Lightweight Verification of Secure Hardware Isolation Through Static Information Flow Analysis (Technical Report)

Andrew Ferraiuolo[1], Rui Xu[1], Danfeng Zhang[2], Andrew C. Myers[1], and G. Edward Suh[1]

[1]Cornell University, Ithaca, NY
Email: {af433,rx37}@cornell.edu, andru@cs.cornell.edu, suh@ece.cornell.edu

[2]Penn State University, University Park, PA
Email: zhang@cse.psu.edu

January 29, 2017

Abstract

Hardware-based mechanisms for software isolation are becoming increasingly popular, but implementing these mechanisms correctly has proved difficult, undermining the root of security. This work introduces an effective way to formally verify important properties of such hardware security mechanisms. In our approach, hardware is developed using a lightweight security-typed hardware description language (HDL) that performs static information flow analysis. We show the practicality of our approach by implementing and verifying a simplified but realistic multi-core prototype of the ARM TrustZone architecture. To make the security-typed HDL expressive enough to verify a realistic processor, we develop new type system features. Our experiments suggest that information flow analysis is efficient, and programmer effort is modest. We also show that information flow constraints are an effective way to detect hardware vulnerabilities, including several found in commercial processors.

1 Introduction

Modern computing systems increasingly rely on hardware-level protection to provide secure environments for critical software components. Protection rings are widely used in practice to isolate supervisor processes from user processes. Recent hardware security architectures such as ARM TrustZone [20], Intel SGX [2], and IBM SecureBlue [1] aim to protect software even when the operating system is malicious or compromised

The complexity of modern processors inevitably leads to bugs and security vulnerabilities. Processor errata often include security bugs [13]. Previous studies found vulnerabilities in implementations of Intel VT-d [40] and system management mode (SMM) [39]. Vulnerabilities are found in safety-critical hardware. For example, an exploitable bug was found in the Actel ProASIC3 [30], which was used in medical, automotive, and military applications including a Boeing 787 aircraft. (cut to save a line)

Since hardware design is error-prone, there is need for approaches that provide formal guarantees about the security properties enforced by hardware. Unfortunately, conventional tools for formal verification are prohibitive in terms of both verification time and hardware designer effort. As a result, commercial hardware designers rely on code reviews and best practices to attempt to avoid vulnerabilities.

This paper demonstrates that the security requirements of complex hardware security architectures can be verified with hardware description language (HDL) information flow control in practice. Information flow control (IFC) is lightweight since checking happens statically at design time, and it is comprehensive since it constraints the movement of data throughout the system.

Using an IFC HDL, we design and verify a multi-core processor which emulates the security features of ARM TrustZone, a widely used commercial security architecture. TrustZone aims to provide isolation between trusted and untrusted security domains, which it calls the "secure world" and the "normal world". We show how to use static analysis of information flow to check that the security goals of this architecture are met. This is the first demonstration that information flow can be used to verify complex processors implementing commercial hardware security features. Since other hardware security architectures [2, 1, 34] have similar goals, the approach and findings of this paper are applicable to a broader class of architectures.

Information flow is verified at design time by the security type system of the HDL. Type systems offer some important benefits. Type checking is fast and compositional, and provides a formal guarantee that the HDL code enforces the information flow policy specified by the designer. Because HDL-level information flow verification is done statically, it has negligible impact on chip area, run-time performance, and power consumption. Verification time and programmer effort are also small compared to model checking.

Recent efforts [18, 17, 42, 36] have applied information flow control to hardware security. However, these tools have only been applied to single-core processors with simple security policies. We study a multicore architecture with both confidentiality and integrity requirements. Applying static information flow analysis to TrustZone is not a straightforward application of previously developed techniques. Three significant technical challenges arise:

- Prior information flow type systems for HDLs are imprecise in their reasoning about packed data structures. We build on the existing SecVerilog [42] language, but improve the precision of previous HDL-level information flow analyses so that efficient designs can be verified. The new features add precise reasoning about the security of individual array elements and individual bits in packed structures such as network packets, whereas prior hardware-level type systems

[18, 17, 42] only allow one security label for each variable.

- Information flow analysis aims to entirely prevent certain flows of information, but like other hardware security architectures, TrustZone has a more nuanced security policy: code running on a trusted secure-world core is allowed to cause potentially dangerous flows. The TrustZone architecture describes access control checks to enforce its policy. Instead, we show that the main security goals of TrustZone can be naturally expressed as information flow constraints. Further, we extend SecVerilog with language-based *downgrading* mechanisms for declassification and endorsement, and show that these features permit localization of design aspects that might enable dangerous flows.

  Despite the use of these downgrading mechanisms, we obtain strong security assurance. In particular, our implementation enforces the property that in the absence of operations by trusted cores, software running on untrusted cores cannot learn anything about secure world state or violate the integrity of the secure world. Hence, information does not leak unless trusted secure-world software performs operations that leak information.

- Prior studies have used information flow to verify simple secure processors, but this work is the first to verify a multi-core processor with shared caches, on-chip interconnects, and a memory interface. Architecture extensions were necessary to statically verify these features. To avoid checking complex invariants about the functionality of the on-chip network, the memory response ports are extended with low-overhead access controls. This extension prevents responses from being accepted unless they originate from a trusted party. We also identify a potential extension to the instruction set architecture (ISA) that would allow trusted software to control hardware-level information flow downgrading.

Hardware designed with IFC HDLs is formally guaranteed to enforce the security property *noninterference*. We demonstrate for the first time that IFC HDLs are also capable of detecting security vulnerabilities in practice. We emulate security bugs found commercial hardware including the Actel ProAsic 3 [30], an AMD Processor [13], and several Intel Processors [40]. We detect all of these vulnerabilities with our approach. We also implement a suite of six other security vulnerabilities to test the limitations of HDL-level IFC. We find that only bugs that affect downgraded signals can go undetected. Our TrustZone implementation uses downgrading sparingly, and we could not construct an undetectable bug that affects downgraded signals in our TrustZone implementation.

We synthesized a multicore processor based on a commercial security architecture and empirically show that HDL-level information flow control has minimal overhead. The performance, area, energy, and design-time productivity overheads of HDL-level IFC are low. Because verification is performed statically at design time, the hardware overhead is minimal; the clock frequency, area, and power consumption of the verified design are almost identical to an unverified one. The programming effort is also small. The secure HDL requires annotations (security labels) for variable declarations and other purely mechanical changes.

The rest of the paper is organized as follows. Section 2 reviews the ARM TrustZone architecture. Using TrustZone as an example platform, Section 3 describes how HDL-level IFC can be used for security verification. Section 4 describes the language extensions needed for verification. Section 5 describes architectural features that aid verification. Section 6 presents the evaluation results. Finally, Section 7 discusses related work, and Section 8 concludes.

2 ARM TrustZone Prototype

2.1 TrustZone Overview

ARM TrustZone is a representative security architecture that is used widely in practice. Its applications include embedded systems and smartphones. TrustZone uses hardware mechanisms to provide an execution environment that isolates high-security software from low-security software. Other commercial security architectures [2, 1] also aim to provide an isolated execution environment, so we believe the findings of this study are applicable to other architectures as well.

TrustZone partitions the hardware and software into two security domains, called the *secure world* and the *normal (non-secure) world*. The high-security software executes in the secure world, and the remaining software executes in the normal world. The software executing in the normal world is prevented from accessing data owned by the secure world. Practical systems need to allow communication between security domains. For this purpose, TrustZone assumes the secure-world software is trustworthy, and allows it to access data in either world. The threat-model of TrustZone does not address timing channel attacks, and the only physical attacks it addresses are simple ones that exploit debug interfaces.

TrustZone isolates the secure and normal worlds by introducing a security tag, called the *NS bit*. TrustZone uses control mechanisms in hardware that check the NS bit. Each processing core stores the NS bit in a program status register (PSR) to indicate which world is currently executing on the core. The Each bus master and slave that may be used in the secure world is also extended with an NS bit. For example, DMA engines and display controllers may have an NS bit. A core can switch its security domain by executing trusted software called the *monitor mode* which executes with secure world privilige. The monitor mode is entered by an explicit instruction or through interrupts. The normal world cannot change any NS bits.

Access to resources is controlled based on the NS bit. The system (AXI) bus appends the NS bit of the bus master to each transaction. Bus slaves inspect the NS bit and prevent the normal world from accessing secure-world resources. For example, main memory (DRAM) is partitioned into secure and normal based on address ranges, and the NS bit is checked for accesses to the secure-world partition. TrustZone protects debug interfaces by preventing normal-world debug requests from affecting the secure world. Similarly, normal-world accesses to secure-world interrupt configuration registers are disallowed. In some implementations of TrustZone, data from both worlds can coexist in caches. Coexistence is permitted by extending each cache line with an NS bit guarded by access control. Similarly, TLBs can be extended to store address mappings from both worlds.

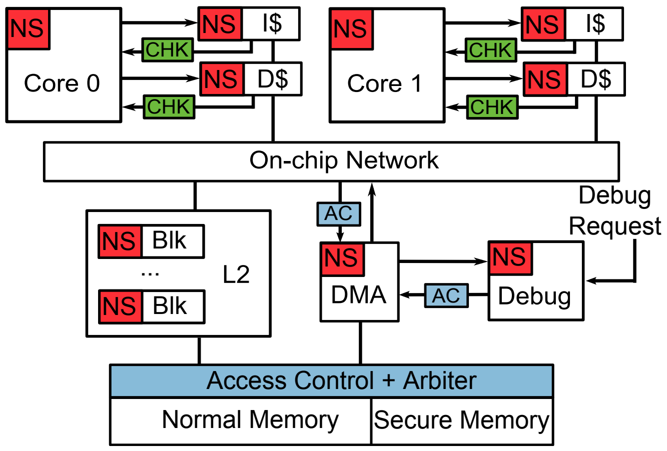

Figure 1: TrustZone prototype implementation.

2.2 Prototype Implementation

TrustZone is an architectural specification that can be implemented in many ways. Our prototype is designed to study the practicality of verification with information flow. We implemented key security features for multicore implementations of TrustZone, but did not include non-essential functions.

Figure 1 shows a block diagram of our implementation of TrustZone. Our implementation includes two five-stage pipelined MIPS processing cores, private L1 caches, a shared L2 cache, a DMA engine, a ring network, and a memory module. Each core has private L1 instruction and data caches, which are connected to a shared L2 cache through a ring network. The L2 cache includes a prefetch buffer. The system includes a DMA engine that can move data between memory locations. The DMA engine takes requests from processing cores through a memory-mapped interface connected to the ring network. It also has an external debug interface. The L2 cache and the DMA engine are connected to the main memory controller through an arbiter. Our processor was implemented in 16,234 lines of Verilog code. This is comparable to other open-source processors such as OpenRISC (31,944 lines) and RISC-V (14,206 lines).

The prototype implements the security features of TrustZone that are necessary to isolate the secure world from the normal world. The processing cores, the DMA engine, and the debug interface include an NS bit to indicate the security domain. The NS bits for the DMA engine and the debug interface can be changed by a secure-world core through a memory-mapped interface. All bus transactions, memory requests, and memory response packets carry the NS bit of the core or DMA engine that initiated the request.

