Register Transfer Level Information Flow Tracking for Provably Secure Hardware Design

Armaiti Ardeshiricham, Wei Hu, Joshua Marxen and Ryan Kastner
Dept. of Computer Science and Engineering, University of California, San Diego

***Abstract*—Information Flow Tracking (IFT) provides a formal methodology for modeling and reasoning about security properties related to integrity, confidentiality, and logical side channel. Recently, IFT has been employed for secure hardware design and verification. However, existing hardware IFT techniques either require designers to rewrite their hardware specifications in a new language or do not scale to large designs due to a low level of abstraction. In this work, we propose Register Transfer Level IFT (RTLIFT), which enables verification of security properties in an early design phase, at a higher level of abstraction, and directly on RTL code. The proposed method enables a precise understanding of all logical flows through RTL design and allows various tradeoffs in IFT precision. We show that RTLIFT achieves over 5× speedup in verification performance as compared to gate level IFT while minimizing the required effort for the designer to verify security properties on RTL designs.**

I. Introduction

Despite decades of research on software and hardware security, today's computing technology remains vulnerable to classic exploit techniques due to the fact that many fundamental computational models and traditional design flows do not consider or prioritize security. In the realm of hardware, security is rarely considered during design space exploration. Security vulnerabilities can originate from design flaws, which can be fully eliminated after a complete verification. Unfortunately, this is impractical due to the scale of modern chips. Furthermore, novel attack vectors like side-channel analysis undermine classic assumptions about the accessibility of internal secret information outside of a computing system. In addition, hardware designs often required incorporating third party IP cores, which may contain undocumented malicious design modifications known as backdoor.

Remedying these security vulnerabilities is best accomplished with a systemic solution. One such solution that has shown promise is information flow tracking (IFT). IFT models how labeled data moves through a system. It provides an approach for verifying that systems adhere to security policies, either by static verification during the design phase or dynamical checking at runtime. Recent work has demonstrated the effectiveness of hardware IFT in identifying and mitigating hardware security vulnerabilities, such as timing channels in cryptographic cores and caches, unintended interference between IP components of different trust, and information leakage through hardware Trojans [7]–[12].

Hardware IFT techniques have been deployed at different levels of abstraction. At the gate level, all logical information flows can be tracked by augmenting each logic primitive in the synthesized design netlist with additional IFT logic. While this simplifies IFT logic generation by breaking down complex language structures to lower level logic constructs, this method does not scale with design size since it relies on gate level verification. Language level methods avoid this problem by generating IFT logic at a higher level of abstraction. Previous language level IFT techniques are accomplished via designing type enforced HDL (Hardware Description Language). These techniques require that designs be rewritten or annotated in a new language in order to verify security properties, which can be a challenging task for hardware designers.

The level of abstraction can also affect the precision of the IFT logic. More specifically, the precision of IFT logic is determined by both the precision of the label propagation rules for logic operations and the granularity of the building blocks over which IFT is deployed. Gate level IFT methods [7] achieve increased precision by defining precise tracking rules for a set of universal gates. On the other hand, gate level methods apply IFT at a fine granularity, and thus cannot detect higher-level dependencies (e.g., variable correlation due to reconvergent fanout) between the signals in the design.

In this work, we propose a Register Transfer Level IFT (RTLIFT) method, which allows a precise understanding of all logical information flows through RTL code. By defining precise label propagation rules for RTL expressions, we show that our method can achieve a higher level of precision as compared to gate level methods. Our IFT model is completely described with standard RTL syntax and thus eliminates the need for acquiring a new type-enforced language. Furthermore, we discuss how RTLIFT allows for separation of implicit and explicit flows, enabling various tradeoff of IFT precision, which cannot be realized using existing IFT methods. Specifically, this paper makes the following contributions:

- Proposing a method for precisely understanding all logical information flows through RTL design;
- Providing techniques that allow trading-off IFT precision and security verification performance;
- Presenting experimental results to show the improvement in IFT precision and security verification time.

The remainder of this paper is organized as follows: Section II summarizes existing IFT methods and discusses how our work differs from them. Section III deliberates the concept of precise IFT and what RTLIFT aims to improve in this work. Section IV describes the implementation details. The experimental results are reported in Section V. We conclude in Section VI.

