HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security

Andrew Ferraiuolo
Cornell University
Ithaca, NY, USA

Mark Zhao
Cornell University
Ithaca, NY, USA

Andrew C. Myers
Cornell University
Ithaca, NY, USA

G. Edward Suh
Cornell University
Ithaca, NY, USA

ABSTRACT

This paper presents HyperFlow, a processor that enforces secure information flow, including control over timing channels. The design and implementation of HyperFlow offer security assurance because it is implemented using a security-typed hardware description language that enforces secure information flow. Unlike prior processors that aim to enforce simple information-flow policies such as noninterference, HyperFlow allows complex information flow policies that can be configured at run time. Its fine-grained, decentralized information flow mechanisms allow controlled communication among mutually distrusting processes and system calls into different security domains. We address the significant challenges in designing such a processor architecture with contributions in both the hardware architecture and the security type system. The paper discusses the architecture decisions that make the processor secure and describes ChiselFlow, a new secure hardware description language supporting lightweight information-flow enforcement. The HyperFlow architecture is prototyped on a full-featured processor that offers a complete RISC-V instruction set, and is shown to add moderate overhead to area and performance.

ACM Reference Format:
Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. 2018. HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security . In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243743

1 INTRODUCTION

Hardware plays a central role in modern software security. Protection rings are widely used to isolate supervisor processes from user processes. Recent hardware security architectures such as Intel SGX [11, 12] aim to protect critical software even when the operating system is malicious or compromised. However, the security of these processors relies on the assumption that the underlying hardware properly enforces necessary security properties.

Unfortunately, microprocessors often contain vulnerabilities that allow security-critical software to be compromised by untrusted software. Software-exploitable vulnerabilities have already been found in SGX [40] and also in the implementations of Intel VT-d [62] and system management mode (SMM) [61]. Moreover, the recent Spectre [32] and Meltdown [37] vulnerabilities show that even if the hardware is correct in a conventional sense — it implements its specification — correctness is not enough to ensure security. Timing channels in microprocessors do not violate the processor specification but can be used to leak information, and in the case of Meltdown, to read the entire memory of the kernel [37].

This paper presents HyperFlow, a processor architecture and implementation designed for timing-safe information-flow security. It is implemented as a full-featured processor with a complete (RISC-V) instruction set, extended with instructions for information flow control. The HyperFlow architecture is carefully designed to remove all disallowed information flows between security levels, including timing channels. The implementation of HyperFlow is verified at design time with a security-typed hardware description language (HDL), providing strong assurance about the security of its design and implementation.

The HyperFlow architecture is designed to support the rich software-level information-flow policies needed to build practical systems. It is standard for label models for information flow security to represent policies as lattices [16], which may change dynamically. HyperFlow uses a novel representation of lattices that can be implemented in hardware efficiently yet supports the full generality of many prior label models used in operating systems [8, 21, 67] and programming languages [1, 42, 47]. HyperFlow permits communication across security levels through controlled *downgrading* in its instruction-set architecture (ISA). Downgrading weakens noninterference but its potential for harm is limited by restrictions. These new features enable the HyperFlow ISA and prototype to support practical OS functionality such as interprocess communication and system calls.

While previous studies on tagged architectures have proposed to enforce information flow security at the hardware level [20, 68], these architectures do not prevent timing channels. Further, their implementations have not been statically checked with a security-typed HDL, and their ISAs have not been designed to be amenable to information-flow verification. On the other hand, prior processors with verified information flow security [26, 35, 36, 56–58] only support simple, fixed security policies, and do not support the expressive security policies needed for practical applications. Further, none of these prior processors support cross-domain communication, needed for interprocess communication (IPC) and system calls, and none constrain the use of downgrades.

We implemented HyperFlow as an extension to the RISC-V Rocket processor, which supports a complete RISC-V ISA with the features necessary to run an operating system. Our prototype implementation shows that the HyperFlow architecture and the

The first two authors are now at Google and Stanford University respectively.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243743

security type system in ChiselFlow can be scaled to complex, full-featured processors. The prototype implementation shows that the new security features in HyperFlow add moderate area overhead, largely due to the additional storage for security tags, and moderate performance overhead due to timing-channel protection.

Some core technical contributions enable high-assurance construction of a practical processor with timing-safe information-flow security:

Security policies and hypercube labels. Prior processors with verified information flow [26, 35, 36, 56–58] only support simple, fixed 2-point or 4-point policy lattices. However, work on information-flow security in operating systems and programming languages suggests that real applications need rich lattice policies that can capture complex trust relationships relating mutually distrusting principals and involving both confidentiality and integrity [8, 21, 42, 67].

Most prior label models for information flow security represent policies using lattices of information flow labels. To support such rich software labels, we introduce a novel representation of lattices over bit vectors, which we call the *hypercube label model*. In this model, software-level labels are mapped to points in a hypercube. Hypercube labels enable efficient comparisons and computations of security levels directly in hardware, and they are amenable to static checking in the security-typed HDL.

We also show how to use the hypercube label model to encode policies expressed in the recently proposed FLAM label model [1] for decentralized information flow control. By translating FLAM labels into hypercube labels, we show that the hypercube label model can flexibly enforce software-defined information flow policies.

Controlled downgrading. To be practical, systems based on information-flow security must allow exceptions to noninterference [28]. Downgrading can usefully relax information-flow restrictions, but uncontrolled downgrading is dangerous. HyperFlow therefore provides *controlled* downgrading. Downgrading of confidentiality policies (declassification) is permitted only when it is *robust* [64] — secrets can be released only if the downgrade can be influenced only by their owners. Dually, downgrading of integrity policies (endorsement) is permitted only when it is *transparent* [6] — endorsed data must be readable by its provider. Together, these conditions ensure that information flow is *nonmalleable*. Nonmalleable information flow [6] is enforced not only at the ISA level but also at the HDL level, strengthening assurance about the implementation.

Secure interprocess communication. Another novel and challenging feature of HyperFlow is its support for secure communication across trust domains. HyperFlow allows but constrains IPC via shared memory. It also supports the secure communication via registers for arguments and return values of system calls.

System calls and shared function libraries present another challenge that HyperFlow addresses — both scenarios require a mechanism by which untrusted code can invoke trusted code. HyperFlow provides secure cross-domain control transfers by extending *call gates* [51, 60] with information flow control.

Tagged architecture for memory protection. Conventionally, virtual memory isolates pages belonging to different applications. However, hardware support for virtual memory is complex and its correctness also depends on other mechanisms such as cache coherence, which is notoriously difficult to implement correctly. HyperFlow augments conventional memory protection with *security tags* associated with each physical page (or frame) of memory. Security tags are mapped to hypercube labels using a mapping defined by the operating system; accesses to memory are then mediated using hypercube labels. The security of this mechanism is checked in the HDL code at design time.

ChiselFlow security-typed HDL. To provide strong information-flow assurance for the construction of HyperFlow, we designed a security-typed HDL called ChiselFlow. ChiselFlow is embedded in Scala, so it inherits the expressiveness of a complex, full-featured language. But ChiselFlow compiles to a small intermediate language that is responsible for the enforcement of security policies, so its trusted component is small. Unlike prior secure HDLs, ChiselFlow provides label inference that reduces programmer effort. The hardware designer provides security labels for the inputs and outputs of hardware modules, but labels of internal signals can be omitted. ChiselFlow also supports multiple mechanisms for describing heterogeneously labeled data structures, which are crucial for practical designs. ChiselFlow also permits downgrades in the HDL syntax, yet constrains them to be robust and transparent, like the ISA of HyperFlow. ChiselFlow is the first HDL for information flow control with controlled downgrading.

In summary, HyperFlow offers a new approach to processor design, in which protection is based on flexible information flow policies rather than standard memory protection, and in which microarchitectural side channels are controlled by implementation using an expressive HDL.

2 SECURITY GOALS AND THREAT MODEL

The goal of HyperFlow is to provide strong timing-safe information flow security in hardware while supporting flexible application-defined information flow policies. A software-defined security policy is expressed as a lattice of security labels that encode information flow constraints for both confidentiality and integrity. The HyperFlow ISA allows security labels to be assigned to software processes running on a processor as well as to state elements such as registers and memory pages. HyperFlow constrains information flow according to the label lattice, but to be practical, it allows downgrading that weakens noninterference, while making all downgrading explicit and nonmalleable [6].

HyperFlow aims to enforce timing-safe information flow security by preventing microarchitectural timing channels that influence the number of CPU clock cycles taken between events observable by the adversary. It therefore considers a strong adversary that can measure timing at cycle granularity. However, we do not consider attacks that require physical proximity, such as physical tampering with hardware or side/covert channels through physical media such as power consumption, electromagnetic fields, or sound.

We assume that there is a trusted label manager that assigns security labels to processes and storage in a way that prevents untrusted applications from harming other applications. A malicious application may execute arbitrary instructions in the HyperFlow ISA but should not be able to violate the nonmalleable information flow control (NMIFC) policy set by the trusted label manager.

The eventual goal of the HyperFlow project is to provide strong formal security assurance for the instruction set architecture (ISA). Ideally, we would prove formally that programs written in the HyperFlow ISA enforce an extensional security property such as noninterference [28] or nonmalleable information flow [6]. To obtain a theorem regarding adversaries that can exploit timing channels, this proof would need to consider not only the ISA, but also low-level details of the microarchitecture implementation. As a first step toward a proof about the ISA and its implementation, we have implemented the HyperFlow processor using an HDL that enforces hardware-level information flow security. We have proved that aside from downgrades, hardware implemented in this HDL enforces a gate-level formulation of noninterference. We leave as future work a proof that the secure HDL enforces nonmalleable information flow at the gate level, and extending this proof to the ISA level.

This paper shows how different aspects of the design and implementation of HyperFlow combine to meet this security goal. The hypercube labels (Section 3) show how arbitrary lattice-model policies can be encoded and expressed in hardware. Section 4 describes the HyperFlow ISA, which specifies how these policies are presented to and enforced by hardware. Section 6 shows how the ISA can be realized while eliminating microarchitectural timing channels. To provide strong assurance that this microarchitecture is timing-safe, it is constructed in the ChiselFlow HDL, which is described in Section 5.

3 SECURITY POLICIES IN HYPERFLOW

HyperFlow enforces information flow security policies directly in hardware. Prior work on label models for information flow security in software support rich policies allowing mutually distrusting principals to communicate [8, 21, 42, 67]. These label models represent policies using lattices of information flow labels. Here, we discuss how general lattice-based policies can be expressed with bit vectors in a way that allows efficient computations and comparisons of labels in hardware.

3.1 Confidentiality and integrity policies

An information-flow label $\ell = (c, i)$ in HyperFlow is a pair of a confidentiality level c and an integrity level i. Confidentiality and integrity levels in HyperFlow both form lattices that are ordered by $\sqsubseteq_C$ and $\sqsubseteq_I$ respectively. The ordering on confidentiality levels specifies constraints on secrecy; if $c \sqsubseteq_C c'$, then c is no more confidential than c'. Similarly, if $i \sqsubseteq_I i'$, then i is at least as trustworthy as i'. The ordering of integrity levels and confidentiality levels is dual: high confidentiality levels are more restrictive than low ones, whereas low integrity levels are more restrictive than high ones. The orderings on confidentiality and integrity levels are lifted to a lattice of labels $\sqsubseteq$; if $c \sqsubseteq_C c'$ and $i \sqsubseteq_I i'$ then $(c, i) \sqsubseteq (c', i')$. We write $C(\ell)$ and $I(\ell)$ to denote just the confidentiality or integrity part of the label respectively.