The prototype supports world switches for cores through an instruction. World-switching is implemented in a way that ensures that the NS bits of in-flight instructions are not corrupted. The pipeline is stalled until all in-flight instructions have completed. Then, the NS bit of the core is changed. Registers which are labeled with NS bits are cleared (set to 0) when the NS bit changes. Clearing prevents the implicit downgrading [42] problem, which we do not have space to describe here. Register files and the PC register are cleared. When the PC is 0 and the core is in normal world, the PC jumps to an offset storing a switch-to-normal handler. Address 0 stores a secure world handler. Arguments to calls between the two worlds must be passed via memory.

The caches allow data from both worlds to coexist. Each line of the L1 and L2 caches is extended with the NS bit to indicate the world that owns the data. On a hit, the NS bit of the access is compared to the NS bit of the cache line. If there is a mismatch, the access is treated as a miss.

The bus slaves use access control. For example, normal-world requests to the DMA engine are rejected when the DMA engine is in the secure world. The main memory includes an access control module. A partition control register in the module partitions the address space between worlds. The partition control register is memory-mapped and can only be modified by a secure-world request.

The prototype implements the security features of TrustZone necessary to protect the confidentiality and integrity of the secure world in a multi-core SoC. Protection includes support for hardware IP modules (e.g., a DMA engine) and a debug interface. However, the security domains in TrustZone are orthogonal to traditional privilege levels and virtual memory, so our processor does not include supervisor/user mode or virtual address translation. Also, optional functions such as additional peripherals, coherent accelerators, tightly coupled memory, and protected interrupts are not implemented.

3 Verification Methodology

3.1 Background: SecVerilog

Our security verification methodology relies on information flow control enforced at the hardware description language (HDL) level. SecVerilog [42] extends Verilog with a syntax for annotating variables (wires and regs) with security labels. Security labels represent security levels such as H (high-security) or L (low-security). In addition to the variable annotations, the programmer defines a lattice of security levels [7] to specify how information may flow among security levels. For example, the lattice might specify that variables labeled H cannot flow to variables labeled L, but that variables labeled L can flow anywhere. More formally, security levels $\ell \in \mathcal{L}$ form a lattice with ordering relation $\sqsubseteq$. If $L \sqsubseteq H$, information is permitted to flow from variables labeled L to variables labeled H. However, information flow is prevented from H to L if $H \not\sqsubseteq L$. The levels $\perp$ and $\top$ denote the least and greatest security levels ordered by the relation $\sqsubseteq$. The variable annotations and lattice form an *information flow policy*, which is enforced by the type system. Checking is done statically at design time, and variable annotations have no affect on the synthesized hardware.

In SecVerilog, security types τ have the syntax shown in Figure 2. Types may either be labels (ℓ), the meet ($\sqcap$) or join ($\sqcup$) of two types, or a dependent type $f(v)$, where f is a type-level function, and v is a variable name. The meet and join represent the greatest lower bound and least upper bound of their operands respectively. We later extend this syntax in Section 4 to handle more practical hardware designs.

$$\text{Types} \qquad \tau ::= \ell \mid f(v) \mid \tau_1 \sqcup \tau_2 \mid \tau_1 \sqcap \tau_2$$

Figure 2: SecVerilog syntax of security labels.

```
1   reg  {L} v, {L} l, {H} h;
2   // LH(0) = L, LH(1) = H
3   wire {LH(v)} shared;
4   ...
5   if (v == 0) l = shared;
6   else h = shared;
7   ...
8   if (h == 0) l = 0;
9   else l = 1;
10  ...
```

Figure 3: SecVerilog code example.

SecVerilog enforces the security policy by constraining assignment statements. For example, Figure 3 shows a code example for the policy that prevents flow from H to L. In the example, *explicit flows* such as `l=h` are disallowed, because they directly violate the security policy. SecVerilog prevents explicit flows with an assignment typing rule that requires the expression on the right of the assignment to be less than the variable on the left according to the ordering relation $\sqsubseteq$. *Implicit flows* such as the insecure assignments on lines 8–9 leak information indirectly through control flow. To prevent implicit flows, the type system associates a security level, `pc`, with each node in the control flow graph. Assignments are allowed only if the `pc` is lower than the level of the assigned variable. The expression typing rules determine the least upper bound of information contained in each expression. For example, the expression `l & h` has type H since this is the lowest type that is at least as restrictive as the types of both `l` and `h`.

In the syntax, $f(v)$ is a fully-applied function representing a dependent type. Dependent types describe static types that depend on the run-time values of variables. They are used to describe hardware which is shared by different security levels over time. In Figure 3, `shared` has a type which depends on `v`. Here, `LH` is a function that is L when `v` is 0 and H when `v` is 1. To perform checking at design time, SecVerilog statically generates predicates to reason about the run-time behavior of dependent types. For example, to check the assignment on line 5, it generates the invariant $\mathtt{v} = 0 \Rightarrow LH(\mathtt{v}) \sqsubseteq L$. These invariants are based on a conservative approximation of strongest postcondition reasoning as described by Zhang et al. [42], and are checked by an automated constraint solver [6].

3.2 Approach

This paper studies the application of language-level information flow control to formally verify secure isolation properties of hardware designs. While security architectures in processors typically rely on access control mechanisms and are not designed for information flow security, this study shows that the key security goals of such processors can be naturally captured by information flow policies.

To use HDL-level IFC, hardware designers first represent the security goal of the

TrustZone Security Policy	Information Flow Policy Label	Downgrading
P1. Normal world core/IP cannot **(C)** read or **(I)** write secure-world memory/IP	CT for secure world core/IP/memory PU for normal world core/IP/memory (dependent type based on the NS bit)	D1-1. Secure world reads/writes to normal-world memory/IP D1-2. Timing dependence (common for all)
P2. **(I)** Normal world cannot change NS bits	PT for NS bits	D2-1. Secure world writes to an NS bit D2.2. Legitimate normal-to-secure NS-bit switches
P3. **(I)** Normal world cannot change TrustZone control registers	PT for TrustZone control registers	D3-1. Secure-world writes to TrustZone control registers

Table 1: Core TrustZone policy expressed as information flow constraints with explicit exceptions (declassification/endorsement). (C) and (I) represent policy for confidentiality and integrity, respectively. IP (Intellectual Property) is a hardware module.

hardware isolation mechanisms with a set of information flow constraints. The designer then annotates the HDL code along with a security lattice to express the information flow constraints as a concrete information flow policy. Because the goal is to remove unintentional bugs, designers and the security policy that they write are trusted. The design is then verified with type checking. If the hardware design passes type checking, the type system formally guarantees that the code enforces noninterference [42, 11] under the given policy.

The formulation of noninterference enforced by SecVerilog is timing-sensitive (i.e., it prevents timing channels), since timing-flows are indistinguishable from non-timing flows at the gate level [37]. However, the threat model of TrustZone *does not* restrict timing flows, perhaps because preventing timing channels imposes performance overheads. As we will see, a challenge is posed by this disparity between the language-level policy and the security goals of the architecture.

3.3 TrustZone as an Information Flow Policy

This subsection uses TrustZone as an example to show how HDL-level IFC can verify an isolated execution environment. TrustZone isolates the secure world from the normal world using access control policies and mechanisms that control normal-world accesses. As shown in the first column of Table 1, the goal of each access control policy is to protect either the confidentiality (C) or the integrity (I) of security-sensitive state.

The high-level security goal of TrustZone can be expressed as information flow constraints that address either confidentiality or integrity requirements. The confidentiality policies specify that no information can flow from secure-world processing cores, memory, or hardware (IP) blocks to normal-world modules. The integrity constraints ensure that no information from a normal-world core/memory/IP can affect a secure-world core/memory/IP or other trusted state such as NS bits and TrustZone control registers.

The above information flow constraints can be translated into an information flow policy expressed with a security lattice and labels in HDL code. To express both confidentiality and integrity levels, we define four security levels: CT, CU, PT, and PU. The first letter represents the confidentiality level (confidential or public) and the second letter represents the integrity level (trusted or untrusted). Then, we define a security lattice that prevents confidential information from flowing to public and

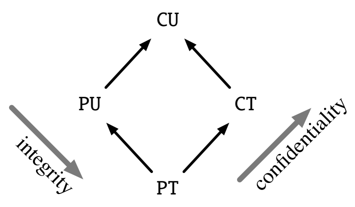

Figure 4: Security lattice for TrustZone.

untrusted information from affecting trusted as shown in Figure 4. In the figure, the arrows represent the direction of allowed information flow.

The second column in Table 1 shows how the TrustZone implementation is labeled using the security levels in the lattice. To protect both confidentiality and integrity of secure-world state, variables in secure-world processing cores, memory, and hardware IP blocks are labeled `CT` while normal-world ones are labeled `PU`. Signals that are statically allocated to one world are annotated with fixed labels. Most hardware resources (e.g., the processing cores) can be switched between the two worlds. The security labels of time-shared modules use a dependent type and are expressed as a function (`world`) of the NS bit (`ns`) associated with that module. Here, `world(ns)` maps the value of `ns` to a security label: 1 maps to `PU` and 0 maps to `CT`. Signals that must be trusted, but are not confidential, are labeled `PT`. For example, NS bits and TrustZone control registers, such as the one that partitions memory among worlds, must be trustworthy. The clock and reset variables are also labeled `PT`.

Unfortunately, strict noninterference is too restrictive for practical systems like TrustZone because it does not allow any communication between security levels. TrustZone prevents the normal world from acting maliciously, but trusts the secure world to release information to or accept information from the normal world correctly. This permitted communication violates noninterference and causes type errors.

To bridge the gap between noninterference and practical security policies, we introduce declassification and endorsement expressions so that designers can explicitly allow exceptions to noninterference. Declassification releases confidential information to the public. Endorsement changes the security level of untrusted information so that it is considered trusted. The term *downgrading* refers to both.

The third column in Table 1 shows how downgrading is used to express the security policy of TrustZone. TrustZone allows the secure world to access the normal world (D1-1). The TrustZone threat model does not include timing-channel attacks, so information flows through timing (D1-2) are allowed. Secure-world writes to NS bits (D2-1) and control registers (D3-1) are allowed even though their values may be read by the normal world. TrustZone allows the normal world to trigger a world switch through a special instruction (D2-2) even though this causes a flow from the normal world to the NS bit. Section 6.1 discusses how downgrading is used in the prototype in more detail.

3.4 Security Assurance with Downgrading

Static information flow analysis with SecVerilog provides a formal security guarantee that the described hardware enforces noninterference [42]. This guarantee is independent of functional correctness. If there is no downgrading, type checking ensures that the security policy specified by the labels is enforced even if there is a functional bug.

Therefore, our verification methodology ensures that the only possible violations to the security policy are through downgrading. This improves security assurance in two ways. First, all potentially dangerous information flows are made explicit and designers must explicitly allow each of them in the source code. Second, the size of the code that can cause vulnerabilities is significantly reduced; only code that affects downgrading expressions can lead to vulnerabilities. Without information flow analysis, bugs in any part of the code may break security.

We argue that our use of downgrading policies permits information to be downgraded only under the authority of the secure world software and that downgrading cannot be controlled by the normal world software. We substantiate this claim by an analysis of the conditions under which downgrading expressions may be removed (see Section 6.1). For example, the downgrading expressions for memory accesses can be removed if there is no access from the secure world. Explicit communication between security levels is only permitted if initiated by the secure world; data is downgraded only if the NS bit of the request is secure. Our analysis shows that these downgrading expressions can be removed if the secure world never executes. This suggests that only the secure world may affect information release.