II. Related Work

Software-mediated IFT, e.g., LIFT [1] and RIFLE [2], use dynamic binary translation to track flows during execution.

The overhead of these methods can be prohibitive, and thus motivated the use of custom architectural modifications to facilitate faster information flow tracking. Such schemes generally fall into one of two categories – integrated pervasive processor modifications (e.g., Minos [3] and Raksha [4]) and modular core additions or coprocessor support (e.g., Flexitaint [5] and Kannan et al. [6]).

Gate level information flow tracking (GLIFT) [7] performs IFT analysis directly on the "original" hardware design. It does this by creating a separate GLIFT analysis logic that is derived from the original logic, but operates independently from it. GLIFT tracks any arbitrary set of flows by labeling different hardware variables as "tainted", and tracking their effect throughout the design. The GLIFT logic is generated once, independent of the security property, and can be used to verify any IFT property. GLIFT is primarily used at design time for testing and verification [8].

Caisson and Sapper are hardware security design languages that directly generate circuits that enforce the desired IFT properties (e.g., separation and isolation). Both add a typing system to a finite state machine (FSM) language which requires that the designer assigns a security label to each register and wire. Caisson [9] uses static types forcing it to conservatively perform replication to restrict flows of information. This can lead to significant increases in logic. Sapper [10] improves Caisson by adding a dynamic type system. This reduces the need for logic replication, but still requires that the designer learn a new language. Sapper and Caisson enforce information flow by restricting transitions between the states. Hence, the user must redesign the hardware using a new language which can be a nontrivial task.

VeriCoq-IFT [11] automatically converts designs from their HDL representation to the Coq formal language, eliminating the need to redesign the hardware. However, the user still needs to annotate the generated Coq code in order to analyze security properties. Furthermore, in all these three methods the flow of information is tracked conservatively since the label propagation rules are defined as updating label of the output of any operation to the highest label of its inputs. As we discuss in this work, this approach overestimates the flow of information by ignoring the functionality of the operation and the exact values of the operands.

SecVerilog [12] extends the Verilog language with an expressive type system. SecVerilog users are required to explicitly add a security label to each variable in the code. These labels are a consequence of the security property that one wishes to verify on the design. It uses a type system to ensure that the specified information flow policy is upheld. Being a thoroughly static tool, SecVerilog uses predicate analysis in order to acquire the hardware state essential for precise flow tracking. This complicates the labeling processing, and inevitably the intricacy of precise predicate analysis leads to loss of precision compared to simulation-based and dynamic approaches [13]. Furthermore, the designer must specify many of the flow rules when she adds labels to the variables. Ideally, this process would be automated, e.g,. as done with GLIFT, otherwise it impedes its use as a hardware security design tool. Often there are many (potentially hundreds or thousands) of IFT properties that one wishes to test or verify on a single design. This would require the user to relabel the design in order to prove each different property. For example, in a cryptographic core proving that the secret key value does not affect the timing of the output signals requires a different labeling from proving that no inputs except the key and the plain text can affect the value of the cipher text.

RTLIFT creates a new methodology that combines the benefits of these previous approaches while eliminating their drawbacks. RTLIFT works directly with existing HDL languages, and thus does not require a designer to learn a new hardware security language. Much like GLIFT, it automatically defines flow relation properties. Yet working at a higher level of abstraction leads to many benefits including faster verification time and more flexibility in defining different types of flow relationships (e.g., implicit versus explicit).

III. Background and Motivation

A. IFT Basics

Information flows from signal X to signal Y if and only if a change in the value of X can influence the value of Y. Information flow can model security properties related to both confidentiality and integrity:

- **Confidentiality:** Assume that X is a secret value while Y is publicly observable. In this case, an attacker can extract sensitive information by observing and analyzing the variations in signal Y. For example, Y could be the "ready" output of a cryptographic core which in a secure design should not depend on the value of the private key stored in X; otherwise there is a timing side channel which can be used to extract the secret key. In this case, we want to insure the property that X does not flow to Y.
- **Integrity:** Assume that X is an untrusted value while Y is trusted. In this scenario, we wish to insure that an attacker cannot gain unauthorized access through Y by modifying the value of X. For example, X may be an openly accessible memory location and we wish to insure that it cannot be used to influence the results of a system control register. Thus, we want to insure again that X cannot flow to Y.

