HybriDIFT: Scalable Memory-Aware Dynamic Information Flow Tracking for Hardware

Flavien Solt
ETH Zürich

Kaveh Razavi
ETH Zürich

***Abstract*—Designing correct and secure hardware is challenging. Dynamic information flow tracking (DIFT) enhances RTL testing flows, for example, by providing formal guarantees on detecting information leakage. However, existing DIFT solutions do not scale to large memories encountered in complex processors. A formal analysis of existing DIFT mechanisms reveals the two factors that fundamentally limit the scalability of instrumenting memories: existing mechanisms enforce that all memory words must be accessible simultaneously, and dependent reads and writes must happen concurrently. These aspects that are detrimental to scalability are all due to precise tracking of implicit flows for every memory word, which is not required in many scenarios of interest. Based on this insight, we design HybriDIFT, a module-level DIFT memory instrumentation based on SRAM deduplication and on a single state bit that tracks implicit information flows. HybriDIFT can automatically identify memories and their protocols by combining static and dynamic analysis. HybriDIFT is precise in practice and scalable to RTL designs that feature large memories. We evaluate HybriDIFT by automatically instrumenting a set of open-source hardware designs. With Verilator, HybriDIFT accelerates build time by $1.06\times$ to $3.5\times$ and simulation by $2.6\times$ to $5.1\times$ on default target configurations, and instruments a larger OpenC910 configuration that was out of reach for the state-of-the-art DIFT mechanisms, while preserving sufficient precision for all known applications.**

I. Introduction

Designing correct hardware is challenging, as witnessed by the number of bugs regularly found in commercial [1] and open-source [2]–[6] designs. Hardware dynamic information flow tracking (DIFT) provides strong guarantees regarding confidentiality and integrity. DIFT has recently been shown to enable the detection of information leakage, of architectural bugs and of microarchitectural vulnerabilities, and to scale to complex designs such as out-of-order processors [7].

Yet contemporary DIFT imposes a significant overhead both when building and simulating designs, because of the instrumentation of deep memories, i.e., memories that contain many words. Hence, a common practice is to modify designs before instrumenting them, typically by minimizing internal memories such as microarchitectural structures [7], [8]. This approach has two major shortcomings. First, parametrizable designs must be specifically configured towards smaller memories to be efficiently instrumentable by DIFT mechanisms. Furthermore, this implies that DIFT mechanisms are effectively unsuited to instrument actual design versions. This is unfortunate since DIFT may miss problematic information flows that may only appear with default memories. The second shortcoming is that some designs are only slightly parameterized [9], [10], resulting in severe scalability problems for DIFT. Given the compelling theoretical promises made by DIFT [7], [11]–[17], it is paramount to have a practical solution that scales with the increasing design complexities and sizes.

We make the observation that the existing DIFT mechanisms will attempt to track implicit information flows in memories with a high precision [7], [8], [18]. Besides poor scalability, we find that precise implicit flow tracking has never been useful for any known hardware DIFT application, as opposed to precise explicit flow tracking, which we will show to be efficiently implementable. As an example, detecting a Meltdown-like vulnerability [19] first requires a legitimate load of data tainted as confidential by a privileged instance into some cache. Precisely tracking this explicit flow of tainted data to the specific target cache word is important, as excessively tainting the cache would mark other data or instructions as confidential and would very easily lead to an intractable number of false positives when they are loaded or executed. As a second step, an attacker speculatively loads the data from the cache and uses it as an address to access a different memory location in the cache. At this point, what exact words of memory will be tainted is of low relevance, as taint explosion usually follows regardless of slight variations in the number of tainted words after the occurrence of the implicit flow [7].

Our formal analysis identifies two factors that fundamentally limit the scalability of precise tracking of implicit information flows: the need for accessing all values together at once, and the need for dependent reads and writes in a single cycle. Hence, to provide scalability while maintaining sufficient precision in practical scenarios, we propose to treat explicit information flows to memory words precisely, while collapsing implicit information flows into a single taint bit. None of the proposed DIFT abstraction levels (gate [8], RTL [18], or macro-cell [7]), however, captures the memory behavior at a level that enables us to treat explicit and implicit flows separately. Indeed, memory is a higher-level construct, and our empirical study shows that in standard designs, the module level captures memories very effectively. We present HybriDIFT, the first DIFT mechanism that instruments memories at the module level, while defering the rest of the instrumentation to existing DIFT mechanisms that operate at a lower abstraction level [7], [8], [18]. HybriDIFT can either automatically identify memories and their protocols using a combined static and dynamic analysis, or rely on a manual annotation of the memory interfaces to instrument them.

We evaluate HybriDIFT on a set of open-source designs and show that it can significantly accelerate both elaboration and simulation. To show that HybriDIFT maintains the precision of existing DIFT mechanisms, we reproduce the variety of use cases demonstrated by the state-of-the-art DIFT instrumentation mechanism [7] and show that the approximation of implicit flows does not deteriorate the results.

In summary, our contributions in this paper are:

- We identify the fundamental limitations of existing DIFT mechanisms when it comes to memory instrumentation.
- We propose a novel memory instrumentation that is scalable and sufficiently precise for practical use cases.
- We introduce HybriDIFT, the first DIFT instrumentation mechanism that operates at module level.
- We evaluate HybriDIFT on a set of open-source designs, and report an acceleration of up to $3.5\times$ for build and $5.1\times$ for simulation.
- We reproduce the variety of use cases demonstrated by state-of-the-art DIFT mechanism [7] and show that the detection results are unaffected.

HybriDIFT is available at https://comsec.ethz.ch/transfuzz/

II. Background

In this section, we provide background regarding abstraction levels in Verilog, hardware DIFT, and hardware memories.