3.5 Security Labeling Errors

The security labels specify the security policy that is verified by type checking. As a result, a security vulnerability may not be detected if the security labels are incorrect. For example, if an entire processing core is labeled `CT` when it is in the normal world, the information flow analysis will not detect a violation which permits that core to read secure memory. However, we found that errors in security labels are typically detected as long as labels are correct at sources and sinks. Labels must be consistent along all possible paths of an information flow to pass the type check. Unintentional mistakes are unlikely to be consistent with other labels.

4 Language Extensions

We introduce SecVerilogBL, which extends the SecVerilog hardware description language with three significant features needed to verify practical systems. These extensions include new type declarations that can specify distinct security labels for each bit of a vector and for each element of an array and a syntax for declassification and endorsement. We prove that SecVerilogBL enforces the same security guarantee as SecVerilog [42], namely, that well-typed programs enforce observational determinism [31].

```
1  wire [0:31] {world(ns)} data;
2  wire [32:41] {PT} addr;
3  wire {PT} ns;
4  wire [0:42] {i -> if (i <= 31) world(ns) PT} packet;
5  assign packet = {ns, addr, data};
```

Figure 5: A packet concatenation example.

4.1 Downgrading

As described earlier, the security policy of TrustZone is described using an integrity/confidentiality diamond lattice. This lattice policy is more conservative than the TrustZone policy, which assumes secure world software is written correctly and makes no mistakes when reading from or writing to normal world data. To relax the lattice policy, we use downgrading, which lowers the type of the downgraded expression within the security lattice.

Downgrading expressions are used only to allow the secure world to read and write normal-world data and to avoid timing channel protection. Since downgrading is used sparingly, SecVerilogBL includes a downgrading syntax, `downgrade`(e, τ) adopted from Sabelfeld et al. [26]. This syntax is used for both declassification and endorsement. Downgrading expressions take two arguments, an expression e and a type τ. The downgrading expression then behaves semantically like e, except that during type checking, its type is treated as if it were τ.

This syntax allows the designer to precisely control where information is released. The type of a downgraded expression is modified only in the context in which it is downgraded. For example, if `x` has type `CT` and `y` has type `PU`, the assignment `y = x` would be rejected even if `z = downgrade(x, PU)` appears elsewhere in the same HDL code. The downgrading syntax also permits precise control over what information is released. For example, the expression `downgrade(x > 3, PU)` reveals whether or not `x` is greater than 3, but it does not reveal the value of `x`.

4.2 Per-Bit Types

Hardware designs often use sequences of bits to describe data structures. For example, a packet in our TrustZone implementation is a collection of bits describing data, an address, and possibly other metadata. SecVerilog reasons imprecisely about individual fields of a packet, since the whole packet must share a single label. Information about the labels of individual wires is lost once they are grouped.

Figure 5 shows an example that creates a packet by concatenating data with an address and the NS bit. For now, ignore the security label on line 4, which uses a new syntax explained later in this section. Grouping variables in this way makes code clearer and more compact. Unfortunately, SecVerilog cannot precisely capture the desired label for the resulting packet. In this example, the address is public, but depending on the value of the `ns` bit, the data could be confidential. Thus, the (static) security levels of some of the bits in the packet depend on the run-time value of other bits.

A dependent type is a natural way to describe a situation like this one, in which a static type depends on run-time values. But the prior dependently-typed HDL, Kinds $\quad k ::= \ell \mid \texttt{int} \rightharpoonup k$

Types $\quad \tau ::= \ell \mid \tau_1 \sqcup \tau_2 \mid \tau_1 \sqcap \tau_2 \mid x \mapsto \tau \mid f\ x$

$\qquad \mid \mathit{if}\ e^\tau\ \tau_t\ \tau_f \mid \mathit{case}\ e^\tau\ \tau_1\ \ldots\ \tau_n$

Figure 6: Syntax of SecVerilogBL labels.

SecVerilog, does not have enough expressive power to describe the example just given. The upper 11 bits of the packet have a different label than the rest of the packet, but SecVerilog applies the same label to all bits in a bit vector. Lowering the type of the entire packet is not a solution. The non-data bits are used for routing decisions which affect both worlds, so this type change would cause SecVerilog to variable an insecure flow.

Our solution is to enrich what can be expressed using dependent labels, as shown in braces on line 4. The label expression specifies that the security class of the i^{th} bit depends both on i and on the value of the `ns` bit. The type of `packet` is a function that takes an integer, i, representing an index to the bit vector. If the index is less than 31, the data is accessed, so it returns a type that depends on the MSB of the packet that corresponds to the `ns` bit. Otherwise, the address or the `ns` bit is accessed, so the returned type is `PT`.

Figure 6 shows the formal type syntax of SecVerilogBL. Crucially, the type system is extended with types of higher kinds. Kinds, written k, can either be levels $\ell \in \mathcal{L}$ or partial functions from integers to other kinds. In the SecVerilog syntax, all types, including dependent types (which are functions that are fully applied to variables) are of the kind ℓ.

Types (i.e., labels) in the extended syntax, written τ, are pure (side-effect-free) expressions signifying security levels. The syntax $v \mapsto \tau$ specifies a mapping from a position in the bit vector to the type of the bit at that position, and is used to specify the type of `packet` in Figure 5. Since $v \mapsto \tau$ is a form of function abstraction, types written with this syntax are higher-kinded. Dependent types may be written by referring to program variables in the type expression. The syntax `if` $e^\tau\ \tau_t\ \tau_f$ and `case` $e^\tau\ \tau_1 \ldots \tau_n$ describe conditional selection between security labels. The syntax of e^τ describes pure expressions and is omitted because it is standard. However, notably e^τ may contain variables declared in the program, and therefore, can be used to write dependent types.

At a high level, the type system is extended to track the bit-width of each variable in addition to its type. Types of kind ℓ are lifted in the obvious way to $\texttt{int} \rightharpoonup \ell$, so that all types become functions from bit indices to labels. During assignment checking, the bit-width of both sides of the assignment is used as a range for quantification.

The type rules of SecVerilogBL are shown in Figure 7. In addition to a standard type environment Γ, a width environment $\mathcal{W}$ maps variables to their bit-widths. The width environment is populated with the declared widths of variables. Bit-widths are static, finite, and specified with integer constants. Since Verilog does not support dynamically-sized bit vectors, ranges are easily determined at compile time. Typing judgments for expressions have the form $\Theta; \Gamma; \mathcal{W} \vdash e : \tau, w$ meaning that under context $\Theta; \Gamma; \mathcal{W}$, expression e has type τ and bit-width w. A kind environment, Θ, is used to make kind judgments of the form $\Theta \vdash \tau : k$.

The rule T-Logical for logical binary operators checks that the widths of both expressions are the same and that both expressions have `int` $\rightharpoonup \ell$ types. The type of the resulting expression is the bitwise join of the types of the operands. The rule T-Arith must track the bits that are propagated by carry bits. The i^{th} bit of the result is affected by all bits below i from both inputs. The rule for concatenations (T-Concat) selects between the type functions of the original subexpressions, shifting the upper expression as needed. The rules for shifting by constants (T-LShift and T-RShift) select the bottom type for the bits of the resulting expression that are constant. For the remaining parts, the type function of the non-constant subexpression is shifted. The rule for indexed arrays (T-ArrIndex) is discussed in Section 4.3.

Per-bit checking is done in the type-checking rule for assignments. The check verifies that the type of each bit of the right side of the assignment (joined with the program counter) is lower than the corresponding bit on the left. To type-check an assignment of some expression with type τ_r to a variable x, with type τ_l, both of these types are applied to each integer within $(0, \mathcal{W}(x))$. If the condition $\forall i \in (0, \mathcal{W}(x)). \tau_r(i) \sqcup pc(i) \sqsubseteq \tau_l(i)$ holds, the check succeeds. We omit the rules for commands since they are straightforward. Notably, pc is a bitwise label that is determined by commands in a straightforward way. Using a bitwise label for pc is more permissive than alternative rules which might compute the join over the bitwise labels of expressions, for example, used as conditionals in if-statements.

4.3 Per-Element Types

Arrays are commonly used in hardware descriptions and are important for our TrustZone implementation. However, SecVerilog has minimal support for labeling arrays; all elements must have the same type. The code segment shown in Figure 8 describes part of the memory array of a cache that implements reads. The input `ns` is the NS-bit of the device originating the read request. The output `read` is the output data, which has a type that depends on `read_ns`. This code is secure, but cannot be written in SecVerilog.

Following common practice, the cache is implemented as an array of memory cells, `mem`. Another array `reg_ns` stores the NS bit of the last device to write to each address of the array. Therefore, the label of a particular memory cell at array position i should depend on `reg_ns`. With support for fine-grained array labels, the memory cells can be implemented conveniently as an array of bit vectors.

To support arrays in which each element has a distinct type, array variables must have kind `int` $\rightharpoonup$ `int` $\rightharpoonup \ell$ when they are declared. In the rule for indexed arrays (T-Arr-Index), the type of the array, τ_x, is applied as a function to the expression that indexes the array, `e`. Doing so produces the security label of the selected element of the array. The requirement imposed on the kind of τ_x ensures that τ_x e is `int` $\rightharpoonup \ell$, which is a mapping from the bit position to the label of that bit. Arrays in Verilog may only be indexed by variables and constants rather than arbitrary expressions, and therefore e can be substituted into τ_x at compile time. If e is a variable, this forms a dependent type. Constraints for these dependent types are generated using a conservative approximation of strongest postcondition analysis as in [42]. The label τ_e' is the (bitwise) join over

$$\textsc{T-Const}\ \frac{}{\Gamma;\mathcal{W};\Theta \vdash n : \bot, w} \qquad \textsc{T-Var}\ \frac{\Gamma(x) = \tau \qquad \mathcal{W}(x) = w}{\Gamma;\mathcal{W};\Theta \vdash x : \tau, w}$$

$$\textsc{T-Logical}\ \frac{\begin{array}{c}\Gamma;\mathcal{W};\Theta \vdash e_1 : \tau_1, w \qquad \Gamma;\mathcal{W};\Theta \vdash e_2 : \tau_2, w \\ \Theta \vdash \tau_1 : \texttt{int} \rightharpoonup \ell \qquad \Theta \vdash \tau_2 : \texttt{int} \rightharpoonup \ell \\ \tau = i \mapsto (\tau_1\ i) \sqcup (\tau_2\ i)\end{array}}{\Gamma;\mathcal{W};\Theta \vdash e_1 \ \mathsf{bop}\ e_2 : \tau, w}\ (\text{when } \mathsf{bop} \in \{\vee \wedge \oplus\})$$

$$\textsc{T-Arith}\ \frac{\begin{array}{c}\Gamma;\mathcal{W};\Theta \vdash e_1 : \tau_1, w \qquad \Gamma;\mathcal{W};\Theta \vdash e_2 : \tau_2, w \\ \Theta \vdash \tau_1 : \texttt{int} \rightharpoonup \ell \qquad \Theta \vdash \tau_2 : \texttt{int} \rightharpoonup \ell \\ \tau = i \mapsto \bigsqcup_{j \in (1,i)} ((\tau_1\ j) \sqcup (\tau_2\ j))\end{array}}{\Gamma;\mathcal{W};\Theta \vdash e_1 \ \mathsf{bop}\ e_2 : \tau, w}\ (\text{when } \mathsf{bop} \in \{+-\})$$