Hardware information flow tracking generally works by adding a security label to each signal, and using that to track the influence of flow (or *taint*) of a set of signals throughout the circuit. The initial taint is set based upon the desired security property, and IFT techniques are used to test or verify whether that taint can move to an unwanted part of the system as specified by the security property.

IFT can be done with various levels of precision. One approach marks the output of each operation as tainted when any of its inputs is tainted. While simple, this method is overly conservative and can inaccurately report existence of flow in certain cases (i.e., *false positives*). This inaccuracy is due to the fact that based on the functionality of the operation, a single untainted input can dominate the output, yielding an untainted output while other inputs are tainted. To avoid this imprecision, the tracking rules need to take into account both the type of the operation and the exact state of the hardware.

To clarify this idea, consider the expression `out = secret & 0x0F`, where sensitive information is stored in an 8-bit variable `secret` and we want to determine if the information from `secret` flows to the variable `out`. The most conservative and least precise approach would mark all bits of `out` as tainted since `secret` is tainted. A more precise strategy gives us slightly different answer: the secret information only flows to the four least significant bits of `out`, and the other bits should not be marked as tainted since their values are zero regardless of the value of `secret`. To achieve this level of precision, separate tracking rules for different operations shall be defined as we discuss in Section IV-A.

B. IFT Precision-Complexity Trade-offs

Precise tracking rules impose more complexity, and hence it might be desirable to deliberately add some false positives to the IFT logic by taking a rather conservative approach. In large designs it might be beneficial to use imprecise and efficient approaches to track the flow through complex arithmetic operations, while preserving the precision for logical operations such as AND, OR, XOR, etc. As opposed to the gate level where the difference between logical and arithmetic operations is lost, considering the high level description of the design, one can define the tracking rules for various operations with different levels of precision and more flexibility.

The notion of taking various levels of precision based on the functionality becomes more important when extended to the different information flow paths in the design: the data flow and the control flow. The data flow represents how the information explicitly flows, while the control flow shows all the paths that might be traversed and hence contains information regarding the implicit flow. For example, in a conditional statement the flow from the right hand side expression to the left hand side variable is explicit and the logic implementing it is within the data path. However, the value of the left hand side variable is also implicitly affected by the conditional variable which is represented in the control path. Implicit and explicit flows are not distinguishable in the gate level netlist. However, we can differentiate between them at the language level. We exploit this idea in order to adjust the complexity of the tracking logic based on the verification objective. Specifically, when searching for timing flows, which are caused by implicit flow, keeping the tracking logic associated with the data path imprecise, – hence reducing the logic complexity – and implementing the control flow's tracking logic precisely, we can realize a smaller tracking logic which does not impose additional false positives for tracking the implicit flow.

C. IFT Precision

In this section we discuss how generating the IFT logic at a higher level of abstraction can improve its precision level. Gate level IFT techniques necessitate synthesizing the design to its gate level netlist before generating the IFT logic. Resource sharing done by the synthesis tool introduces reconvergent paths to the netlist which are not present at the language level. Reconvergent paths lead to false positives in the tracking logic since the tracking rules cannot easily take into account the exact relationship between multiple inputs of an operation. To clarify the source of such imprecision, we deliberate the gate level and high level tracking rules for a 2:1 multiplexer.

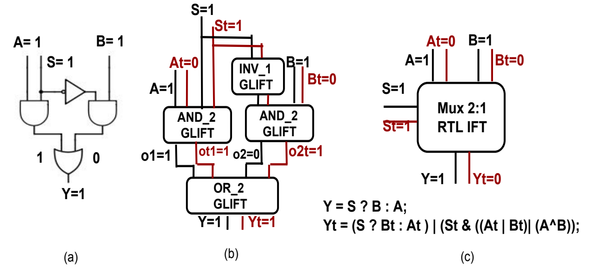

Fig. 1: Different IFT logic for 2:1 mux. (a) Gate level structure of a mux. (b) IFT logic generated using GLIFT. (c) Precise IFT tracking logic for a 2:1 mux. RTLIFT can use either of these (or more) as an IFT library element.

