CELLIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL

Flavien Solt
ETH Zurich

Ben Gras
Intel Corporation

Kaveh Razavi
ETH Zurich

Abstract

Dynamic Information Flow Tracking (dynamic IFT) is a well-known technique with many security applications such as analyzing the behavior of a system given an input and detecting security violations. While there are many widely used open dynamic IFT solutions that scale to large software, the same level of support is unfortunately lacking for hardware. This gap is becoming more pronounced with the increasing complexity of open-source hardware and the plethora of recent hardware attacks.

We introduce CELLIFT, a new design point in the space of dynamic IFT for hardware. CELLIFT leverages the logical macrocell abstraction (e.g., an adder) to achieve scalability, precision and completeness when instrumenting a given Register Transfer Level (RTL) hardware design. Cell-level dynamic IFT does not suffer from the scalability problems that are inherent to lower levels of abstraction such as gates, yet it achieves completeness given the limited number of cell types. We show the versatility of CELLIFT by instrumenting five distinct RISC-V designs, one of which is a complete SoC. The only existing complete solution already fails to instrument two of these designs. Our extensive evaluation using microbenchmarks and standard RISC-V benchmarks on the instrumented designs shows that CELLIFT is 21× to 61× faster than the state of the art in terms of simulation runtime without losing precision. We further show-case concrete applications of CELLIFT in four scenarios by detecting: 1) sources of microarchitectural information leakage, 2) microarchitectural bugs such as Meltdown, 3) speculative vulnerabilities such as Spectre-BCB, and 4) SoC-wide architectural design flaws. We release CELLIFT as open source to enable RTL-level security research for the wider community.

1 Introduction

Despite substantial design verification efforts, there is a continual discovery of security-critical hardware design flaws that are exploitable from software [1, 6, 8, 11, 37, 38, 46, 48, 55, 56, 59, 60, 62, 69, 71, 72]. These vulnerabilities are often difficult to mitigate post-silicon, leaving systems exposed for long periods of time. While critical, current tools and techniques are unfortunately incapable of capturing these issues in large designs. There is hence an urgent need for empowering Electronic Design Automation (EDA) tools for detecting security vulnerabilities during hardware design.

Dynamic Information Flow Tracking. A promising approach for analyzing the state of a system given an input and verifying certain security properties is Information Flow Tracking (IFT). Static IFT either needs to consider all possible states which does not scale beyond very simple designs [13, 17, 20] or it over-approximates, leading to precision problems [65]. In contrast, dynamic IFT only considers changes to the state made in a single execution, allowing it to potentially scale to larger hardware designs for checking design-wide security properties. As an example, many of the recently discovered security flaws [8, 37, 38, 55, 56, 69, 71, 72] can be formulated as a dynamic IFT constraint. While there are many popular dynamic IFT tools that scale to large software [36, 47, 66], the same level of support is currently lacking for hardware. With the increasing popularity of open-source hardware, an open-source dynamic IFT solution that scales to larger designs has the potential to significantly improve security testing and enable new security applications.

CELLIFT. It is possible to instrument a given design at one of the two extreme abstraction levels for dynamic IFT: either by considering its low-level gate netlist [67]; or by considering the high-level Hardware Description Language (HDL) [2]. Unfortunately, both abstraction levels come with significant shortcomings: instrumenting the gates has severe *scalability* issues while instrumenting the HDL requires managing complex language constructs, making it challenging to achieve *completeness*. We make a key observation that instrumenting the *macrocells*, which are at a slightly lower abstraction level than HDL, preserves the benefits of both extremes without

their shortcomings. There are a manageable number of cell[1] types (e.g., adder, shifter, multiplexer, etc.) for which we can create *shadow cell types* that precisely track the information flows and scale comfortably to larger designs. Unlike gates, these shadow cells map efficiently to large units available in commodity CPUs (e.g., registers or arithmetic and logic units), significantly improving the performance of dynamic IFT during simulation. To design these shadow cells, we introduce a generic precise information flow tracking logic scheme called *m-replica* based on cell replication. To make this scheme scale efficiently with increasing cell widths while preserving precision, we exploit the cells' mathematical properties of *monotonicity*, *transportability* and *translatability* that we formally define and leverage for the first time. We then prototype CellIFT, our dynamic IFT solution that instruments a given Register Transfer Level (RTL) design by leveraging the design of our shadow cell types.

To show the versatility of CellIFT, we use it to instrument five RISC-V cores, one of which integrated in a System on a Chip (SoC) which we also instrument to show that CellIFT can successfully be applied on various complex and heterogeneous designs. The only existing (gate-level) solution that can handle generic designs [67] already fails at instrumenting and simulating two of these five designs. Our evaluation shows that compared to instrumenting gates, CellIFT is more precise, more than 5× faster with instrumentation and synthesis while requiring 5× less memory, and more than 21× faster during simulations. We show-case the benefits of CellIFT in four different scenarios using our instrumented designs: first, we show how CellIFT can be used to measure changes to the microarchitectural state as a result of executing an instruction or a memory access. This information can be used to detect various sources of timing-based information leakage [40, 52, 75]. Second and third, we show how CellIFT can enable the detection of microarchitectural vulnerabilities such as Meltdown [38] or MDS [59, 71], or speculative execution attacks such as Spectre [37] respectively. Finally, we use CellIFT to detect design flaws that can lead to architectural security vulnerabilities using various scenarios from a hardware hacking challenge [17].

In summary, we make the following contributions:

- We present a scalable, precise and complete cell-level dynamic IFT design.
- We implement CellIFT based on this new design as new passes into the Yosys open-source synthesizer [74].
- We evaluate CellIFT on five RISC-V designs: Ibex [43], Ariane [76], Rocket [3], BOOM [4] and the PULPissimo SoC from the Hack@DAC'18 competition [17].
- We use CellIFT in several scenarios to show the benefits of scalable and precise dynamic IFT support as part of the hardware design toolchain.

[1] We use the terms cell and macrocell interchangeably.

Open sourcing. To enable reproducibility and to let researchers and practitioners benefit from CellIFT, we publish the source code of CellIFT, the experiments and the instrumented designs at this URL:
https://comsec.ethz.ch/cellift.

2 Background

In this section, we provide a brief background on existing techniques for detecting architectural vulnerabilities, their (in)effectiveness against recent microarchitectural vulnerabilities, and discuss how hardware dynamic IFT mechanisms can provide a better alternative.

2.1 Detecting architectural vulnerabilities

Hardware designers employ various methods in an attempt to detect flaws in the RTL representation. These methods are manual or automatic; automatic methods are local or global.

Manual inspection. Dessouky et al. [17] recently showed that existing verification methods do not effectively cover many of the cross-layer bugs resulting from subtle hardware-software interactions. Hardware designers often resort to manual inspection of the RTL and simulation of hand-crafted input to detect these complex cases. Unfortunately, this approach is cumbersome, error-prone, and incompatible with any form of continuous integration that can catch subtle vulnerabilities introduced after the initial hardware design. Errata of recent complex designs such as 12th generation Intel® Core™ processors contain an overwhelming proportion of such bugs [14].

Local methods. Hardware designers often rely on SystemVerilog assertions (SVA) [50] to ensure correct behavior and capture unintended behavior that can lead to bugs at all design stages. These assertions express local properties such as compliance to a given bus protocol, or compliance of a state machine with some expected properties. Assertion-based verification formal methods such as Formal Property Verification (FPV) [22, 23] can provably check whether these assertions can be triggered in a given RTL design and provide examples when they do. These formal methods unfortunately suffer from state explosion due to their static nature, limiting their scalability. Therefore, hardware designers and verification engineers often deploy Constrained Random Verification (CRV) extensively [49]. CRV tries out a series of signals while respecting some constraints, such as complying with some input bus protocol, to empirically find out whether local assertions can be triggered.

Global methods. Many security properties are not expressible in the form of local assertions. For instance, a local assertion cannot infer the origin or destination of some data transfer and therefore cannot verify confidentiality properties in the general case. Formal methods such as Security

Path Verification (SPV) [9] and Formal Security Verification (FSV) [64] are designed to statically catch unauthorized information flows. However, these methods are reported to suffer from the same scalability issues as local formal methods and typically requires black-boxing parts of the design [15], hampering their ability to detect information flows across the entire design.

2.2 Microarchitectural vulnerabilities

While architectural security vulnerabilities already pose a challenge for existing tools and techniques, microarchitectural security vulnerabilities that do not explicitly collide with architectural specifications pose an even greater challenge. These security vulnerabilities are not only legion [1, 6, 8, 11, 37, 38, 46, 48, 55, 56, 59, 60, 62, 69, 71, 72], but industry leaders still struggle to mitigate them, sometimes requiring multiple generations of attempts before reaching an effective mitigation as seen in the recent MDS class of vulnerabilities [11, 59, 70, 71]. Such vulnerabilities can lead to confidentiality breaches, effectively enabling arbitrary read primitives. Recent work [24] attempts to detect microarchitectural vulnerabilities by exploiting the RTL description, but requires tailoring to a specific design and vulnerability, without providing exploitability insights. We hence urgently need a design-agnostic solution that can detect different classes of vulnerabilities with little effort.

2.3 Dynamic hardware IFT

Hardware dynamic IFT provides the possibility of following how information flows propagate in a design [2, 67]. Confidentiality, integrity, isolation, constant time and design integrity properties are canonical properties covered by dynamic IFT [29]. As opposed to static methods, dynamic IFT does not consider the entire set of possible states in a given design, but instead allows to dynamically prove properties in a specific context. Consequently, it is immune to the state explosion problem.

Dynamic IFT requires: a *mechanism* for tracking information flows and *policies* expressed on top of this mechanism. According to certain policies, signals are temporarily or permanently labeled as *taint sources* or *taint sinks* during runtime, and an alarm is triggered when an information flow from a taint source to a taint sink is detected through the mechanism. Confidentiality policies inspect the data flow from secret data locations (taint sources) to unauthorized entities (taint sinks). Conversely, integrity policies inspect data flows from unauthorized entities (taint sources) to sensitive locations (taint sinks). As an example, Meltdown-type [10] class of vulnerabilities [8, 11, 38, 55, 56, 59, 69, 71, 72] can be expressed as a policy that disallows memory loads from a different domain.

Two hardware dynamic IFT mechanisms have been proposed so far: GLIFT [67] and RTLIFT [2]. They add new elements to the design to support dynamic IFT, but at different levels: GLIFT *instruments* the design at the level of elementary logic gates (AND, NOT, OR and multiplexers), and RTLIFT proposes to instrument the HDL directly. We will show that GLIFT has critical scalability problems, and (to the best of our knowledge) a complete RTLIFT has never been implemented due to the tremendous engineering effort required.

Table 1 summarizes all the verification techniques discussed in this section. While dynamic IFT is an attractive alternative for detecting hardware vulnerabilities, existing solutions such as GLIFT [67] and RTLIFT [2] have severe limitations that hamper their adoption. In the following section, we analyze these limitations and discuss how our new hardware dynamic IFT design addresses them.

Table 1: Classification of design verification methods.

	Local	Global
Static	FPV [22, 23]	SPV [9], FSV [64]
Dynamic	CRV [49]	GLIFT [67], RTLIFT [2]

3 Dynamic Hardware IFT Using Cells

There are three properties that are significant for any hardware dynamic IFT mechanism to see adoption: first, it should be able to operate on any given (valid) digital design (i.e., the *completeness* property). Second, it should scale to large designs with high instrumentation performance and usable simulation overhead (i.e., the *scalability* property). Finally, it should faithfully propagate tainted signals while minimizing the taint spilled to additional signals in the design (i.e., the *precision* property). Unfortunately, none of the existing techniques cover all these important properties together.