A. Levels of abstraction in Verilog

Hardware designs are generally described at the Register Transfer Level (RTL) in languages such as Verilog. RTL is an abstraction level describing the behavior of digital circuits in terms of binary data flows between registers, and the arithmetic or logical operations that are performed on the data. Modules describe the hierarchy of a design [20]. All digital circuits can be described down to a (large) netlist of logic gates, such as AND, OR, and simple stateful elements such as flip-flops and latches. Electronic Design Automation (EDA) software often first transforms the RTL description into a netlist of macro-cells, which is the preferred intermediate representation for optimizations, transformations and synthesis [21].

B. Hardware DIFT

An information flow (also known as taint flow) between a source and a sink signifies that changes in the valuation of the source signal can affect the valuation of the sink signal. Computing whether a signal A should be tainted is equivalent to iterating through all valuations of all tainted signals, and observing whether A is affected. Information Flow Tracking (IFT) can be used to detect information leaks like side-channel attacks or to detect bugs in a design [7], [8], [11], [13], [22]. Dynamic IFT (DIFT) instruments designs by concretely inserting additional logic to track the flow of information through the original design, while the original design's functionality remains unaffected. This instrumentation allows following information flows through the design at runtime, typically during simulation, or when proving formal properties. Several abstraction levels have been proposed for DIFT instrumentations: gate level (GLIFT) [8], RTL (RTLIFT) [18], and most recently the Yosys macro-cell level (CellIFT) [7].

The existing instrumentation mechanisms [7], [8] hit a scalability wall when instrumenting deep memories (i.e., memories with many words) such as the default caches of CVA6 [9], Rocket [23], BOOM [24] and OpenC910 [10]. Consequently, such designs are sometimes adapted specifically for instrumentation [7]. Some open-source designs are not much parametrizable in terms of memory sizes [9], [10], making their instrumentation difficult in practice, by incurring either a heavy manual effort, or significant performance overheads.

C. Memories

At design time, the exact memory implementation (mega-cell for ASICs, BRAM for FPGAs) is usually abstracted away and only the memory interface is described. The memory is instantiated as a black-box module that emulates the functionality for simulation purposes. In particular, apart from a clock signal, memories have a data input and output, an address input, and some enable signals. SRAMs are particularly area- and power-efficient memories, at the cost of providing access to only one element at a time, as opposed to standard cells, which may provide access to multiple elements concurrently [25]–[28].

III. Observations and Overview

In this section, we first analyze the scalability issues of existing DIFT mechanisms and the precision requirements of known DIFT applications. From these observations, we describe challenges that will guide the design of HybriDIFT.

A. Observations

We analyze the scalability of existing instrumentation mechanisms and the role of their precision in all known use cases.

a) Scalability: To understand the scalability of existing DIFT mechanisms, we consider a 32-bit wide memory module and instrument it for various depth values, i.e., various numbers of words. We measure the performance of instrumentation and simulation for memories of increasing depth and report the results in Figure 1. The state of the art (CellIFT) [7] could not instrument memories with a depth of 64kWord or more in 12 hours. While we conducted the experiment both with GLIFT and CellIFT, we only report the results for CellIFT, as the latter systematically shows significantly better performance and non-inferior precision by construction. Note that the L2 cache of OpenC910 [10] contains 8kWord memories by default, but can be configured to contain memories of up to 64kWords (`spsram_65536x128`). Evidently, the latter cannot be instrumented or simulated in reasonable time with CellIFT given the results reported in Figure 1. Future RISC-V CPUs with deeper pipelines, microarchitectural components and caches will certainly all face similar scalability issues.

Observation 1. Instrumenting deep memories with state-of-the-art DIFT mechanisms is expensive.

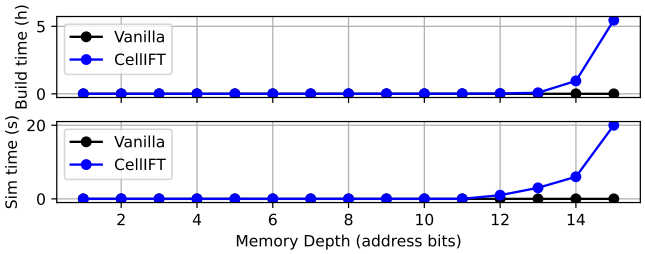

Fig. 1. Performance of building and simulating memories of various depths. Vanilla represents the absence of instrumentation.

b) Precision: To understand the precision requirements of DIFT for memories, we enumerate all the known use cases of hardware DIFT. In the original work that presents a gate-level DIFT mechanism [8], the authors specifically designed a CPU and an ISA to ensure that memory addresses will not be tainted, to avoid taint explosion. W. Hu et al. [11], [14], [29] and RTLIFT [18] only showed applications in memory-less designs. J. Oberg et al. [30] instrumented a cache, but they only monitor whether it is reached by tainted signals. Clepsydra [22] relies on imprecise memory taints only. In the use cases demonstrated by CellIFT [7] memory addresses or enable signals are either never tainted (detection of microarchitectural leakage and of architectural bugs), or the detection indeed relies on the taint explosion induced by tainting memory control signals, typically memory addresses (detection of microarchitectural vulnerabilities).

Observation 2. There is no known use case of hardware DIFT that makes use of precise information flow tracking from memory addresses or enable signals.

B. Overview

We observed the scalability issues related to memories in existing DIFT mechanisms. Additionally, no known use case of hardware DIFT requires precise taint flows from memory addresses or enable signals. To provide a solution that is both efficient and practical, we tackle the following challenges.

Challenge 1. Analyze the potentials and limitations of precise memory DIFT instrumentation.

In Section IV, a formal analysis of precise memory DIFT reveals the two aspects that fundamentally limit the scalability of existing DIFT mechanisms. Both are related to precise implicit flow tracking, which are exactly the flows that can be tracked imprecisely in all known hardware DIFT use cases.