Fig. 1(a) shows the gate level structure of a multiplexer and Fig. 1(b) represents its precise gate level tracking logic. Even though the flow is precisely tracked through each single gate, when combined together the multiplexer's IFT logic contains false positives. To examine when a false positive happens, we analyze the case where both data inputs A and B are one, and their security labels At and Bt are zero, indicating being untainted. The control signal S and its security label St are both equal to one, indicating being tainted. Analyzing the gate level tracking logic, the output of both AND gates have high security labels while only one of them is on. Consequently, both inputs to the OR gate are tainted resulting in a tainted output. Conceptually, the output of the OR gate is marked as tainted because flipping the value of its high input will change its result to zero. However, the missing part is that this flip cannot happen unaccompanied by a flip on the other input, which forces the final output to remain the same. This imprecision, in the presence of precise tracking logic for all the gates, occurs due to the reconvergence path at the input of the OR gate. The false positives provoked by the reconvergnece paths at the gate level can be avoided by generating the tracking logic at a higher level as we can see in Fig. 1(c).

Reconvegence paths also exist at the language level, which inevitably results in false positives. We can define precise tracking rules for gates or language constructs; however, this precision is based on the independency of the inputs. By generating the tracking logic at a higher level of abstraction and hence utilizing higher level tracking rules, we can overcome the dependency between the intermediate variables which improves the precision level. Fundamentally, precise information flow tracking is an undecidable problem as shown by Denning and Denning [14]. Nonetheless, we improve IFT precision level in two ways: First, we have precise tracking rules specifically defined for each operation which take into account the exact state of hardware; second, we avoid a large class of false positives caused by the reconvergent paths in the gate level netlist by analyzing the design from a higher level of abstraction.

IV. IMPLEMENTATION

In this section we elaborate details of RTLIFT implementation. RTLIFT software receives a synthesizable Verilog code along with flags specifying the precision level of the data flow IFT logic and the control flow IFT logic, and

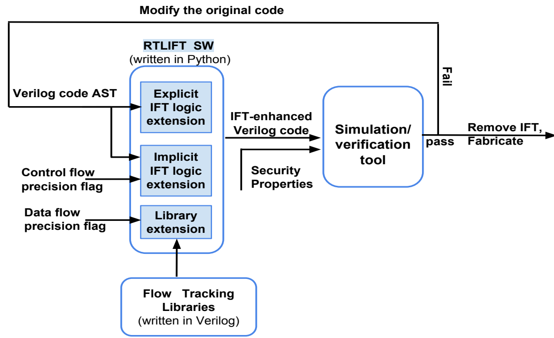

Fig. 2: RTLIFT overview

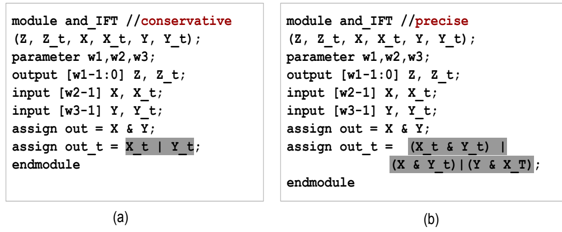

Fig. 3: Flow tracking libraries

generates functionally equivalent Verilog code instrumented with information flow tracking logic. Since the generated code is synthesizable, it can be analyzed by standard EDA test and verification tools allowing us to leverage decades of research on functional testing to assess security properties of hardware designs. If the IFT-enhaced design passes the security properties the original code can be used for fabrication. Otherwise, the original code should be modified and analyzed again. Fig. 2 gives an overview of how the tool is used. RTLIFT is realized through the following steps: Designing flow tracking libraries; enhancing the combinational circuit with tracking logic; enhancing the conditional statements with logic required for tracking the implicit flow. We describe each of these steps in the rest of this section.