The most common strategy is instrumenting designs with IFT logic at the gate level [30–33, 67]. While achieving completeness due to the limited number of different gates, it suffers from scalability problems: since IFT logic construction occurs after logic synthesis (i.e., once the design is expressed as a list of elementary logic gates), it incurs an exponential worst-case time complexity at instrumentation time [33]. As an example, instrumenting Ibex [43], a small RISC-V design, requires 72× more elements when instrumenting the design with gates, in comparison with the non-instrumented design. Perhaps more importantly, the gate-level approach also incurs high overhead at simulation time, as it forces simulators to simulate the design and the shadow logic gate by gate, while a significant speed-up would result from simulating higher-level constructs such as additions or comparisons.

Another issue with gate-level IFT logic is its precision. While precise IFT for a given design is an undecidable problem [16], doing so at the (low) level of gates exacerbates the problem. While there exists precise IFT logic for individual gates, the interaction of IFT logic from different gates leads to imprecision (i.e., overtainting).

To improve this situation, it is possible to generate the IFT logic at a higher level of abstraction. Previous work on

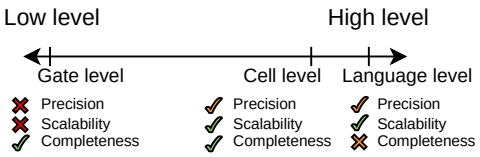

Figure 1: Different levels of instrumentation and their characteristics.

elevating the abstraction level uses HDL [2, 77]. They either introduce new language features that require porting by a human expert at significant time and backward-compatibility cost [77], or it is challenging to make them complete given the many possible language constructs to be supported [2].

IFT logic based on macrocells. We argue that to gain the benefits of both strategies, we need to operate at a different level of abstraction than gates or HDL. This level should be high enough to achieve high performance and precision, while generic enough to achieve completeness and backward-compatibility. Macrocells, to which we refer as *cells*, denote higher-level intermediate representations of synthesizable hardware primitives (such as an adder, comparator, etc.). Cells are limited in types but are parametrizable in widths and in other important properties such as signedness. Whereas working at gate-level requires breaking these primitives, cells are close to HDL and often map directly to HDL constructs. Therefore, cells form typical intermediate representations in hardware tools, explicitly in LLHD [58] and Yosys [74], and in Verilator which maps similar constructs to the simulating machine's ISA [73]. This makes cells an ideal candidate for generating the IFT logic as shown in Figure 1. Because all cells correspond to HDL constructs, CellIFT's shadow logic design is also a *necessary* basis for any HDL instrumentation that would strive to achieve completeness in the future.

Designing a scalable and precise IFT logic for general-purpose digital designs, however, poses certain challenges. First, it is unclear whether we can follow a generic approach for designing a precise IFT logic for any given cell, leading us to our first research question:

> **RQ1.** Is there a generic IFT logic pattern that could precisely instrument any combinational cell?

Second, while a generic approach will enable us to achieve completeness as we will soon discuss, it will come at a high cost. To reduce this cost, we can perhaps make use of the structure of the cells themselves, leading to our second research question:

> **RQ2.** Do cells have certain logical properties that we can exploit to scale the resulting IFT logic?

Section 4 answers the first question by introducing a novel m-replica architecture which can be adapted to implement the IFT logic with perfect precision for any given combinational cell. This architecture, however, scales exponentially with the cell's width. Section 5 answers the second question by introducing three fundamental logical properties of different cells, namely *monotonicity*, *transportability*, and *translatability* that can be leveraged to adapt the m-replica architecture for creating efficient per-cell IFT logic.

4 The Canonical M-replica Architecture

We introduce a replication-based architecture that can be used to generate IFT logic for any given cell with perfect precision.

4.1 Precise information propagation

Let C be a combinational cell, and C^t its IFT logic. C has some input bits $(I_j)_{0\le j<N_i}$ and output bits $C(I) := (Y_j)_{0\le j<N_o}$[2]. We define $(I^t_j)_{0\le j<N_i}$ and $C^t(I,I^t) := (Y^t_j)_{0\le j<N_o}$ to be the taint signals corresponding to C's input and outputs, respectively. The terms N_i and N_o represent the number of input bits and the number of output bits of the cell. This means that $I^t_j = 1$ if the input signal I_j is tainted, and similarly $Y^t_j = 1$ if the output signal Y_j is tainted.

Because some cells, such as adders, have two inputs A and B of identical width, we define $A_j := I_j$ and $B_j := I_{j+\frac{N_i}{2}}$. We refer to the index j in A_j or B_j as *operand position*, as opposed to the position in the aggregated input I.

Precise information propagation rule. Information propagates from the set of tainted input signals $\{I_j \mid I^t_j = 1\}$ to some output signal Y_j if there exists another input vector $\tilde{I}$ for C, which differs from I only on tainted input bits, and such that the two input vectors I and $\tilde{I}$ cause distinct values for Y_j. Equation 1 formalizes this rule. $\oplus$ denotes exclusive or.

$$C^t(I,I^t)_j = 1 \iff \exists \tilde{I} \mid (I \oplus \tilde{I}) \wedge \overline{I^t} = 0 \text{ and } C(\tilde{I})_j = \overline{C(I)_j} \tag{1}$$

Equivalently, for a given input I with taint vector I^t, and for a given output bit Y_j, Y_j is tainted if the value of Y_j can be changed by only changing the value of I at some tainted indices. We use this insight in the design of IFT logic using cell replication.

4.2 Replication-based design

Any digital circuit can be represented as an interconnection of state-holding cells (flip-flops and latches) and purely combinational (stateless) cells. We discuss how we can instrument these different cell types using replication.

State-holding cells. A simple replication suffices to instrument state-holding cells: when a signal is delayed by entering such a cell, the same delay affects the information carried by this signal, as shown in Figure 2a. This means that for every state-holding cell, we simply need an additional shadow cell that stores the taint information for that cell.

[2]We use the := notation when defining new terms in this paper.

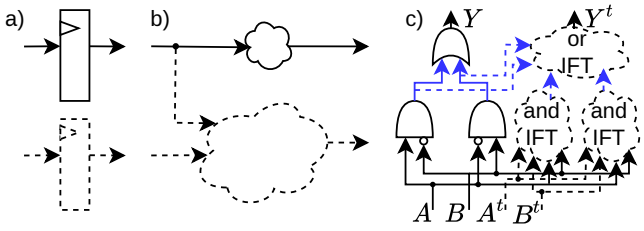

Figure 2: Instrumentation of a) a state-holding cell, b) a combinational block and c) an exclusive-or cell with single input bit at gate level. Signals generated to feed the gate-level IFT logic but absent at cell level are drawn in blue. The t exponent indicates taint signals.

Combinational cells. Combinational blocks should be instrumented with combinational logic as illustrated in Figure 2b. As shown in Figure 2c, dividing combinational blocks results in simulating more elements individually to generate intermediate signals and limits performance and precision.

Based on Equation 1, precise information flow tracking can be achieved by trying each possible input $\tilde{I}$, filtered by the input taints I^t. This can be achieved by replicating 2^{N_i} instances of C. This *canonical* design can create precise IFT logic for any cell, given that it uses copies of the cells.

Figure 3 shows the design of such an IFT logic for a combinational cell C with 2 inputs and 3 outputs. The four instances $C^{00}, C^{01}, C^{10}, C^{11}$ are identical instances of C supplied with distinct inputs depending on the input taint assignment. Equation 2 formalizes these instances as $C^v(I, I^t)$, with v in unsigned binary representation. In Equation 2, I^t has the role of selector in multiplexers between I and v.

$$C^v := C^v(I, I^t) := C((I \wedge \overline{I^t}) \vee (v \wedge I^t)) \tag{2}$$

We call such IFT logic architectures m-replica, where m is the number of instances of the original cell present in the IFT logic. The canonical m-replica architecture requires an exponential number of copies of C in the input size N_i. In the following sections, we specialize this generic structure using cells' mathematical properties to improve its scalability.

5 Exploiting the Logical Properties of Cells

We exploit monotonicity (Section 5.1), transportability (Section 5.2) and translatability (Section 5.3) properties to enhance the performance of replication-based cell IFT logic by reducing its size without loss of precision.

5.1 Monotonic cells

Prevalent cells such as comparators and some logical reductions (e.g., multi-bit OR cells) feature a property which we call monotonicity. Monotonicity allows pruning of the canonical replica-based IFT logic to obtain a constant-complexity 2-replica IFT logic. We define three monotonicity properties:

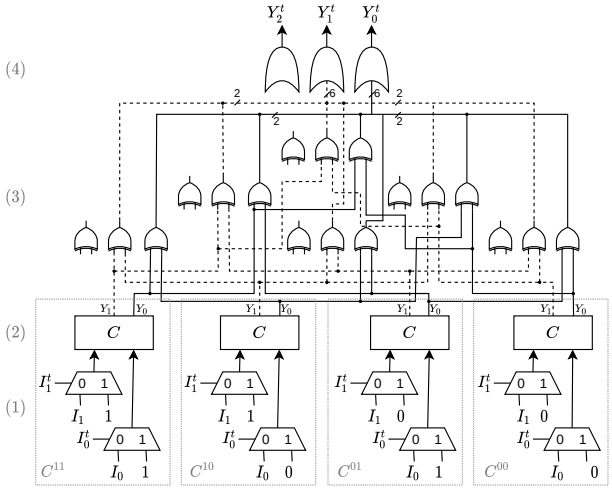

Figure 3: IFT logic for a cell C with 2 input bits and 3 output bits. The wires corresponding to the highest order output bit are omitted, and those of the second output bit are dashed. Stage (1) replaces the tainted inputs with all possible value combinations. In stage (2), C is replicated 2^{N_i} times to take all input combinations. Stage (3) compares the outputs of all replicas. Finally, for each output bit index, a taint is set if two replicas have different output bits at the corresponding index.

1. *bitwise non-decreasing.* A cell is bitwise non-decreasing in input offset j if no output bit of the cell can fall from 1 to 0 when I_j is raised from 0 to 1. For example, an OR cell is bitwise non-decreasing in all its input bits.

2. *bitwise non-increasing.* A cell is bitwise non-increasing in input offset j if no output bit of the cell can raise from 0 to 1 when I_j is raised from 0 to 1. For example, an inverter cell is bitwise non-increasing in all its input bits.

3. *bitwise monotonic.* A cell is monotonic if with respect to each of its input bits, the cell is non-increasing or non-decreasing.