$$\textsc{T-Concat}\ \frac{\begin{array}{c}\Gamma;\mathcal{W};\Theta \vdash e_1 : \tau_1, w_1 \qquad \Gamma;\mathcal{W};\Theta \vdash e_2 : \tau_2, w_2 \\ \Theta \vdash \tau_1 : \texttt{int} \rightharpoonup \ell \qquad \Theta \vdash \tau_2 : \texttt{int} \rightharpoonup \ell \\ \tau = i \mapsto \texttt{if}(i > w2)\ (\tau_1\ i - w_2 + 1)\ (\tau_2\ i)\end{array}}{\Gamma;\mathcal{W};\Theta \vdash \{e_1; e_2\} : \tau, (w_1 + w_2)}$$

$$\textsc{T-LShift}\ \frac{\begin{array}{c}\Gamma;\mathcal{W};\Theta \vdash e : \tau, w \qquad \Theta \vdash \tau : \texttt{int} \rightharpoonup \ell \\ \tau' = i \mapsto \texttt{if}\ (i > n)\ (\tau\ i - n + 1)\ \bot\end{array}}{\Gamma;\mathcal{W};\Theta \vdash e << n : \tau', w} \qquad \textsc{T-RShift}\ \frac{\begin{array}{c}\Gamma;\mathcal{W};\Theta \vdash e : \tau, w \qquad \Theta \vdash \tau : \texttt{int} \rightharpoonup \ell \\ \tau = i \mapsto \texttt{if}\ (i > w - n)\ \bot\ (\tau\ i + n)\end{array}}{\Gamma;\mathcal{W};\Theta \vdash e >> n : \tau, w}$$

$$\textsc{T-ArrIndex}\ \frac{\begin{array}{c}\Gamma(x) = \tau_x \qquad \mathcal{W}(x) = w \\ \Gamma;\mathcal{W};\Theta \vdash e : \tau_e, w_e \\ \Theta \vdash \tau_x : \texttt{int} \rightharpoonup \texttt{int} \rightharpoonup \ell \\ \Theta \vdash \tau_e : \texttt{int} \rightharpoonup \ell \\ \tau'_e = i \mapsto \bigsqcup_{i \in (1,w_e)} \tau_e\ i\end{array}}{\Gamma;\mathcal{W};\Theta \vdash x[e] : \tau'_e \sqcup (\tau_x\ e), w}$$

Figure 7: Typing rules for SecVerilogBL expressions.

```
...
reg [0:31] {world(ns)} read;
reg [0:31] { i -> j -> world(reg_ns[i]) } mem[0:1023];
reg {PT} reg_ns  [0:1023];
...
if(ns == 1) begin
    read = (reg_ns[read_addr] == 1) ?
        mem[read_addr] : 32'b0;
end else begin
...
```

Figure 8: A register file code segment.

the `int` $\rightharpoonup \ell$ label of `e`. Since the value of `e` determines which element of `x` is selected, each bit of `e` affects the value of `x[e]`. So τ_e' is used to elevate the label of `x[e]` to reflect that each bit of `e` has influenced `x[e]`

On line 3 of Figure 8, `mem` has an array type that maps the index of the array to a type that depends on the value stored in `reg_ns` at the corresponding index. Although the function does not depend on j, it is still written as a type function of kind `int` $\rightharpoonup$ `int` $\rightharpoonup \ell$, so that when the array index is applied, the type can serve as a mapping from bits to types. To support arrays of bit vectors where each element has a different mapping from indices to types, the type function can be written to depend on both i and j.

Per-bit and per-element type declarations enable better component designs that are not otherwise possible and reduce programmer effort. For example, without per-element types there is no way to describe a queue that is dynamically shared among security domains. With per-element types, a securely shared queue can be described with an array of queue data entries and a queue of security tags that correspond to the data. A prior implementation without shared queues required separate queues for each security domain and for each port. The improved design with shared queues has six fewer queues, three fewer arbiters, and four fewer demultiplexers than the other design. The improved design required 392 fewer lines of code. Per-bit types also made the description of network packets clearer and more compact.

4.4 Soundness

Any well-typed SecVerilogBL program without downgrading obeys observational determinism [42, 31], a generalization of noninterference [11] for nondeterministic systems. The formulation of observational determinism enforced is the timing-sensitive one presented by Zheng et al. [42]. Informally, it states that if the hardware begins execution from two states which are indistinguishable to an attacker, then on each clock cycle and beginning from either initial state, the hardware will produce states which are also indistinguishable to the attacker.

In the rest of this section, we provide a formal definition and proof of this theorem. The proof is accomplished by a translation from well-typed SecVerilogBL programs into well-typed SecVerilog programs. SecVerilogBL bit vectors are simulated with 1-bit SecVerilog variables and a corresponding translation of SecVerilogBL environments into SecVerilog environments. SecVerilogBL expressions of width w translate into a vector of w SecVerilog expressions. Assignments of w-bit expressions are unrolled into w assignments. The translation of commands other than assignment statements merely propagate the translation of assignments. The translation is clearly semantics-preserving, and the security result is obtained by showing that the translation is also type-preserving.

The translation splits non-array variables, x, into single-bit representations x_1,...,x_n, where x_i stores the i^{th} bit of the original variable x. The notation $[\![\Gamma; \mathcal{W}; \Theta]\!]$ denotes the translation of SecVerilogBL environment $\Gamma; \mathcal{W}; \Theta$ into a SecVerilog environment Γ' containing 1-bit variables. Each variable $x_j \in \Gamma$ is translated into $\mathcal{W}(x_j)$ 1-bit variables whenever $\Gamma(x_j)$ has kind `int` $\rightharpoonup \ell$. Otherwise, $\Gamma(x_j)$ has kind `int` $\rightharpoonup$ `int` $\rightharpoonup \ell$, and x_j represents an array, translated into $n \times \mathcal{W}(x_j)$ 1-bit variables where n is the declared length of the array:

$$\llbracket \Gamma; \mathcal{W}; \Theta \rrbracket = \llbracket ..., x_j : \tau_j, ...; ..., x_j : w_j, ..., ; \tau_j : k_j, ... \rrbracket \triangleq \{..., X_j, ...\}$$
$$\text{where } X_j = \begin{cases} x_{j,1} : \tau_{j,1}, ..., x_{j,wj} : \tau_{j,wj} & \text{if } k_j = \texttt{int} \rightharpoonup \ell \\ x_{j,1,1} : \tau_{j,1,wj}, ..., x_{j,n,wj} : \tau_{j,n,wj} & \text{if } k_j = \texttt{int} \rightharpoonup \texttt{int} \rightharpoonup \ell \end{cases}$$

The translation of expressions and assignment statements is shown in Figure 9. The translation for other commands than statements merely propagates the translation of statements. The translation for both statements and expressions also include Γ, $\mathcal{W}$, and Θ as arguments. For notational convenience, define $\mathcal{W}\llbracket e \rrbracket_{\Gamma;\mathcal{W};\Theta} = w \iff \Gamma; \mathcal{W}; \Theta \vdash e : \tau, w$ for some τ. Each translation of a SecVerilogBL expression produces a vector of SecVerilog expressions. The meta-syntax $\mathcal{E}\llbracket e \rrbracket_{\Gamma;\mathcal{W};\Theta}(i)$ selects the i^{th} element of the vector produced by $\mathcal{E}\llbracket e \rrbracket_{\Gamma;\mathcal{W};\Theta}$. The notation $\vec{e}|_{i\in(n_1,n_2)}$ constructs a new vector by replacing the free occurrences of i in e with each integer in the range (n_1, n_2).

Note that in the rules for logical and arithmetic operators $\mathcal{W}\llbracket e_1 \rrbracket_{\Gamma;\mathcal{W};\Theta} = \mathcal{W}\llbracket e_2 \rrbracket_{\Gamma;\mathcal{W};\Theta}$. In the rule for the translation of addition and subtraction, C_{n-1}, is a straightforward bit-level representation of the carry-out from the summation of the digits at $n - 1$. The translation for arrays produces a nested conditional assignment that dynamically checks the value of the indexing expression (e) to select from among n w-size vectors where n is the declared size of the array and w is the width of each array element. The translation for assignment unrolls the assignment into separate assignments for each of the 1-bit variables. Although not shown, integer constants translate into their binary representations.

Figure 10 shows the type-directed translation from SecVerilogBL typing derivations to SecVerilog types. The translation is only defined for types of well-typed expressions. The rules have the form

$$\frac{\mathcal{P}_1 \qquad ... \qquad \mathcal{P}_n}{\mathcal{T}\llbracket \Gamma; \mathcal{W}; \Theta \vdash e : \tau_{SecVerilogBL}, w \rrbracket} \\ \hookrightarrow \\ \overrightarrow{\tau_{SecVerilog}}$$

where $\mathcal{P}_i$ is a premise in a type rule of SecVerilogBL, and

$$\mathcal{T}\llbracket \Gamma; \mathcal{W}; \Theta \vdash e : \tau_{SecVerilogBL}, w \rrbracket \hookrightarrow \overrightarrow{\tau_{SecVerilog}}$$

denotes that the typing derivation surrounded in brackets translates into the vector of SecVerilog types $\overrightarrow{\tau_{SecVerilog}}$, and the meta-syntax $\overrightarrow{\tau_{SecVerilog}}(i)$ denotes the i^{th} element in the vector of types. The notation $\mathcal{T}\llbracket \tau_{SecVerliogBL} \rrbracket$ is used for recursive translation of types and it denotes $\overrightarrow{\tau_{SecVerilog}}$ such that

$$\mathcal{T}\llbracket \Gamma; \mathcal{W}; \Theta \vdash e : \tau_{SecVerilogBL}, w \rrbracket \hookrightarrow \overrightarrow{\tau_{SecVerilog}}$$

where Γ, $\mathcal{W}$, Θ, and e are all always clear from context. The target-language type may contain types mentioned in the inferrence rules of the derivation.