A. Flow Tracking Libraries

For tracking the flow of information through an RTL code, each operation should be instrumented so it can operate both on the Boolean values and security labels of the operands. Hence, for each operation `OP` such that `Z = X OP Y` is a valid statement in Verilog, we have defined a module `OP_IFT` which receives inputs `X` and `Y` along with their security labels `X_t` and `Y_t`, and generates the output `Z` along with its security label `Z_t`. These modules are predefined and given to the RTLIFT software as an input file called "flow tracking libraries", as shown in Fig. 2. These libraries serve two goals: first, improving IFT precision by enabling operation-specific label propagation as opposed to the tracking rules in Caisson [9], Sapper [10] and VeriCoq-IFT [11]; second, automating the computation of security labels in contrast to the approach taken in SecVerilog [12].

We have defined two different sets of libraries, each of which calculating `Z_t` output with a different level of precision. In the *conservative* library, the label propagation rules overestimate the existence of flow by marking the output of each operation as tainted when any of its inputs are tainted, yielding a small tracking logic modeled with an OR expression. In the *precise* library, the label propagation rules are designed to minimize the number of false positives through each operation in exchange for a more complex tracking logic. The user selects which library should be used by specifying the precision flag for the data path logic. Other libraries with various precision-complexity balances can be added if required. Fig. 3(a) and (b) shows the IFT-enhanced modules for the AND operation available in the *conservative* and *precise* libraries respectively. In Section V we explain the design of IFT-enhanced modules for arithmetic operations in more detail.

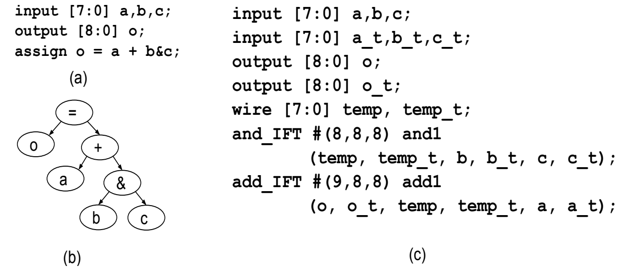

Fig. 4: Explicit flow tracking. (a) sample Verilog code. (b) Data Flow Graph of the code. (c) IFT-enhanced Verilog code.

B. Tracking Explicit Flows

Flow tracking starts by extending each bit of data, i.e., wires and registers in a given Verilog code, with a label that carries out information regarding the security properties of the data (Lines 1-3 in Algorithm 1). In this work, we consider a single bit label, where a high value indicates either secret or untrusted value, depending on whether we want to verify confidentiality or integrity IFT properties. To obtain a smaller IFT logic and further speed up the verification, it is possible to make a label for a multi-bit variable; this is the power of the library based approach.

After extending the variables with security labels, we replace every HDL operation with an IFT-enhanced operation as described earlier. To do so, we examine the node of each assignment statement via in-order traversal. The data flow graph is acquired using *Yosys Verilog frontend* [17] to transform Verilog code to its Abstract Syntax Tree (AST) representation. For each operation, a module from the available libraries is instantiated (Lines 4-13 in Algorithm 1). This process is shown for a simple code in Fig. 4. For sequential circuits modeled as *always blocks* in Verilog language, the same approach is taken by calculating the flow of the right hand side expression outside the *always block* and updating the label of the left hand side variable using the original *always block* structure.

C. Tracking Implicit Flows

Merely tracking the explicit flow might inaccurately report the absence of flow in conditional statements by ignoring existence of implicit flows. To track these flows, for each assignment we obtain a list of variables which affect the execution of the statement. Having this list, we generate the logic required for tracking the implicit flow as shown by Algorithm 2. This logic can be generated with different levels of precision specified by "$flag_im$". If we wish to employ

```
input : verilog code AST file, IFT libraries
input : flag_im, flag_ex
output: IFT instrumented verilog code
1  for each variable V[n : 0] do
2  |   define new variable V_t[n : 0];
3  end
4  for each assignment Ai: Vk ::= Exp(Xi, ···, Xj) do
5  |   Traverse the DFG of Ai in order;
6  |   for each operation OP: Yp = Xm OP Xn ∈ Exp do
7  |   |   instantiate module
   |   |   OP_IFT (Xm, Xm_t, Xn, Xn_t, Yp, Yp_t);
8  |   |   if (OP == DFG.root) then
9  |   |   |   Vk ::= Yp ;
10 |   |   |   Vk_t ::= Yp_t | Imp_Flow(Ai, flag_im);
11 |   |   end
12 |   end
13 end
```