Challenge 2. Introduce a novel memory instrumentation that is scalable and with sufficient precision.

In Section V, given these formal insights, we propose a novel practical DIFT memory instrumentation that precisely tracks explicit flows and conservatively tracks implicit flows. This instrumentation is based on memory deduplication, and on the addition of a new aggregated state bit τ to track all implicit information flows at once.

Challenge 3. Detect memories and their protocols and provide an automated hybrid instrumentation solution.

In Section VI, we introduce HybriDIFT, a hybrid instrumentation solution that relies on manual annotations or automatic memory and protocol identification by a combined static and dynamic approach, and instruments them at module level, while delegating the rest of the design to an existing DIFT mechanism. We finally show the performance and practicality of HybriDIFT on a set of designs and scenarios.

IV. Precise Memory Instrumentation

In this section, we perform a formal analysis of precise memory DIFT and deduce the aspects that impair scalability.

We split the analysis between reads and writes. We observe that each of these two operations is made of an explicit and an implicit flow component. Such a separation has already been proposed in other software [31]–[35] and hardware [11], [18] DIFT applications. Note that this distinction between implicit and explicit components does not exist in current hardware DIFT instrumentations [7], [8], [18], which operate at a too low level of abstraction to capture it, but is conceptual separation that we introduce for the formal analysis.

Let us refer to memory bits as $M[S]_i$, where S is some word selector signal and i is the bit index in this word. We introduce the signals D_i and Y_i for the input and output data. We introduce, without loss of generality, W_i and R_i for bit-level write and read enable. Memories may for example feature both a write enable and a byte enable signal, which we combine into W_i for simplicity, as they would complicate the equations without affecting the conclusions of our analysis. We discuss in Section V-B how the analysis extends to slightly more complex memory interfaces. Finally, for some signal A, we denote its taint signal as A^t, which is a parallel signal with its own computation and propagation rules, as introduced in previous hardware DIFT work [7], [8], [18].

a) Precise memory writes: When a memory bit is tainted, then this taint is persistent until it is overwritten, because the value in the original memory will also be preserved until then. The update rule for writes is described by Equation 1.

The explicit flow $M[S]^t_{i,ex}$ corresponds to tainted data input, while the implicit flow $M[S]^t_{i,im}$ corresponds to tainted control input, i.e., addresses or enable signals.

$$M[S]^t_{i,next} = M[S]^t_{i,ex} \vee M[S]^t_{i,im} \tag{1}$$

In explicit flows, taints represent variability in *what* value is written, as expressed by Equation 2. Said otherwise, in explicit flows, changing the valuation of a tainted bit (i.e., changing the value of some input D_i, if $D^t_i = 1$) will only affect the value written to the specified memory word $M[S]$.

$$M[S]^t_{i,ex} = W_i \ ? \ D^t_i : M[S]^t_i \tag{2}$$

To express implicit flows, we involve the notation $A \overset{t}{=} B$ from [7]. It stipulates that two signals A and B match on the non-tainted bits indices, as described in Equation 3.

$$A \overset{t}{=} B \Leftrightarrow \forall j, (A_j = B_j) \vee A_j^t \vee B_j^t \tag{3}$$

In implicit flows, taints represent variability in *whether* the memory bit of interest could toggle, depending on potential changes in the valuations of tainted control input bits (addresses or write enable signals). Equation 4 expresses implicit flows for write signals. Implicit flows can occur due a tainted write enable signal ($W_i^t = 1$) or the word selector S being tainted, which implies that changes in valuations of tainted bits of S will decide whether $M[\tilde{S}]$ or another word will be written to. The selector $\tilde{S}$ is a conceptual memory index, hence it is never tainted. $\tilde{S}$ complies with the equation $S \overset{t}{=} \tilde{S}$, i.e., S could be changed into $\tilde{S}$ by only flipping tainted bits.

The second line of Equation 4 expresses that depending on the valuation of bits in control inputs whose taint signal counterpart is 1, the memory word at address $\tilde{S}$ could be written or not. The last line of Equation 4 expresses the three cases that could taint the memory given all the possibilities provided by the second line. Either the input data is tainted and the second line allows writing to the memory by choosing some valuation of tainted bits (D_i^t), or the memory word at address $\tilde{S}$ is already tainted ($M[\tilde{S}]_i^t$) and the second line allows not to write to the memory, or the input data and the present data are distinct at bit i ($D_i \oplus M[\tilde{S}]_i$), and hence the variation tolerated by line 2 between writing and not writing the word at address $\tilde{S}$ induces a change in the memory bit value, and hence a taint at this location in memory. Consequently, each of the two lines of Equation 4 represents respectively the existence of variations in *whether* the word at address $\tilde{S}$ is written (when changing valuations of bits that are tainted), and whether these variations can effectively impact the value of the memory bit at index i, hence the equation covers all cases.

$$\begin{aligned} \forall \tilde{S}, M[\tilde{S}]_{i,im}^t = \\ ((W_i^t \vee (W_i \wedge \bigvee S^t \wedge (S \overset{t}{=} \tilde{S})))\wedge \\ (D_i^t \vee M[\tilde{S}]_i^t \vee (D_i \oplus M[\tilde{S}]_i))) \end{aligned} \tag{4}$$

b) Precise memory reads: Contrary to writes, reads never leave a persistent taint in the memory.