3.2 Lattices via bit vectors

To support efficient computations and comparisons of labels in hardware, HyperFlow represents lattices over bit vectors. We first explain the ordering of confidentiality levels. Levels are mapped to a point in a hypercube, expressed using a bit vector. A bit vector b is split into D dimensions, each of K bits. Bit vectors are then functions from $[1, D]$ to $[0, 2^K - 1]$, and the notation $b(i)$ represents the value in the i^{th} dimension of b. Bit vectors b_1 and b_2 are ordered in the confidentiality lattice, written $b_1 \sqsubseteq_C b_2$ if each dimension of b_1 is numerically less than or equal to the corresponding element of b_2, as shown in Figure 1. As an example, if b_1 and b_2 are each bit vectors of four 2-bit dimensions, and b_1 is `10100111` and b_2 is `10010010`, then $b_2 \sqsubseteq_C b_1$. The join ($\sqcup_C$) and meet ($\sqcap_C$) of two confidentiality components are respectively computed by taking the maximum or the minimum over the corresponding dimensions of each vector. The integrity lattice ordering is exactly dual to that for confidentiality, as shown in Figure 2: $b_1 \sqsubseteq_C b_2 \iff b_2 \sqsubseteq_I b_1$.

$$b_1 \sqsubseteq_C b_2 \triangleq \forall d \in [1, D], b_1(d) \leq b_2(d)$$
$$(b_1 \sqcup_C b_2)(d) \triangleq \max\{b_1(d), b_2(d)\}$$
$$(b_1 \sqcap_C b_2)(d) \triangleq \min\{b_1(d), b_2(d)\}$$

Figure 1: Confidentiality ordering over bit vectors.

$$b_1 \sqsubseteq_I b_2 \triangleq \forall d \in [1, D], b_1(d) \geq b_2(d)$$
$$(b_1 \sqcup_I b_2)(d) \triangleq \min\{b_1(d), b_2(d)\}$$
$$(b_1 \sqcap_I b_2)(d) \triangleq \max\{b_1(d), b_2(d)\}$$

Figure 2: Integrity ordering over bit vectors.

We write $(c, i) \sqcup (c', i') \triangleq (c \sqcup_C c', i \sqcup_I i')$ and $(c, i) \sqcap (c', i') \triangleq (c \sqcap_C c', i \sqcap_I i')$ to denote the join and meet over labels, respectively. We use $\top$ and $\bot$ to denote a sequence of all `1`'s and all `0`'s, respectively. In the confidentiality ordering, $\top$ and $\bot$ are completely secret and completely public respectively; in the integrity ordering, $\top$ and $\bot$ are completely trusted and completely untrusted respectively. Hence, the labels $(\bot, \top)$ and $(\top, \bot)$ are the least and most restrictive labels in the information-flow ordering ($\sqsubseteq$).

Other representations of lattices in computer systems have been studied in the past [27]. Because HyperFlow controls information flows via timing channels, lattice comparisons and computations need to be done throughout the implementation, making it particularly important to be able to efficiently update and compare labels directly in hardware. Some prior representations of lattices, such as adjacency lists and matrices, are less space-efficient. Approaches that use opaque identifiers and cache recent operations [20, 68] require software intervention for lattice operations, potentially creating timing channels. The hypercube lattice is most similar to the skeletal representation, also known as the Fidge and Mattern vector clock [31]. Vector clocks have not been used to represent lattices in hardware in prior work.

3.3 Nonmalleable downgrading

Systems for information flow control are often intended to enforce noninterference, which prevents all information flows that violate lattice policies. However, noninterference is too restrictive for practical systems. For example, data computed using secrets may eventually need to be released publicly. Noninterference may be weakened through *downgrading*, which relaxes information flow labels. Downgrading that weakens confidentiality is said to *declassify* whereas downgrading that weakens integrity is said to *endorse* [66].

Because downgrading weakens noninterference, effort has been made to constrain downgrading to limit its potential to cause harm [50]. In this work, we permit communication that weakens noninterference as long as the downgrading it causes is *nonmalleable* [6]. Nonmalleable information flow subsumes two security conditions, robust declassification and transparent endorsement. These security conditions constrain downgrading but have not been enforced previously by hardware.

Robust declassification [64] only permits information to be downgraded by parties that have authority over that information. As in prior work on defining robust declassification [6, 9], authority (trust, privilege) is represented by integrity; only a principal at least as trusted as $I(p)$ can declassify data with confidentiality $C(p)$. This constraint is useful for decentralized systems such as microkernels. A principal A can declassify its data to a principal B, and as long as B does not have integrity $I(A)$, B can observe A's data but is prevented from releasing it elsewhere.

In HyperFlow, a process with label ℓ_{cur} can declassify a label ℓ to ℓ' only if the following condition holds:

$$C(\ell) \sqsubseteq_C C(\ell') \sqcup_C \left(I(\ell_{cur}) \sqcup_I I(\ell)\right)$$

This condition follows directly from prior work on defining robust declassification in the context of programming languages [6, 9]. Intuitively, it allows the confidentiality $C(\ell)$ of the data being declassified to be compensated for by the integrity of the data and of the current process. Hence, only sufficiently trusted processes can influence whether or not secrets are declassified. When ℓ can be robustly declassified to ℓ' by a process with label ℓ_{cur}, we write $\ell \xrightarrow[\ell_{cur}]{C} \ell'$. Notably, the condition includes a confidentiality join involving an integrity label component; this condition is well-defined because $\sqcup_C$ is an operation defined over bit vectors, and regardless of whether the vectors represent confidentiality or integrity.

The dual of robust declassification is transparent endorsement [6]. It sets a maximum confidentiality on endorsements to prevent opaque writes that could enable attacks. A write is opaque if a principal could have written data but not read it. In HyperFlow, a process with label ℓ_{cur} can endorse a label ℓ to ℓ' if,

$$I(\ell) \sqsubseteq_I I(\ell') \sqcup_I \left(C(\ell_{cur}) \sqcup_C C(\ell)\right)$$

This condition follows directly from prior work on defining transparent endorsement for a functional programming language [6]. When ℓ can be transparently endorsed to ℓ' by a process with label ℓ_{cur}, we write $\ell \xrightarrow[\ell_{cur}]{I} \ell'$. When $\ell \xrightarrow[\ell_{cur}]{I} \ell'$ and $\ell \xrightarrow[\ell_{cur}]{C} \ell'$ we say that ℓ can be nonmalleably downgraded to ℓ' by a process with label ℓ_{cur} and we write $\ell \xrightarrow[\ell_{cur}]{} \ell'$.

4 THE HYPERFLOW ARCHITECTURE

Policies in the HyperFlow ISA are defined using the hypercube labels described in Section 3. HyperFlow is realized as a tagged architecture in which security labels are explicitly represented as hardware tags for processes, registers, and memory pages. HyperFlow augments conventional memory protection enforced by virtual memory with security tags that are associated with each physical page (or frame) of memory. Tagged physical memory enables static checking of information flow with the security type system of ChiselFlow. Traditional virtual memory does not ensure noninterference; it is possible for the same physical page to be mapped to virtual addresses owned by distrusting processes. Even if the mapping did ensure noninterference, it would not be possible to prove noninterference purely by inspecting the hardware design because the mapping is software-defined. By simultaneously supporting isolation based on both virtual memory and information flow, HyperFlow supports incremental adoption.

4.1 Security labels

Processes executing in HyperFlow are associated with a security label, ℓ_{cur}. The label $C(\ell_{cur})$ represents the highest secrecy that the process can observe, and $I(\ell_{cur})$ represents the most trusted information it can affect.

In addition to authorizing explicit data access, ℓ_{cur} also constrains data leakage through control flow, instruction fetches, and timing channels. HyperFlow also associates a security label (ℓ_r) with each general-purpose register (r). Similarly, a page (m) in physical memory has a security label $\mathcal{M}_\ell(m)$.

The idea of dynamically associating processes with a current security label that constraining information flow is quite old [5]. In an approach suggested by the Orange Book [18], as well as in many recent systems (e.g., [21, 29]), processes have *floating labels* that change as processes access sensitive information. By contrast, HyperFlow is a fixed-label system, in which neither the current process label, nor the labels of registers, change unless the process explicitly changes them. The fixed-label approach has also been taken by IFC-based operating systems [33, 67], for the same reason: labels that change can be an information channel.

HyperFlow uses information-flow security to authorize reads and writes to memory. In order for the currently executing process to read a page of memory, m, we require $\mathcal{M}_\ell(m) \sqsubseteq \ell_{cur}$, where $\mathcal{M}_\ell$ is a mapping from pages to their information flow labels. Similarly, to write to m, we require that $\ell_{cur} \sqsubseteq \mathcal{M}_\ell(m)$.

HyperFlow associates security labels with registers to facilitate two kinds of communication that are needed in processors: 1) communication between user-space applications and the operating system during system calls, and 2) interprocess communication in memory. During system calls, arguments and return values are communicated between the application and system call handler via registers. HyperFlow permits communication using registers by associating labels with registers and through instructions that downgrade registers' labels. Assuming an application is untrusted, the trusted call handler can endorse the registers storing the arguments after inspecting them. At the end of the system call, the call handler can declassify the registers storing the return values before returning to the application.

Because information flow labels are used to enforce security, HyperFlow must ensure that the labels accurately reflect the security of the data they protect. To store the content of register r to an address in memory page m, we require that the precondition $\ell_r \sqsubseteq \mathcal{M}_\ell(m)$ hold. Similarly, loading a word from m into r requires $\mathcal{M}_\ell(m) \sqsubseteq \ell_r$.

However, these security preconditions sometimes interfere with necessary communication among distrusting principals. To provide a familiar software interface, HyperFlow permits interprocess

communication among distrusting principals via shared memory. Shared-memory reads and writes that would violate noninterference require downgrading. HyperFlow supports downgrading at the granularity of an individual word with downgrading load and store instructions. They work just like conventional loads and stores but downgrade data as it is copied. HyperFlow also supports page downgrades for zero-copy sharing of entire pages.

The instructions fetched by the current process come from labeled memory, which has security implications. A process should not execute low-integrity instructions because then untrusted adversaries might influence the executed code. Conversely, confidentiality can be violated by code that depends on secrets, including control decisions based on secrets. Information leaks through control flow are called *implicit flows*. HyperFlow prevents implicit flows because ℓ_{cur} includes the security label of the fetched instruction and any control flow decisions. Branches cannot depend on a register r unless $\ell_r \sqsubseteq \ell_{cur}$. For all instructions that write to a register r, the precondition $\ell_{cur} \sqsubseteq \ell_r$ ensures that the label of ℓ_r securely reflects its influences.

4.2 Information-flow call gates

The restriction on branch conditions and on writes to registers together prevent an untrusted or secret process from invoking code that is trusted or public. However, untrusted applications need to be able to call trusted code when making system calls, and secret applications need to be able to call public functions from libraries. HyperFlow supports control transfers of this form with an extended form of *call gate* [51]. Call gates in HyperFlow tightly couple the entry point (program counter) that initiates the code with an information-flow label representing the privilege level of that code. A process at level ℓ_{cur} can register a call gate at level ℓ' as long as $\ell_{cur} \sqsubseteq \ell'$. Another process can then invoke a call gate, at which point the program counter is set to the entry point of the gate, and ℓ_{cur} is set to the level at which the gate was registered. To allow protected returns from call gates, invoking a call gate also pushes the previous program counter value and level of ℓ_{cur} onto a hardware stack. The executing process can then invoke a return instruction to pop the stack, jumping to the old pc value and privilege level.

Uniquely, call gates in HyperFlow replace conventional hierarchical privilege levels with lattice-based information-flow labels. By generalizing privilege levels, HyperFlow securely supports control transfers with fewer privilege changes than in a conventional processor while simultaneously providing more fine-grained separation of privilege. For example, a network driver might register a call gate at a security level ℓ_{net} that is incomparable with other system labels. When an application sends a packet over the network, it can directly invoke the call gate, transferring immediately to ℓ_{net}. In a conventional processor, the network driver can either run in supervisor mode, in which case the application implicitly trusts the entire kernel, or the network driver can run in user space. In the second case, the application must first make a system call causing a transition to supervisor mode before the kernel delegates to the user-space driver. In this case, the application must trust the kernel to delegate to the driver and also incur performance penalty because of the extra privilege changes.

4.3 Current label bounds

Using just a single level, ℓ_{cur}, for a given process should be sufficient for many applications — particularly, legacy applications that do not use information flow labels internally. However, other applications may require the ability to operate on data at multiple security levels. To make the label of executed instructions more flexible, HyperFlow allows the active process to move the level of ℓ_{cur} within a space of labels bounded between ℓ_{lwr} and ℓ_{upr}. When setting the value of ℓ_{cur}, we require $\ell_{lwr} \sqsubseteq \ell_{cur} \sqsubseteq \ell_{upr}$. Label $C(\ell_{upr})$ represents the most secret information that the process can observe, and $I(\ell_{upr})$ is the least trustworthy information it can be influenced by. On the other hand, $C(\ell_{lwr})$ and $I(\ell_{lwr})$ represent the least secrecy the process can claim it has observed and the most trusted information that the process can affect. Upper bounds on information flow labels within process are known as *clearance labels* [54, 67].