Note that SecVerilog types are a strict subset of SecVerilogBL types, and that SecVerilogBL types which are not SecVerilog types are abstractions, if-types, and

$$\mathcal{E}[\![x]\!]_{\Gamma;\mathcal{W};\Theta} \triangleq \overrightarrow{x_1, \ldots, x_{\mathcal{W}(x)}}$$

$$\mathcal{E}[\![e_1 \ \texttt{bop} \ e_2]\!]_{\Gamma;\mathcal{W};\Theta} \triangleq \overrightarrow{\mathcal{E}[\![e_1]\!]_{\Gamma;\mathcal{W};\Theta}(i) \ \texttt{bop} \ \mathcal{E}[\![e_2]\!]_{\Gamma;\mathcal{W};\Theta}(i)}\Big|_{i \in (1, \mathcal{W}[\![e_1]\!]_{\Gamma;\mathcal{W};\Theta})}$$

$$\mathcal{E}[\![e_1 + e_2]\!]_{\Gamma;\mathcal{W};\Theta} \triangleq \overrightarrow{\mathcal{E}[\![e_1]\!]_{\Gamma;\mathcal{W};\Theta}(i) \oplus \mathcal{E}[\![e_2]\!]_{\Gamma;\mathcal{W};\Theta}(i) \oplus C_{i-1}}\Big|_{i \in (1, \mathcal{W}[\![e_1]\!]_{\Gamma;\mathcal{W};\Theta})}$$

$$\mathcal{E}[\![e_1 - e_2]\!]_{\Gamma;\mathcal{W};\Theta} \triangleq \overrightarrow{\mathcal{E}[\![e_1]\!]_{\Gamma;\mathcal{W};\Theta}(i) \oplus \neg\mathcal{E}[\![e_2]\!]_{\Gamma;\mathcal{W};\Theta}(i) \oplus C_{i-1}}\Big|_{i \in (1, \mathcal{W}[\![e_1]\!]_{\Gamma;\mathcal{W};\Theta})}$$

$$\mathcal{E}[\![\{e_1, e_2\}]\!]_{\Gamma;\mathcal{W};\Theta} \triangleq \overrightarrow{\mathcal{E}[\![e_1]\!]_{\Gamma;\mathcal{W};\Theta}(i - w_2 + 1)}\Big|_{i \in (w_2, w_1 + w_2)} :: \overrightarrow{\mathcal{E}[\![e_2]\!]_{\Gamma;\mathcal{W};\Theta}(i)}\Big|_{i \in (1, w_2)}$$

$$\text{(where } w_1, w_2 = \mathcal{W}[\![e_1]\!]_{\Gamma;\mathcal{W};\Theta}, \mathcal{W}[\![e_2]\!]_{\Gamma;\mathcal{W};\Theta}\text{)}$$

$$\mathcal{E}[\![e \ll n]\!]_{\Gamma;\mathcal{W};\Theta} \triangleq \overrightarrow{\mathcal{E}[\![e]\!]_{\Gamma;\mathcal{W};\Theta}(i - n)}\Big|_{i \in (1, \mathcal{W}[\![e]\!]_{\Gamma;\mathcal{W};\Theta} - n)} :: \underbrace{0 \ldots 0}_{\text{n times}}$$

$$\mathcal{E}[\![e \gg n]\!]_{\Gamma;\mathcal{W};\Theta} \triangleq \underbrace{0 \ldots 0}_{\text{n times}} :: \overrightarrow{\mathcal{E}[\![e]\!]_{\Gamma;\mathcal{W};\Theta} \ i}\Big|_{i \in (n, \mathcal{W}[\![e]\!]_{\Gamma;\mathcal{W};\Theta})}$$

$$\begin{aligned}
\mathcal{E}[\![x\texttt{[}e\texttt{]}]\!]_{\Gamma;\mathcal{W};\Theta} \triangleq\ & \mathcal{E}[\![e]\!]_{\Gamma;\mathcal{W};\Theta} == \mathcal{E}[\![0]\!]_{\Gamma;\mathcal{W};\Theta} \ ? \ \overrightarrow{x_{0,1}, \ldots, x_{0, \mathcal{W}[\![x[e]]\!]_{\Gamma;\mathcal{W};\Theta}}} : \\
& \quad \mathcal{E}[\![e]\!]_{\Gamma;\mathcal{W};\Theta} == \mathcal{E}[\![1]\!]_{\Gamma;\mathcal{W};\Theta} \ ? \ \overrightarrow{x_{1,1}, \ldots, x_{1, \mathcal{W}[\![x[e]]\!]_{\Gamma;\mathcal{W};\Theta}}} : \\
& \ldots \\
& \quad \mathcal{E}[\![e]\!]_{\Gamma;\mathcal{W};\Theta} == \mathcal{E}[\![n-1]\!]_{\Gamma;\mathcal{W};\Theta} \ ? \ \overrightarrow{x_{n-1,1}, \ldots, x_{n-1, \mathcal{W}[\![x[e]]\!]_{\Gamma;\mathcal{W};\Theta}}} : \\
& \quad \overrightarrow{x_{n,1}, \ldots, x_{n, \mathcal{W}[\![x[e]]\!]_{\Gamma;\mathcal{W};\Theta}}} \\
& \text{(where } n = 2^{\mathcal{W}[\![x[e]]\!]_{\Gamma;\mathcal{W};\Theta}} - 1\text{)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\![x = e]\!]_{\Gamma;\mathcal{W};\Theta} \triangleq\ & \\
& \texttt{begin} \\
& \quad \mathcal{E}[\![x]\!]_{\Gamma;\mathcal{W};\Theta}(1) = \mathcal{E}[\![e]\!]_{\Gamma;\mathcal{W};\Theta}(1); \\
& \quad \ldots; \\
& \quad \mathcal{E}[\![x]\!]_{\Gamma;\mathcal{W};\Theta}(\mathcal{W}[\![x]\!]_{\Gamma;\mathcal{W};\Theta}) = \mathcal{E}[\![e]\!]_{\Gamma;\mathcal{W};\Theta}(\mathcal{W}[\![x]\!]_{\Gamma;\mathcal{W};\Theta}); \\
& \texttt{end}
\end{aligned}$$

Figure 9: Translation from SecVerilogBL expressions to SecVerilog expressions.

$$\textsc{TTrans-Const}\ \frac{}{\mathcal{T}[\![\Gamma;\mathcal{W};\Theta \vdash n : \bot, w]\!] \hookrightarrow \bot} \qquad \textsc{TTrans-Var}\ \frac{\Gamma(x) = \tau \qquad \mathcal{W}(x) = w}{\mathcal{T}[\![\Gamma;\mathcal{W};\Theta \vdash x : \tau, w]\!] \hookrightarrow \overrightarrow{\mathcal{T}[\![\tau]\!](i)}\Big|_{i\in(1,w)}}$$

$$\textsc{TTrans-Logical}\ \frac{\begin{array}{c}\Gamma;\mathcal{W};\Theta \vdash e_1 : \tau_1, w \qquad \Gamma;\mathcal{W};\Theta \vdash e_2 : \tau_2, w \\ \Theta \vdash \tau_1 : \mathtt{int} \rightharpoonup \ell \qquad \Theta \vdash \tau_2 : \mathtt{int} \rightharpoonup \ell \\ \tau = i \mapsto (\tau_1\ i) \sqcup (\tau_2\ i)\end{array}}{\begin{array}{c}\mathcal{T}[\![\Gamma;\mathcal{W};\Theta \vdash e_1\ \mathsf{bop}\ e_2 : \tau, w]\!] \\ \hookrightarrow \\ \overrightarrow{\mathcal{T}[\![\tau_1]\!](i) \sqcup \mathcal{T}[\![\tau_2]\!](i)}\Big|_{i\in(1,w)}\end{array}}\ (\text{when } \mathsf{bop} \in \{\vee \wedge \oplus\})$$

$$\textsc{TTrans-Arith}\ \frac{\begin{array}{c}\Gamma;\mathcal{W};\Theta \vdash e_1 : \tau_1, w \qquad \Gamma;\mathcal{W};\Theta \vdash e_2 : \tau_2, w \\ \Theta \vdash \tau_1 : \mathtt{int} \rightharpoonup \ell \qquad \Theta \vdash \tau_2 : \mathtt{int} \rightharpoonup \ell \\ \tau = i \mapsto \bigsqcup_{j\in(1,i)} ((\tau_1\ j) \sqcup (\tau_2\ j))\end{array}}{\begin{array}{c}\mathcal{T}[\![\Gamma;\mathcal{W};\Theta \vdash e_1\ \mathsf{bop}\ e_2 : \tau, w]\!] \\ \hookrightarrow \\ \overrightarrow{\bigsqcup_{j\in(1,i)} (\mathcal{T}[\![\tau_1]\!](j) \sqcup \mathcal{T}[\![\tau_2]\!](j))}\Big|_{i\in(1,w)}\end{array}}\ (\text{when } \mathsf{bop} \in \{+-\})$$

$$\textsc{TTrans-Concat}\ \frac{\begin{array}{c}\Gamma;\mathcal{W};\Theta \vdash e_1 : \tau_1, w_1 \qquad \Gamma;\mathcal{W};\Theta \vdash e_2 : \tau_2, w_2 \\ \Theta \vdash \tau_1 : \mathtt{int} \rightharpoonup \ell \qquad \Theta \vdash \tau_2 : \mathtt{int} \rightharpoonup \ell \\ \tau = i \mapsto \mathtt{if}(i > w2)\ (\tau_1\ i - w_2 + 1)\ (\tau_2\ i)\end{array}}{\begin{array}{c}\mathcal{T}[\![\Gamma;\mathcal{W};\Theta \vdash \{e_1; e_2\} : \tau, (w_1 + w_2)]\!] \\ \hookrightarrow \\ \overrightarrow{\mathcal{T}[\![\tau_1]\!](i - w_2 + 1)}\Big|_{i\in(w_1, w_1+w_2)} :: \overrightarrow{\mathcal{T}[\![\tau_2]\!](i)}\Big|_{i\in(1,w_2)}\end{array}}$$

$$\textsc{TTrans-LShift}\ \frac{\begin{array}{c}\Gamma;\mathcal{W};\Theta \vdash e : \tau, w \qquad \Theta \vdash \tau : \mathtt{int} \rightharpoonup \ell \\ \tau' = i \mapsto \mathtt{if}\ (i > n)\ (\tau\ i - n + 1)\ \bot\end{array}}{\begin{array}{c}\mathcal{T}[\![\Gamma;\mathcal{W};\Theta \vdash e << n : \tau', w]\!] \\ \hookrightarrow \\ \overrightarrow{\mathcal{T}[\![\tau]\!](i - n + 1)}\Big|_{i\in(1,w-n)} :: \underbrace{\bot \ldots \bot}_{\text{n times}}\end{array}} \qquad \textsc{TTrans-RShift}\ \frac{\begin{array}{c}\Gamma;\mathcal{W};\Theta \vdash e : \tau, w \qquad \Theta \vdash \tau : \mathtt{int} \rightharpoonup \ell \\ \tau = i \mapsto \mathtt{if}\ (i > w - n)\ \bot\ (\tau\ i + n)\end{array}}{\begin{array}{c}\mathcal{T}[\![\Gamma;\mathcal{W};\Theta \vdash e >> n : \tau, w]\!] \\ \hookrightarrow \\ \underbrace{\bot \ldots \bot}_{\text{n times}} :: \overrightarrow{\mathcal{T}[\![\tau]\!](i + n)}\Big|_{i\in(n,w)}\end{array}}$$

$$\textsc{TTrans-ArrIndex}\ \frac{\begin{array}{c}\Gamma(x) = \tau_x \qquad \mathcal{W}(x) = w \\ \Gamma;\mathcal{W};\Theta \vdash e : \tau_e, w_e \\ \Theta \vdash \tau_x : \mathtt{int} \rightharpoonup \mathtt{int} \rightharpoonup \ell \\ \Theta \vdash \tau_e : \mathtt{int} \rightharpoonup \ell \\ \tau_e' = i \mapsto \bigsqcup_{j\in(j,w_e)} \tau_e\ j\end{array}}{\begin{array}{c}\mathcal{T}[\![\Gamma;\mathcal{W};\Theta \vdash x[e] : \tau_e' \sqcup (\tau_x\ e), w]\!] \\ \hookrightarrow \\ \Big(\bigsqcup_{i\in(1,w_e)} \mathcal{T}[\![\tau_e]\!](i)\Big) \sqcup \mathcal{T}[\![\tau_x\ e]\!]\end{array}}$$

Figure 10: Type-directed translation from SecVerilogBL types to SecVerilog types.

case-types. The if-types and case-types are straightforwardly translated into functions fully-applied to program variables (which are SecVerilog types) so these translations are not shown. Also note that in the translation rules shown in Figure 10, all resulting types have kind ℓ and are therefore SecVerilog types.