For simplicity and without loss of generality, we assume that the memory returns 0 when it is not actively being read, and that the memory is being read combinationally, i.e., without latency, hence we avoid potential delays that would clutter the equations. Like for writes, the taint for reads is described by Equation 5, where Y is the data output signal.

$$Y_i^t = Y_{i,ex}^t \vee Y_{i,im}^t \tag{5}$$

Like for writes, taints in read explicit flows $Y_{i,ex}^t$ represent variability in *what* value is read, as expressed by Equation 6.

$$Y_{i,ex}^t = R_i \ ? \ M[S]_i^t : 0 \tag{6}$$

To express implicit read data flows, we first define in Equation 7 the set $\mathcal{S}(S, S^t)$ of all possible words that could be addressed in memory, given the word selector signal and its taint, if we could flip tainted bits of S, i.e., the bits of S at indices j such that $S_j^t = 1$.

$$\mathcal{S}(S, S^t) := \{M[\tilde{S}] \mid S \overset{t}{=} \tilde{S}\} \tag{7}$$

From there, we express the implicit read data flows in Equation 8. This expression is made of two components, where taint represents variability in *whether* something is read, and *where* in memory it is read from. Note that if we had not formulated the assumption that Y_i is always 0 when the memory is not actively being read, then the $M[S]_i$ term in the first component ($R_i^t \wedge (M[S]_i \vee M[S]_i^t)$) would have to be XORed with the value of Y_i if no read would happen. Hence we fixed this value to 0 for the reader's convenience. The read output is tainted by the second component if, depending on which address would be selected if tainted bits were flipped, the output bit could flip. For instance, if the memory is full of zeros, the read address will not influence the read output. On the contrary, if the memory contains a word full of ones and one word full of zeros, assuming that both words would be readable by flipping some tainted bits of S, then the read output will be tainted, as changes in valuations of tainted control bits (in that case in S) would influence the output value. Specifically, the last line of Equation 8 expresses that there are two memory words at two addresses reachable by flipping tainted bits of S that differ at bit i, or at least one of them is tainted at bit i.

$$\begin{aligned} Y_{i,im}^t = \\ (R_i^t \wedge (M[S]_i \vee M[S]_i^t)) \vee ((R_i \vee R_i^t)\wedge \\ (\bigvee\nolimits_{m_i \in \mathcal{S}(S,S^t)} m_i \vee m_i^t) \wedge (\bigvee\nolimits_{m_i \in \mathcal{S}(S,S^t)} \overline{m_i} \vee m_i^t)) \end{aligned} \tag{8}$$

c) Conclusions on precise memory instrumentation: From this analysis, we draw two observations that underline the structural implications of precise memory instrumentation for explicit and implicit flows, and hence the scalability implications. The first observation concerns explicit flows.

Observation 3. Precise explicit flows only require access to a single memory word per clock cycle.

The second observation concerns implicit flows. First, as shown by Equations 4 and 8, calculating precise information flows requires writing ($\forall \tilde{S}, M[\tilde{S}]_{i,im}^t = \cdots$) and reading ($\bigvee_{m_i \in \mathcal{S}(S,S^t)} \cdots$) as many words as the whole memory in

each clock cycle. Second, as expressed by the term $D_i \oplus M[\tilde{S}]_i$ in Equation 4, if changing valuations of tainted input bits can affect *whether* a write will happen, the memory must first be read, XORed and then written again, in a single clock cycle, to know whether this decision can impact the stored value.

Observation 4. Precise implicit flows require being able to read or write as much data as the whole memory size in a single cycle, and to perform dependent reads and writes in a single clock cycle.

These two observations are fundamental to the scalability aspect of precise memory instrumentation and will guide the design of our practical memory instrumentation.

V. Practical memory instrumentation

We observed that precise implicit flows have two fundamental properties that make them hard to implement in practice. This explains the scalability limits that we observed earlier in Section III-A-a. Yet these constraints are only there to satisfy precise information flow tracking from control signals. We have shown in Section III-A-b that such a precise tracking of implicit flows is not required in existing DIFT applications.

A. Adaptations for practicality

We propose to instrument memories by conservatively approximating information flows coming from the address and enable signals to resolve these issues. To efficiently aggregate these implicit information flows, we introduce a unique state bit τ that tracks whether some implicit flow from some control bits happened at write time. After τ is set, τ can only be unset by a reset signal. This state $\tau = 1$ models the typical enormous amount of taint that would occur from implicit address flows.

a) Practical memory writes: The update rule of τ, ignoring resets which unset it, is described by Equation 9.

$$\tau_{next} = \tau \vee (W_i \wedge \bigvee S^t) \vee W_i^t \tag{9}$$

Despite the simplicity of its update rule, τ captures all implicit flows for write signals, while requiring none of the two expensive constraints that we identified earlier. Equation 10 expresses the practical taint propagation from writes, where the explicit flow $M[S]^t_{i,ex}$ remains as defined in Equation 2 from Section IV.

$$M[S]^t_{i,next} = M[S]^t_{i,ex} \vee \tau \tag{10}$$

b) Practical memory reads: Since τ contains an over-approximation of the effects of implicit flows that occur during write operations, which would be stored in $M[S]^t_{i,next}$ in precise instrumentations, we must add a τ component in the read taint. Concretely, any read will be tainted if τ is set, which again makes an intuitive correspondence between $\tau = 1$ and the memory being potentially excessively tainted by implicit flows. Regarding the read enable signal taint flow, a precise instrumentation would require potentially reading all memory at once, hence we taint the output data whenever the read enable signal or the read address is tainted. Concretely, we transform Equation 8 into the more efficient Equation 11.