4.4 Instruction set extensions

HyperFlow introduces new instructions as well as new control and status registers. Security levels in HyperFlow are represented as a pair of confidentiality and integrity components, as described in Section 3. Levels ℓ_{lwr}, ℓ_{cur}, and, ℓ_{upr} are each stored in control and status registers (CSRs) and are accessed with conventional CSR instructions. To prevent a process from circumventing its own bounds, the bounds can only be modified when the processor is in the most public and trusted level, that is $\ell_{cur} = (\bot, \top)$. However, ℓ_{cur} can be modified by a process as long as $\ell_{lwr} \sqsubseteq \ell_{cur} \sqsubseteq \ell_{upr}$.

Table 1 shows the new instructions added in HyperFlow. The first column gives the instruction name and operands, the second column describes the preconditions for the instruction, and the third column describes the instruction's effect.

The instructions `DECLREG` and `ENDOREG` downgrade registers. The `DECLREG` instruction declassifies the value stored in r_1 to the confidentiality level stored in r_2, but it permits the declassification only if it is robust. The first restriction prevents implicit flows by ensuring that ℓ_{cur} can write to the new level of r_1. The second restriction ensures that r_1 can be robustly declassified from ℓ_{r_1} to $(r_2, I(\ell_{r_1}))$.

The third restriction is more subtle — it prevents potential information flow violations that might be caused by the use of r_2 as a label. In general, confidentiality or integrity can be affected by information flow via labels represented at run time. Inspecting labels releases information, and if the labels are not trusted, it is tricky to rely on them for security. Type systems can control such information flow via labels [41, 71], but we do not assume that HyperFlow programs are statically checked. Hence, as in the Breeze language [29], we treat register labels and memory labels as fully public and fully trusted. Thus, because `DECLREG` allows the value stored in r_2 to influence a label, it must be permitted to influence fully public and trusted data.

A natural way to ensure this influence is to simply require that $\ell_{r_2} = (\bot, \top)$. However, this restriction would often require extra instructions to first downgrade r_2 before downgrading r_1. Instead, we enforce a less restrictive, but equally secure condition — it must be possible to downgrade r_2 to $(\bot, \top)$ using robust declassifications and transparent endorsements. This relaxed restriction does not weaken security because when the restriction on the label of r_2 holds, it is always possible to first downgrade the label of r_2. The

Instruction	Restrictions	Behavior
DECLREG R1, R2	$\ell_{cur} \sqsubseteq (r_2, I(\ell_{r_1}))$ $\quad \ell_{r_1} \xrightarrow[\ell_{cur}]{C} (r_2, I(\ell_{r_1}))$ $\quad \ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$C(\ell_{r_1}) \leftarrow r_2$
ENDOREG R1, R2	$\ell_{cur} \sqsubseteq (C(\ell_{r_1}), r_2)$ $\quad \ell_{r_1} \xrightarrow[\ell_{cur}]{I} (C(\ell_{r_1}), r_2)$ $\quad \ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$I(\ell_{r_1}) \leftarrow r_2$
RSTREG R1	None	$\ell_{r_1} \leftarrow \ell_{cur}$ $r_1 \leftarrow 0$
LWDWN R2, IMM(R1)	$\mathcal{M}_\ell(r_1 + \text{IMM}) \xrightarrow[\ell_{cur}]{} \ell_{r_2}$ $\mathcal{M}_\ell(r_1 + \text{IMM}) \xrightarrow[\ell_{cur}]{} \ell_{cur}$	$r_2 \leftarrow \mathcal{M}(r_1 + \text{IMM})$
SWDWN R2, IMM(R1)	$\ell_{r_2} \sqcup \ell_{cur} \xrightarrow[\ell_{cur}]{} \mathcal{M}_\ell(r_1 + \text{IMM})$	$\mathcal{M}(r_1 + \text{IMM}) \leftarrow r_2$
SETMEM R2, IMM(R1)	$\ell_{cur} = (\bot, \top)$	$\mathcal{M}_\ell(r_1 + \text{IMM}) \leftarrow r_2$ $\mathcal{M}(r_1 + \text{IMM}) \leftarrow 0$
DECLMEM R2, IMM(R1)	$\ell_{cur} \sqsubseteq (r_2, I(\mathcal{M}_\ell(r_1 + \text{IMM})))$ $\mathcal{M}_\ell(r_1 + \text{IMM}) \xrightarrow[\ell_{cur}]{C} (r_2, I(\mathcal{M}_\ell(r_1 + \text{IMM})))$ $\ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$ $\quad \ell_{r_1} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$C(\mathcal{M}_\ell(r_1 + \text{IMM})) \leftarrow r_2$
ENDOMEM R2, IMM(R1)	$\ell_{cur} \sqsubseteq (C(\mathcal{M}_\ell(r_1)), r_2)$ $\mathcal{M}_\ell(r_1 + \text{IMM}) \xrightarrow[\ell_{cur}]{I} (C(\mathcal{M}_\ell(r_1 + \text{IMM})), r_2)$ $\ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$ $\quad \ell_{r_1} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$I(\mathcal{M}_\ell(r_1 + \text{IMM})) \leftarrow r_2$
REGLGATE R1, R2	$(\ell_{cur} \sqcup \ell_{r_1} \sqcup \ell_{r_2}) \sqsubseteq r_2$ $\quad \ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$ $\quad \ell_{r_1} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$T[r_1] \leftarrow r_2$
LCALL IMM	None.	$S \leftarrow S :: (\text{pc} + 4, \ell_{cur}, \ell_{lwr}, \ell_{upr})$ $\quad \text{pc} \leftarrow \text{pc} + \text{IMM}$ $\ell_{cur} \leftarrow T[\text{pc} + \text{IMM}]$
LCALLR IMM(R1)	$\ell_{r_1} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$S \leftarrow S :: (\text{pc} + 4, \ell_{cur}, \ell_{lwr}, \ell_{upr})$ $\quad \text{pc} \leftarrow r_1 + \text{IMM}$ $\ell_{cur} \leftarrow T[r_1 + \text{IMM}]$
LRET	None.	$(pc, \ell_{cur}, \ell_{lwr}, \ell_{upr}) \leftarrow \text{tail(S)}$ $\quad S \leftarrow \text{head(S)}$
SETBOUNDS	$\ell_{cur} = (\bot, \top)$	$\ell_{cur} \leftarrow \ell_{ncur}$ $\quad \ell_{lwr} \leftarrow \ell_{nlwr}$ $\quad \ell_{upr} \leftarrow \ell_{nupr}$

Table 1: New instructions added in HyperFlow.

ENDOREG instruction works similarly, but it endorses rather than declassifies the value stored in r_1.

The register labels of HyperFlow resemble tags in an information flow tracking architecture [14, 55], but these processors typically have floating labels, propagating tags automatically. For example, following an ADD RS3, RS2, RS1 instruction, we would like to compute the join of the labels of RS1 and RS2 and store the result in RS3 without needing explicit instructions to set the tag of RS3. However, when and whether the labels of general purpose registers are updated both depend on control signals that are labeled with ℓ_{cur}, but the updated labels are public and trusted. In other words, the act of dynamically updating security tags causes information flow from ℓ_{cur} to public-and-trusted. Because HyperFlow aims to eliminate timing channels, tag updates must be done by explicit instructions such as DECLREG and ENDOREG. Automatic tag updates can potentially be inserted by the compiler.

The instruction RSTREG allows a process to reclaim a register without downgrading by setting the level of the register r_1 to ℓ_{cur}. In order to avoid possibly downgrading the value stored in r_1, r_1 is cleared. Because this instruction takes no arguments other than r_1 and it happens unconditionally, it has no preconditions.

The LWDWN instruction works like a normal load-word instruction but relaxes restrictions on labels. It permits the load if the label of the source page could be downgraded to the label of the destination register, and to ℓ_{cur}. Similarly, SWDWN works like a store instruction that permits the store if the register could be downgraded to the label of the destination page. Both instructions are useful for interprocess communication via shared memory.

Memory labels can be reset by totally trusted and public software via a SETMEM instruction, which takes two arguments: the page-aligned physical address in register r_1 and a new label in r_2. The SETMEM instruction can only be executed when $\ell_{cur} = (\bot, \top)$. Trusted software that uses this instruction can clear the contents of the page to avoid implicit downgrades.

Entire pages can also be declassified/endorsed by user-space applications through the DECLMEM and ENDOMEM instructions, which are similar to SETMEM except that they require the changes in memory labels to be robust/transparent as in DECLREG and ENDOREG. As with DECLREG and ENDOREG, information flow violations through labels are also prevented by requiring that the arguments that influence labels can be downgraded securely.

The REGLGATE instruction registers a new call gate with a pc value of r_1 and a label of r_2 by adding it to a table T that records call gates by mapping pc values to labels. The first restriction, $(\ell_{cur} \sqcup \ell_{r_1} \sqcup \ell_{r_2}) \sqsubseteq r_2$, checks that the process creating the gate and the arguments from which the gate is constructed are no more secret and are at least as trusted as the label of the gate. The entries in the call gate table are public and trusted (though the labels of individual gates may be more restrictive), because processes that attempt to use call gates must be able to see whether or not they exist. Therefore, the last two restrictions check that the active process can downgrade the register arguments to public and trusted because they influence the creation of a call gate entry.

The LCALL and LCALLR instructions execute a call gate and have the same instruction formats as conventional JAL and JALR instructions. The LCALL instruction specifies the call-gate entry point with an immediate that is added to the current pc value, whereas the LCALLR instruction specifies the entry point by adding an immediate to a register argument. For both instructions, if the specified entry point is found in the call gate table, the address of the instruction following the call and the value of ℓ_{cur} prior to the call are pushed onto a hardware stack S. The processor then sets the pc value to the entry point of the gate and sets ℓ_{cur} to the label of the gate. If the gate does not exist, the instruction is converted to a NOP. The instruction LRET pops the stack S and returns to the most recent pc value and label.

When a call gate terminates with an LRET instruction, information is potentially leaked — by returning, the gate leaks when

Instruction Type	Invariant
Load instructions	$\ell_{r_a} \sqcup \mathcal{M}_\ell(m) \sqcup \ell_{cur} \sqsubseteq \ell_{r_d}$ $\wedge \mathcal{M}_\ell(m) \sqsubseteq \ell_{cur}$
Store instructions	$\ell_{r_a} \sqcup \ell_{r_v} \sqcup \ell_{cur} \sqsubseteq \mathcal{M}_\ell(m)$
Execute unit	$\ell_{r_{s1}} \sqcup \ell_{r_{s2}} \sqcup \ell_{cur} \sqsubseteq \ell_{r_d}$
Value-dependent branches	$\ell_{r_{s1}} \sqcup \ell_{r_{s2}} \sqsubseteq \ell_{cur}$
All instructions	$\mathcal{M}_\ell(m_i) \sqsubseteq \ell_{cur}$

Table 2: Instruction preconditions

and whether or not it has finished executing. Indeed, this is represented by a single microarchitectural downgrade, as described in Section 7.1. To avoid leaks, the execution time of the gate should not depend on secrets. One way to accomplish this would be for gates to act as execution leases [56] by forcing a return at a predetermined time. A less error-prone alternative is for the call gate to use timing mitigation [53, 69], restricting the set of possible execution time so that the channel capacity of the leak is acceptably limited. To implement mitigation, a gate can be begun with a timer read and delayed until at least a predetermined value before returning. ISA mitigation support is a natural future extension.

Finally, the SETBOUNDS instruction permits software that is fully public and fully trusted to set the label bounds by atomically copying CSRs ℓ_{ncur}, ℓ_{nlwr}, ℓ_{nupr} to ℓ_{cur}, ℓ_{lwr}, and ℓ_{upr}. Atomicity is needed because writing individually to bound registers could temporarily violate the invariant $\ell_{lwr} \sqsubseteq \ell_{cur} \sqsubseteq \ell_{upr}$.

4.5 Semantic changes to existing instructions

Existing RISC-V instructions are also modified so that they enforce information flow restrictions. Several preconditions must hold for each instruction that is executed, depending on the instruction. Table 2 summarizes these preconditions, which serve two purposes: 1) to implement memory protection, and 2) to ensure that the labels of registers and memory pages accurately capture the confidentiality and integrity of their data.

Memory protection is enforced by ensuring that when a process with label ℓ_{cur} loads from a page m, the condition $\mathcal{M}_\ell(m) \sqsubseteq \ell_{cur}$ holds, preventing reads that would violate security. On stores to m, we require $\ell_{cur} \sqsubseteq \mathcal{M}_\ell(m)$, which is subsumed by the precondition enforced by store instructions, listed in the table.