IFT logic for bitwise non-decreasing cells. Let C be bitwise non-decreasing in all its input bits. We build a 2-replica IFT logic using *polarization* as described in Equation 3.

$$Y^t = C^{0\ldots0} \oplus C^{1\ldots1} \tag{3}$$

Proof. Let C be a bitwise non-decreasing cell in all its input bits. Let $0 \leq j < N_o$ (N_o is the output width of the cell), and consider a fixed input taint vector I^t. If the output bit Y_j is zero for some input I, then Y_j is also zero for the input $\tilde{I}^0 := I \wedge \overline{I^t}$ given the non-decreasing property. Conversely, if Y_j is one for some input value $\tilde{I}$ such that $I \wedge \overline{I^t} = \tilde{I} \wedge \overline{I^t}$ (i.e., I and $\tilde{I}$ differ only on tainted bits), then Y_j is also one for the input $\tilde{I}^1 := I \vee I^t$, again given the non-decreasing property. It follows that all output bits that can be toggled by applying two inputs I and $\tilde{I}$ of tainted bits are also toggled between applying $\tilde{I}^0$ and $\tilde{I}^1$. This allows us to reduce m-replica to 2-replica using polarization: $Y^t = C(\tilde{I}^0) \oplus C(\tilde{I}^1) = C^{0\ldots0} \oplus C^{1\ldots1}$. □

Table 2: 2-replica-based instrumentation of monotonic cells. Logical reductions represent multi-input OR or AND cells with single output. Comparisons represent $<$, $\leq$, $\geq$ and $>$.

Cell	Replica 1		Replica 2
Reductions	$C^{0\ldots0}$	$\oplus$	$C^{1\ldots1}$
Unsigned comparisons	$C^{00.0;11.1}$	$\oplus$	$C^{11.1;00.0}$
Signed comparisons	$C^{10.0;01.1}$	$\oplus$	$C^{01.1;10.0}$

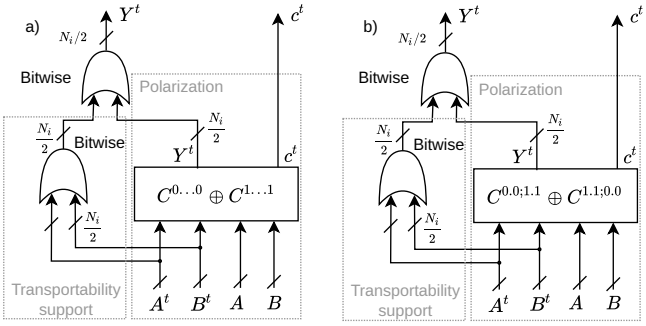

Figure 4: 2-replica-based IFTL for a) an adder cell and b) a subtractor cell.

IFT logic for bitwise monotonic cells. We now extend polarization to all bitwise monotonic cells. If C is non-increasing with respect to an input bit at index i, then it is non-decreasing in the negation of this input bit. Because bit inversion does not affect taint propagation, any monotonic cell can similarly be instrumented according to Equation 4, where $d_i := 1$ if C is non-increasing in input bit i, and $d_i := 0$ if C is non-increasing in input bit i.

$$Y^t = C^{d_{N_i-1}\ldots d_0} \oplus C^{\overline{d_{N_i-1}\ldots d_0}} \tag{4}$$

Summary. Table 2 shows the 2-replicas used to instrument logic reductions and comparisons. Unsigned comparisons, and or-/and-reductions are bitwise non-decreasing cells in all input bits. Signed comparison cells are bitwise non-decreasing on all bits except the most significant bit of the operands, which is non-increasing.

5.2 Transportable cells

Abundant arithmetic cells such as addition and subtraction cells can be instrumented in constant complexity thanks to their *transportability* property. Transportable cells have the same width for each operand, and information from an operand bit always flows to the corresponding output bit. We can instrument these cells using polarization (similar to monotonic cells) complemented with *transportability*-supporting IFT logic that implements a conjunction of the input bits.

5.2.1 Adder cells

We design the IFT logic of the adder cell as described by Equation 5 and illustrated in Figure 4-a using polarization conjuncted with the transportability term $A^t \vee B^t$. Appendix B provides a proof of correctness and precision of this architecture. Our proof considers a ripple carry adder which exposes

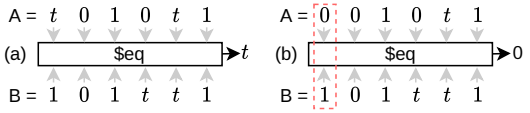

Figure 5: Examples of an equality (a) with tainted output, and (b) without a tainted output. The equality cells compare the top input bits with the corresponding bottom input bits.

an induction property in the taint propagation from the least significant bits to the most significant bits of the operands A and B. This proof generalizes to any adder implementation, since different architectures provide the same mathematical function for addition.

$$Y^t = \left[C^{0\ldots0} \oplus C^{1\ldots1}\right] \vee A^t \vee B^t \tag{5}$$

5.2.2 Subtractor cells

Similarly to the adder cell, we base the IFT logic of the subtractor cell on a 2-replica polarized architecture supplemented with transportability logic to form the IFT logic given in Figure 4-b. The first polarization term takes operand A with tainted bits set to zero and operand B with tainted bits set to one, and conversely for the second polarization term. A proof by induction in all aspects is similar to the proof given in Appendix B since subtraction can be formulated for this architecture based on the identity $A - B = A + \overline{B} + 1$.

5.2.3 Negation cells

We define I as the data concatenation of the inputs (i.e., $\{A,B\}$), and I^t as the taint input concatenation (i.e., $\{A^t,B^t\}$) The negation cell defined by $Y = \overline{I} + 1$ exposes the same property as an adder, except that there is a single operand and no carry bit in the negation cell. Therefore, we instrument it in constant complexity using the same polarization replicas, conjuncted with the transparency term I^t that corresponds to a single operand.

5.2.4 (In)equality cells

(In)equality cells are transportable under the condition that all non-tainted bit pairs are equal. More precisely, the output bit of an (in)equality cell is tainted if and only if the two following conditions are fulfilled: 1. At least one input bit is tainted. 2. For each operand position where no input bit is tainted, the two input bits are equal.

Figure 5 shows two equality cells. The equality cell (a) has its output tainted because it fulfills the two conditions. However, cell (b) has its output non-tainted, because the leftmost bits do not match and are not tainted. Note that the value of tainted input bits never matters. We design the IFT logic of the equality cell as in Equation 6, where $T_{AB} := \overline{A^t \vee B^t}$ represents which bits are neither tainted in A nor in B. The term $\bigvee I^t$, where $I^t := \{A^t,B^t\}$, is one if and only if any of the input bits is tainted.

$$Y^t = C(A \wedge T_{AB}; B \wedge T_{AB}) \wedge \bigvee I^t \tag{6}$$

Table 3: 2-replica-based instrumentation of transportable (addition, subtraction, negation) or conditionally transportable (equality, inequality) cells.

Cell	Polarization	Junction	Transportability
Add	$C^{0...0} \oplus C^{1...1}$	$\vee$	$A^t \vee B^t$
Sub	$C^{0.0;1.1} \oplus C^{1.1;0.0}$	$\vee$	$A^t \vee B^t$
Neg	$C^{0...0} \oplus C^{1...1}$	$\vee$	I^t
Eq/Neq	$C(A \wedge T_{AB}; B \wedge T_{AB})$	$\wedge$	$\bigvee I^t$

5.2.5 Summary

We summarize the IFT logic of transportable and conditionally transportable cells in Table 3.

5.3 Translatable cells

Some cells do not present powerful properties such as monotonicity or transportability for generating efficient and precise IFT logic. In this section, we take a different approach to tackle the problem: instead of relying on the replication mechanism immediately, we consider each output bit, and examine which input condition results in tainting output bits.

5.3.1 Input decomposition

Examining each output bit of a cell and its relationship with input taints and values results in complex formulas that are not efficient to implement for wide-input cells that can be present in real digital designs. It is often convenient to make simplifying assumptions such as some operand being non-tainted. For instance, for the left shift operator $A \ll B$, if we assume that $B^t = 0$, then $Y^t = A^t \ll B$. We show that the combination of two properties, namely *translatability* and *taint combination surjectivity*, allows us to construct new architectures that support such simplifying assumptions.

Translatability. A two-input cell C is said to be *right side translatable* (over addition) if it satisfies Equation 7 for all inputs A and B.

$$C(A, B' + B'') = C(C(A, B'), B'') \tag{7}$$

As an example, a left shift by an unsigned offset provides this property: $A \ll (B' + B'') = (A \ll B') \ll B''$.

Additionally, the following decomposition holds: $B = B^0 + B \wedge B^t$, where $B^0 := B \wedge \overline{B^t}$ (i.e., B where all the tainted input bits are zeros). This leads to Equation 8, which has two instances of C on the right-hand side: one instance with a non-tainted second operand B^0, and one instance where all non-tainted bits of the second operand are zero.

$$C(A, B^0 + B \wedge B^t) = C(C(A, B^0), B \wedge B^t) \tag{8}$$

Taint decomposition surjectivity. Equation 7 relies on cell composition. However, it is known that in the general case, cell composition does not preserve precision of the IFT logic [2,67]. We introduce the *taint decomposition surjectivity* property: a cell's IFT logic L is taint decomposition surjective if and only if precision is unaffected when performing IFT through this cell. Assuming k output bits are tainted, if L can generate the 2^k outputs by changing the data at tainted input bits, then L is taint decomposition surjective. In Appendix D, we formalize this property, and use it to prove the precision of the IFT logic for unsigned-offset logical shift cells.

Table 4: 1-replica-based instrumentation of translatable cells. The IFT logic is the sequence of two components. Shift cells in this table have an unsigned interpretation of B.

Cell	Comp. 1	Comp. 2 (bitwise)
$\ll$	$C(A, B^0)$	$\bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \stackrel{t}{=} k \right] \wedge \left[A_j \stackrel{t}{\neq} A_{j-k} \right] \right)$
$\gg$ (logic)	$C(A, B^0)$	$\bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \stackrel{t}{=} k \right] \wedge \left[A_j \stackrel{t}{\neq} A_{j+k} \right] \right)$
$\gg$ (arith)	$C(A, B^0)$	$\bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \stackrel{t}{=} k \right] \wedge \left[A_j \stackrel{t}{\neq} A_{min(j+k, N_A-1)} \right] \right)$

5.3.2 Logical unsigned-offset shift cells

We show that logical shifts by an unsigned offset $C(A, B) := A \ll B$, or $C(A, B) := A \gg B$ can be decomposed in two successive shifts, each instrumented separately, without losing precision. We proceed according to Equation 8, which implies decomposing C into two cells copies, instrumented separately: the first cell in the decomposition is $C(A, B^0)$, where the offset B^0 is independent of the tainted value assignments. The IFT logic for this replica is given by $C(A^t, B^0)$, i.e. $A^t \ll B^0$ and $A^t \gg B^0$ for logical left and right shifts respectively.

The second cell in the decomposition is $C(A', B \wedge B^t)$, where A' is the output of the first cell. In Appendix C, we provide a precise bitwise IFT logic for this second cell. Because logic unsigned-offset shift cells are taint combination surjective as shown in Appendix D, the sequence of the two IFT logics is precise.

5.3.3 Arithmetical unsigned-offset shift cells

The arithmetical unsigned-offset shift cell benefits from the same right-side translatability as the logical shifts. We decompose them in two cells before instrumentation, similarly to their logical shift counterparts. The only difference is the propagation of the most significant bit. The IFT logic of the first cell is identical to the logical counterpart. Appendix C provides a bitwise IFT logic of the second cell.

5.3.4 Summary

In this section, we used the translatability property to decompose some cells into two identical cells with simplifying assumptions on operands as summarized in Table 4, while preserving perfect precision.

6 Implementation

In this section, we provide additional details and describe the implementation of CellIFT.