It is straightforward to check that the SecVerilog program after transformation is semantically equivalent. The soundness result is obtained by showing that the translation is also type-preserving. That is, well-typed SecVerilogBL programs that do not contain downgrading translate into well-typed SecVerilog programs. Since well-typed SecVerilog programs enforce observational determinism, SecVerilogBL programs share the same result. We now show a proof of the type-preservation result for commands.

In addition to the well-formedness requirements of type environments in SecVerilog, SecVerilogBL environments also require the following for a SecVerilogBL environment to be well-formed:

Definition 1 (Well-Formedness of Environments) *An environment $\Gamma; \mathcal{W}; \Theta$ is well-formed, written $\vdash \Gamma; \mathcal{W}; \Theta$, if for all $x \in \Gamma$ such that $\Gamma(x) = \tau$, $\mathcal{W}(x) = w$, and $\Theta \vdash \tau : \texttt{int} \rightharpoonup \ell$, we have $(\tau\ i)$ is in the image of τ for all $i \in (1, w)$. Otherwise, $\Theta \vdash \tau : \texttt{int} \rightharpoonup \texttt{int} \rightharpoonup \ell$ and there is some n such that $(\tau\ j)$ is in the image of τ for all $j \in (1, n)$ and $(\tau\ j\ i)$ is in the image of $(\tau\ j)$ for all $i \in (1, w)$.*

Lemma 1 *If e is a SecVerilogBL expression, then for all $\Gamma, \mathcal{W}, \Theta$ such that $\vdash \Gamma; \mathcal{W}; \Theta$, $\Gamma; \mathcal{W}; \Theta \vdash e : \tau, w$,and $\Theta \vdash \tau : \texttt{int} \rightharpoonup \ell$, let $(\tau\ i)$ is defined denote that $(\tau\ i)$ is in the image of τ, then $(\tau\ i)$ is defined and of kind ℓ for all $i \in (1, w)$.*

Proof. By induction over the type rules of SecVerilogBL

- T-Const: trivial.
- T-Var: by the definition of $\vdash \Gamma; \mathcal{W}; \Theta$
- T-Logical: By typing rule $\mathcal{W}[\![e]\!]_{\Gamma;\mathcal{W};\Theta} = \mathcal{W}[\![e_1]\!]_{\Gamma;\mathcal{W};\Theta} = \mathcal{W}[\![e_2]\!]_{\Gamma;\mathcal{W};\Theta} = w$ $\Theta \vdash \tau_i \vdash \texttt{int} \rightharpoonup \ell$ for $i \in \{1, 2\}$. By the induction hypothesis $(\tau_i\ j)$ is defined for all $j \in (1, w)$ and $i \in \{1, 2\}$. Therefore, $(\tau\ j)$ is defined for all $j \in (1, w)$. By typing rule τ is of kind $\texttt{int} \rightharpoonup \ell$ and so its application to i is of kind ℓ.
- T-Arith: similar to T-Logical.
- T-Concat: $\mathcal{W}[\![e]\!]_{\Gamma;\mathcal{W};\Theta} = w_1 + w_2$. We show that $(\tau\ i)$ is defined for $i \in (1, w_2)$ and for $i \in (w_2, w_1 + w_2)$. $(\tau\ i) = (\tau_2\ i)$ for $i \in (1, w_2)$ and by the induction hypothesis $(\tau_2\ i)$ is defined and so is $(\tau\ i)$. $(\tau\ i) = (\tau_1\ i - w_2 + 1)$ for $i \in (w_2, w_1 + w_2)$, and since $i - w_2 + 1 \in (1, w_1)$ for $i \in (w_2, w_1 + w_2)$, $(\tau_1\ i - w_2 + 1)$ is defined and so is $(\tau\ i)$. By typing rule τ is of kind $\texttt{int} \rightharpoonup \ell$ and so its application to i is of kind ℓ.
- T-LShift: $\mathcal{W}[\![e«n]\!]_{\Gamma;\mathcal{W};\Theta} = w$. For the case in which $n < w$ we show that $(\tau'\ i)$ is defined for $i \in (1, n)$ and for $i \in (n, w - n)$. $(\tau'\ i) = (\tau\ i - n + 1)$ for $i \in (n, w + n)$ and since $i - n + 1 \in (1, w)$ $(\tau\ i - n + 1)$ is defined by the induction hypothesis. Otherwise $(\tau'\ i) =\perp$. If $n >= w$, then $(\tau'\ i) =\perp$. By typing rule τ' is of kind $\texttt{int} \rightharpoonup \ell$ and so its application to i is of kind ℓ.

- T-RShift: Similar to T-LShift.

- T-ArrIndex: By the definition of $\vdash \Gamma; \mathcal{W}; \Theta$, there is some n such that $(\tau_x\ j\ i)$ is defined for all $j \in (1, n)$ and for all $i \in (1, w)$. Though not shown in the typing rule for conciseness, we require that n is greater than the maximum number expressible in w_e bits, so τ_x is defined over the range from 1 to all possible valuations of e and $\tau_x\ e$ is defined over $(1, w)$ By the induction hypothesis τ_e is defined over $(1, w_e)$, and so τ_e' is defined and of kind $\texttt{int} \rightharpoonup \ell$. Since τ_x is applied to e it also of kind $\texttt{int} \rightharpoonup \ell$, the application of $\tau_e' \sqcup (\tau_x\ e)$ to an i is of kind ℓ.

□

Lemma 2 (Type Preservation of Expressions) *If e is a SecVerilogBL expression, then for all $\Gamma, \mathcal{W}, \Theta$ such that $\vdash \Gamma; \mathcal{W}; \Theta$, $\Gamma; \mathcal{W}; \Theta \vdash e : \tau, w$, $\Theta \vdash \tau : \texttt{int} \rightharpoonup \ell$, and there exists a derivation of $\mathcal{T}[\![\Gamma; \mathcal{W}; \Theta \vdash e : \tau, w]\!] \hookrightarrow \overrightarrow{\tau_{SV}}$, let $\mathcal{E}[\![e]\!]_{\Gamma;\mathcal{W};\Theta} = e_1, ..., e_n$ and $\overrightarrow{\tau_{SV}} = \tau_{SV,1}, ...\tau_{SV,m}$, then $[\![\Gamma; \mathcal{W}; \Theta]\!] \vdash e_1 : \tau_{SV,1}, ..., [\![\Gamma; \mathcal{W}; \Theta]\!] \vdash e_w : \tau_{SV,w}$*

Proof. By induction over the translation rules of expressions using the type translation rules of SecVerilogBL, the type rules of SecVerilog [41], Lemma 1, the definition of well-formedness, and the definition of $[\![\Gamma; \mathcal{W}; \Theta]\!]$. As before, let $(\tau\ i)$ is defined denote that $(\tau\ i)$ is in the image of τ.

- v: by the definition of $[\![\Gamma; \mathcal{W}; \Theta]\!]$, TTrans-Var, and Lemma 1.

- e_1 $\texttt{bop}$ e_2: By T-Logical (and T-Arith) $\mathcal{W}[\![e_1]\!]_{\Gamma;\mathcal{W};\Theta} = \mathcal{W}[\![e_2]\!]_{\Gamma;\mathcal{W};\Theta} = w$, and by Lemma 1, $(\tau_1\ i)$ and $(\tau_2\ i)$ are both defined and of kind ℓ for $i \in (1, w)$. Since e is well-typed, by TTrans-Logical or TTrans-Arith, the typing derivation of e translates into $\overrightarrow{(\mathcal{T}[\![\tau_1]\!](i)) \sqcup (\mathcal{T}[\![\tau_2]\!](i))}\Big|_{i \in (1,w)}$ if $\texttt{bop}$ is logical or $\overrightarrow{\bigsqcup_{j \in (1,i)} (\mathcal{T}[\![\tau_1]\!](j) \sqcup \mathcal{T}[\![\tau_2]\!](j))}\Big|_{i \in (1,w)}$ if $\texttt{bop}$ is arithmetic.In either case Lemma 2 holds of e_1 and e_2 by the induction hypothesis. The typing rule of binary operators in SecVerilog requires that the type of e is the join of the types of e_1 and e_2. For both arithmetic and logical operators the translation is the join of the type of e_1 at index i with the type of e_2 at index i, and so Lemma 2 holds of e.

- $\{e_1, e_2\}$: By Lemma 1, $(\tau_1\ i - w_2 + 1)$ is defined and of kind ℓ for $i \in (w_2, w_1 + w_2)$ and $(\tau_2\ i)$ is defined and of kind ℓ for $i \in (1, w_2)$. The formal language SecVerilog does not support concatenations. However, the typing derivation of e translates into $\overrightarrow{\mathcal{T}[\![\tau_1]\!](i - w_2 + 1)}\Big|_{i \in (w_1, w_1+w_2)} :: \overrightarrow{\mathcal{T}[\![\tau_2]\!](i)}\Big|_{i \in (1, w_2)}$ which is the concatentation of the types of e_1 and e_2 shifted appropriately. By the induction hypothesis Lemma 2 holds of e_1, and so the bits of e in range $(w_1, w_1 + w_2)$ are well-typed. Similarly, Lemma 2 holds of e_2 and so the remaining bits are also well-typed. Since by the definition of $\mathcal{E}[\![\{e_1, e_2\}]\!]_{\Gamma;\mathcal{W};\Theta}$, the bits of e_1 and e_2 are concatenated in the same way as their types, each bit of $\mathcal{E}[\![e]\!]_{\Gamma;\mathcal{W};\Theta}$ is well-typed in SecVerilog by the corresponding bit of $\mathcal{T}[\![\tau]\!]$ (by the SecVerilog typing rule of variables and the definition of $[\![\Gamma; \mathcal{W}; \Theta]\!]$).

- e'«n, e'»n: The formal language SecVerilog does not support shifts. This case is similar to $\{e_1, e_2\}$, noting that the bits of e which are not bits of e' are always zero, and by the SecVerilog typing rule for constants are well-typed with type $\bot$.

- x[e]: By T-ArrIndex $\mathcal{W}[\![e]\!]_{\Gamma;\mathcal{W};\Theta} = w_e$, and the maximum valuation of e is n as defined in the translation rule. By the definitions of $\vdash \Gamma; \mathcal{W}; \Theta$ and $[\![\Gamma; \mathcal{W}; \Theta]\!]$ and Lemma 1 ($\tau_x\ j\ i$) is defined and of kind ℓ for $j \in (1, n)$ and $i \in (1, w)$, so $P(x_{i,j})$ holds for $j \in (1, n)$ and $i \in (1, w)$. The definition of $\mathcal{E}[\![x[e]]\!]_{\Gamma;\mathcal{W};\Theta}$ evaluates to some SecVerilog variable $x_{i,j}$ so by the definition of $[\![\Gamma; \mathcal{W}; \Theta]\!]$ and the typing rule of SecVerilog variables, Lemma 2 holds of x[e].