For all instructions, HyperFlow must enforce the precondition $\mathcal{M}_\ell(m_i) \sqsubseteq \ell_{cur}$, where m_i is the memory page where the instruction is fetched from. This prevents information from leaking to the process via fetched instructions and ensures that the integrity of fetched instructions is sufficiently high.

The rest of the preconditions ensure that the information flow labels are accurate. For load instructions, the precondition

$$\ell_{r_a} \sqcup \mathcal{M}_\ell(m) \sqcup \ell_{cur} \sqsubseteq \ell_{r_d}$$

must also hold, where r_a is the source register that contains the base address and m is the page that contains the data being loaded. This precondition ensures that the level of the destination register accurately reflects the level of the data it stores. Similarly, store instructions enforce the precondition

$$\ell_{r_a} \sqcup \ell_{r_v} \sqcup \ell_{cur} \sqsubseteq \mathcal{M}_\ell(m)$$

where r_v is the register that contains the value being written, and m is the page being written to. This precondition ensures that the policy described by the level of the page being written to is also not violated by the data being written to the page or by the address.

Computation instructions — arithmetic and logical instructions, and multiplier unit instructions — write the result of computation into a destination register. For these instructions, the precondition

$$\ell_{r_{s1}} \sqcup \ell_{r_{s2}} \sqcup \ell_{cur} \sqsubseteq \ell_{r_d}$$

must hold, where r_{s1} and r_{s2} are the source registers and r_d is the destination register. The data is influenced by the values of both the source registers (bounded by $\ell_{r_{s1}}$ and $\ell_{r_{s2}}$) as well as by the process executing the instruction (bounded by ℓ_{cur}).

Value-dependent control-flow instructions such as conditional branches must enforce the precondition

$$\ell_{r_{s1}} \sqcup \ell_{r_{s2}} \sqsubseteq \ell_{cur}$$

where r_{s1} and r_{s2} are the source registers used to determine the branch condition. Because ℓ_{cur} represents the security level of the current control flow (program counter) as well as the security level of a process, a change to the program counter can only be affected by information that can flow into ℓ_{cur}. This precondition prevents branch instructions that would violate the information flow policy.

When the restrictions on the new HyperFlow instructions or the preconditions on existing instructions are violated, it might seem natural to raise an exception to be handled by software. However, exceptions can, in general, also cause information flow violations [17]. Type systems can control exception-based information flow [41, 45], but HyperFlow code is not statically checked. Hence, to avoid the possibility of nested exceptions, HyperFlow converts instructions that would otherwise cause information flow violations into NOP instructions, a strategy originally proposed by Fenton [22].

Turning instructions into NOPs preserves security but can change the behavior of programs. However, legacy applications will generally run at one level, with labels assigned by the operating system, so their instructions cannot turn into NOPs. Communication among legacy applications will involve downgrading mediated by trusted libraries or the operating system. For label-aware applications that mix information from multiple security domains, static information-flow checking can be used to ensure they do not violate security; if so, their instructions also will not turn into NOPs.

For timing-channel protection, ℓ_{cur} also represents the security level for the timing of the current process execution, and is used to control the latency of individual instructions. For example, the latency of an instruction cannot depend on its source operand value r_{s1} unless $\ell_{r_{s1}} \sqsubseteq \ell_{cur}$. Similarly, the latency of a load/store instruction cannot depend on memory accesses from another process with label ℓ'_{cur} unless $\ell'_{cur} \sqsubseteq \ell_{cur}$.

5 HDL-LEVEL INFORMATION FLOW CONTROL WITH CHISELFLOW

To provide assurance that the HyperFlow microarchitecture eliminates timing channels, we have implemented it in ChiselFlow, an HDL for static information flow security. HDL-level information flow control applies techniques from language-based security [48] to hardware design [25, 35, 36, 70]. Variables in the code that describe the hardware design are annotated with security labels, L,

```
class ExampleIO extends Bundle {
  val id      = Input(UInt(4.W), L(public, trusted))
  val data_in = Input(UInt(32.W), hlvl(id, id))
  val data_out = Output(UInt(32.W), hlvl(id, id))
}

class ExampleModule extends Module {
  val io = IO(new ExampleIO)
  val secretMask = Reg(init = 0x2.U, L(secret, trusted))
  val publicMask = Reg(init = 0x1.U, L(public, trusted))
  when (secret confFlowsTo id) {
    io.data_out := secretMask & io.data_in
  }.otherwise {
    io.data_out := publicMask & io.data_in
  }
}
```

Figure 3: ChiselFlow example.

which describe static restrictions on where information contained in that signal can flow. A security type system then enforces these restrictions.

Type systems for information flow security can enforce noninterference [28], which ensures that a signal with a label L can only be influenced by signals with labels that are less restrictive than L. For example, if the label `public` is less restrictive than the label `secret`, then a `secret` signal cannot influence a `public` signal. When a label L is no more restrictive than another label L', it is said that L flows to L', written $L \sqsubseteq L'$.

HDLs for static information flow security ensure that the hardware is secure for all executions at design time, before hardware fabrication. HDLs for information flow security can enforce a particularly strong, timing-safe variation of noninterference [35, 36, 70]. HDLs usually describe hardware at the register transfer level (RTL). Because HDLs give cycle-level descriptions of changes to hardware state, the information-flow type system can enforce cycle-level timing-channel freedom.

ChiselFlow extends Chisel [4], an HDL embedded in Scala. ChiselFlow therefore gains much of the expressiveness of Scala, but does not include this complex language in its trusted computing base. Like Chisel, ChiselFlow generates a simpler, compiled intermediate representation (IR) that can then be used to produce hardware designs. The IR for ChiselFlow is called SIRRTL; it extends Chisel's IR, FIRRTL, with information-flow annotations. The enforcement mechanisms of ChiselFlow operate entirely on SIRRTL.

5.1 ChiselFlow Example

Figure 3 shows an example of ChiselFlow code, which looks much like Chisel code aside from the parts in bold font. As in Chisel, ChiselFlow describes hardware modules with classes that extend `Module`. The example shows a module called `ExampleModule`, which takes two inputs: `data_in`, a data value, and `id`, a 4-bit signal indicating the security level of `data_in` and `data_out`. The module outputs `data_out`, which is `data_in` masked with a secret value when `id` indicates that `data_out` can observe secrets.

ChiselFlow also controls implicit flows at the HDL level, in the standard way: the compiler associates an information flow label with each basic block of the code (i.e., the *pc* label) and uses the label of the basic block to constrain side effects.

Signals in ChiselFlow are annotated with security labels that have confidentiality and integrity components. Here, the label `L(public,trusted)` means that `id` is fully public and fully trusted. However, ChiselFlow also supports software-defined security policies that depend on the run-time values of variables. The ability to express security policies that change at run time enables hardware implementations with low area overhead, because it allows hardware modules to be shared among security domains over time. For example, `hlvl(id, id)` describes the interpretation of the signal `id` as a hypercube label. Labels of this form depend on run-time values of signals, but the type system statically reasons about the behavior of these run-time types.

The interface for `ExampleModule` is `ExampleIO`, which describes a record type in which `id`, `data_in`, and `data_out` are records with distinct security labels as described. The register `secretMask` is a fully secret, fully trusted register. The body of ExampleModule is secure because access control ensures that the value of `secretMask` does not flow to `io.data_out` unless `secret` flows to `id`. The program analysis statically determines that in the branch in which this access check succeeds, the security label of `io.data_out` is `secret`, so the assignment from the secret value is secure.

5.2 Security results for SIRRTL

We prove that, aside from downgrades, well-typed hardware modules written in a core subset of SIRRTL enforce a timing-sensitive variant of observational determinism [46, 65]. Supplementary technical material formalizes the core of SIRRTL and includes full proofs [23]. We elide some features from our formalization of SIRRTL, such as record labels (i.e., Bundles), arrays, and modules, because these are either uninteresting or apply techniques studied in prior work. SIRRTL supports heterogeneous record labels and heterogeneously labeled arrays by adopting a proposal by Ferraiuolo et al. [26]. SIRRTL also constrains downgrades; declassifications must be robust and endorsements must be transparent [6]. The language design for downgrades is similar to the NMIFC language by Cecchetti et al. [6], though the type rules are adapted to a cycle-directed hardware language as described in the supplementary material. However, we leave a proof that the language enforces non-malleable information flow control to future work.

We state the main theorem here. We first define low-equivalence of states. A state, σ, is a mapping from HDL variables, $x \in \mathcal{V}$, to values, which are bit vectors. Type environments, Γ, map variables to labels. The syntax of labels in SIRRTL includes functions (from bit vectors to labels) that are fully applied to variables, written $f(x)$. Because labels in SIRRTL include functions over program variables, the valuations of labels depend on the state. We use the metasyntax $\mathcal{T}(\ell, \sigma)$ to denote the valuation of label ℓ in state σ.

We use a low-equivalence relation $\approx_L$ to relate two states that appear the same to an attacker that can observe and modify all variables that flow to L. When two states, σ_1 and σ_2 are low-equivalent to an attacker at security level L, we write $\sigma_1 \approx_L \sigma_2$. Low-equivalence at level L is defined as,

$$\sigma_1 \approx_L \sigma_2 \triangleq \forall x \in \mathcal{V}.(\mathcal{T}(\Gamma(x), \sigma_1) \sqsubseteq L \iff \mathcal{T}(\Gamma(x), \sigma_2) \sqsubseteq L) \\ \wedge \mathcal{T}(\Gamma(x), \sigma_1) \sqsubseteq L \implies \sigma_1(x) = \sigma_2(x)$$

The semantics of SIRRTL emits traces, t, that are sequences of states, each time-stamped with the clock cycle in which it was reached. Traces have the syntax

$$t ::= \epsilon \mid (T, \sigma) \mid t_1; t_2$$

where T is a clock cycle counter represented by a positive integer, and ϵ is the empty trace. Traces are low-equivalent, written $t_1 \approx_L t_2$, when for each element of the trace, the corresponding clock cycle counters are equal, and the states are low-equivalent.

Programs in SIRRTL are single statements s. We give a small-step semantics of SIRRTL that operates on configurations of the form $\langle T, \sigma, s, t\rangle$. Transitions between configurations are denoted by $\longrightarrow_{\mathcal{S}}$, in which $\mathcal{S}$ represents the statement that is the initial syntactic description of the program.

Theorem 1 (Observational Determinism). *If Γ is a type environment, s is a statement without downgrades, pc is a label, L is a fully evaluated security label, and σ_1 and σ_2 are states, then*

$$\begin{aligned} &\vdash \Gamma \wedge \Gamma; pc \vdash s \wedge \sigma_1 \approx_L \sigma_2 \wedge \\ &\langle 0, \sigma_1, s, \epsilon\rangle \longrightarrow_{\mathcal{S}} \langle T, \sigma_1', s', t_1\rangle \wedge \\ &\langle 0, \sigma_2, s, \epsilon\rangle \longrightarrow_{\mathcal{S}} \langle T, \sigma_2', s', t_2\rangle \\ &\qquad\qquad \Longrightarrow \sigma_1' \approx_L \sigma_2' \wedge t_1 \approx_L t_2 \end{aligned}$$

This theorem precludes timing channels because the low-equivalence relation on traces distinguishes traces that differ in timing.

5.3 ChiselFlow Implementation

ChiselFlow is implemented as a 13K-line extension of the Chisel HDL, most of which implements information-flow checking in SIRRTL.

Types that depend on the run-time values of signals are statically checked by using a program analysis that calls out to the SMT solver Z3 [15] to dispatch proof obligations. The use of an SMT solver to handle dependent types is similar to the implementation of SecVerilog [70]. However, the program analysis of SIRRTL differs from SecVerilog in that it models the fact that RTL code is not evaluated in program order; rather, signals are propagated in parallel. The program analysis generates a set of Z3 constraints that model the value assigned to each signal — for each signal, a single expression is generated by unrolling conditional statements. Cases in which sequential variables retain their value from the previous cycle are modeled by an auxiliary variable that represents the old value from the previous cycle. In accompanying supplementary material, we formalize a core subset of SIRRTL and prove that well-typed hardware modules do not leak secrets if they do not explicitly downgrade.

5.4 Heterogeneously labeled data structures

Hardware modules written in Chisel commonly group signals together in bundles, which are analogous to structs or record types. Unlike prior HDLs for information flow security including Sapper [35], Caisson [36], and SecVerilog [25, 26, 70], ChiselFlow supports heterogeneously labeled bundles. The syntax of ChiselFlow allows each signal within a bundle to take an additional argument that describes the label of that individual field. Bundle labels in ChiselFlow are similar to the path labels in Jif [10]. Bundle labels must be represented in Z3; we represent them using Z3's algebraic datatype theory.