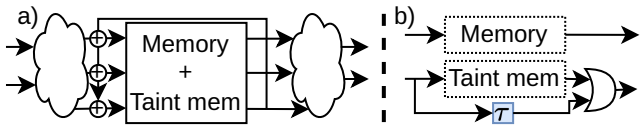

Fig. 2. Structural comparison between (a) precise and (b) practical memory instrumentations. The block in the precise implementation must be implemented as a complex monolithic block of interconnected standard cells, while the dotted lines represent traditional SRAMs.

$$Y^t_{i,im} = R^t_i \vee (R_i \wedge \tau) \tag{11}$$

B. Implementation

a) Adaptations to concrete memory interfaces: Memories may feature additional interface signals, such as a byte enable signal B, which we have abstracted away for simplicity, but which must be taken into account in practice. For example, if a write would happen only if $W_i \wedge B_i = 1$, using bitwise notations for consistency where B_i concerns the data word bit i, then even if $W^t_i = 1$, the memory bit would not be tainted if $B_i = 0$. To handle all these cases, we must first understand the exact memory protocol, which we do in Section VI, and then we apply the DIFT rules introduced by GLIFT [8] to the memory interface signals. With a byte enable signal B, the condition $W^t_i = 1$ in the equations from Section V-A will be replaced by $(W^t_i \wedge B_i) \vee (W_i \wedge B^t_i) \vee (W^t_i \wedge B^t_i) = 1$ when instrumenting a concrete memory module with such a B input.

b) Structural aspects: The practical memory instrumentation addresses the two main issues of the precise memory instrumentation that we identified: the need for dependent read and writes in a single clock cycle and the need to read or write the whole memory in a single cycle. As illustrated in Figure 2, the practical memory instrumentation is essentially made of two SRAMs, one for the memory data and one for the explicit taints, in addition to some simple logic described in Equation 9 and Equation 10. On the other hand, the precise implementation must be implemented as an enormous and complex interconnection of standard cells. Note that the practical memory instrumentation complies with gate-level instrumentation [8] principles (Section V-B-a) and with the replication-based principle for cell-level instrumentation [7].

c) Meta-reset: Often, memory modules do not feature a reset signal, but such a signal is required to unset τ. To solve this problem, we introduce a meta-reset similar to the RFUZZ implementation by K. Laeufer et al. [36]. It corresponds to a parallel reset of all memory modules in the design and is triggered by the reset signal of the design.

VI. HybriDIFT

Based on the new practical module-level memory instrumentation mechanism that we proposed in Section V, we

TABLE I
MEMORY INPUT SIGNAL DEFINITIONS FOR ALL ANALYZED DESIGNS COMBINED.

Name	Description
c_addr	Address for any operation.
r_addr	Address for reads. Incompatible with c_addr.
w_addr	Address for writes. Incompatible with c_addr.
b_en	Byte write enable.
en	Enable any operation.
w_en	Write enable. Supersedes en.
w_mask	Write mask. Similar to single-entry b_en.

introduce HybriDIFT, a practical automatic DIFT instrumentation for memories. HybriDIFT first automatically identifies memories as well as their interface protocols and then instruments them at module level or can alternatively rely on manual annotations. In this section, we first analyze memory integrations in multiple open-source designs. From this analysis, we understand their similarities and divergences. We then propose a two-step approach to automatically detect memories and determine their interface protocols in arbitrary designs. Finally, we explain the position of HybriDIFT in typical DIFT instrumentation workflows.

A. Analysis of Memory Integrations

We manually analyze the integration of memories in multiple open-source designs from diverse vendors: CVA6 [9] (OpenHWGroup), Rocket [23] and BOOM [24] (Chipyard [37]) and OpenC910 [10] (T-Head). Our analysis shows that the SRAMs are systematically integrated as a specific separate module. This complies with classical design practices, where memories are later instantiated as BRAMs (for FPGAs) or as SRAMs (for ASICs). We additionally observe that these modules systematically expose a single read data output port, a single write data input port, some clock signals and diverse types of enable signals. Table I summarizes all the encountered non-clock/reset signals in the memory modules of the analyzed designs. In terms of port dimensions, we observe that data width can be as small as a single bit, while all memories feature at least 4 words.

B. Static analysis

To identify memories, HybriDIFT starts with a static analysis. Since a memory module may behave differently depending on the Verilog parameters provided to it, we first enforce the parametrization. Concretely, HybriDIFT uses the Yosys synthesizer for this purpose. This pass creates a new module type for each module instance with different Verilog parameters. Then, HybriDIFT identifies memories by filtering modules as follows: (a) a memory module never contains a memory module as a submodule, (b) a memory module has a single output port, and an input port that has the same width as the output port, (c) a memory module has at least one clock signal and another single-bit signal for controlling reads and writes.

While this static analysis filters out the vast majority of modules, it is not sufficient as it may include false positives. Furthermore, a static look at the interface does not provide sufficient information about the role of each signal in the protocol, for example, if there are multiple single-bit inputs.

C. Dynamic analysis

To determine the functionality of each input signal and incidentally identify any potential false positive from the static approach, we introduce an automatic dynamic module analysis platform as part of HybriDIFT. HybriDIFT first isolates each module flagged by the static analysis from the rest of the design. Note that this module has parameters already enforced, as described in Section VI-B. HybriDIFT then generates an RTL simulation testbench tailored for the module interface signals. This simulation testbench is kept minimal for compatibility across RTL simulators, and essentially parses the descriptions of inputs for the module, and logs the module output valuations.

Once the module is ready for RTL simulation with a high-level interface indicating which wire will be supplied with which value at each cycle, the next task of HybriDIFT is to design a sequence of inputs to automatically discriminate the functionalities of each input signal. The approach that HybriDIFT adopts is to first list all candidate mappings of input wires to functionalities, such as write enable, with regards to the acceptable bit widths of each functionality. Then, HybriDIFT generates a short sequence of input values to the candidate role of each input wire, which consists in a short sequence of write and read transactions exerting the input wires with each candidate functionality. HybriDIFT observes the output of the module for each transaction and compares it with the sequence of expected outputs given the mapping of wires to functionalities. Alternatively, HybriDIFT can rely on manual annotations, which is a one-time process. Once the input protocol is identified, HybriDIFT instruments the memory module with the mechanism described in Section V.