□

Lemma 3 (Type Preservation) *If c is a SecVerilogBL command,* Γ *is a type context, and* $\mathcal{W}$ *is a width environment such that* $\Gamma; \mathcal{W}; \Theta \vdash c$*, then* $[\![\Gamma; \mathcal{W}; \Theta]\!] \vdash c$*.*

Proof. By induction on the translation of commands. The only interesting case is assignment. Because the translation for expressions is type-preserving when those expressions have `int` $\rightharpoonup \ell$ types (Lemma 2), and all expressions appearing in commands have such types (by assignment rule), and the SecVerilogBL type rule applies the type function for both sides of the assignment to each integer less than the width of the assigned expression, w, then the translation will result in w separate SecVerilog assignments, which will all type-check. □

Theorem 1 (Observational Determinism) *If c is a SecVerilogBL command,* Γ *is a type context, and* $\mathcal{W}$ *is a width environment such that* $\Gamma; \mathcal{W}; \Theta \vdash: c$*, then c obeys observational determinism.*

Proof. This theorem follows directly from Lemma 3 and the proof of observational determinism for SecVerilog [42]. □

5 Architecture Extensions

5.1 Return Response Access Controls

In the baseline TrustZone prototype, memory transactions are controlled with access controls on requests but not on responses. This design is secure if the processor's memory hierarchy is functionally correct; normal-world cores should never receive a response containing secure-world data because the corresponding request would have been denied. Unfortunately, such functional correctness is difficult to prove statically, especially for the complex memory hierarchies of modern processors.

To address this challenge, the prototype is extended with run-time access controls that ensure the NS bit of each memory response matches the NS bit of the receiving core. The added access controls enforce the invariant that secure-world responses cannot be read by normal-world cores, and enable the type system to statically prove the security of the processor. Note that typechecking fails if the response check is functionally incorrect and does not enforce the invariant.

5.2 Potential ISA Extensions

In TrustZone, secure-world programs are trusted and allowed to access both secure and normal memory. The secure world software's intention to access either secure or normal memory is communicated implicitly through the phyiscal memory address. Existing TrustZone implementations assume that secure-world software intends to downgrade the accessed data if the memory address points to the normal-world memory. As a result, a bug in the memory address calculation may cause information flow to/from normal-world memory even when secure-world software did not intend to downgrade information.

Such incorrect downgrading can be avoided by extending the ISA with loads and stores that explicitly indicate whether the software intends to access normal world memory or secure world memory. Indicating the intended world requires a single bit. This extension allows downgrading to be performed based on explicit information provided by the software rather than relying on information implicit in the possibly erroneous memory address. These new instructions would remove a functional correctness assumption about the hardware.

6 Evaluation

6.1 Verification with No Bugs

Here, we discuss the results of the information flow analysis for our TrustZone implementation when no security vulnerabilities are introduced. Later, we introduce bugs into the processor to evaluate the effectiveness of information flow at detecting vulnerabilities. The implementation of TrustZone passes type checking when analyzed by SecVerilogBL. Typechecking formally guarantees that, aside from variables which affect downgraded expressions, there is no violation of the information flow policy expressed in the code. Downgraded expressions explicitly permit exceptions to the policy

Table 2 summarizes the uses of downgrading in each microarchitecture component of our implementation of TrustZone. The table categorizes these downgrading expressions as confidentiality exceptions ($C \to P$), integrity exceptions ($U \to T$), or both. The downgrade expressions are further classified by the type of variable – data (Data), address or control (AddrCtrl), NS-bit (NS), or control register (CReg) – at the source and destination using the notation `source` →`destination`. Address or control variables include valid/ready variables in the network, and instruction decode outputs and stall variables in the cores. The numbers in parenthesis indicate the number of downgrade expressions of that form.

SecverilogBL enforces a timing-sensitive security property; however, the threat model of the architecture under study does not address timing channels. Timing channels will cause type errors even though they are not considered threats. Therefore, timing channels must be distinguished from other kinds of illegal information flows. Categorizing flows based on variable type is useful for doing so; timing channels in hardware typically originate from an address or control variable, but are not directly derived from data.

Component Name	$C \rightarrow P$	$U \rightarrow T$
Core Pipeline	$AddrCtrl \rightarrow NSB$ (1)	
	$AddrCtrl \rightarrow AddrCtrl$ (1)	
L2 Cache	$Data \rightarrow CReg$ (1)	
	$AddrCtrl \rightarrow AddrCtrl$ (2)	
	$AddrCtrl \rightarrow NSB$ (12)	
Network	$AddrCtrl \rightarrow AddrCtrl$ (4)	
	$AddrCtrl \rightarrow NSB$ (1)	
DMA Engine	$AddrCtrl \rightarrow NSB$ (1)	
Debug Interface	$AddrCtrl \rightarrow NSB$ (1)	
Memory Arbiter	$AddrCtrl \rightarrow AddrCtrl$ (2)	
Memory Access Control Module	$Data \rightarrow CReg$ (1)	
	$AddrCtrl \rightarrow NSB$ (4)	
Main Memory	$Data \rightarrow Data$ (1)	$Data \rightarrow Data$ (1)
	$AddrCtrl \rightarrow AddrCtrl$ (2)	

Table 2: Downgrading expressions in our prototype.

The following paragraphs summarize why downgrading was needed based on the variable type categorization.

Data → Data TrustZone allows the secure world (CT) to read or write normal-world (PU) memory, contrary to the lattice policy. When secure world writes to normal-world memory, confidentiality is violated. Similarly, secure-world reads from normal-world could violate integrity. The data must be downgraded to permit the intended behavior. This use of downgrading is safe because the secure world is trusted.

Data → CReg Control registers are labeled PT, because they are used to control both secure-world and normal-world operations. However, control registers can also be modified by the secure-world. This is a violation of the confidentiality policy (since it is a flow from CT to PT), and must be explicitly permitted with downgrading. Note that normal world is still prevented from setting control registers since downgrading is performed only for a write from the secure world.

AddrCtrl → Data, AddrCtrl → AddrCtrl Illegal flows from address/control variables to address, control, and data variables are caused by timing interference between security levels. Timing interference leaks information from address/control variables ($AddrCtrl$) but not data variables ($Data$).

AddrCtrl → NSB Resources which are used by both worlds cause flows from control variables to the NS bit. Figure 11 shows a representative example of this type of flow. It shows a bus arbiter that accepts requests from both cores that could be executing in either world. The output NS bit becomes the NS bit of the core that is granted access. Here, the NS bit is labeled PT because its integrity needs to be protected from the normal world, leading to information flow from CT/PU to PT. In the core, the NS bit is changed by an instruction with label `world(ns)`, similarly requiring downgrading. Here, downgrading affects the timing of the NS bit change, but does not introduce a vulnerability.

The information flow analysis in SecVerilogBL formally proves timing-sensitive noninterference. However, explicit uses of downgrading expressions are used to weaken

```
input          {world(ns1)} cpu1_valid;
input          {world(ns2)} cpu2_valid;
input          {PT}           ns1;
input          {PT}           ns2;
output         {PT}           ns_out;
...
if      (cpu1_valid == 1) ns_out <= ns1;
else if (cpu2_valid == 1) ns_out <= ns2;
...
```

Figure 11: A flow from control to NS due to resource arbitration.

```
input          {PT}             ns;
input [dw-1:0] {world(ns)}      data_in;

reg   [dw-1:0] {PT}             part_reg;
    ...
    // Detected bug.
    part_reg <= data_in;
    ...
    // Correct code.
    if (ns == 0) part_reg <= downgrade(data_in, PT);
    ...
```

Figure 12: A detected access control omission.

noninterference. We argue that in our implementation, downgrading is used only under the authority of the secure world, and therefore that information release cannot be controlled by the normal world. To show that this is true, we note that information is never downgraded if the secure world never performs an operation. In other words, downgrading can be removed if the secure world is hard-coded to not execute. Both $Data \rightarrow Data$ (secure-world reads/writes to memory) and $Data \rightarrow Creg$ (secure-world writes to control registers) flows happen under an `if` condition that checks if an access is from the secure world. These downgraded information flows never happen if there is no secure-world access. The flows from *AddrCtrl* variables cause timing contention. Downgrading is unnecessary if there is no secure-world access because `ns_out` will always be 1 (normal world). Since information is never downgraded when the secure world is inactive, this suggests that information release cannot be controlled by the normal world.

6.2 Security Bug Detection

Here, we study the effectiveness of the proposed information flow analysis at detecting security bugs. We developed a set of security bugs based on reported vulnerabilities in commercial products [39, 40] as well as possible mistakes.

Bugs 1-5: Access Control Omission In TrustZone, access controls ensure that trusted/confidential state can only be accessed by the secure world. If access control checks are left out, security is violated. We model five bugs, which omit access controls for 1) the control register that partitions main memory between worlds, 2) the main memory, 3) the debug interface, 4) the L2 cache prefetch buffer, and 5) the L2 cache blocks. Bug 3 is inspired by a back door in the Actel ProASIC3 [30], Bug 4 models a vulnerability found in an AMD processor [13], and Bug 5 models a privilege escalation attack in Intel processors that support SMM mode [40]. Figure 12 shows how omitted access controls for the partition register are detected. `data_in` has the label `CT` when `ns` is `0` and `PU` when `ns` is `1`. The code on line 7 is detected as a bug because the type system cannot prove that `data_in` is trusted. The correct code on line 10 adds access controls to check that the `ns` bit is 0, implying that `data_in` is trusted in this context. Bugs 2-4 are detected and fixed similarly.

Bug 5: Cache Poisoning We emulate and detect a subtle vulnerability found in Intel processors [40]. The vulnerability allows a user-mode process to execute arbitrary code in System Management Mode (SMM), the highest privilege level. SMM mode is only used to execute SMM handlers – interrupt handlers requiring such high privilige. In the vulnerable processor, the region of physical memory which stores SMM handlers is protected by access controls in the memory interface. A control register can mark this region as uncacheable, and it does so by default. However, the control register can be modified without SMM privilege, allowing an attacker to make the SMM memory cacheable. Then, the attacker can write to the address of an SMM interrupt handler. Memory access controls reject the write, but it is cached. Subsequent executions of the handler address will hit in the cache and execute the attacker's code. We modeled this vulnerability in our processor by removing the NS tags and access controls from the L2 while keeping memory access controls. We added a control register that sets cacheability for the secure world but can be modified by either world. The bug is detected because the cache lines can receive data that is from either world, but no access controls are present.

Bug 6: NS-bit Flip Memory requests are transmitted with an NS bit that indicates the security level of the request. This bug inverts the NS bit so that a memory request from a normal-world core is interpreted as a secure-world access. This bug is detected because flipping an NS bit changes the type of dependently typed variables. In the network, the input and output data variables both have types that depend on the NS bit. If the input and output NS bits do not match, the security labels of the input and output data will also not match causing an error. Even if the bit is flipped in multiple places, the error will be detected because eventually the input and output types will not match. This demonstrates the benefit of information flow analysis, which tracks the propagation of data throughout the design.

Bug 7: Network Routing Bug This bug models a network implementation that leaks secrets by incorrectly routing a response from the secure-world memory to a normal-world core. In our TrustZone implementation, the bug is prevented by the memory response access control checks and the L1 cache tags. To test the bug, we removed these access controls and used an L1 cache that keeps data from only one security domain at a time. This bug is detected at the interface between the L1 caches and the on-chip network. Without access controls at the response ports, the type system cannot prove that the NS bit of a response matches the NS bit of a cache.