Chisel also includes the `Vec` type for describing arrays. In addition, Chisel has a `Mem` type for describing memories, which can either be used to instantiate BRAMs on an FPGA, SRAMs in an ASIC design, or arrays of registers. To support heterogeneously labeled `Vecs` and `Mems` in ChiselFlow, we adopt a recent proposal in which arrays in a secure hardware language can be labeled with functions that map the index of the array to the label of the element at that index [26].

5.5 Nonmalleable downgrades

Much like the downgrading instructions in HyperFlow, ChiselFlow supports robust declassification and transparent endorsement with syntax $\texttt{decl}(e, \ell)$ and $\texttt{endo}(e, \ell)$, in which e is an expression and ℓ is a label. The typing judgments for these expressions closely resemble those used in a recent NMIFC language [6]. One difference between the typing judgments of downgrades in ChiselFlow and NMIFC is that ChiselFlow must use a program analysis to reason about the run-time valuation of signals mentioned in the labels.

5.6 Label inference

The implementation of the baseline processor implementation that HyperFlow extends contains many lines of code. To aid in the effort needed to label HyperFlow, ChiselFlow supports label inference. In ChiselFlow, only module ports need to be explicitly annotated with labels whereas labels of internal registers and wires are inferred. ChiselFlow is the first security-typed hardware language with support for label inference, though the label inference algorithm is similar to those of prior security typed languages for software [41]. Initially, all internal signals that are not explicitly labeled are given variable labels $v \in \mathit{VarLabel}$ that represent unknown labels. Initially, variable labels have the least restrictive label $\top$. Label inference then iteratively lowers the estimate of the final label of each variable label, based on the labels of other signals it influences, until a solution or contradiction is found.

6 MICROARCHITECTURE AND LABELING

Section 4 describes how information flow security policies can be passed to hardware through an ISA. We now describe how a microarchitecture can implement this ISA, and in particular, how it prevents hardware-level timing channels. The HyperFlow instruction set architecture can be realized by many implementations and microarchitectures. Our prototype implementation is based on a single-core configuration of the RISC-V Rocket Chip processor. The prototype implementation label-checks with ChiselFlow and successfully runs all of Rocket Chip's ISA and application unit tests. The implementation includes many microarchitecture features absent from previous information-flow-secured processor designs [26, 35, 36, 70]: e.g., a pending store buffer, pipelined caches, branch prediction, virtual memory, and atomic memory operations.

HyperFlow requires microarchitecture extensions that must be labeled in the secure HDL. Section 8 provides more details on the design trade-offs we considered to make the implementation pass the information flow security analysis, and how the process of implementing hardware with a security-typed HDL differs from that of conventional hardware designs.

6.1 Prototype processor features

The processor is pipelined, with branch prediction and branch target prediction. The branch history table has 2 bits of state per entry

and a global history register. The branch target predictor is fully associative. Execution units include an ALU, a multi-cycle multiplier, and a floating-point unit (FPU) as an independent coprocessor.

The processor has a 32-bit virtual address space divided into 4KiB pages. The baseline processor has L1 instruction and data caches, each with 64 sets and 4 ways and 64B cache blocks. Both L1 caches have 2 pipeline stages. The data cache has a two-slot pending store buffer. Both caches are virtually indexed and physically tagged. The caches include cache controllers. Separate instruction and data TLBs store level-1 page table entries for each cache. A single hardware page-table walker refills both TLBs on misses and caches recently used level-2 page-table entries.

Many of these microarchitectural features are absent in prior information-flow secured processor implementations. To the best of our knowledge, HyperFlow is the first to include TLBs, a PTW, branch and branch target prediction, and a pending store buffer. Most of these features introduce subtle timing channels that we needed to address in order to satisfy the type system. HyperFlow is also the first to include data bypassing with fine-grained information flow labels. This necessitates dynamic label bypassing, which we must also label-check. The prototype implementation of HyperFlow includes all of the aforementioned features as well as the ISA and the microarchitectural extensions described in Sections 4 and 6. The HyperFlow prototype does not include hardware accelerators and relies on a hard-wired memory controller on an FPGA for off-chip DRAM accesses. The processor also does not include support for memory-mapped IO (MMIO). The MMIO support could be added with an on-chip table that provides information flow labels for address ranges that are mapped to IO devices.

6.2 Labeling signals

Every signal in the HDL implementation has a label, though many of them are inferred. The ISA-visible security tags of registers and memory locations and ℓ_{cur} that have already been described in Section 4 also represent type-level information-flow labels in ChiselFlow. The remaining type-level labels that protect other signals in the implementation must be consistent with the ISA-visible labels.

HDLs for information flow control prevent information flow violations through explicit changes in data values such as the storage of a value in a register or memory location, and through the timing of events such as the assertion of a valid or ready signal. In the remaining discussion in this section, we coarsely separate labels in the HDL syntax into *data labels* that protect values, and *timing labels* that protect the timing of events. Examples of data labels include register labels and bypass value labels. The valid bits of data cache entries have timing labels. We also note that this is a coarse characterization; timing flows cannot be cleanly separated from other kinds of information flows in hardware. For example, the values of the data operands of a conventional multiplier influence the time that multiplication takes.

6.3 Labels in the core and label bypassing

In the processing core, the security label of the current process (ℓ_{cur}) is stored in a new control status register. The confidentiality and integrity components, $C(\ell_{r_i})$ and $I(\ell_{r_i})$, of general purpose registers r_i are stored in register banks adjacent to the registers. These label registers can be modified only by the `DECLREG` and `ENDOREG` instructions, which are guarded by logic that checks the nonmalleability conditions, and the `RSTLREG` instruction, which can only set the label of a register to ℓ_{cur}.

The HyperFlow core supports data bypassing. To function correctly, the security labels must be bypassed with the data. For immediate values, the bypassed label is ℓ_{cur}. For a value from a register or the data cache, its label travels with the bypassed data. The bypassed labels are themselves labeled with ℓ_{cur} because they might be stalled or updated by the current process.

6.4 Memory protection and labels

In our prototype, the memory page labels $\mathcal{M}_\ell(m)$ are stored in an on-chip table that maps page numbers to labels. The current security label ℓ_{cur} is attached to each memory transaction so that information flow and downgrading can be checked in the memory system. When returning data from the memory, the label of the page being read is fetched from the table and appended to the memory response transaction. The label of the accessed data is used to enforce the invariants pertaining to load instructions. Write transactions that modify data in memory are similarly appended with a label that protects the data being stored. The label of this data payload is generated from the processing core initiating the request. The label of the data in a write transaction is compared against the label of the destination page, which is stored in the memory label table. Write transactions that would cause information flow violations are dropped. Initially, every page is mapped to the most public and trusted label $C(\bot) \wedge I(\top)$, but the `SETMEM`, `DECLMEM`, and `ENDOMEM` instructions can modify the mapping.

Note that our prototype represents one possible implementation and alternate designs can also be secure. For example, memory labels could be stored in off-chip memory instead of an on-chip table. For a system where the initial state of memory cannot be trusted, the off-chip memory may be initially labeled untrusted while boot code is placed in a trusted on-chip ROM.

6.5 Cache labels

In the data cache, a data label is added to each cache line to track the memory label for the physical address stored in the cache line. The memory label is appended to a cache refill transaction from memory. The data cache is blocking, so memory tags are always brought into the cache before any data is modified or returned to the core. For a load, the cache only returns data if the data label of the accessed cache line flows to ℓ_{cur}. The core updates the destination register only if the label of the returned cache data flows to the label of the destination register. The security of a store is enforced by checking that the label of a pending store buffer entry flows to the label of the cache line, and that the label of a memory transaction flows to the memory page label.

6.6 Timing-channel protection

ℓ_{cur} is also used as a timing label to prevent timing channels through microarchitectural state. That is, $C(\ell_{cur})$ is an upper bound on the level of secrecy that the process is permitted to observe by measuring timing. Any microarchitectural state that influences the timing of instructions is protected by ℓ_{cur}. Cache entries, in-flight

instructions and cache transactions, translation-lookaside buffer (TLB) and page table walker (PTW) entries, and branch predictor state are examples of state that influences instruction timing. Because the security type system in ChiselFlow enforces timing-sensitive noninterference, timing channels must be removed for the hardware to type-check.

When the value of ℓ_{cur} moves downwards in the lattice, the level of secrecy that the process can observe is decreasing. HyperFlow must prevent secrets owned by the previous level of ℓ_{cur} from leaking to the new one. The processor pipeline is drained to prevent high instructions from stalling low instructions as well as other subtle timing channels through register bypassing. In-flight transactions in pipelined caches are also drained when ℓ_{cur} is lowered.

The pending store buffer in the data cache also introduced a subtle and unexpected timing channel. Outstanding cache-write requests in the pending store buffer are serviced opportunistically when there is no in-flight read request. The store buffer can cause a stall either when the content of the buffer might have a read-after-write hazard or when the buffer is full. To prevent a timing channel, we enforce that all entries of the pending store buffer have the same label, and the buffer is drained before lowering ℓ_{cur}.

Caches may also cause timing channels when they are shared among security levels. For instruction caches, the timing channel can be removed by simply clearing and invalidating cache lines when lowering ℓ_{cur}. However, in the data cache, dirty cache lines must be written back when they are evicted, and cannot be simply invalidated. In our implementation, we require software to issue a cache flush instruction to write-back dirty cache blocks before executing an instruction to lower ℓ_{cur}. When ℓ_{cur} is lowered, the data cache is invalidated in a single clock cycle without writebacks. We note that the flush is needed for correctness, but not security.

While our prototype implementation uses flushing to remove cache timing channels, cache partitioning can also be used to lower the flushing overhead on a label change. With partitioned caches, each partition can have a register for its own security label. Then, the logic for a cache read only searches partitions with labels ℓ_P such that $\ell_P \sqsubseteq \ell_{cur}$. When the security label of a partition changes downwards, only that partition will need to be invalidated.

HyperFlow has both *branch prediction*, which predicts whether or not branches are taken, and *branch target prediction*. The branch target predictor in HyperFlow is fully associative. The branch history table (BHT) has two-bit states per index and a global history register. Prior work has demonstrated that both forms of branch prediction create timing channels capable of leaking secrets from Intel SGX enclaves [34]. To prevent timing channels, when ℓ_{cur} moves downward, the branch target predictor is invalidated and cleared, the BHT is cleared, and the global history register is reset.

6.7 Virtual memory

The HyperFlow implementation includes support for virtual memory. While HyperFlow protects memory using memory labels, the virtual memory system provides a familiar interface with a view of private and contiguous memory and permits legacy application software to run on HyperFlow unmodified. Virtual memory support includes instruction and data TLBs as well as a hardware page table walker (PTW). TLBs influence timing because they are caches of recently used Level-1 (L1) page table entries (PTEs). L1 PTEs store mappings from virtual to physical addresses. The PTW serves as a cache of L2 PTEs, which store pointers to L1 PTEs. The TLB and PTW state are labeled with ℓ_{cur}, and the state is cleared when ℓ_{cur} moves downward in the lattice.

Because the TLB and PTW state is labeled with ℓ_{cur}, PTEs must be stored in a memory page with a label that flows to ℓ_{cur}, and managed by software whose integrity is high enough. This restriction must be satisfied by the software that manages the page tables. One simple option is to label the memory pages for page tables with the least restrictive information-flow label, $(C(\bot), I(\top))$. The page table could also be compartmentalized, storing different fragments of the table with different labels to provide finer-grained protection.

7 EVALUATION

We evaluated our HyperFlow prototype in various ways. Because downgrading and dynamic information-flow checks need special scrutiny to avoid security or functionality errors, we studied where and how often these features needed to be used. We also evaluated the power, performance, and area overhead incurred by the added information flow security mechanisms. And we evaluated the usability of the architecture by writing information-flow-rich code in the ISA.

7.1 Uses of downgrades

The RTL code for HyperFlow performs downgrades at various points. The design of ChiselFlow is intended to ensure that these downgrades are nonmalleable [6], as described in Section 5.3 and more fully in the supplementary material. As a result, the potential for these downgrades to cause harm is limited. Our formal results in the technical report imply that insecure information flows can only arise because of these downgrades [49]. A stronger result would prove that well-typed hardware implementations enforce a timing-sensitive variant of NMIFC. While we do not prove such a result, we expect that it should hold, because SIRRTL's rules for downgrades closely follow those used in a language proved to enforce NMIFC [6].