VII. EVALUATION

In this section, we evaluate HybriDIFT in terms of scalability, performance, practicality and precision and compare it with the state of the art [7]. In Section VII-A, we quantify the scalability of HybriDIFT's practical memory instrumentation using microbenchmarks. In Section VII-B, we evaluate the performance of the memory module detection and protocol identification technique on CVA6 and OpenC910. In Section VII-C, we evaluate the build performance and instrumentation ability of HybriDIFT by instrumenting the designs and measuring the build time. In Section VII-D, we evaluate the simulation performance of the designs instrumented with HybriDIFT by measuring the average wallclock time per simulated clock cycle. Finally, in Section VII-E, we show no practical loss of precision by reproducing the experiments conducted by the state-of-the-art dynamic IFT mechanism [7].

Evaluation setting: We conducted the evaluations on a machine equipped with two AMD EPYC 7H12 processors at 2.6 GHz with a total of 256 logical cores and 1 TB of DRAM. We use Verilator 5.023 for the simulations, and

TABLE II
DESIGNS USED IN THE EVALUATION.

Design	Version	Commit
PULPissimo [38]	Hack@DAC'18 [39]	`d21d7b9` [7]
CVA6 [9]	5.0.1	`c51fad1`
Rocket [23]	1.6	`c45f449` (Chipyard [37])
BOOM [24]	3.0.0	`c45f449` (Chipyard [37])
OpenC910 [10]	N.A.	`e0c4ad8`

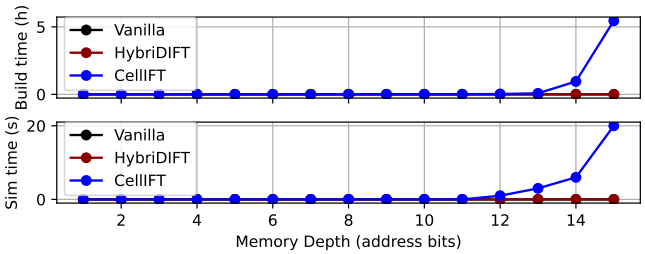

Fig. 3. Microbenchmark: scalability of the memory instrumentation. Vanilla represents the absence of instrumentation. The performance of Vanilla and HybriDIFT is very close hence indistinguishable.

Yosys 0.39 to embed the instrumentation passes. Regarding the hardware designs, we use the designs listed in Table II, in their default configuration. We additionally consider the larger available version of OpenC910, which contains memories of a depth of 64k words in the L2 cache. Note that since the memories were already manually replaced with models for instrumenting PULPissimo in [7] to be able to inject programs into the SoC memory, we do not include this design in the performance evaluations. We implemented HybriDIFT as 2 k lines of Python code and less than 500 lines of C++ code.

A. Scalability

To quantify the scalability of the practical memory instrumentation that we introduced, we measure the time required to build simulation models of 32-bit wide memories of various depths and to run simulations and collect the results in Figure 3. We do not include GLIFT [8] in the figure, as it scales significantly worse than CellIFT [7]. With CellIFT, the build performance significantly degrades for memories of depth larger than 2^{13} words, and the simulation performance significantly degrades for memories deeper than 2^{11} words. In the figure, HybriDIFT and the original non-instrumented design (vanilla) are superposed because HybriDIFT's memory instrumentation as two SRAMs and a single state bit are very efficient to simulate, essentially as two arrays.

B. Identifying the memory modules and protocols

We measure the time taken by HybriDIFT to identify the memory modules and protocols. The automatic memory identification always takes less than 2 minutes, and the memory sizes do not significantly impact the identification time. In comparison with the absolute build and elaboration durations reported in Section VII-C, the overhead is small.

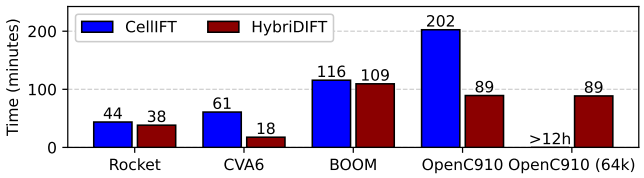

Fig. 4. Build performance for CellIFT and HybriDIFT. The last design is the OpenC910 core configured with 64kWord memories in the L2 cache.

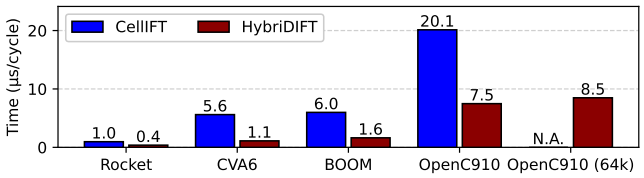

Fig. 5. Simulation performance measured by the average number of microseconds required to simulate one clock cycle. The last design is the OpenC910 core configured with 64kWord memories in the L2 cache.

C. Build performance

Build performance is critical with continuous integration and testing of hardware designs. We measure the time taken to build designs instrumented by CellIFT [7] and HybriDIFT and report the results in Figure 4. While CellIFT could not instrument the larger OpenC910 version in reasonable time, as expected from Section III-A-a, HybriDIFT could instrument it without a significant overhead over the default version. The build speedup provided by HybriDIFT over the state of the art on default design configurations varies from 1.06× for BOOM to 3.5× on CVA6.

D. Simulation performance

We measure the simulation performance of designs instrumented with the state of the art and with HybriDIFT by measuring the average time taken to simulate one clock cycle, over 1000 clock cycles when simulating the non-trivial RISC-V ELFs generated by a state-of-the-art CPU fuzz testing tool [5]. Figure 5 shows the results. The simulation speedup provided by HybriDIFT on default design configurations varies from 2.6× for Rocket to 5.1× on CVA6.

