mathcal{P}} C'_{\mathbf{2u}}$$

Case 3) $l \in \mathcal{S} \cap \mathcal{T}$

For any two trusted equivalent ($=_{\mathcal{T}}$) configurations then this will result in the same label checking result and same modifications to L. Any configurations which were trusted equivalent before this instruction are still trusted equivalent.

$$C_{\mathbf{s1}} =_{\mathcal{T}} C_{\mathbf{s2}} \implies C'_{\mathbf{s1}} =_{\mathcal{T}} C'_{\mathbf{s2}}$$

In this case, we know that $L'_{\mathbf{s1}} =_{\mathcal{T}} L'_{\mathbf{s2}}$ and we assume in the premise that $L'_{\mathbf{11}} =_{\mathcal{P}} L'_{\mathbf{21}}$. Therefore, $L'_{\mathbf{12}} \approx L'_{\mathbf{11}} \approx L'_{\mathbf{21}} \approx L'_{\mathbf{22}}$.

Next, we will consider the four possible quadrants for l':

Case 1) $l' \in \mathcal{P} \cap \mathcal{T}$

In this case, l' has the same value in each resulting configuration by transitivity of low-equivalent labels ($L'_{12} =_{\mathcal{P} \cap \mathcal{T}} L'_{22}$). Regardless of whether or not configurations $C_{\mathbf{12}}$ and $C_{\mathbf{22}}$ both successfully executed the `dwnlbl` instruction, they resulted in updating the same location to the same label value. Therefore, $C'_{\mathbf{12}}$ and $C'_{\mathbf{22}}$ are public equivalent.

Case 2) $l' \in \mathcal{P} \cap \mathcal{U}$

If this instruction succeeds in $C_{\mathbf{12}}$ it will modify both the $\mathcal{T}$ domain and the $\mathcal{P}$ domain (i.e. a label is going from high to low in confidentiality and low to high in integrity). However, if it fails then it will not modify either set of domains. Since $L'_{12} \approx L'_{22}$ this success or failure must be the same in both of these configurations (failure in one but not the other would imply $L'_{12} \not\approx L'_{22}$). Therefore it must either succeed or fail in both of these configurations. If it fails, L'_{12} and L'_{22} are trivially low-equivalent. If it succeeds, L'_{12} and L'_{22} are modified in the same way (since l' has the same value across all four configurations) and are still public equivalent.

Case 3) $l' \in \mathcal{S} \cap \mathcal{T}$

In this case, whether or not each configuration successfully executed the `dwnlbl` instruction has no bearing on the public equivalence of L'_{12} and L'_{22}. Since l' is a secret label, its exact value may differ in these two label mappings. Therefore, $C'_{\mathbf{12}}$ and $C'_{\mathbf{22}}$ remain public equivalent.

Case 4) $l' \in \mathcal{S} \cap \mathcal{U}$

By the same logic as in case 3, $C'_{\mathbf{12}}$ and $C'_{\mathbf{22}}$ remain public equivalent in this case as well.

Case 4) $l' \in \mathcal{S} \cap \mathcal{U}$

By Lemma 1, if $l \in \mathcal{S} \cap \mathcal{U}$, then l is compromised and the label check does not pass. Therefore, L is not updated in any configuration and the resulting configurations remain low-equivalent.

Dwncalls:

Now we show that the `dwncall` and `dwnret` instructions maintain nonmalleability.

Based on the pc_l of the starting configurations, there are 3 cases to prove, based on its quadrant (since pc_l is *valid* it cannot be in $\mathcal{S} \cap \mathcal{U}$, by Lemma 1).

Case 1) $\boldsymbol{pc_l} \in \mathcal{P} \cap \mathcal{T}$

First, if the $L(r_{s1}) \sqsubseteq pc_l$ check fails then this will fail in all configurations and low equivalence is preserved. Similarly, if the check that this call is going *down* in the lattice ($pc'_l \sqcup t'_l \sqsubset pc_l$) fails, this will fail in all four configurations.

Otherwise, pc' and t' will be the same in all four configurations, since they reference the same entry in the gate registry.

If the `dwncall` executes successfully in $C_{\mathbf{12}}$ because $CS_{\mathbf{12}}$ is empty, then $CS_{\mathbf{11}}$ must either also be empty or contain only public entries. By Lemma 2, it cannot contain any secret-untrusted entries and by trusted equivalence with $CS_{\mathbf{12}}$, it cannot contain any secret-trusted entries. By the premise ($C'_{\mathbf{11}} =_{\mathcal{P}} C'_{\mathbf{21}}$) $CS_{\mathbf{21}}$ must be empty if $CS_{\mathbf{11}}$ is empty and must otherwise contain public entries. By trusted equivalence, $CS_{\mathbf{22}}$ must also be either empty or contain only public entries. Since $CS_{\mathbf{12}} =_{\mathcal{P}} CS_{\mathbf{22}}$, the latter must also be empty.

This argument is symmetric (we could have started reasoning with $CS_{\mathbf{22}}$ and concluded that $CS_{\mathbf{12}}$ must be empty), the `dwncall` succeeds in $C_{\mathbf{12}}$ if and only if it succeeds in $C_{\mathbf{22}}$.

In the case where the instruction succeeds, both of the configurations will push on public equivalent call stack entries and will jump to the same new pc_l and pc_v based on the gate registry entry.

In any case where the instruction fails in both configurations, no state changes other than the pc_v and they will therefore remain low-equivalent.