The downgrades are statically checked to be nonmalleable by the type system. Table 3 summarizes the uses of RTL-level downgrades. The first column shows the ISA-visible event to which the downgrade is tied, the second column states the number of downgraded expressions in the RTL code, and the third gives a brief description of what is downgraded. For all downgrades other than downgrades of data caused by explicit downgrade instructions, both an endorsement and a declassification happen.

We expand upon these descriptions here. When the processor resets (1), the register file tags are all initialized to $(\bot, \top)$ and the registers are initialized to zero. This initialization requires explicit writes to the tags because the register file labels are implemented as a sequential memory that can be synthesized as a BRAM on an FPGA. However, this initialization is secure because the processor is initially public and trusted and boots public and trusted code. For convenience, copies from the FPU to the integer register file (2) are automatically downgraded if the labels of the data coming out of the FPU can be nonmalleably downgraded. When ℓ_{cur} moves downwards in the lattice (3), it is possible for a single outstanding

	When is Information Downgraded	Number of Downgrades	What is Downgraded
1	On Reset	1	Register tags (for initialization)
2	FPU to Int instructions	1	Values copied from the FPU to integer registers (when nonmalleable)
3	ℓ_{cur} lowers	2	Presence of one outstanding finish coherence transaction
4	Memory instructions	1	Address is downgraded to ℓ_{cur}
5	CSR file writes	1	Data written to CSR file is downgraded to ℓ_{cur}
6	DECLREG, ENDOREG	7 (1 + 3 each)	Register contents, control signals, arguments
7	LWDWN	2	RF writeback data, dcache bypass data
8	SWDWN	1	P-store buffer data
9	DECLMEM, ENDOMEM	9 (1 + 4 each)	Page contents, control signals, arguments
10	RSTLREG	1	Control signal
11	REGLGATE	8	Control signals, arguments, pipelined data labels
12	LCALL, LCALLR	3	Control signal, arguments, pc value
13	LRET	1	Control signal

Table 3: Uses of Downgrades in HyperFlow.

cache coherence transaction to remain in a pending transaction buffer, causing timing interference. We resolve this with a downgrade, but nothing is leaked if the software is written as described in Section 4; prior to lowering ℓ_{cur}, the software should issue a cache flush to flush any buffered coherence transaction. When a memory transaction is issued (4), the data used to compute the address is downgraded to ℓ_{cur} because the address affects the timing of the cache transaction; this downgrade is for convenience because the address can otherwise be downgraded with an instruction. The label of the address is still protected by the data label, and so the store invariant in Table 2 is enforced. To permit use of performance counters, writes to the CSR file (5) are downgraded to ℓ_{cur}.

The downgrading instructions (DECLREG, ENDOREG, LWDWN, SWDWN, DECLMEM, and ENDOMEM) downgrade the stored data and the arguments to the instructions (6–9). These downgrades are done under a conditional statement that checks that these values are downgraded nonmalleably. As described in Section 4, the arguments are also downgraded to $(\bot, \top)$ because the arguments influence changes to public and trusted labels — this downgrade is also guarded by a nonmalleability check. The labels of the arguments can also be bypassed, and bypassed labels are labeled ℓ_{cur}. Because the bypassed labels are inspected by the nonmalleability check, which influences whether or not the downgrade happens, the labels of the bypassed labels are also downgraded from ℓ_{cur} to $(\bot, \top)$. The control signals that induce the downgrades are also downgraded to $(\bot, \top)$ — this downgrade is always nonmalleable because these control signals are labeled ℓ_{cur}. The LWDWN instruction downgrades the data in two places in the core: the bypass data from the cache and the register file writeback data from the cache. The SWDWN instruction downgrades the stored data from the label in the pending store buffer to the label indicated by the memory tag in the cache. Neither LWDWN nor SWDWN changes the value of any label, so these instructions do not induce downgrades of control signals or arguments.

Similarly, for instructions RSTLREG, REGLGATE, LCALL, LCALLR, and LRET (10–13), control signals are downgraded because these instructions affect public and trusted state. The REGLGATE instruction also includes a nonmalleability check on pipelined labels. The LCALL and LCALLR instructions store the old pc value in a public and trusted stack, so the pc is downgraded from ℓ_{cur} to $(\bot, \top)$.

7.2 Uses of dynamic information-flow checks

As an alternative to downgrading, it is possible to use dynamic label comparisons to prevent information flow violations. These comparisons, called *dynamic checks* should never to be violated at run time, and convert information flow violations into correctness violations. Although such dynamic checks do not weaken security, they must be used with care. It is important to note that dynamic checks differ from the dynamic label comparisons discussed in Table 1 and Table 2 to enforce information flow security. These architectural dynamic label comparisons are necessary because the values of security labels are only known at run time.

Dynamic checks are used in HyperFlow to establish that $\ell_{lwr} \sqsubseteq \ell_{cur}$. This invariant is established in the control and status register (CSR) file where those registers are stored. However, the truth of this invariant is not visible to components outside the CSR file.

Another use of dynamic checks is to convince the type system that we have prevented timing channels caused by floating-point computation. Because the FPU computes on register values, and the time taken to finish a floating-point computation is data-dependent, stall signals from the FPU are also data-dependent. We prevent such flows by disallowing floating-point operations on registers with labels that do not flow to ℓ_{cur}. The stall signals are at a later stage in the pipeline than this check, complicating type-checking the dependently typed stall signals. The stall signals are (redundantly) modified to hide the stalls whenever operand labels do not flow to ℓ_{cur}. This dynamic check would convert information flow violations into a correctness bug, but we ensure such violations do not happen with checks in an earlier pipeline stage.

Another dynamic check is used to convince the type system that the bypass value from the data cache does not cause timing channels; this dynamic check forces the bypass value from the cache to 0 if the timing label from the data cache response does not flow to ℓ_{cur}, but permits the actual data value to be returned otherwise. In practice, this dynamic check does not cause a functional error because when ℓ_{cur} is lowered, the data cache pipeline is stalled and cannot emit responses. Both the regular data cache bypass value and a downgraded bypass value produced by LWDWN are covered by the dynamic check.

7.3 RTL synthesis results

We synthesized the baseline processor and HyperFlow using Vivado v2016.2 targeting the 7z020clg484-1 FPGA found on the Zedboard Zynq 7000 development board. The baseline processor uses 34,508 LUTs (64.9%) on the FPGA, whereas HyperFlow uses 40,205 LUTs (75.6%), a LUT utilization overhead of 16.7%.

The baseline processor uses 13 (9%) of the block RAM tiles whereas HyperFlow utilizes 19.5 (14%). The majority of the overhead is due to the security tags stored with each cache entry, the tag table that associates tags with memory pages, and dynamic label comparisons, which are used for either access controls or dynamic checks. For both the baseline processor and HyperFlow, Vivado is able to meet a target clock frequency of 25MHz. For both designs, the critical path is through the FPU multiplier, so we expect that the minimum clock period is the same for both designs.

7.4 Size of labels

These synthesis results are for 8-bit labels (4-bit confidentiality and integrity), with 2-bit dimensions. This configuration of labels was sufficient to implement DIFC applications with system calls and IPC as described in Section 7.6. Because the OS can virtualize labels, hardware labels only need different representations for *truly parallel* processes that execute at the same time on different cores or in different SMT threads, and not for active processes not currently running. Area overhead grows linearly with the label size. Because our implementation stores memory labels on-chip, label sizes do not affect performance. If memory labels are in DRAM, granularity does not affect on-chip area overhead, but finer granularity increases off-chip storage and bandwidth usage. Performance does not change if an additional memory transaction is issued to fetch a label on *each* last-level cache miss.

7.5 CPI results

Although HyperFlow has no clock frequency overhead, timing channel protection does add performance overhead. We measured the cycles per instruction (CPI) for HyperFlow when executing the RISC-V benchmark suite compared to the baseline Rocket Chip processor. The results are summarized in Table 4. For the HyperFlow processor, the processor executes with the same security level during the entire execution of the program. It incurs a performance penalty because unlike Rocket Chip, the multiplier unit always executes in the worst-case number of cycles. This performance penalty can be removed by disallowing multiplications of operands whose data labels do not flow to ℓ_{cur}. The mm benchmark is matrix-matrix multiply, spmv is double-precision sparse matrix vector multiply, median is a median filter, multiply does multiplications, qsort does quicksort on an array of integers, towers solves a towers of Hanoi puzzle, vvad adds two vectors, and dhrystone is the classic synthetic benchmark. The benchmark with the highest overhead is multiply, naturally, with a geometric mean overhead of 12.4%. HyperFlow also introduces performance overhead by flushing or invalidating hardware state on label changes, but this occurs infrequently, at context switches. The time between context switches is tens of milliseconds, so this overhead should be amortized over execution.

Benchmark name	HyperFlow CPI	RISC-V CPI	Overhead
mm	1.089	1.063	2.4%
spmv	1.748	1.678	4.2%
median	1.631	1.284	27%
multiply	1.899	1.115	69%
qsort	1.542	1.531	0.7%
towers	1.052	1.030	2.14%
vvad	1.161	1.094	6.12%
dhrystone	1.206	1.187	1.6%

Table 4: Performance (CPI) results.

7.6 Usability

HyperFlow is designed to support communication among mutually distrusting principals in an environment managed by an operating system, while supporting the expressive information-flow label models that have been proposed for prior operating systems and languages for information flow control. In this section, we demonstrate that HyperFlow supports shared memory interprocess communication, communication through registers for system calls, and enforcement of rich information flow policies.

We demonstrate how labels represented in the flow-limited authorization model (FLAM) [1] model can be enforced as hypercube labels. FLAM is a recent label model that supports decentralized information-flow policies. To illustrate the usability of the HyperFlow architecture, we implemented a simple, classic application based on decentralized information flow control (DIFC) [42].

The application emulates a tax-preparation service in which a user ("Bob") sends data to a tax preparer and gets a tax form back. Both the tax preparer and the user are distrusting. Although the tax-preparer process is allowed to perform computation on the user's data, HyperFlow prevents it from sharing the user's data or any values derived from it to any party other than the user. In our implementation, the tax-preparer process and the user process communicate through shared memory via IPC. Both processes are managed by trusted software implemented as a single system call that manages labels for the two parties. The application is implemented as assembly code that runs in RTL simulations of our processor prototype. This result suggests that the HyperFlow ISA and prototype are sufficient to enforce practical application-defined information flow control policies with IPC and system calls.

7.6.1 Background: The FLAM label model and downgrading. FLAM unifies authorization and information flow policies. Principals p can delegate to each other; given principals p and q, if p *acts for* q, written $p \succeq q$, then p trusts q. Compound principals can be constructed from primitive principals. The conjunctive principal $p \wedge q$ denotes the combined authority of both p and q. Similarly, the disjunctive principal $p \vee q$ represents the authority of either p or q. Principals with $\succeq$ form a lattice, and $p \wedge q \succeq p \succeq p \vee q$ for any p and q.

In FLAM, principals are also information flow labels. The confidentiality of p is written $p^{\rightarrow}$, and intuitively represents the authority to observe secrets owned by p. The integrity of p, written $p^{\leftarrow}$, represents the authority to affect information owned by p. A second ordering on principals, defines permitted information flows. The statement p flows to q, written $p \sqsubseteq q$, denotes that information is permitted to flow from p to q. The ordering $\sqsubseteq$ forms another lattice over principals, which is orthogonal to the authority lattice.

$$\mathcal{B}[\![p]\!] \triangleq (b_p, b_p)$$
$$\mathcal{B}[\![p^{\rightarrow}]\!] \triangleq (\mathcal{B}_c[\![p]\!], b_{max}) \qquad \mathcal{B}[\![p^{\leftarrow}]\!] \triangleq (b_{min}, \mathcal{B}_i[\![p]\!])$$
$$\mathcal{B}[\![p \wedge q]\!] \triangleq (\max_b\{\mathcal{B}_c[\![p]\!], \mathcal{B}_c[\![q]\!]\}, \max_b\{\mathcal{B}_i[\![p]\!], \mathcal{B}_i[\![q]\!]\})$$
$$\mathcal{B}[\![p \vee q]\!] \triangleq (\min_b\{\mathcal{B}_c[\![p]\!], \mathcal{B}_c[\![q]\!]\}, \min_b\{\mathcal{B}_i[\![p]\!], \mathcal{B}_i[\![q]\!]\})$$
$$\mathcal{B}[\![p \sqcup q]\!] \triangleq (\max_b\{\mathcal{B}_c[\![p]\!], \mathcal{B}_c[\![q]\!]\}, \min_b\{\mathcal{B}_i[\![p]\!], \mathcal{B}_i[\![q]\!]\})$$
$$\mathcal{B}[\![p \sqcap q]\!] \triangleq (\min_b\{\mathcal{B}_c[\![p]\!], \mathcal{B}_c[\![q]\!]\}, \max_b\{\mathcal{B}_i[\![p]\!], \mathcal{B}_i[\![q]\!]\})$$

Figure 4: Representing FLAM labels with hypercube labels.