E. Reproduction of the practical experiments

To demonstrate that the precision of the DIFT instrumentation provided by HybriDIFT is sufficient, we reproduce the practical experiments conducted by CellIFT [7] on PULPissimo, CVA6, Rocket and BOOM. Regarding the detection of leakage-prone paths, we find the same components as reported in the original study. Intuitively, we note that when loading tainted data, we expect no memory address or enable signal to be tainted. We also reproduce the architectural bug detection experiments on PULPissimo. When the bugs are present, we indeed detect the bugs. Most importantly, when we fix the bugs, detection stops, which shows that also in this experiment, no false positive occurs. Finally, we also reproduce the microarchitectural vulnerability detection experiments using Spectre and Meltdown [19], [40] on BOOM.

Already clear from a theoretical viewpoint, we experimentally confirm that the precision of the DIFT instrumentation provided by HybriDIFT is sufficient for all these policies.

VIII. Related Work

This work is at the intersection between two disciplines: circuit reverse engineering and performance-precision trade-offs in hardware DIFT instrumentations.

a) Circuit Reverse Engineering: In the literature, a large effort in reverse engineering has been dedicated to understanding the behavior of FPGA bitstreams [41]–[43] or the logic implemented on chips [44]. Another direction is using reverse engineering to detect hardware trojans [45]–[48]. The reverse engineering of module-level netlists has so far been vastly unexplored [49] and generally focuses on the deobfuscation of encrypted logic circuits [50], [51] or finite state machines [52]. Subramanyan et al. [53] proposed a method to draw lines between modules and identify, among others, memories using a structural analysis of read and write logic in circuits that are already synthesized. HybriDIFT benefits from a higher-level design representation and can hence rely on module interfaces for its analysis. In particular, it can then adopt the approach based on dynamic analysis to determine the exact interface protocols of memories and confirm the nature of memories.

b) Precision-performance trade-offs in hardware DIFT: The precision of hardware DIFT mechanisms has been a topic of interest, starting with gate-level instrumentation [8]. RTLIFT exposes performance/precision trade-offs for a few HDL operators [18]. Hu et al. [29] propose a binary search on precision to find an optimal performance based on a specific information flow property. CellIFT [7] proposes a new instrumentation that improves the known performance/precision trade-offs at the cell level. By operating at the module level, HybriDIFT can provide a new trade-off between precision and performance in memory instrumentations, which is not possible at the lower abstraction levels in previous work.

IX. Discussion

We first discuss the structural assumptions made for automatizing HybriDIFT, then discuss manual memory and protocol annotations, and finally discuss the precision achieved by HybriDIFT's memory instrumentation.

a) Structural assumptions: The automatic memory and protocol identification technique is based on observations made on a wide range of open-source designs and on our experience with practical VLSI implementations. We cannot exclude that designs could use memories with vastly unusual interfaces, yet this would also imply different ways of integrating such memories. Standard-cell memories are a typical example of memories with potentially custom interfaces. Such memories can either be manually instrumented based on the practical memory instrumentation presented in Section V, or future work could extend the memory and protocol identification of HybriDIFT to recognize such memories and their protocols and instrument them accordingly.

b) Manual annotations: Given that the state-of-the-art DIFT mechanisms [7], [8], [18] do not rely on any manual annotations, it is beneficial that HybriDIFT can automatically recognize some memories and their protocols. However, in many use cases, it is easy and inexpensive to manually annotate memories. This allows simplifying the flow in practice and maximizes the predictability of the resulting instrumentation. As such, in scenarios where the design is well-understood by the user, manual annotations might be preferable to automatic identification.

c) Precision sufficiency: Our study of existing applications of hardware DIFT shows that precise tracking of implicit flows is currently unnecessary. Hence, HybriDIFT instruments memories at module level to conservatively approximate these specific flows that are extremely expensive to track precisely when memories contain many words.

One potential use case of precise tracking of implicit flows in memories would be to dynamically test the security of software on specific hardware, where the software isolates several tenants of the same SRAM. In that case, for example, the upper bits of an address could determine the tenant, and the lower bits the location within the tenant's memory. In this example, a tenant's isolation would only be compromised if the upper address bits are tainted. On the other hand, writing data when only the lower address bits are tainted cannot affect other tenants. This distinction is not captured by HybriDIFT, but it is possible to split the memory into several SRAMs, or τ could easily be refined to improve the precision of the implicit flow tracking up the requirements of the use case.

X. Conclusion

We performed a formal analysis of precise DIFT in memories and discovered fundamental aspects that make scaling the precise tracking of implicit flows in memories challenging. Fortunately, our study of the existing DIFT applications shows that none of them requires precise tracking of implicit flows in memories. Hence, we introduced a new module-level memory DIFT, called HybriDIFT, that approximates the information flows coming from the address and enable signals, while tracking explicit flows precisely. Evaluation using four popular RISC-V designs shows that HybriDIFT accelerates build time by $1.06\times$ to $3.5\times$ and simulation performance by $2.6\times$ to $5.1\times$ on default target configurations, while maintaining sufficient precision for all existing hardware DIFT applications. We show that HybriDIFT can instrument a larger OpenC910 configuration that was unattainable by the state of the art.

Acknowledgements

The authors would like to thank Katharina Ceesay-Seitz and the anonymous reviewers for their valuable feedback. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

REFERENCES

[1] F. Solt, P. Jattke, and K. Razavi, "Rememberr: Leveraging microprocessor errata for design testing and validation," in *MICRO*, 2022.

[2] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, and C. Wang, "Morfuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation," in *USENIX Security*, 2023.

[3] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "Thehuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," in *USENIX Security*, 2022.

[4] 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.

[5] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: Cpu fuzzing via intricate program generation," *USENIX Security*, 2024.

[6] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi, "Processorfuzz: Processor fuzzing with control and status registers guidance," in *HOST*, 2023.

[7] F. Solt, B. Gras, and K. Razavi, "Cellift: Leveraging cells for scalable and precise dynamic information flow tracking in rtl," in *USENIX Security*, 2022.

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

[9] 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.