Case 2) $\boldsymbol{pc_l} \in \mathcal{P} \cap \mathcal{U}$

If the $L(r_{s1}) \sqsubseteq pc_l$ check fails in $C_{\mathbf{12}}$ then it will also fail in $C_{\mathbf{22}}$. Similarly, if the check that this call is going *down* in the lattice ($pc'_l \sqcup t'_l \sqsubset pc_l$) fails in $C_{\mathbf{12}}$ it will also fail in $C_{\mathbf{22}}$.

Otherwise, pc' and t' will be the same in both configurations, since they reference the same entry in the gate registry.

The reasoning from Case 1 about call stack equivalence holds in this case as well since it does not depend at all on the current pc_l of the configurations. Therefore, $C_{\mathbf{12}}$ and $C_{\mathbf{22}}$ will either both fail or both succeed on this `dwncall` instruction. In the event the instruction succeeds, since their pc_l are public equivalent, the call stack entries that are pushed will be public equivalent as well, resulting in public equivalent configurations.

Case 3) $\boldsymbol{pc_l} \in \mathcal{S} \cap \mathcal{T}$

In this case, if either $C_{\mathbf{12}}$ or $C_{\mathbf{22}}$ successfully executes a `dwncall`, and the resulting pc labels (pc'_l) are both secret, then the resulting configurations are public equivalent. By Lemma 6 (with $\mathcal{L} = \mathcal{P}$ and $\mathcal{H} = \mathcal{S}$), the resulting call stacks are public equivalent. Similarly, the pc and t_l are public equivalent, since they remain secret.

Alternatively, either configuration may execute a `dwncall` such that $pc'_l \in \mathcal{P} \cap \mathcal{T}$. It is impossible for pc'_l to be untrusted, since `dwncall` requires that $pc'_l \sqsubseteq pc_l$.

In this case, $C_{\mathbf{11}}$ must also be empty. Since *CS **11** is* valid*, the label of its first entry must be ordered with respect to* pc_l*. Since no label in* $\mathcal{S} \cap \mathcal{T}$ *flows to any label in* $\mathcal{P} \cap \mathcal{U}$ *and no label in* $\mathcal{P} \cap \mathcal{U}$ *flows to any label in* $\mathcal{S} \cap \mathcal{T}$*, that first entry must not be public and untrusted. However, by trusted equivalence with CS* **12**, it cannot contain any secret entries either, and must therefore be empty.

By combining this with the call stack equivalence reasoning used in cases 1 and 2, this means that either all four call stack configurations are empty, or they all contain public, trusted entries.

Therefore, if $C_{\mathbf{12}}$ executes a `dwncall` successfully, then so must $C_{\mathbf{11}}$, since all other arguments to the instruction flow to the pc_l and are equal between the two configurations, and as reasoned above, both must have empty call stacks.

Since the call gate entries will be the same across $C_{\mathbf{12}}$ and $C_{\mathbf{11}}$ they will both end up with the same pc_v and pc_l in their new configurations. Similarly, they will push trusted equivalent call stack entries (whose labels are secret and trusted) and $C'_{\mathbf{12}} =_{\mathcal{T}} C'_{\mathbf{11}}$.

Furthermore, $pc'_l \in \mathcal{P} \cap \mathcal{T}$, which implies that $C'_{\mathbf{21}}$ will have the same pc'_l and pc'_v as configuration $C'_{\mathbf{12}}$, and will also have a secret, trusted call stack entry. Since $C_{\mathbf{21}}$ must have successfully executed the same `dwnlbl` instruction to end up in this state, by trusted equivalence $C_{\mathbf{22}}$ also must execute the same `dwnlbl` instruction and will also end up with the same new pc'_l and pc'_v as the other configurations. Lastly, since the new call stack entries are all secret and trusted, $CS'_{\mathbf{12}} =_{\mathcal{P}} CS'_{\mathbf{22}}$ because neither contains any public entries.

Dwnret:
This is essentially analogous to `dwncall` instruction, except we are restoring saved pc_v and pc_l from the stored call stack entries. Since the reasoning is so similar we will omit a full proof and include only a sketch.

Case 1) $pc_l \in \mathcal{P}$

In this case both $C_{\mathbf{12}}$ and $C_{\mathbf{22}}$ will either execute the `dwnret` or not, since the INUPCALL check must have the same result in both cases; if either configuration executes a `dwnret` successfully, then all four configurations must successfully execute a similar `dwnret`.

In the case where the call stack entry is public, then both configurations must have the same call stack entry and will end up in public equivalent configurations. If the call stack entry is secret and trusted, then by similar transitive arguments for `dwncall`, all four configurations will execute a `dwnret` to secret, trusted *pc*s and will all have empty call stacks.

Case 2) $pc_l \in \mathcal{S} \cap \mathcal{T}$

If $C_{\mathbf{12}}$ executes a `dwnret` successfully, the resulting pc'_l must also be secret and trusted. Similarly, its new call stack is empty and therefore still public equivalent to the call stack of $C_{\mathbf{22}}$ (which must have only contained secret entries, or no entries at all). In this case, $C'_{\mathbf{12}} =_{\mathcal{P}} C'_{\mathbf{22}}$ since it must also have a secret pc (the only instruction for lowering the pc_l is `dwncall` and by the earlier portion of this proof, it could not have executed a successful `dwncall`).

Similarly, if $C_{\mathbf{12}}$ executes a `dwnret` which does not pass label checking, the result will remain public equivalent with $C'_{\mathbf{22}}$.

□