6.1 CellIFT flow implementation

We integrated CellIFT into the Yosys synthesizer as a synthesis pass over the design's internal representation. Since

Yosys does not have a complete understanding of all the SystemVerilog constructs, we first pre-process the designs using the open-source sv2v tool that converts SystemVerilog (IEEE 1800-2017) to Verilog (IEEE 1364-2005) [61]. We refer to the processing by Yosys as the *instrumentation* step.

For simulation, we use the open-source Verilator simulator [73]. For emulation, we use Xilinx Vivado. We refer to the processing by Verilator to produce a simulation binary, or the processing by Vivado until the end of the FPGA implementation, as the *synthesis* step.

The CellIFT Yosys pass is made of 4575 lines of C++ code. CellIFT supports a total of 181 Yosys cell types, of which many are variations of state-holding elements with reset, enable and clear signals. 8 unsupported cell types are memories and their ports, because they are substituted in a previous stage and do not reach the CellIFT pass. The 33 remaining cell types are not supported because they have never been encountered when experimenting on various heterogeneous designs. Next, we provide more information on how we support memories in CellIFT and describe simple techniques for instrumenting the remaining cells other than the ones described in Section 5.

6.2 Instrumenting other cells

Memory models. In digital designs, memories are typically substituted with specific components depending on the design's target: simulation model (simulation), block RAM (FPGA), or SRAM (ASIC). CellIFT implements memory models with *conservative* IFT rules as follows: (a) Taints are written and read along with the corresponding data. (b) Memory read from a tainted address also returns a tainted value. (c) Memories are marked fully tainted as soon as the inputs and their taints authorize a write operation to a tainted address. We also implemented memory models that only perform *explicit* tainting, i.e., when only rule (a) applies. This allows us to learn which microarchitectural components are dependent on the tainted address of a load instruction.

Lower-level cells. Exclusive ORs propagate taint as soon as at least one input bit is tainted. Logic gates ((N)ANDs, (N)ORs and inverters) are parametrizable in width to profit from wider instructions on the simulating machine. Multiplexers with multiple selector bits are decomposed into a tree of single-bit selector multiplexers without losing precision. Then, for a multiplexer of formula $Y = S?B : A$, we design the IFT logic expressed in Equation 9, where S, $\overline{S}$ and S^t are replicated to have the same width as A and B.

$$Y^t = (A^t \wedge \overline{S}) \vee (B^t \wedge S) \vee ([A \oplus B] \wedge S^t) \quad (9)$$

6.3 Imprecise cell instrumentations

Imprecise shift cells. As opposed to their unsigned counterpart, signed-offset shift cells are not right-side translatable. Because a precise implementation of these shift cells is expensive, CellIFT implements a replication-based approximate propagation policy: taint Y completely whenever B is tainted, else shift the taints of A by B.

Additionally, on all the designs that we considered, we noted that right shifts are never larger than 10 bits, whereas left shifts are often larger in five reference open-source designs. Therefore, for unsigned shift offsets, in CellIFT we instrument right shifts precisely and left shifts imprecisely.

Imprecise multiplier cell. As proposed in [2], we decompose the multiplier into a sequence of adders before instrumenting it. This results is an imprecise IFT logic, but this does not affect overall precision much since multipliers are mostly present only on data paths. For emulation targets, to decrease the critical paths we used a shadow logic made of a single OR reduction, without practical decrease in precision.

6.4 Summary

CellIFT instruments all the 22 combinational cell types that we encountered in our diverse experiments. From these 22 cell types, 3 are similar to GLIFT when they have a single bit per input. Appendix A summarizes the cell composition of non-instrumented, and instrumented designs[3].

7 Evaluation

In this section, we evaluate CellIFT in terms of performance, precision and completeness by instrumenting heterogeneous and complex designs. We use microbenchmarks to show how CellIFT compares in terms of precision and scalability with previous work [2, 67] (Section 7.1). To show the scalability of the instrumentation phase, we instrument five heterogeneous designs of various complexities (Section 7.2). To show the performance and precision of CellIFT-instrumented designs, we simulate standard RISC-V benchmarks (Section 7.3). Finally, we show FPGA portability of the CellIFT-instrumented designs (Section 7.4).

Evaluation setting. The performance results were obtained on a machine equipped with an AMD EPYC 7H12 processor at 2.6 GHz equipped with 256 logical cores and 1 TB of DRAM. We used Verilator 4.212 and g++11.2 with the -O0 compiler flag similar to what is used for certain OpenTitan designs [44, 45]. Using compiler optimizations causes spurious segmentation faults in the Verilator simulation binaries.

Baselines. To the best of our knowledge, no mature open-source or transparent-enough commercial implementations of gate-level or language-level hardware IFT is available.

[3] One year after publication of this article, RTLIFT authors finally shared their code with us. For comparisons and equality testing, RTLIFT taints whenever one input bit is tainted. It treats logic operators AND, OR and NOT operators similarly to CellIFT. It instruments AND and OR reductions using basic gates. It instruments the shift in the same way as CellIFT's conservative version. RTLIFT is not able to instrument any of the designs in this study because of the small number of supported Verilog contructs.

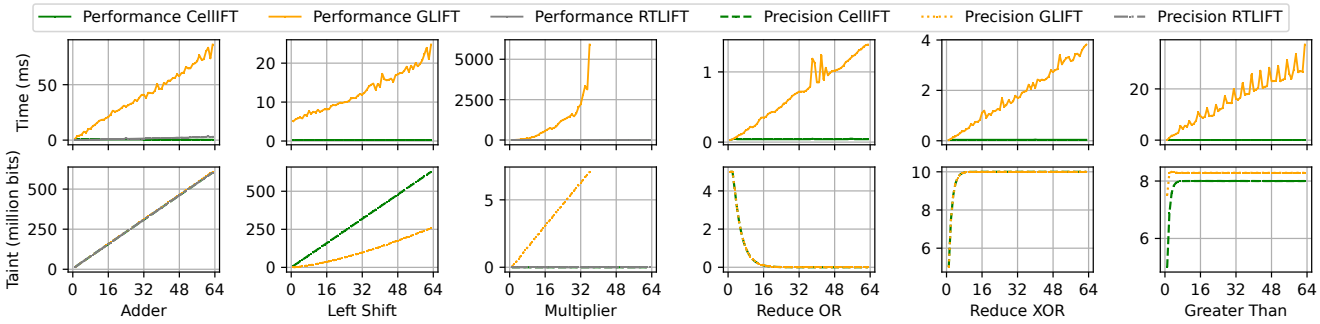

Figure 6: Cell microbenchmarks: runtime performance (top) and precision (bottom) depending on the cell input width, measured from 10 parallel identical cells to reduce time measurement noise. *Reduce* cells are multi-bit input, single-bit output logic cells (e.g., AND). Precision numbers represent the cumulative number of tainted output bits. The left shift cell has an 8 bits wide offset operand, which is common in the designs that we examined. RTLIFT only instruments *add* and *mul*.

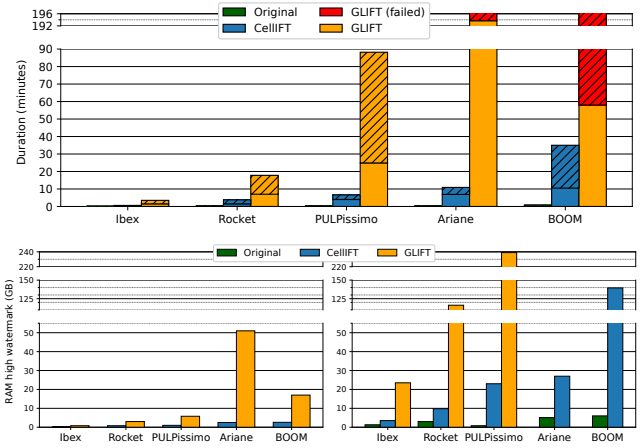

Figure 7: Top: duration of Yosys instrumentation and Verilator synthesis. Hashed rectangles represent synthesis. Bottom: RAM usage of each mechanism on each design. Left: instrumentation. Right: synthesis. Note: vertical axes are broken.

Therefore, we implemented GLIFT as described in the original paper [67] for comparison purposes. As the authors of RTLIFT [2] were reluctant to share their implementation or provide additional details, we re-implemented the few operators described in [2]. Our implementation reproduces their operator-level performance and precision results.

7.1 Microbenchmarks

To evaluate the performance and precision of CellIFT, we instrument and simulate individual combinational cells with various widths to evaluate scalability. We apply one million random inputs to each cell and randomize the taint bits. We measure precision by counting the tainted output bits.

Figure 6 shows the performance and precision results of cell microbenchmarks: CellIFT provides a massive speedup in simulation over existing mechanisms. RTLIFT and CellIFT-instrumented adders and multipliers have the same precision, whereas CellIFT-instrumented cells are faster. By design, left shifts suffer from poor precision in this benchmark, which is based on randomly tainting all input bits: we traded off some empirically superfluous precision in this cell for speed. This design point is not fundamental as we showed a perfectly precise shift implementation in Section 5. All the other cells are instrumented at least as precisely as GLIFT and RTLIFT. Comparison cells are more precise with CellIFT compared to GLIFT. Since these cells can have large taint fanouts, this precision improves overall taint results significantly, which is crucial in practical scenarios as we will show in Section 8. Superior scalability of CellIFT is made evident by its steadily low runtime for any cell width.

7.2 Instrumentation

Configurations. We built simple SoCs for Ibex [43] (in its default *Small* configuration) and Ariane [76] (with 4-way associative 8 kB instruction and data L1 caches) by adding memory models and protocol adapters at the design top levels (caches remain untouched), and inserted memory models in PULPissimo (Hack@DAC'18 version) in place of the L2 SRAM. We used the Rocket chip generator to create a SoC to interface with the Rocket core [3] and the BOOM core [12] with reduced cache sizes. These configurations allow us to run standard software on these designs. We additionally replaced PULPissimo's technology-dependent oscillator with a model. Code and data are preloaded into the memory models prior to any measurement.

Performance. We attempt to instrument and synthesize each design with both CellIFT and GLIFT, and report the wall clock durations and the resident memory consumption in Figure 7. Verilator failed to synthesize Ariane (out of memory) and BOOM (timeout after 96 hours) instrumented with GLIFT due to the excessive complexity of the GLIFT instrumentation. While piecewise instrumentation of Ariane by GLIFT at a greater engineering cost could lead to an eventually successful gate-level instrumentation, these results make the limitations of gate-level instrumentation apparent.

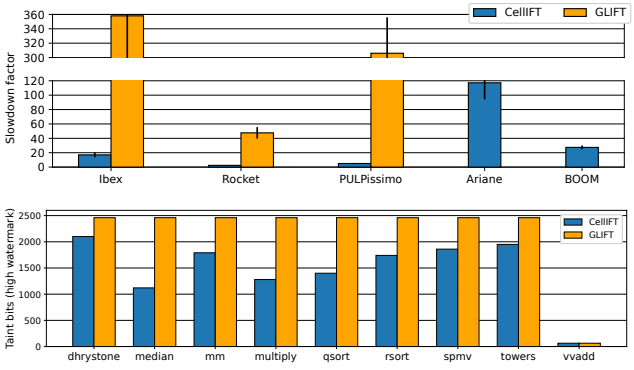

Figure 8: Top: average wall clock slowdown of the RISC-V benchmarks of each design compared with the original designs. Bottom: number of tainted stateful elements on Ibex.