[10] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen, *et al.*, "Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance risc-v processor with vector extension: Industrial product," in *ISCA*, 2020.

[11] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in i2c and usb," in *DAC*, 2011.

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

[13] J. Oberg, S. Meiklejohn, T. Sherwood, and R. Kastner, "Leveraging gate-level properties to identify hardware timing channels," *TCAD*, 2014.

[14] W. Hu, A. Ardeshiricham, M. S. Gobulukoglu, X. Wang, and R. Kastner, "Property specific information flow analysis for hardware security verification," in *ICCAD*, 2018.

[15] W. Hu, D. Mu, J. Oberg, B. Mao, M. Tiwari, T. Sherwood, and R. Kastner, "Gate-level information flow tracking for security lattices," *TODAES*, 2014.

[16] 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.

[17] W. Hu, J. Oberg, J. Barrientos, D. Mu, and R. Kastner, "Expanding gate level information flow tracking for multilevel security," *IEEE Embedded Systems Letters*, 2013.

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

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

[20] I. C. Society, "Ieee standard for verilog hardware description language," *IEEE*, 2006.

[21] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Austrochip*, 2013.

[22] A. Ardeshiricham, W. Hu, and R. Kastner, "Clepsydra: Modeling timing flows in hardware designs," in *ICCAD*, 2017.

[23] 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.

[24] K. Asanovic, D. A. Patterson, and C. Celio, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," tech. rep., UC Berkeley, 2015.

[25] A. Teman, D. Rossi, P. Meinerzhagen, L. Benini, and A. Burg, "Power, area, and performance optimization of standard cell memory arrays through controlled placement," *TODAES*, 2016.

[26] P. Meinerzhagen, C. Roth, and A. Burg, "Towards generic low-power area-efficient standard cell based memory architectures," in *MWSCAS*, 2010.

[27] P. Meinerzhagen, S. Y. Sherazi, A. Burg, and J. N. Rodrigues, "Benchmarking of standard-cell based memories in the sub-v_t domain in 65-nm cmos technology," *IEEE CASS*, 2011.

[28] M. Marazzi, P. Jattke, F. Solt, and K. Razavi, "Protrr: Principled yet optimal in-dram target row refresh," in *IEEE SP*, 2022.

[29] W. Hu, B. Mao, J. Oberg, and R. Kastner, "Detecting hardware trojans with gate-level information-flow tracking," *Computer*, 2016.

[30] J. Oberg, S. Meiklejohn, T. Sherwood, and R. Kastner, "A practical testing framework for isolating hardware timing channels," in *DATE*, 2013.

[31] J. Shin, H. Zhang, J. Lee, I. Heo, Y.-Y. Chen, R. Lee, and Y. Paek, "A hardware-based technique for efficient implicit information flow tracking," in *ICCAD*, 2016.

[32] Y. Liu and A. Milanova, "Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows," in *CSMR*, 2010.

[33] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *ICISS*, 2008.

[34] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *TOCS*, 2014.

[35] A. Russo, A. Sabelfeld, and K. Li, "Implicit flows in malicious and non-malicious code," in *Logics and Languages for Reliability and Security*, 2010.

[36] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *ICCAD*, 2018.

[37] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *MICRO*, 2020.

[38] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini, "Quentin: an ultra-low-power pulpissimo soc in 22nm fdx," in *S3S*, 2018.

[39] 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 Security*, 2019.

[40] 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.

[41] R. V. Narayanan, A. N. Venkatesan, K. Pula, S. Muthukumaran, and R. Vemuri, "Reverse engineering word-level models from look-up table netlists," in *ISQED*, 2023.

[42] A. Nathamuni-Venkatesan, R.-V. Narayanan, K. Pula, S. Muthukumaran, and R. Vemuri, "Word-level structure identification in fpga designs using cell proximity information," in *VLSID*, 2023.

[43] S. Muthukumaran, A. N. Venkatesan, K. Pula, R. V. Narayanan, R. Vemuri, and J. Emmert, "Reverse engineering of rtl controllers from look-up table netlists," in *ISVLSI*, 2023.

[44] S. E. Quadir, J. Chen, D. Forte, N. Asadizanjani, S. Shahbazmohamadi, L. Wang, J. Chandy, and M. Tehranipoor, "A survey on chip to system reverse engineering," *JETC*, 2016.

[45] T. Zhang, J. Wang, and Z. Chen, "A reverse engineering-based framework assisting hardware trojan detection for encrypted ips," in *IMCCC*, 2018.

[46] W. Danesh, J. Banago, and M. Rahman, "Turning the table: Using bitstream reverse engineering to detect fpga trojans," *Journal of Hardware and Systems Security*, 2021.

[47] S. Wallat, M. Fyrbiak, M. Schlögel, and C. Paar, "A look at the dark side of hardware reverse engineering-a case study," in *IVSW*, 2017.

[48] M. Ludwig, A.-C. Bette, and B. Lippmann, "Vital: Verifying trojan-free physical layouts through hardware reverse engineering," in *PAINE*, 2021.

[49] M. Fyrbiak, S. Strauß, C. Kison, S. Wallat, M. Elson, N. Rummel, and C. Paar, "Hardware reverse engineering: Overview and open challenges," in *IVSW*, 2017.

[50] R. Tofighi-Shirazi, M. Christofi, P. Elbaz-Vincent, and T.-H. Le, "Dose: Deobfuscation based on semantic equivalence," in *SSPREW*, 2018.

[51] F. Farahmandi, O. Sinanoglu, R. Blanton, and S. Pagliarini, "Design obfuscation versus test," in *ETS*, 2020.

[52] T. Meade, S. Zhang, and Y. Jin, "Netlist reverse engineering for high-level functionality reconstruction," in *ASP*, 2016.

[53] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse engineering digital circuits using functional analysis," in *DATE*, 2013.