The meet and join in the information flow order are written $\sqcap$ and $\sqcup$. Any FLAM principal can be represented as a conjunction of a confidentiality projection and integrity projection $p^{\rightarrow} \wedge q^{\leftarrow}$. Labels of this form are said to be in normal form.

7.6.2 Mapping FLAM labels to hypercube labels. FLAM labels are easily represented in the hypercube model using bit vectors. FLAM labels in normal form map directly to confidentiality and integrity components of hypercube labels. Primitive principals p are mapped to numeric constants, b_p. For example, if there are four 1-bit dimensions and p and q are mutually distrusting, one might map p to `1000` and q to `0100`. Figure 4 shows how compound principals can be mapped to hypercube labels. Here, $\mathcal{B}[\![p]\!]$ denotes the representation of p as a pair of its hypercube label components. The confidentiality component is the first in the pair, and the integrity component is the second. $\mathcal{B}_c[\![p]\!]$ denotes the confidentiality component of p and $\mathcal{B}_i[\![p]\!]$ is the integrity component. The values b_{min} and b_{max} are the lowest and greatest bit vectors that can be represented with the width of a label; they are respectively a sequence of all 1s and a sequence of all 0s. Here, $\max_b$ is a function that computes the dimension-wise maximum of two hypercube labels, and $\min_b$ similarly computes a minimum.

7.6.3 Tax-preparer application. To test the usability of HyperFlow, we implemented the tax-preparer application in assembly using the HyperFlow ISA. Bob has ℓ_{lwr} and ℓ_{upr} labels that are $B^{\leftarrow}$ and $B^{\rightarrow}$ respectively, and generally operates with a ℓ_{cur} label of B. The tax-preparer generally operates with ℓ_{cur} of $(B \wedge P)^{\rightarrow} \wedge P^{\leftarrow}$ because it is an instance of the tax preparation service specifically for handling Bob's requests, so it needs to be able to observe Bob's data. Its ℓ_{lwr} and ℓ_{upr} labels are $P^{\leftarrow}$ and $(B \wedge P)^{\rightarrow}$ respectively.

Before either Bob or the tax-preparer executes, a label manager that is fully trusted and public registers the `switch_process` call gate and initializes the memory label. Bob computes his tax form and sends the message to the preparer using shared memory, as described in Section 7.7. Bob then yields the processor to the preparer by calling the `switch_process` gate. The tax preparer receives the message and computes the form using its proprietary data before declassifying the result. The preparer sends the result back to Bob via IPC and yields the processor back to Bob by calling `switch_process` again. Finally, Bob receives the computed form.

7.7 Interprocess Communication

Figure 5 shows an example of how messages are communicated among processes in the tax-preparer application, and more generally, shows how shared memory IPC works in HyperFlow. In the

```
# Set Page Labels. cur_lvl: {\bot-> & \top <-}
li x1, 0x84       # {B-> & P<-}
li x2, 0x48       # {P-> & B<-}
la x3, prep_to_bob
la x4, bob_to_prep
setmem x1, 0(x3)
setmem x2, 0(x4)
...
# Bob Sends. cur_lvl: {B}
la x5, bob_to_prep
swdwn x6, 0(x5) #decl {B} to {P-> & B<-}
...
# TP Receives. cur_lvl: {(B&P)-> & P<-}
la x5, bob_to_prep
lwdwn x6, 0(x5) #endo {P-> & B<-} to {(B&P)-> & P<-}
```

Figure 5: IPC Example.

example, $0x88$ represents the principal Bob (B), and $0x44$ represents the Tax Preparer principal (P). A page of memory is allocated for Bob to send messages to Preparer with label $B^{\rightarrow} \wedge P^{\leftarrow}$, and for Preparer to send messages to Bob with label $P^{\rightarrow} \wedge B^{\leftarrow}$. Public and trusted code initializes the labels of the pages used for IPC. In the code segment shown, Bob has a ℓ_{cur} label of B. Because the Tax Preparer is an instance of the tax preparation service specifically for handling Bob's requests, it has a ℓ_{cur} label of $(B \wedge P)^{\rightarrow} \wedge P^{\leftarrow}$ so that it can see Bob's data. For Bob to send a message to Preparer, it simply performs a `SWDWN` instruction on a register with label B which downgrades the register contents to the label of the destination page ($P^{\rightarrow} \wedge B^{\leftarrow}$). This downgrade is robust because Bob has enough integrity to remove the $B^{\rightarrow}$ component of the label. The Preparer receives the message by doing a `LWDWN` instruction, which endorses the integrity of the message to $P^{\leftarrow}$.

In some cases, it is possible to receive a message through IPC by first downgrading a register and then doing a conventional load instruction to the downgraded register. However, this example demonstrates that this workaround is not always possible; the `LWDWN` instruction is necessary for expressiveness of the ISA. The tax preparer cannot endorse the integrity of a register to $B^{\leftarrow}$, because its label does not flow to $B^{\leftarrow}$. However, it can endorse the $P^{\rightarrow} \wedge B^{\leftarrow}$ data to $P^{\leftarrow}$, so it can receive the data with a `LWDWN` instruction.

This example uses separate pages for communication in each direction. However, it is conventional for processes to share a single page that both processes can both read and write. The label model of HyperFlow is also expressive enough to support bidirectionally shared pages — a single page could be labeled $B \vee P$. With this label, both B and P can write to the page, and both process can read from the page by endorsing it. However, the aforementioned numerical representations of B and P cannot represent $B \vee P$ as a label distinct from the fully public and fully distrusted label, which any other process could read and write. By representing B as $0b01001$ and P as $0b00101$, $B \vee P$ becomes the more restrictive label, $0b00001$. Therefore, by adding an extra bit, we can distinguish disjunctions from the bottom label. Other representations are also possible. For example, using 2-dimensional rather than 4-dimensional hypercubes offers a compact encoding while retaining unique disjunctions; however, it reduces the total number of physical principals that can be represented simultaneously.

```
# Gate Registration: cur_lvl {\bot-> & \top<-}
la x1,      switch_process
li x2,      0x0F # {\bot-> & \top<-}
reglcall x1, x2

# Bob. cur_lvl: {B-> & B<-}
...          # compute form, store in shared page
li x4, 0x0
la x3, swith_process
declreg x1, x4 # Flag to choose Bob or Preparer
declreg x2, x4 # Address to jump to after call
lcallr 0(x3)

# Process Switch Call: cur_lvl {\bot-> & \top<-}
switch_process:
li x4, 0xF
endoreg x1, x4 # Flag to choose Bob or Preparer
endoreg x2, x4 # Address to jump to after call
...           # Set labels, jump to target
```

Figure 6: Syscall Example.

7.8 System Calls

In the tax preparer, a single trusted system call is used to manage Bob and the Preparer. This system call starts a new process by initializing the labels for the process and then jumping to the entry point of the process. This system call is implemented as a call gate. Figure 6 shows a small segment of the call gate as well as how the gate is registered and called. More generally, the example shows how system calls can be implemented in HyperFlow. Initially, fully trusted and fully public code registers the call gate at address `switch_process` and with label $\bot^{\rightarrow} \wedge \top^{\leftarrow}$. The call gate takes two arguments. One describes whether labels should be initialized for Bob or the Preparer, and the other is the PC value that is the entry point for the next process. When Bob is done computing, it executes the call gate. Before doing so, it must declassify the confidentiality of the two arguments from $B^{\rightarrow}$ to $\bot^{\rightarrow}$. It then simply calls the gate with an `LCALLR` instruction.

At the start of the call gate, the handler must endorse the integrity of the two arguments from $B^{\leftarrow}$ to $\top^{\leftarrow}$ because the system call handler is fully trusted, and it takes a conditional branch based on the value of the argument. The call gate handler then sets the tags of all registers and the levels of ℓ_{nlwr}, ℓ_{ncur}, and ℓ_{nupr} to values that depend on the next principal to execute. It then does a `SETBOUNDS` instruction before jumping to the entry point of the next process.

In conventional processors, system calls work by first jumping to trusted code that contains a system call handler table — the particular call handler to execute is selected by using a register argument that contains the call number. This model is also supported by HyperFlow. However, because HyperFlow replaces conventional privilege modes with lattice model information flow labels, the call gates of HyperFlow are more general and can both improve performance and the precision with which access controls are enforced.

8 DISCUSSION

We now discuss some of the design and implementation trade-offs we made in order to satisfy the goal of making our HyperFlow prototype label-check with ChiselFlow. We also discuss how the process of implementing a processor that label-checks differs from conventional hardware design.

Labels of labels. In ChiselFlow, wires can be used to represent information flow labels that can change at run time. Because these wires are still wires, they are also labeled. Many information flow systems assume that information flow labels are fully trusted and fully public. However, at the HDL level, we found it necessary to give more restrictive labels to some signals that represent data labels. For example, the per-page data labels are stored in the cache and used as security types for the data in the cache. Most control signals in the cache are labeled with ℓ_{cur} because they represent signals that affect timing. Because the values of these control signals influence the time that per-page data labels are brought into the cache, the labels themselves must be labeled with ℓ_{cur}.

Automatic label propagation. Initially, we expected that security labels could be propagated automatically. For example, following an `ADD RS3, RS2, RS1` instruction, we would like to compute the join of the labels of `RS1` and `RS2` and store the result in `RS3` without needing explicit instructions to set the label of `RS3`. In fact, this form of dynamic label propagation is common in tagged hardware architectures [14, 55]. However, whether general purpose registers are updated depends on control signals that are labeled with ℓ_{cur}, when the labels of the registers are public and trusted. In other words, dynamically updating the security labels themselves introduces subtle timing channels. We found it better to allow register labels to be updated only by explicit instructions. These label updates could be inserted automatically by the compiler.

Multi-cycle execute unit stall signals. Our HyperFlow prototype has two execute units that take multiple cycles to perform a computation: the multiplier and the FPU. The time to complete these computations depends on data values. If the labels of these data values do not flow to ℓ_{cur}, the computation time might create timing channel vulnerabilities if not controlled. In the HDL code, this timing channel is visible as a flow from the operands from the register file, whose security levels depend on their labels, to the stall signal, which has the label ℓ_{cur}. We address this timing channel for each of the two execute units in different ways. For the FPU, we do not permit computations on operands with labels that do not flow to ℓ_{cur}. For the multiplier, we permit computations on operands that do not flow to ℓ_{cur}, but the operations always complete in the worst-case time. This presents a tradeoff between the expressiveness of the ISA and performance. We took different approaches for each primarily to demonstrate that either can be statically checked with the information-flow type system.

9 FUTURE WORK

This paper takes a first step toward a new approach to designing and building secure processors; we now discuss future steps. A natural next goal for HyperFlow is to obtain end-to-end security results about software that executes on HyperFlow. In addition, we expect future work to study software that takes advantage of the flexible policies enforceable by HyperFlow. We also sketch a few potential use cases for HyperFlow, which include the enforcement of enclaves, information flow operating systems, and hardware support for language-based information flow security. Constructing these applications is left to future work.

9.1 Security Results

The HyperFlow ISA is designed to enforce timing-safe information flow security. Ideally, the security guarantees should be formalized, for example, as a variant of noninterference [28], among security levels. One would then show that for all executions of valid HyperFlow-ISA programs beginning from a reasonable initial state, that this property is enforced. We leave such an ISA-level proof of security to future work.

However, to ensure that confidentiality is not violated through timing channels, we must also consider low-level details about the hardware implementation that are invisible at the ISA level. As a step toward ensuring that HyperFlow can eliminate timing channels, we have constructed HyperFlow in ChiselFlow, an HDL for information flow security, and proved a timing-safe noninterference result. A remaining open problem is to connect this low-level security guarantee to a future ISA-level security guarantee.

ChiselFlow is designed to ensure that downgrades in the implementation are non-malleable, but we have not yet shown this formally. We expect that ChiselFlow will support timing-safe variant of the NMIFC [6] property, though formalizing timing safety in the presence of downgrades is also an open problem. Similarly, the ISA is designed to prevent malleable downgrades; an ISA-level proof about HyperFlow should also offer assurance about this.