7.3 Benchmarks

We run a standard series of RISC-V benchmarks [57] on each design to assess the performance and precision of CellIFT. We run the single-threaded benchmarks, where we skip the *pmp* benchmark, which is not supported by Ibex in its default configuration. We taint some relevant part of each benchmark: the first data element in *median*, *mm*, *multiply*, *qsort*, *rsort* and *spmv*, the first instruction in *dhrystone*, and the number of discs in *towers*. Because running benchmarks on a simulated RTL takes very long, we resized the benchmarks' inputs so that each experiment point finishes under 30 minutes. This translated to simulating 4 M cycles for Ibex, 200 k cycles for PULPissimo, and 40 k cycles for Ariane, Rocket and BOOM. As we will show in Section 8 simulating this number of cycles is enough to enable many interesting scenarios.

Figure 8 shows the performance results on all designs and the number of tainted stateful elements (i.e., precision) on Ibex. Since the performance variation between the benchmarks is small, we only indicated the average and standard deviation between the benchmarks. CellIFT is significantly faster than GLIFT in simulation. Regarding precision, some benchmarks such as *multiply* taint the control flow in Ibex, resulting in abundant tainting [67]. Manual investigation shows that some ALU operations influence subsequent branch predictions, which CellIFT legitimately revealed. CellIFT's precision allows exploration of different scenarios in Section 8 without observing any false positive.

7.4 FPGA emulation

While we optimized CellIFT's shadow logic for simulation, we show-case its flexibility by porting the five instrumented designs to an FPGA. For the FPGA flow, we use Vivado-2019-03 on a machine equipped with a Intel Xeon Gold 6146 CPU with 48 logical cores at 3.20 GHz with 196 GB of DRAM, for licensing reasons. We target a xcvu440-flga2892-3-e FPGA at 100MHz. We set a time limit of 48 hours for the synthesis.

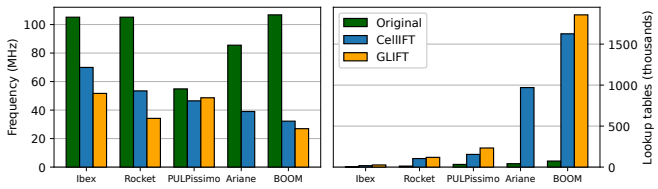

Figure 9: FPGA results. Left: Achieved FPGA frequency. Right: 6-input lookup table usage.

Results. We successfully port the CellIFT-instrumented designs to the target FPGA. The synthesis of GLIFT-instrumented BOOM requires 27.57 hours and Ariane times out still at an early stage. With CellIFT, it lasts respectively 5.93 and 4.57 hours to synthesize BOOM and Ariane. Figure 9 summarizes the achieved frequencies and resource requirements for each design. CellIFT usually shortens the critical path compared to GLIFT, providing a frequency increase of up to 39% over the state of the art. Only PULPissimo shows a slight frequency decrease of less than 5%. We obtain an FPGA acceleration over simulation of 256.2 k× and 94.1 k× for Ariane and BOOM respectively. Since CellIFT is the only dynamic IFT mechanism to instrument Ariane and to port it to an FPGA, and improves frequency and utilization compared to the state of the art, these results advocate for a cell-level (or higher) instrumentation for FPGA emulation.

7.5 Summary

We showed that CellIFT is at least as precise and significantly faster than the state of the art. It can instrument designs that were so far inaccessible to the existing solutions, without aggressive and imprecise approximations [30]. In the next section, we show some of the new opportunities offered by CellIFT thanks to its completeness, correctness, performance, and precision.

8 Scenarios

We now show how CellIFT could be used to enable new applications. While there are many applications possible with hardware dynamic IFT, here we focus on detecting different classes of hardware vulnerabilities with CellIFT.

8.1 Discovering microarchitectural leakage

Changes made to the microarchitectural state from secret-dependent data or control paths can be exploited to leak secret data [40, 52, 75]. Cache partitioning schemes attempt to mitigate such attacks [26, 39, 78], but there are other components that can leak information, such as Translation Lookaside Buffers (TLBs) [25] or branch prediction schemes [18] to name a few. CellIFT can be used to detect microarchitectural components that can leak information. This information is valuable for both attackers looking to discover a new source of leakage and for the defenders that want to protect the components that might leak sensitive information. We execute a

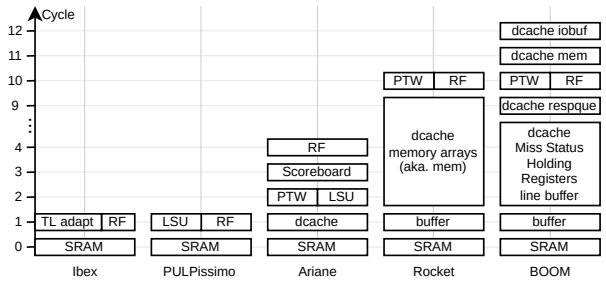

Figure 10: Affected microcomponents for a load with tainted address or tainted data. RF: Register File, TL: TileLink, LSU: Load-Store Unit, PTW: Page Table Walker.

memory load of some tainted data in memory. The taint propagation in the design gives us a cycle-accurate knowledge of when, and which bits of the design are tainted.

Figure 10 shows a chronological list of the components that become tainted in designs that we have instrumented with CellIFT. We make several observations. First, taints make it very easy to see when any buffer in the design contains sensitive information. Second, taints provide a convenient way of measuring latencies in a system. We detected that PULPissimo and Ibex load data in a single cycle, while the other designs require respectively 4, 10 and 10 cycles. Third, privilege level and page misses do not affect which components are tainted in any of the considered designs. Finally, CellIFT is precise in this scenario since the control flow was legitimately never tainted.

8.2 Detecting Meltdown-type vulnerabilities

Meltdown-type vulnerabilities have been haunting CPUs since their introduction in 2018 [1, 8, 11, 38, 55, 56, 59, 69, 71, 72] At the core, this class of vulnerabilities allows an invalid load to transiently access data from a different (higher) privilege level. CellIFT can detect such loads by checking that that they do not access data from a higher privilege.

Ariane. Ariane features an MMU and may suffer from Meltdown, Foreshadow or MDS [8, 11, 38, 59, 71]. We craft the following test cases to trigger these three potential issues: 1. An unprivileged load of a supervisor page (triggering Meltdown [38]). 2. An unprivileged load of a supervisor page with the present bit unset (triggering Foreshadow [8]). 3. A load from an invalid address without the address bits in the page table entry (triggering MDS [11, 59, 71]).

To see which of these cases trigger the relevant problem, we taint a target (supervisor) memory page and configure the page table entry according to one of the above scenarios. Prior to measuring the malicious load, we access the tainted page to ensure it is in the cache, and possibly in the microarchitectural buffers that also have been used for it. The results from these different cases show that in none of the cases signals get tainted as a result of the malicious access, showing that Ariane

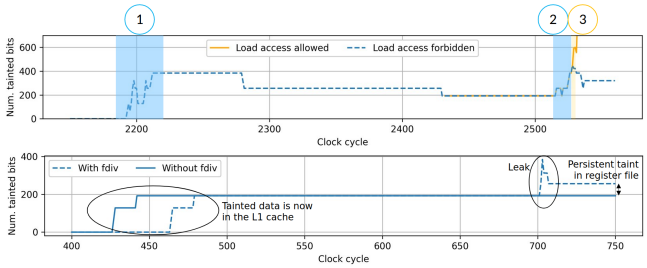

Figure 11: Tainted bits in a BOOM SoC. Top: Meltdown-type leak: (1) Tainted privileged data is brought into the L1 cache. (2) The user loads the privileged data into a physical register. (3) The user immediately attempts to load using the tainted data as an address. This load is either allowed (orange, realigned) or forbidden (blue). Bottom: Spectre-type leak (at cycle 700) occurs if the speculation window is large enough.

does not suffer from Meltdown-type vulnerabilities in any of these specific cases. We contacted the authors who confirmed our observations that Ariane indeed does not suffer from this class of vulnerabilities.

BOOM. BOOM v2.2.3 was reported to be susceptible to Meltdown-type vulnerabilities [24]. We instrument the exact same version, and use CellIFT for detecting Meltdown-type leakages. Contrary to previous work [24], we do not use any knowledge of the design's microarchitecture. We run a simple Meltdown experiment by transiently loading privileged data in unprivileged mode and using it as an address for a subsequent load. We ensure that the privileged page table entry is present in the TLB, and we taint the privileged word. Given that the load address will be tainted, a load will spread taint to all sets of the L1 cache among other elements. Conversely, if we see no taint in the cache, we know that the load fails, and any leakage from this load cannot be observed using a cache covert channel in a later stage.

In our experiments, we observe that when the user tries to load privileged data, (a) If the data is not in the L1 data cache, then the data is fetched into the cache's load buffer regardless of any privilege mismatch, and (b) If the data resides in the L1 data cache already, then the data is loaded into the physical register file. According to [24], these constitute a Meltdown-type leaks of L and R classes, respectively.

To assess exploitability, we first establish a benign baseline. We first do a legal load of a tainted address (orange curve in Figure 11, top). As predicted, this taints a large number of elements. Next, we repeat this experiment with a secret-dependent load, using the privileged (tainted) data as an address. Figure 11 summarizes the taint propagation during the attack, aligned on the load event. We observe that the taint propagation is blocked when the page table entry shows a privileged page (blue curve), and the level of tainting is immediately restored as it was before the malicious load. While exploitability remained an open question in [24], this result shows that in this specific case (R1 [24]), the leakage is not ex-

ploitable because it does not leave any observable change. We repeated the exact same experiments on the newest version of BOOM (v3.0), and draw the exact same conclusions.

8.3 Detecting Spectre-type vulnerabilities

To show how CellIFT can detect Spectre in complex designs, we consider a Spectre-BCB exploit on BOOM where we taint the secret data. Our exploit consists of two steps, visible in Figure 11 (bottom). First, the secret is brought to the L1 data cache. Second, the data is speculatively loaded into the physical register file. We run two experiments. In the first case, the mispredicted branch relies on a simple condition (solid line in figure). In the second case, this branch condition is made more complex by adding four dependent floating-point divisions, as in a reference exploit [7], which enlarges the period of speculative execution (dashed line). Intuitively, the first experiment may not reveal Spectre-type leakage because the speculation window may be too short. We observe that the Spectre-type leakage only happens in the second experiment, consistent with this intuition. As opposed to classical techniques such as [7], CellIFT does not require a cache attack to assess whether data leaks to microarchitectural elements.

8.4 Detecting architectural vulnerabilities

We show how simple policies built on top of CellIFT can detect a large number of bugs in the PULPissimo-based faulty design used in the Hack@DAC'18 contest, some of which are not detectable by common verification flows. Detailed bug descriptions are provided in the corresponding paper [17].

Address space violations. We build two policies that check for correct behavior in the interfaces of the memory-mapped components: (a) All components in the address map must comply to the specified boundaries. (b) No aliasing must occur; may it be in the specification or because of an implementation bug. To force these policy checks in the CellIFT-instrumented design, we perform store operations of tainted data from the CPU to the addresses before and after each address space boundary. These policies reveal bugs 1, 2, 6, 8 and 22 which could not be expressed by either SPV or FPV [17].

Reachability violations. Our reachability policy checks that certain instructions do not affect certain components by executing a tainted instruction. Then, after N cycles, all the components that *can* be affected by the CPU are tainted. This allows us to check the integrity and reachability of components against a specification, revealing bugs 4 and 27. We did not find bug 24 with this method, although we had expected it. Manual code inspection and simulation showed that the bug was not present in the open-source version of the faulty design [27]. Similarly, we discovered that bug 7 was inserted in a module which is never instantiated in the design.