Algorithm 1: IFT Logic Generation

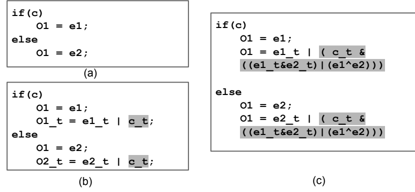

Fig. 5: Implicit flow tracking. (a) Verilog code. (b) Imprecise IFT-extended Verilog code. (c) Precise IFT-extended Verilog code. Highlighted parts show the tracking logic for implicit flow tracking.

a conservative IFT approach, any use of tainted condition should yield a tainted output (Lines 5-7 in Algorithm 2). While this approach captures all possible flows of information, it overestimates the actual flow. For a more precise flow tracking, we need to traverse the control flow graph in order to figure out what other outcomes are possible for the right hand side of the assignment, assuming the conditions were flipped. Now we can model each conditional statement with a multiplexer and acquire the taint of the output using the high level flow tracking rule for multiplexer (Lines 7-10 in Algorithm 2). To better understand the idea, we analyze implicit flow tracking through a simple code shown in Fig. 5(a). The highlighted parts in Fig. 5(b) and (c) show the logic added to track the implicit flow while `e1_t` and `e2_t` represent the explicit flows from the right hand side expressions `e1` and `e2`. As it can be seen in Fig. 5(b), the imprecise approach marks the output of a conditional statement as tainted if the condition is tainted by employing an OR logic. Taking a precise approach, information flows from the condition to the output only if the tainted condition occurs when both inputs are tainted or they have different Boolean values.

V. Experimental Results

We have used RTLIFT to analyze security properties on several benchmarks, and here we compare RTLIFT and GLIFT in terms of required time for verifying security policies and the precision of the generated IFT logic.

A. Security Proofs

1) Cryptographic Cores: Table I shows the required time for proving IFT properties on cryptographic cores. As depicted in Fig. 2, we have used RTLIFT to generate the IFT logic for the design under test, which is then given to Quetsa Formal

```
input : assignment Ai: Vk ::= Exp(Xi, ···, Xj)
input : flag_im, verilog code AST file
output: Verilog expression ImpFlow
1  Extract Vk_Conds by traversing CFG of Ai;
2  if (Vk_Conds==Null) then
3  |   ImpFlow ::= 0;
4  else
5  |   if (flag_im==Imprecise) then
6  |   |   ImpFlow ::= ImpFlow | ci_t; ∀ ci ∈ Vk_Conds
7  |   else
8  |   |   Extract OtherInps by traversing the CFG;
9  |   |   ImpFlow ::= Mux_IFT(Vk_Conds,OtherInps);
10 |   end
11 end
12 return ImpFlow;
```

Algorithm 2: Implicit flow tracking logic generation

Verification tool. To inspect if information flows from input *X* to output *Y*, we need to set *X*'s label high while all other inputs' labels are low, and observe Y's label which tells of if information can flow from X to Y or if they are isolated from each other. We have proved two properties on 32 bit and 128 bit RSA cores: 1) flow from the secret key to the cipher text and 2) flow from secret key to "ready" signal. While the former is expected as it is secured through encryption, the latter reveals an unintended flow. Since the Boolean value of the " ready" signal is not affected by the key value, the detected flow reveals a timing channel. The timeout is set for one hour for this experiments.

Furthermore, we have used the tool to check confidentiality properties on a number of trust-HUB AES benchmarks that contain hardware Trojans which leaks the secret key to an output other that the cipher text. IFT techniques can be used to detect hardware Trojans that cause unintended flows of information [15]. Specifically, in a cryptographic core information from the secret key should only flow to the cipher text, and its flow to any other output is undesirable. Hence, we have used the tool to specify if there is a flow from the secret key to any output besides the cipher text. Our method is capable of detecting such hardware Trojans while considerably reducing the verification time compared to GLIFT (taken from reference [15]), as reported in Table I.