Bug 8: World Switch Bug For a world switch (i.e., context switch) from normal world to secure world, the processor pipeline must complete all in-flight instructions before changing the NS bit. Otherwise, in-flight normal-world instructions will execute with escalated privilege. We model a vulnerable mode switch by omitting the pipeline drain step. This bug is detected because changing the NS bit causes the labels of

```
// 0x0000-0x8000 is the secure-world memory.
// the rest is the normal-world memory.

// Code common to both bugs
wire [0:31] {world(ns)} addrout, addrin;
wire [0:31] {world(ns)} datain;
reg  [0:31] {CT} data_sec;
reg  [0:31] {PU} data_norm;
always@(*) begin
    if((ns == 0) && (addrout > 'h8000))
        data_norm = downgrade(datain, PU);
    else
        data_sec  = datain;
end
// Bug 9-1: not detected
assign addrout = (addrin <= 'h8000) ?
                 addrin + 'h8000 : addrin;
// Bug 9-2: detected
wire {PU} normal_world_trigger;
assign addrout = (normal_world_trigger ?
                 addrin + 'h8000 : addrin;
```

Figure 13: Bug 9: memory address change bugs.

the dependently typed registers to change. SecVerilogBL prevents label changes from leaking information by dynamically clearing register contents when labels are changed.

Bug 9: Memory Address Change Bug To understand the limitations of HDL-level IFC, we constructed two bugs that change the memory address at the memory interface. Figure 13 illustrates the bugs. In both cases, the address is changed from the secure-world region to the normal-world region so that a write into secure-world memory gets stored in normal-world memory. This allows the normal world to read data that should be stored in the secure-world memory. Bug 9-1 is not detected, because downgrading allows the secure-world access to write data into the normal-world memory. On the other hand, Bug 9-2 is detected, because the change in the secure-world memory address is triggered by a normal-world variable (`normal_world_trigger`). The examples show that functional bugs in the secure world may lead to undetected bugs through downgrading, but only if there is no influence from the normal world. Vulnerabilities that do not affect downgraded variables are always detected.

Other Bugs Hicks et al. [13] proposed SPECS, a run-time bug detector. They evaluated it by implementing 14 bugs in the OpenRISC processor. The bugs included privilege escalation, register target/source redirection, interrupt-register contamination, interrupt disabling, code injection, jump instruction disabling, and others. While we could not implement those bugs in our processor architecture (e.g., our prototype does not have protection rings or an MMU), we reviewed the HDL code studied for SPECS. These bugs all allow a user-mode process to change supervisor-mode variables. Therefore, these bugs should all be detected by information flow analysis if user-mode variables are labeled `PU` and supervisor-mode variables are labeled `CT`.

6.3 Overhead

Programming Overhead Table 3 shows the number of lines of code for the unverified version of our processor (Unverified) and the verified version with security labels

Component	Unverified	Verified	Percentage
Top-Level Module	1391	1412	1.5%
Processor	3474	3504	0.86%
L1 Cache	1250	1308	4.6%
Access Control	0	75	N/A
On-chip Network	2122	2557	1.7%
L2 Cache	2976	3093	3.9%
DMA controller	525	549	4.6%
Debug interface	350	369	5.4%
Main memory	974	1015	4.2%
Library Modules	2780	2818	1.4%
Total	16234	16700	2.9%

Table 3: Programming overhead (lines of code).

(Verified). We emphasize that the verification procedure is purely static and and performed at compile time. However, the implementation changes slightly 1) to add extra variables specifically for encoding dependent types and 2) to aid the program analysis phase that estimates the run-time values of dependent types. The code increases by 2.9%.

Area, Power, and Performance Overheads The processor was synthesized using Cadence Design Compiler using a standard 90nm library to obtain performance, area, and power results. We found that the extra code for security verification has no impact on the maximum clock frequency. Also, there is no CPI difference between the unverified and verified versions. Therefore, the performance is identical for the unverified and verified designs. The area and power overheads are negligible (0.37% and 0.32%).

7 Related Work

Secure Hardware Architecture Though this work studied an implementation of TrustZone [20], we believe our methodology is also applicable to other secure architectures proposed in industry [2, 1, 3] and academia [33, 9, 10, 5]. These architectures are similar to TrustZone since they also aim to isolate critical software. Information flow checking can identify violations of strict isolation.

Sinha et al. [29, 28] verify the security of programs which use SGX enclaves to ensure that hardware security features are used correctly. Gollamudi et al. [12] use information flow in software to partition programs into TrustZone worlds or SGX enclaves. This work is complementary to ours, which verifies the hardware security features.

Information Flow Tracking in Hardware. DIFT [32] is the earliest use of information flow tracking in hardware. It applies information flow tracking coarsely at the architecture level. GLIFT [37, 23, 24, 35, 36, 14] performs information flow analysis on hardware at the gate level. The earliest GLIFT [37] approach incurs high performance, area, and energy overheads by inserting additional logic. Later work applies GLIFT to simulated circuits [23, 24] and avoids overhead in synthesized hardware. Because simulating every possible state in large designs is infeasible, the approach is used to check either small designs or only a set of states reachable by particular software that is designed together with hardware [36]. Oberg et al. [25] propose a technique to separate timing flows from non-timing flows. Sapper [17], Caisson [18], and SecVerilog [42] all apply information flow type systems at the hardware description language level. SecVerilog is used in this work, but our methodology can also be applied to other secure HDLs.

This study is the first to use information flow at the HDL level to verify a multi-core security architecture. The processor includes a shared cache, pipelines, a shared network, a DMA engine, and a debug interface. Other studies have used information flow to verify simpler hardware. Oberg et al. use GLIFT to verify an I2C hub and USB controller [24]. Tiwari et al. [36] identify a small, security-critical portion of a CPU and use StarLogic to verify it. However, the authors do not include shared un-core components. Previous studies verified hardware designs that enforce strict noninterference. In contrast, this work solves challenges in enforcing relaxed policies used in practice. For example, commercial architectures are seldom concerned with timing attacks [2, 1, 20].

Hardware Security Bugs Countless vulnerabilities have been found in real hardware designs. Manufacturers release errata documents enumerating known bugs [4, 8]. Hicks et al. [13] analyzed 301 bugs from commercial errata documents and found that 28 were security-critical. Wojtczuk et al. [39] found a software-exploitable hardware vulnerability that allows an attacker to escalate from user-space privilege to a privilege level above the kernel in an Intel processor. The same authors [40] use faulty DMA transaction messages to escape VM-isolation in processors that support Intel VT-d. Lee et al. [15] used uninitialized data vulnerabilities in GPUs to leak data from co-resident users. Several of these vulnerabilities formed the basis for bugs that we demonstrate can be detected by SecVerilog.

Language-Level Information Flow Control Language-based information flow control is a widely studied area [27]. Yet, applying information flow type systems to hardware description languages is a relatively new research direction, only studied by a few papers [42, 17, 18]. One important question in IFC HDL designs is how to allow security domains to share hardware. Sapper [17] inserts dynamic checks that securely permit sharing, but may induce functional errors at run time. To permit sharing with purely static checking, SecVerilog [42] uses a dependent type system.

The use of dependent types for accurate tracking of information flow started with JFlow [21], which uses value-indexed labels. Later systems [38, 43, 16, 22, 19] have introduced more expressive forms of dependent labels, exploring trade-offs between needed expressive power and tractability of analysis. For expressive type systems to be useful in practice, type checking must be tractable and efficient for real-world code. The per-bit and per-element dependent labels we add to SecVerilog are not supported by previous dependent type systems for imperative languages. They are an important feature for future security-typed HDLs because they offer valuable expressive power while remaining tractable: a sweet spot in the trade-off space. The proposed type system extensions are also non-trivial because they need to statically and precisely propagate bit vector labels through bit-level operations that may combine or shift bits.

8 Conclusion

This work shows that the security of an ARM TrustZone implementation on a multi-core processor can be verified at design time using information flow analysis. This study is the first to verify a complex security architecture with information flow. We found that static verification of practical security policies on a multi-core introduces new technical challenges, and show how to solve them through novel type system extensions and the addition of simple, inexpensive run-time hardware checks. This verification methodology provides strong assurance that the processor provides secure isolation for critical software.

9 Acknowledgments

We thank Daniel Lo and the anonymous reviewers for their review of and suggestions about this paper. This research has been sponsored by NSF grant 1513797 and NASA grant NNX16AB09G.

References

[1] Rick Boivie. SecureBlue++: CPU Support for Secure Execution, 2012.

[2] Intel Corporation. Intel Software Guard Extensions Programming Reference, 2014.

[3] Intel Corporation. Intel Trusted Execution Technology Software Development Guide, 2015.

[4] Intel Corporation. Intel Xeon Processor E7-8800/4800/2800 Product Families: Specification Update, 2015.

[5] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security*, 2016.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TCAS*, 2008.

[7] Dorothy E. Denning. A Lattice Model of Secure Information Flow. In *Communications of the ACM*, 1976.

[8] Advanced Micro Devices. Revision Guide for AMD Athlon 64 and AMD Opteron Processors, 2005.

[9] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *MICRO*, 2014.

[10] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *STC*, 2012.

[11] J.A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE S&P*, 1982.

[12] Anitha Gollamudi and Stephen Chong. Automatic Enforcement of Expressive Security Policies Using Enclaves. In *OOPSLA*, 2016.

[13] Matthew Hicks, Cynthia Sturton, Samuel T. King, and Jonathan M. Smith. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *ASPLOS*, 2015.

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

[15] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *IEEE S&P*, 2014.

[16] Peng Li and Steve Zdancewic. Downgrading Policies and Relaxed Noninterference. In *POPL*, 2005.

[17] 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.

[18] 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 *PLDI*, 2011.

[19] Luísa Lourenço and Luís Caires. Dependent information flow types. In *POPL*, 2015.

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

[21] Andrew C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *POPL*.

[22] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *SSP*, 2011.

[23] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Theoretical Analysis of Gate Level Information Flow Tracking. In *DAC*, 2010.

[24] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Information Flow Isolation in I2C and USB. In *DAC*, 2011.

[25] Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. A practical testing framework for isolating hardware timing channels. In *DATE*, 2013.

[26] Andrei Sabelfeld and Andrew C. Myers. A Model for Delimited Information Release. In *IEEE S&P*, 2004.

[27] Andrei Sabelfeld and Andrew C. Myers. Language-based Information-flow Security. *IEEE Journal on Selected Areas in Communications*, 2006.

[28] Rohit Sinha, Manuel Costa, Akash Lal, Nuno Lopes, Sanjit Seshia, Sriram Rajamani, and Kapil Vaswani. A Design and Verification Methodology for Secure Isolated Regions. In *PLDI*, 2016.

[29] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *CCS*, 2015.

[30] Sergei Skorobogatov and Christopher Woods. Breakthrough Silicon Scanning Discovers Backdoor in Military Chip. In *CHES*, 2012.

[31] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *CSFW*, 2003.

[32] G. Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ACM Sigplan Notices*, 2004.

[33] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *ISCA*, 2005.

[34] Jakub Szefer and Ruby B. Lee. Architectural Support for Hypervisor-Secure Virtualization. In *ASPLOS*, 2012.

[35] 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.

[36] 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.

[37] 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 *ASPLOS*, 2009.

[38] Stephen Tse and Steve Zdancewic. Run-Time Principals in Information-Flow Type Systems. In *IEEE S&P*, 2004.

[39] Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning. Technical Report invisiblethingslab.com/resources/misc09/smm*cache*fun.pdf, 2009.

[40] Rafal Wojtczuk and Joanna Rutkowska. Following the White Rabbit: Software Attacks Against Intel VT-d Technology. Technical Report http://theinvisiblethings.blogspot.com/2011/05/following-white-rabbit-software-attacks.html, 2011.

[41] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for efficient control of timing channels. Technical Report http://hdl.handle.net/1813/36274, Cornell University, 2014.

[42] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *ASPLOS*, 2015.

[43] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2–3), March 2007.