Reset violations. The reset policy checks that the reset signal clears the state in the design. We check for the violations of this policy by tainting state holding elements in the design, and then applying a reset signal. This reveals which registers and buffers are not cleared during reset, and which ones are accidentally cleared. This reveals bug 5, but not bug 12 (the corresponding register has been optimized out because it was not used and not connected to a clock or reset signal) and bug 16, which lacks a clear specification.

Privilege violations. Our privilege policy checks that instructions execute at the right privilege level according to the specification. We check for violations by executing a tainted instruction. Because an unprivileged user requires to trap to supervisor mode to access privileged locations, these locations will be tainted some cycles after unprivileged locations. We did not see such an expected timing difference in tainting of the CSRs (Control and Status Registers), which hinted to the existence of bug 25.

Summary. We showed that the ability of CellIFT to instrument a complete design efficiently and precisely enables the implementation of novel techniques to detect bugs; some of these bugs are known to be difficult or impossible to express in SPV [9] and FPV [22, 23]. For completeness, we like to mention that hardware dynamic IFT is not suitable for detecting hardcoded design parameters (e.g., bugs 15 and 19) or checking functionality (e.g., bugs 9 and 15). Some bugs such as 11, 13, 14 and 20, 30 or 31 require more advanced policies. We leave the design of systematic methods to detect such bugs as future research.

9 Discussion

We discuss some observations we made while developing CellIFT and provide more information for its future users.

Maturity of the open source flow. During the course of this work, we provided feedback to the developers of Verilator [73] and sv2v [61]. None of these two tools was originally mature enough for the dynamic IFT flow to complete for all the designs under study. These tools have been improved according to our feedback and have reached a sufficient maturity to instrument the complex designs that we evaluated.

Applications. CellIFT aims to provide scalable hardware dynamic IFT. Alone, this mechanism does not aim at discovering new microachitectural vulnerabilities: an important additional element is the scenarios running on top of CellIFT. These scenarios go beyond the verification of handcrafted information flow policies. As an example, CellIFT can be leveraged to provide a new coverage metric, enabling the development of new hardware fuzzers, which are still in their infancy [68]. Another example is enabling system-wide confidentiality and integrity policies for generic processors. We leave the exploration of these directions to future work.

Instrumentation effort. CellIFT can instrument any digital design without modification, after parsing by the Yosys SystemVerilog parser [74] into intermediate cells. However, it is common not to synthesize the memories, and to instead replace them with a model. Typically, the option of ignoring memories during synthesis is available, as this is common practice when mapping a design to an FPGA or to an ASIC flow, where memories are mapped to specific components.

10 Related work

We briefly discuss work related to CellIFT in five areas.

Hardware dynamic IFT. GLIFT [67] is the first hardware-level dynamic IFT mechanism and operates at gate level. Previous work discusses the scalability problems of GLIFT [33] and our results indeed show that it does not scale to the state-of-the-art RISC-V processors. Various techniques try to address the scalability issues of GLIFT by using dedicated hardware [35], static analysis [5], policy-specific IFT logic [41,42], and trading precision with simpler IFT logic [30]. Orthogonally, CellIFT solves the scalability problems of GLIFT by generating the IFT logic at a higher level of abstraction. It would be interesting to see how previous techniques that scale GLIFT, can be used to scale CellIFT even further.

RTLIFT [2] aims to address scalability and precision problems of GLIFT by instrumenting the HDL code directly. Unfortunately, existing HDLs are complex, and it is challenging to achieve completeness by instrumenting in HDL directly. In comparison, CellIFT achieves completeness by choosing a slightly lower-level yet generic cell abstraction, and outperforms RTLIFT as shown in Section 7.1. Because all cells correspond to simple HDL constructs, CellIFT provides a strong basis for any future HDL-level instrumentation.

Non-synthesizable hardware fuzzing of simulation binaries was recently proposed [68]. However, because the simulation binary's control flow depends on values in the design, software dynamic IFT will lead to critical overtainting. Another major drawback of this approach is its restriction to Verilator.

Support for software dynamic IFT. Previous work proposes to provide hardware support for software dynamic IFT [53, 54]. These solutions are more lightweight but are only suited to find issues in software, not in hardware.

Static analysis of hardware. Static analysis in combination with model checking or verification of security properties is a common technique for improving hardware design security [9, 19, 22, 23, 64]. These techniques, however, have scalability issues due to the state explosion problem. To make static analysis tractable in certain cases, previous work introduces a new type system to an existing HDL [21] or a new HDL [77] for checking security properties. These techniques are not backward compatible to existing designs. In comparison, CellIFT provides a scalable alternative for checking security properties in unmodified RTL designs.

Model-based Meltdown detection. IntroSpectre [24] also detected Meltdown-type leakages on BOOM. It uses a Gadget Fuzzer to generate fuzzing rounds, uses known values instead of taints, and augments BOOM with a Leakage Analyzer, an ad-hoc IFT mechanism. Whereas this mechanism may have a faster simulation runtime, CellIFT proposes a trade-off with several advantages. First, CellIFT is not built into a simulator. It is therefore compatible with any tool flow. Second, CellIFT is design-agnostic: it does not require the engineering effort and precise knowledge about the design to be deployed; CellIFT even reveals the relevant microarchitectural elements traversed by tainted signals. Third, CellIFT supports processed secrets (for instance, the result of an addition with a secret) and taints in the control paths, whereas the Leakage Analyzer in IntroSpectre only monitors unchanged secret data in the data path. This last feature is essential in analyzing exploitability: because no taint reached back caches or control path, we concluded that R1 [24] is not exploitable, whereas it remained an open question with IntroSpectre.

Hardware testing. Test cases are an effective tool used by practitioners to combat hardware bugs. To increase the coverage of test cases, random testing or more guided methods can be used to increase the coverage [34, 49, 51, 68]. CellIFT can complement these approaches by providing a mechanism to detect when vulnerabilities are triggered.

SPECS [28] dynamically checks security-critical state using policies derived from the ISA to combat post-silicon vulnerabilities. These policies could be leveraged by CellIFT to check the entire state during pre-silicon testing. Post-silicon, CellIFT can be used as a low-overhead alternative to existing hardware dynamic IFT techniques for enforcing security properties in the entire system [63].

11 Conclusion

We presented CellIFT, the first hardware dynamic IFT mechanism that can scale to complex state-of-the-art open-source RISC-V processors. CellIFT instruments the RTL using the cell abstraction, which is high-level enough for high performance and precision, yet generic enough to handle generic and heterogeneous digital designs without modification. To achieve this, CellIFT leverages the logical properties of cells such as *monotonicity*, *transportability*, and *translatability*. Our evaluation using five real RISC-V designs with various complexities shows the superior scalability, precision and performance of CellIFT. We further used CellIFT in different scenarios to show-case its effectiveness in detecting various classes of flaws and vulnerabilities. CellIFT is the first open-source dynamic IFT solution, enabling hardware security research for the wider community.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable feedback. This work was supported in part by a Microsoft Swiss JRC grant and by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE). As to the opinions and positions in this document that the authors express or to which the authors contributed, they are those of the authors and do not represent the views of any current or previous employer, including Intel Corporation or its affiliates.

References

[1] CVE-2018-3639. Available from MITRE, CVE-ID CVE-2018-3639, 2018.

[2] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner. Register transfer level information flow tracking for provably secure hardware design. In *DATE*, 2017.

[3] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. The rocket chip generator. *Tech. Rep. UCB/EECS-2016-17*, 2016.

[4] Krste Asanovic, David A Patterson, and Christopher Celio. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical report, University of California at Berkeley Berkeley United States, 2015.

[5] S. Banerjee, D. Devecsery, P. M. Chen, and S. Narayanasamy. Iodine: fast dynamic taint tracking using rollback-free optimistic hybrid analysis. In *IEEE SP*, 2019.

[6] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *ACM SIGSAC*, 2019.

[7] RISC-V BOOM. Boom speculative attacks. https://github.com/riscv-boom/boom-attacks. [Online; accessed 16-May-2022].

[8] J.V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel sgx kingdom with transient out-of-order execution. In *USENIX SEC*, 2018.

[9] Cadence. Jaspergold security path verification app. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html. [Online; accessed 16-May-2022].

[10] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX SEC*, 2019.

[11] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *ACM SIGSAC*, 2019.

[12] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovic. Boomv2: an open-source out-of-order risc-v core. In *CARRV*, 2017.

[13] E. M. Clarke, W. Klieber, M. Novacek, and P. Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*. Springer, 2011.

[14] Intel Corporation. 12th generation Intel® core™ processor, document number: 682436-006.

[15] O. Demir, W. Xiong, F. Zaghloul, and J. Szefer. Survey of approaches for security verification of hardware/software systems. *IACR Cryptol.*, 2016.

[16] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications ACM*, page 504–513, 1977.

[17] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran. Hardfails: Insights into software-exploitable hardware bugs. In *USENIX SEC*, 2019.

[18] D. Evtyushkin, R. Riley, N. CSE Abu-Ghazaleh, ECE, and D. Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[19] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz. Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking. In *DATE*, 2019.

[20] F. Farahmandi, Y. Huang, and P. Mishra. Formal approaches to hardware trust verification. In *The Hardware Trojan War*. Springer, 2018.

[21] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh. Verification of a practical hardware security architecture through static information flow analysis. In *ASPLOS*, 2017.

[22] L. Fix. *Fifteen Years of Formal Property Verification in Intel*. 2008.

[23] L. Fix and K. McMillan. Formal property verification. In *EDA for IC System Design, Verification, and Testing*. 2018.

[24] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu. Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *ACM ISCA*, 2021.

[25] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX SEC*, 2018.

[26] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX SEC*, 2017.

[27] Hack@DAC. Phase 2 Buggy SoC. https://github.com/hackdac/hackdac_2018_beta, 2018. [Online; accessed 16-May-2022].

[28] M. Hicks, C. Sturton, S. King, and J. Smith. Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs. In *ASPLOS*, 2015.

[29] W. Hu, A. Ardeshiricham, and R. Kastner. Hardware information flow tracking. *ACM CSUR*, 2021.

[30] W. Hu, A. Becker, A. Ardeshiricham, Y. Tai, P. Ienne, D. Mu, and R. Kastner. Imprecise security: quality and complexity tradeoffs for hardware information flow tracking. In *ICCAD*, 2016.

[31] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. An improved encoding technique for gate level information flow tracking. In *IWLS*, 2011.

[32] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. Theoretical fundamentals of gate level information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2011.

[33] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. On the complexity of generating gate level information flow tracking logic. *IEEE Transactions on Information Forensics and Security*, 2012.

[34] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *IEEE SP*, 2021.

[35] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor. In *DSN*, 2009.

[36] V. P Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *ACM SIGPLAN/SIGOPS*, 2012.

[37] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, 2019.

[38] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, et al. Meltdown: Reading kernel memory from user space. In *USENIX SEC*, 2018.

[39] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE HPCA*, 2016.

[40] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, 2015.

[41] Tortuga Logic. Emulation based security verification. https://tortugalogic.com/radix-m/. [Online; accessed 16-May-2022].

[42] Tortuga Logic. Simulation based security verification. https://tortugalogic.com/radix-s/. [Online; accessed 16-May-2022].