9.2 Enclaves

Trusted execution environments (TEEs) such as Intel SGX [7, 11–13] provide protection for software modules called *enclaves*, whose confidentiality and integrity are protected even if the OS is compromised. TEEs employ a *reference monitor* that allows the operating system to construct and manage enclaves, but rejects operations that might violate the confidentiality or integrity of the enclaves. The reference monitor is typically implemented with instruction extensions for managing enclaves [11], or in software [13, 24, 39]. At minimum, software implementations of TEEs require hardware support for storage that is inaccessible to the operating system, support for attestation, and optionally defense against physical attacks [24].

However, most prior architectures for TEEs do not defend against timing channel attacks, which have already been exploited [34, 59]. HyperFlow enforces information flow policies in a timing-safe way, intending to eliminate microarchitectural timing channel attacks. Though timing channel attacks are subtle, by implementing HyperFlow in an HDL for information flow control, we provide strong assurance that we have succeeded in eliminating them. HyperFlow is designed to provide sufficient support for implementing enclaves. HyperFlow augments virtual memory protection with information flow protection that operates on physical pages of memory. As a result, a TEE can be implemented in HyperFlow by storing enclave metadata in a region of memory that is more confidential and trusted than the set of labels the OS can manipulate. A TEE executing on HyperFlow would benefit from resilience against timing-channel attacks.

9.3 Information-flow operating systems

Beyond microarchitectural timing channels, other side channels are not addressed by enclave systems. For example, they do not address *passive address translation attacks*, in which the operating system correlates the enclave's page faults usage with secrets [12]. Because enclave systems do not prevent side channels, the operating system is ultimately still trusted for confidentiality.

Addressing side channels therefore requires a trustworthy operating system that can control OS-level side channels and executes on a processor such as HyperFlow that can control hardware timing channels. Microkernels for information flow security [8, 21, 67] further improve security by replacing authorization mechanisms based on access controls with information flow control, and can prevent covert channels in the OS [68]. By propagating the information flow policies of microkernels down to the hardware, HyperFlow can provide defense in depth and control side channels. Because HyperFlow generalizes conventional, hierarchical privilege levels to lattice-model information flow labels, it offers more fine-grained separation of privilege compared to conventional implementations.

Constructing an OS for HyperFlow would also provide an opportunity for more performance measurements. In this paper, we provide performance results for applications executing within a single security level. By constructing an OS, future work can experimentally evaluate multi-program systems that operate at and communicate via different security levels.

9.4 Compilers for information-flow security

Hardware and operating system defenses alone do not suffice to prevent timing channels that might violate application confidentiality, because neither the OS or hardware has complete information about the application's security policies. Languages for information flow security can address implicit flows as well as other security policies that are internal to a program [48]. Prior work has also shown that timing channels can be addressed by propagating language-level security policies down to the hardware [69]. Though many label models can express DIFC policies [1, 6, 8, 21, 42, 67], prior label models represent policies as lattices. Hence, HyperFlow can enforce policies from any of these models.

Security-typed languages typically rely on an assumption that the compiler and assembler are semantics- and type-preserving. By enforcing policies in hardware, HyperFlow can potentially reduce these assumptions and support unsafe languages like C.

10 RELATED WORK

Gate-level information flow tracking. Gate-level information flow tracking [30, 43, 44, 56–58] applies information flow control to hardware designs at the gate level. The earliest variations of GLIFT [58] augment each hardware gate with additional gates to track information flow, incurring significant area and energy overhead. Later GLIFT versions apply gate-level information flow tracking to simulated hardware designs [57], rather than to the implementation. This reduces overhead, but increases development effort compared to a conventional processor design flow. Because simulating every state in large designs is intractable, prior efforts to use simulation-based GLIFT check either small components [44], or limit the simulation to cover just the state space reachable with software that is co-designed with the hardware [56, 57].

Security-typed HDLs. Recently, security-typed hardware description languages have been developed to check that information-flow

policies are enforced at design-time. Unlike simulation-based approaches, type systems can ensure that the entire design is secure in just seconds. Sapper [35] and Caisson [36] are security-typed hardware description languages that generate GLIFT logic, but use a static analysis of the HDL code to minimize the amount of information flow tracking logic generated. Sapper and Caisson enforce security dynamically by adding run-time checks, which convert security violations to functional correctness errors. SecVerilog [25, 70] is a security-typed hardware description language that allows security policies to depend on run-time values, but checks information flow security statically by generating type errors at design time. By enforcing security policies statically, hardware designers can avoid unexpected functional errors at run time; if dynamic checks are necessary for security, designers need to explicitly add them to pass the type check. The design of ChiselFlow closely follows the design of SecVerilog, and checks security statically. ChiselFlow ports the security type system to Chisel and extends SecVerilog with new features: nonmalleable downgrades, type inference, etc.

Processors secured with information flow. HyperFlow provides assurance for strong information flow security by statically checking hardware-level information flow using ChiselFlow. Tiwari et al. [57] built the first processor with strong information flow security guarantees using a simulation-based approach to GLIFT. The processor supports just two security levels, and communication across trust domains is not allowed. Similarly, Zhang et al. construct a processor with two security domains [70]. Xun et al. [35, 36] construct a processor that supports a diamond lattice. None of these processors support communication across security domains. Ferraiuolo et al. [26] implement a processor that permits communication that weakens information flow security, but it does not constrain downgrades. HyperFlow provides better assurance with downgrades because they enforce nonmalleable information flow control [6]. None of these processors provide memory protection or privilege levels that can be arbitrary lattice-model information flow labels. Prior information-flow secured processors also do not have mechanisms for downgrading registers, or for control-flow transfers between different security domains; both mechanisms are necessary to support system calls.

Language-based information flow. ChiselFlow applies ideas from the extensive literature on static language-based information-flow security, which started with Denning and Denning [17] and is surveyed by Sabelfeld and Myers [48]. ChiselFlow's expressive dependent labels build on previous work that enriched static labels with expressive run-time labels [25, 38, 41, 70, 71].

On the other hand, the HyperFlow ISA enforces secure information flow using dynamic rather than static checking, since the goal is to support code not generated by a trusted compiler. The enforcement mechanism for the ISA thus fits into the long history of work on dynamic information-flow tracking. In ignoring instructions with illegal information flows, we follow the approach originally proposed by Fenton [22], analogous to the approach of DIFC operating systems like HiStar [67], in which processes ignore messages received from more sensitive contexts. Alternatively, execution can be halted at the point of violation [2, 63], though information may still leak if halting is observable. Other approaches to dynamic information-flow control include secure multi-execution [19], faceted execution [3], and monads for tracking information flow [52, 54]. Like HyperFlow, these approaches introduce the possibility that invalid information flows either change computational results or make them unavailable for use.

11 CONCLUSION

This paper presents HyperFlow, a processor architecture for timing-safe information flow security and a prototype implementation that is statically checked with an HDL-level security type system. The results show that it is feasible to redesign modern microprocessors to enforce strong information flow security in hardware while supporting rich security policies specified in software. The architecture enables low-level information flows such as timing channels to be controlled by software, while the type system provides assurance about the implementation. As future work, we envision a formalization of the ISA that combines with HDL-level security to ensure timing-safe information flow for the software. In that sense, we believe that this work represents a step toward the goal of end-to-end information flow security for a full system comprising both hardware and software.

12 ACKNOWLEDGMENTS

We thank Skand Hurkat, Drew Zagieboylo, and Ethan Cecchetti, along with Cătălin Hriţcu and the reviewers, for their feedback and suggestions on this work. This work was partly sponsored by NSF grant CNS-1513797, NASA grant NNX16AB09G, 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] O. Arden and A. C. Myers. A calculus for flow-limited authorization. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, June 2016.
[2] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *4th Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 113–124, 2009.
[3] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *39th ACM Symp. on Principles of Programming Languages (POPL)*, pages 165–178, January 2012.
[4] 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 Conf. (DAC)*, 2012.
[5] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as DTIC AD-A023 588.
[6] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. Nonmalleable information flow control. In *ACM Conf. on Computer and Communications Security (CCS)*, 2017.
[7] David Champagne. *Scalable Security Architecture for Trusted Software.* PhD thesis, Princeton University, 2010.
[8] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in Aeolus. In *USENIX ATC*, 2012.
[9] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *IEEE Computer Security Foundations Workshop (CSFW)*, 2006.
[10] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *16th USENIX Security Symp.*, 2007.
[11] Intel Corporation. Intel software guard extensions programming reference, 2014.
[12] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, February 2016. http://eprint.iacr.org/2016/086.
[13] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symp.*, 2016.
[14] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Int'l Symp. on Computer Architecture (ISCA)*, 2007.
[15] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*

(TACAS), pages 337–340, March 2008.

[16] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 1976.

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

[18] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD (The Orange Book) edition, December 1985.

[19] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *IEEE Symp. on Security and Privacy*, pages 109–124, May 2010.

[20] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and André DeHon. Pump: A programmable unit for metadata processing. In *3rd Int'l Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, June 2014.

[21] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *20th ACM Symp. on Operating System Principles (SOSP)*, 2005.

[22] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[23] Andrew Ferraiuolo. Security results for sirrtl, a hardware description language for information flow security. Technical report, Cornell University, 2017.

[24] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *26th ACM Symp. on Operating System Principles (SOSP)*, 2017.

[25] Andrew Ferraiuolo, Weizhe Hua, Andrew C Myers, and G Edward Suh. Secure information flow verification with mutable dependent types. In *54th Annual Design Automation Conference 2017*, page 6. ACM, 2017.

[26] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Verification of a practical hardware security architecture through static information flow analysis. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[27] Vijay K. Garg. *Introduction to Lattice Theory with Computer Science Applications*. Wiley Publishing, 2015.

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

[29] Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C Pierce, and Greg Morrisett. All your IFCException are belong to us. In *IEEE Symp. on Security and Privacy*, pages 3–17. IEEE, 2013.

[30] Wei Hu, Dejun Mu, Jason Oberg, Baolei Mao, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Gate-level information flow tracking for security lattices. *DAES*, 2014.

[31] Colin J. Fidge. Timestamps in message-passing systems that preserve partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, 1988.

[32] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[33] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *21st ACM Symp. on Operating System Principles (SOSP)*, 2007.

[34] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symp.*, pages 557–574, 2017.

[35] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A language for hardware-level security policy enforcement. In *ASPLOS*, 2014.

[36] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A hardware description language for secure information flow. In *32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2011.

[37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[38] Luísa Lourenço and Luís Caires. Dependent information flow types. In *42nd ACM Symp. on Principles of Programming Languages (POPL)*, 2015.

[39] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *3rd ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2008.

[40] CVE-2017-5691, July 2017.

[41] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *26th ACM Symp. on Principles of Programming Languages (POPL)*, 1999.

[42] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 2000.

[43] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Theoretical analysis of gate level information flow tracking. In *Design Automation Conf. (DAC)*, 2010.

[44] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Information flow isolation in I2C and USB. In *Design Automation Conf. (DAC)*, 2011.

[45] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1), January 2003.

[46] A. W. Roscoe. CSP and determinism in security modelling. In *IEEE Symp. on Security and Privacy*, pages 114–127, May 1995.

[47] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2009.

[48] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[49] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *IEEE Symp. on Security and Privacy*, 2004.

[50] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. Computer Security*, 17(5):517–548, 2009.

[51] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 1974.

[52] Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. Faceted dynamic information flow via control and data monads. In *5th Int'l Conf. on Principles of Security and Trust (POST)*, pages 3–23, 2016.

[53] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. Addressing covert termination and timing channels in concurrent information flow systems. September 2012.

[54] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in the presence of exceptions. *J. Functional Programming*, 2017.

[55] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[56] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *MICRO*, 2009.

[57] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *ISCA*, 2011.

[58] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

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

[60] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[61] Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM memory via Intel CPU cache poisoning, 2009.

[62] Rafal Wojtczuk and Joanna Rutkowska. Following the white rabbit: Software attacks against Intel VT-d technology, 2011.

[63] Steve Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University Department of Computer Science, August 2002.

[64] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *IEEE Computer Security Foundations Workshop (CSFW)*, 2001.

[65] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *IEEE Computer Security Foundations Workshop (CSFW)*, 2003.

[66] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

[67] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2006.

[68] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *8th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2008.

[69] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2012.

[70] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[71] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *Int'l J. Information Security*, 6(2–3), March 2007.