[43] lowRISC C.I.C. Ibex: An embedded 32 bit risc-v cpu core. https://github.com/lowRISC/ibex. [Online; accessed 16-May-2022].

[44] lowRISC contributors. Aes s-box verilator testbench. https://github.com/lowRISC/opentitan/tree/master/hw/ip/aes/pre_dv/aes_sbox_tb. [Online; accessed 16-May-2022].

[45] lowRISC contributors. Req/ack syncronizer verilator testbench. https://github.com/lowRISC/opentitan/tree/master/hw/ip/prim/pre_dv/prim_sync_reqack. [Online; accessed 16-May-2022].

[46] Arm Ltd. Speculative processor vulnerability. https://developer.arm.com/Arm%20Security%20Center/Speculative%20Processor%20Vulnerability, 2022. [Online; accessed 16-May-2022].

[47] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN*, 2005.

[48] G Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *ACM SIGSAC*, 2018.

[49] A. B. Mehta. *Constrained Random Verification (CRV)*. 2018.

[50] Ashok B Mehta. *SystemVerilog Assertions and Functional Coverage*. Springer, 2020.

[51] X. Meng, S. Kundu, A. K. Kanuparthi, and K. Basu. RTL-ConTest: Concolic Testing on RTL for Detecting Security Vulnerabilities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[52] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*. Springer, 2006.

[53] C Palmiero, G. Di Guglielmo, L. Lavagno, and L.P. Carloni. Design and implementation of a dynamic information flow tracking architecture to secure a risc-v core for iot applications. In *IEEE HPEC*, 2018.

[54] C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni. Tainthls: High-level synthesis for dynamic information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5), 2018.

[55] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX SEC*, 2021.

[56] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *IEEE SP*. Institute of Electrical and Electronics Engineers Inc., 2021.

[57] RISC-V. Architectural testing framework. https://github.com/riscv-non-isa/riscv-arch-test. [Online; accessed 16-May-2022].

[58] F. Schuiki, A. Kurth, T. Grosser, and L. Benini. Llhd: A multi-level intermediate representation for hardware description languages. In *ACM PLDI*, 2020.

[59] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross privilege boundary data sampling. In *ACM SIGSAC*, 2019.

[60] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss. Netspectre: Read arbitrary memory over network. In *ESORICS*, 2019.

[61] Zachary Snow. sv2v: Systemverilog to verilog. https://github.com/zachjs/sv2v/. [Online; accessed 16-May-2022].

[62] J. Stecklina and T. Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[63] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *ACM SIGPLAN*, 2004.

[64] Synopsys. Vc formal. https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html. [Online; accessed 16-May-2022].

[65] J. Taneja, Z. Liu, and J. Regehr. Testing static analyses for precision and soundness. In *CGO*, 2020.

[66] L. Team. Dataflowsanitizer design document. https://clang.llvm.org/docs/DataFlowSanitizerDesign.html, 2019. [Online; accessed 16-May-2022].

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

[68] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks. Fuzzing hardware like software. *arXiv preprint arXiv:2102.02308*, 2021.

[69] Jo Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *IEEE SP*, 2020.

[70] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Addendum to ridl: Rogue in-flight data load, 2019. https://mdsattacks.com/files/ridl-addendum.pdf.

[71] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Ridl: Rogue in-flight data load. In *IEEE SP*, 2019.

[72] S. Van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. Cacheout: Leaking data on Intel cpus via cache evictions. In *IEEE SP*, 2021.

[73] Veripool. Verilator, the fastest Verilog/SystemVerilog simulator. https://veripool.org/verilator/. [Online; accessed 16-May-2022].

[74] C. Wolf. Yosys open synthesis suite, 2016.

[75] Y. Yarom and K. Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX SEC*, 2014.

[76] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE VLSI*, 2019.

[77] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A hardware design language for timing-sensitive information-flow security. *Acm SIGPLAN*, 2015.

[78] Z. Zhou, M. K. Reiter, and Y. Zhang. A Software Approach to Defeating Side Channels in Last-Level Caches. In *ACM SIGSAC*, 2016.

A Cell Statistics

Figure 12 provides details on the cell composition of the evaluated designs before instrumentation, and after instrumentation by each mechanism. For readability, the size of the cells is not indicated.

B Transportability Architecture Proof

We prove the correctness and precision of the adder IFT logic illustrated in Figure 4 and equivalently given by Equation 5.

We conduct a proof by induction on a ripple carry adder. Because any adder implementation fulfills the same mathematical function, the proof holds for any adder implementation.

Notations. We denote by $W := \frac{N_i}{2}$ the width of each input A and B. As illustrated in Figure 13 we adopt the following notations: for the jth full adder inputs: A_j, B_j and c_{j-1}, and outputs Y_j and c_j, with the carry bit sequence $c := \{c_W, ..., c_0, c_{-1}\}$ where $c_{-1} := 0$. We define $\tilde{I}^0 := I \wedge \overline{I^t}$ (tainted inputs deasserted) and $\tilde{I}^1 := I \vee I^t$ (tainted inputs asserted). The generic notation X represents Y or c.

Proof by induction. We consider the following induction property H_{X_j}, for $0 \leq j < W$, for a fixed input I with taints I^t:

«[$\exists\ \tilde{I}$ such that $\overline{(I \oplus \tilde{I}) \wedge \overline{I^t}}$ (match_on_nontaint property) and $X_j(I) \neq X_j(\tilde{I})$ (toggled$_{X_j}$)] $\iff$ [$X_j(\tilde{I}^0) \neq X_j(\tilde{I}^1)$ (polarization$_{X_j}$) or $A^t_j \vee B^t_j$ (transport$_j$)] »: H_{X_j}.

Intuitively, we want to prove, for each output bit Y_j and each carry bit c_j (including the intermediate carry bits), that if I and some other input $\tilde{I}$ match on non-tainted bits and this bit Y_j or c_j is toggled between inputs I and $\tilde{I}$, then polarization and transportability taint the information flow (*correctness*), and conversely (*precision*).

Proof. Let us first prove H_{c_0} and H_{Y_0}. Because $Y_0 = A_0 \oplus B_0$ and $c_0 = A_0 \wedge B_0$, then for Y_0 or c_0 to be tainted, at least one of A_0 and B_0 must be tainted, and conversely. Therefore H_{Y_0} and H_{c_0} hold by (transport$_0$).

Let us now prove H_{c_j} and H_{Y_j} for a given $1 \leq j < W$, assuming that $H_{c_{j-1}}$ holds. We start by showing the implication: $\exists \tilde{I}$ such that (match_on_nontaint) and (toggled$_{X_j}$) $\implies$ (polarization$_{X_j}$) or (transport$_j$), corresponding to correctness.

If A^t_j or B^t_j, then H_{c_j} and H_{Y_j} hold by (transport$_j$). Let us suppose from now that $A^t_j = B^t_j = 0$ and suppose the existence of $\tilde{I}$ that satisfies the conditions (match_on_nontaint) and (toggled$_{c_j}$). Because A_j and B_j are not tainted and therefore identical in I and $\tilde{I}$ by (match_on_nontaint), and because $c_j = [A_j + B_j + c_{j-1} \geq 2]$, it results that c_{j-1} is toggled when applying $\tilde{I}$ instead of I. By $H_{c_{j-1}}$, (a) Either c_{j-1} is toggled between $\tilde{I}^0$ and $\tilde{I}^1$ (polarization$_{c_{j-1}}$), (b) Or $A^t_{j-1} \vee B^t_{j-1}$ (transport$_{j-1}$). Hence, in both cases (a) and (b), c_j is also toggled between $\tilde{I}^0$ and $\tilde{I}^1$, i.e., (polarization$_{c_j}$) holds.

Let us now prove: $\exists \tilde{I}$ such that (match_on_nontaint) and (toggled$_{X_j}$) $\impliedby$ (polarization$_{X_j}$) or (transport$_j$), corresponding to precision. If (polarization$_{X_j}$), then $\tilde{I} = \tilde{I}^0$ or $\tilde{I} = \tilde{I}^1$. If (transport$_j$), then because $c_j = [A_j + B_j + c_{j-1} \geq 2]$ and Y_j is $[A_j + B_j + c_{j-1}] \mod 2$, $\tilde{I} : I \oplus (1 \ll j)$ is a candidate.

Therefore, H_{Y_j} and H_{c_j} hold. We have proved correctness and precision of the IFT logic described by Equation 5. □

C Pivoting for Precise Shift Cells

In this appendix, we compute an IFT logic for the right shift cell by an unsigned offset by introducing a pivoting technique. This appendix relies on the assumption that the shift offset B and its taint vector B^t verify $B \wedge \overline{B^t} = 0$ (i.e., B is zero at all non-tainted indices). This property is provided in the second shift cell obtained from the translatability property.

Logical shifts. We introduce the notation $\overset{t}{=}$, which denotes that two vectors match on non-tainted bits, as defined in Equation 10, and its counterpart $\overset{t}{\neq}$ in Equation 11. The decomposition by translability guarantees that all the non-tainted bits in the shift offset $B \wedge B^t$ of this cell are zero, which substantially simplifies the IFT logic computation. We prove that an IFT logic for this second cell can be expressed by Equation 12 for right shifts and by Equation 13 for left shifts.

$$U \overset{t}{=} V :\iff U^t \vee V^t \vee \overline{[U \oplus V]} \tag{10}$$

$$U \overset{t}{\neq} V :\iff U^t \vee V^t \vee [U \oplus V] \tag{11}$$

$$Y^t_j = \bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \overset{t}{=} k \right] \wedge \left[A_j \overset{t}{\neq} A_{j+k} \right] \right) \tag{12}$$

$$Y^t_j = \bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \overset{t}{=} k \right] \wedge \left[A_j \overset{t}{\neq} A_{j-k} \right] \right) \tag{13}$$

Focusing on a logical right shift, the jth output bit Y_j is tainted if either (*explicit tainting*) the taint results from shifting some tainted bit A_k to the right to the index j, or (*implicit tainting*) some offset $\tilde{B}$ that matches with B on non-tainted bits gives a different value for Y_j, than the offset B.

Because we know that the full-zero vector satisfies the matching condition of $\tilde{B}$, there is always some *pivot* $\tilde{B}^0 = 0$ that maps A_j to Y_j. We iterate with some integer k through all the 2^{N_B} values of $\tilde{B}$, assuming the worst case where the taint vector B^t is full of ones. We complete A with zeroes beyond the most significant bit to simplify the equations without loss

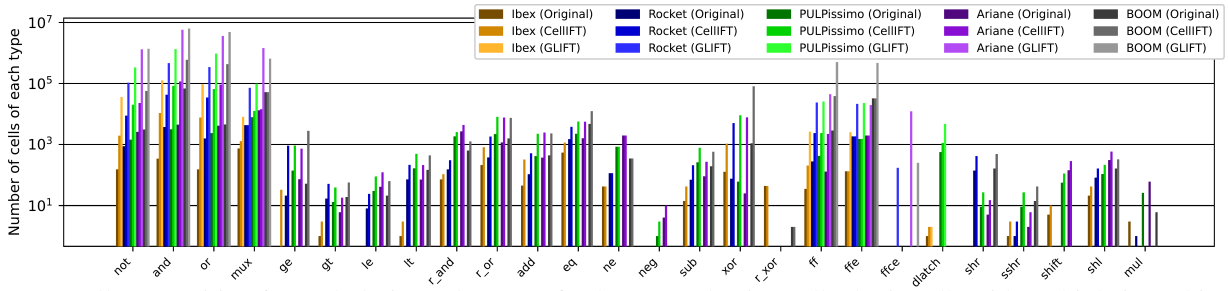

Figure 12: Cell composition for each design. The *r_* prefix denotes reduction cells (logic cells with multiple input bits but a single output bit. Note that the Y axis is logarithmic.)

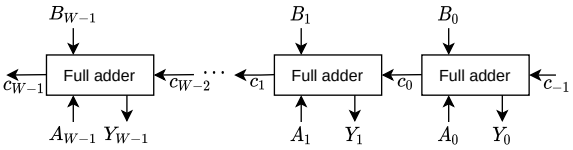

Figure 13: Notations for the ripple carry adder.

of generality. If there is any integer k in this range such that (*implicit tainting*) there is a $\tilde{B}$ that can bring A_{j+k} to Y_j, then Y_j is tainted if $A_j \neq A_{j+k}$ (because there are two $\tilde{B}$ instances that result in different values for Y_j), or (*explicit tainting*) if A_j or A_{j+k} is tainted, because then there is some $\tilde{B}$ that shifts a tainted value to the output bit Y_j, leading to Equation 12.

Arithmetical right shift. The IFT logic of the second cell in the translatability decomposition of an arithmetical right shift can be expressed by Equation 14.

$$Y_j^t = \bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \stackrel{t}{=} k \right] \wedge \left[A_j \stackrel{t}{\neq} A_{min(j+k,N_A-1)} \right] \right) \tag{14}$$

The only difference with logical right shift tainting relies in the *implicit tainting* part. Instead of completing A with zeroes, A must be completed with values equal to its most significant bit A_{N_A-1}. From there, a calculation similar to the logical counterpart leads to the IFT logic described in Equation 14.

We now sketch a proof of the precision of the composition of the two IFT logic instances resulting from the decomposition of the arithmetical right shift.

The $N_A - B^0$ most significant bits of $C(A, B^0)$'s output benefit from the taint combination surjectivity of the first cell's IFT logic, because these bits are shifted by a non-tainted offset. Therefore, the second cell's output taints corresponding to these inputs in step 2 are precise (see Appendix D). The only tainted signals that cannot be surjectively enumerated are the $N_A - B^0$ most significant bits, which take A's most significant bit's value. However, although a comparison between these could lead to an erroneous intermediate result, they will be eventually tainted by the $\stackrel{t}{=}$ operator.

Therefore, the composition of the two instances is precise.

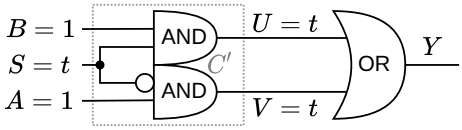

Figure 14: Multiplexer as the succession of a cell C' and an OR gate. $A = B = 1$, $A^t = B^t = 0$ and $S^t = 1$, therefore the value of S does not matter for the IFT logic.

D Taint combination surjectivity

We discuss taint combination surjectivity as a tool for understanding and proving precision of cell compositions.

Let C be a cell with IFT logic $L := C^t$. L is said to be *taint combination surjective* if for all inputs I and all taint vectors I^t, it satisfies Equation 15. Informally, such an IFT logic does not create any new interdependencies between taint signals, because all potential outputs $\tilde{Y}$ that match with Y on non-tainted output bits Y^t are obtainable from inputs $\tilde{I}$ that match with I on non-tainted input bits I^t.

$$\forall \tilde{Y} \exists \tilde{I} \mid C^t((I \wedge \overline{I^t}) \vee (\tilde{I} \wedge I^t)) = (Y \wedge \overline{Y^t}) \vee (\tilde{Y} \wedge Y^t) \tag{15}$$

A composition of cells $C'(C(I))$, each with precise IFT logic L and L', can be precisely instrumented with $L' \circ L$ if L is taint combination surjective. A composition of taint combination surjective IFT logics is also taint combination surjective: the first IFT logic allows to generate all the intermediate combinations for the second by surjectivity.

Intuition. We introduce a counterexample, where C^t is not taint combination surjective. The multiplexer is known to be imprecise [67] if it is instrumented as the composition of one OR gate after two AND gates. Figure 14 shows a multiplexer made of two cells: C' and an OR gate. The outputs U and V of C' are tainted, given the precise taint propagation rule. C'^t is not taint combination surjective, because for $A = B = 1$, $A^t = B^t = 0$ and $S^t = 1$, there is no input $\tilde{I} = \{\tilde{A}, \tilde{B}, \tilde{S}\}$ which matches with I on non-tainted bits (i.e., on A and B), that produces the output $\tilde{Y} := \{U = 0, V = 0\}$.

Logical shifts. All logical shift cells with untainted offset are taint combination surjective. We provide a proof for the left shift. The proof for the right shift is similar.

Proof. Let B be the untainted offset (i.e., $B^t = 0$). Let A be the shifted input, with taints A^t. Then, $Y = A \ll B$ and $Y^t = A^t \ll B$. Let $\tilde{Y}$ be a vector of same width as Y, with $Y \wedge \overline{Y^t} = \tilde{Y} \wedge \overline{Y^t}$. Then, $\tilde{A} := \tilde{Y} \gg B$ is a preimage of $\tilde{Y}$. □

E Artifact

E.1 Abstract

In our Artifact, we provide the source code of CellIFT, a native RISC-V toolchain, and other dependencies. We also provide the framework for performing all the experiments described in this paper and analyzing the obtained results. Everything is packaged as a Docker image to allow for optimal reproducibility. To reproduce the experiments, we expect a machine with 256 GB memory and 500 GB of free storage.

E.2 Artifact checklist

- **Algorithm:** CellIFT is a newly developed algorithm to efficiently generate IFT shadow logic as part of a Yosys pass.
- **Program:** We use a set of five external RISC-V CPU designs (Ariane, BOOM, Ibex, Rocket, PULPissimo) as evaluation targets, as well as benchmarks from the RISC-V Architectural testing framework. All of this code is included in our artifact.
- **Compilation:** We include the required compilers and interpreters.
- **Transformations:** We include the required Verilog transformations (CellIFT and GLIFT), implemented as Yosys passes.
- **Binary:** We include prebuilt Verilator binaries of the five CPU designs in all instrumentation modes (i.e., vanilla, CellIFT, and GLIFT) where possible. Note that GLIFT instrumentation or synthesis sometimes fails, as explained in Section 7.2.
- **Run-time environment:** The bulk of our artifact is a Docker image that runs on Linux. We tested our image on an Ubuntu 22.04 system with 5.15.0-37-generic kernel.
- **Hardware:** We do not require any special hardware, but do need a relatively large amount of DRAM (256 GB) to run all the experiments.
- **Metrics:** The experiments record runtime performance and IFT precision for microbenchmarks for CellIFT as well as GLIFT. Further experiments record execution time and memory footprint of the instrumentation and synthesis process for all instrumentation modes. We also measure the simulation performance on for all instrumentation modes. Lastly, we show resource usage and clockable frequency after FPGA synthesis for all the five CPU designs under all instrumentation modes.
- **Output:** For all experiments used in the Evaluation section of this paper (Section 7), we include code to regenerate the charts. Also, we include code to reproduce all results in the Scenarios section of this paper (Section 8).
- **Experiments:** With the exception of the FPGA results, all experiments are executed automatically when building the Docker image. This means the way to reproduce all experiments is encoded in the Dockerfile, and a Docker container based on this Dockerfile would contain the generated results, and can be used to re-run individual experiments if desired.
- **How much disk space required:** The docker image with all the layers is 330 GB, and Xilinx Vivado requires around 150 GB for downloading and installation. In total, we estimate a total of 500 GB of free storage is required.
- **How much time is needed to prepare workflow:** To prepare the workflow, conscious effort is only needed to retrieve the Git repository and the Docker image, which should take only a few minutes.
- **How much time is needed to complete experiments:** Reproducing the experiments takes approximately 3 days.
- **Publicly available:** Stable URL: https://github.com/comsec-group/cellift-artifacts/commit/eaa9a26ae85fd6a7ae8cd248416315414ae4c135. The README points to a stable (sha256-verified) Dockerhub Docker image that contains the rest of the code and data, namely docker.io/ethcomsec/cellift-artifact-evaluation@sha256:9a15d4070d321026ad4d5d9ba5a236842c6c456279f9c08f4fa4132de7b399ce.
- **Code licenses:** CellIFT is licensed under GPL3.
- **Workflow frameworks used:** Docker, Make, Luigi.

E.3 Description

E.3.1 How to access

The project is located at https://comsec.ethz.ch/cellift. Our artifact is a single Git repository designed primarily to build a Docker image that has run all the experiments automatically. This Git repository is hosted at the 'Publicly available' checklist entry. The README.md in that repository contains further instructions to obtain the prebuilt Docker image from Dockerhub.

E.3.2 Hardware dependencies

The artifact will run all experiments on a machine with 256 GB of memory.

E.3.3 Software dependencies

We tested the Docker image on Ubuntu 22.04 LTS kernel 5.15.0-37-generic, but we expect it to work on a wide range of Linux distributions.

To reproduce the FPGA experiments in the paper, we furthermore depend on the Xilinx Vivado FPGA synthesis tool (version 2019.3).

E.4 Installation

The installation of our artifact requires the following two steps:

1. Cloning the git repository specified in the checklist and using its README.md to pull the Docker image artifact hosted on Dockerhub.
2. Reproducing the FPGA experiments, requires the installation of the full edition of Vivado 2019.3 from the Xilinx website and a license.

E.5 Experiment workflow

Follow the instructions in the git repository README.md that specifies in detail how to start a Docker container with the image, and how to reproduce each experiment, and examine the results.

In principle, cloning the git artifact repository and rebuilding the Docker image using the Dockerfile in the git repository will rebuild all CELLIFT code and designs from scratch and perform the experiments (except the FPGA experiments). For maximum reliability, we also provide the prebuilt Docker image with all code, binaries and results that we have found to work, which can be used to reproduce all the experiments (and use CELLIFT in general if desired).

To run the FPGA experiments, first source the settings64.sh file from the Vivado installation dir, and follow the instructions in the Artifact README.md.

E.6 Evaluation and expected results

The key results from our experiments are as follows. For each result, we point to scripts (Python or bash) that drive the experiments and show the analysis.

1. Instrumented designs that we can synthesize to C++ (i.e. be compiled) for all five RISC-V CPU designs, contrary to GLIFT, and with less CPU time and memory (follows from plot_instrumentation_performance.py and plot_rss.py), and with higher tainting precision (follows from plot_num_tainted_states_ibex.py).
2. For the designs that can be compiled in all instrumentation modes, we show that CELLIFT has lower performance overhead than GLIFT (follows from plot_benchmark_performance.py).
3. The Meltdown and Spectre simulations reproduce Figure 11, showing they can both be detected (follows from plot_tainted_elements.py).
4. We show several bug scenarios detected by CELLIFT (run_scenarios.sh).
5. We show FPGA synthesis results, showing that CELLIFT instrumented designs can be synthesized, with fewer resources than the GLIFT instrumented designs.

We refer to the README.md of the artifact git repository for the detailed steps to reproduce each of the key results described above.

E.7 Experiment customization

There is ample customization opportunity in the Docker image, because the code of the instrumentation tool as well as the target designs are there and can be modified and rebuilt. This does require a deeper knowledge that goes beyond this appendix.

E.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.