2) WISHBONE: IFT can be used to detect timing flows in SoC benchmarks [16]. Here, we have used RTLIFT to inspect timing flows between cores that are connected together via the WISHBONE bus architecture. WISHBONE is a relatively simple protocol developed by the Opencores community [18], and allows multiple devices to interact with each other by sharing a bus. The transaction starts by a master core requesting access from a device by asserting its "cyc" signal. If the slave device is idle, access is granted to the master by setting its "ack" signal. We want to indicate if a certain master's "ack" signal is affected by the requests coming from other masters. To test this, we assume one of the master cores, $m1$, to be untrusted by setting its "cyc_t" signal high. Next, we observe "ack_t" signal from one of the trusted masters, $m2$. "ack_t" being high indicates a timing flow since we have not marked data values as tainted and $m1$ requests are affecting the time that $m2$ can start and finish its computation. This timing flow is a threat to system integrity since it can violate the real-time constraint of the master cores.

TABLE I: Verification Time.

Benchamrk	property	RTLIFT	GLIFT
32bit RSA	Key flows to cipher text	02:38	10:08
32bit RSA	Key flows to ready	02:14	10:17
128bit RSA	Key flows to cipher text	9:43	timeout
128bit RSA	Key flows to ready	9:36	timeout
AES_T100	Key leaks to output	01:05	06:48
AES_T1000	Key leaks to output	01:06	6:49
AES_T1100	Key leaks to output	01:07	6:46
AES_T1200	Key leaks to output	01:06	6:50

We generated both the conservative and precise IFT logic for comparison. As discussed throughout the paper, the conservative IFT overestimates the existence of information flow resulting in false positives. Our approach is to start the verification process by the conservative IFT which is smaller in terms of area. If isolation can be proved using the conservative IFT, there is no need to verify the properties on the precise version. However, if flow is detected using the conservative approach, we need to repeat the experiments using the precise IFT to avoid getting false positives.

For the original WISHBONE architecture with round robin arbiter, both conservative and precise IFT indicate existence of flow. Next, we have modified WISHBONE arbiter to enforce timing isolation. In our first model, we have implemented a TDMA arbiter. Here, the conservative IFT can prove timing isolation, eliminating the need to test the precise IFT. In our second model, we have divided the masters to two groups with time multiplexed access between the groups and round robin within each group. In this scenario, the conservative IFT reports existence of flow between the two groups, while using the precise IFT we can prove isolation. This final example shows the importance of precision of IFT logic for reducing false positives.

B. Precision Analysis

We have compared the precision and complexity of the IFT logic generated by RTLIFT and GLIFT for data path operations addition and multiplication, and control path logic modeled as *case statements* in Verilog language. The precision is measured by comparing the number of tainted outputs during simulation for 2^{20} random input samples. As shown by table II, high level tracking rules result in less tainted flow. False positive percentage is reported as the ratio of the difference in the number of tainted flows to the total number of simulations. The complexity is reported as IFT logic area, which gives a first order estimate on testing and verification time.

We briefly explain how IFT-enhanced addition and multiplication operations are designed for the flow tracking library which is given to RTLIFT as an input file. First we have designed a full adder which receives three inputs `A`, `B` and `Cin` along with their labels `A_t`, `B_t` and `Cin_t` and generates outputs `Sum` and `Co` along with their labels `Sum_t` and `Co_t`. To find Boolean expression describing `Sum_t` and `Co_t`, we need to consider Boolean expressions of `Sum` and `Co` and find the circumstances under which the output can be flipped. Based on the Boolean equation $\texttt{Sum} = \texttt{A} \oplus \texttt{B} \oplus \texttt{Cin}$, the output Sum is tainted when any of the inputs are tainted since each input to an XOR operation can control the output. The `Co` output is high when more than two inputs are high. Hence

TABLE II: Precision & complexity of RTLIFT vs. GLIFT.

Operation	RTLIFT		GLIFT		
	#tainted flows	Area	#tainted flows	Area	%FP
8-bit adder	8477103	271	8535900	222	5.6%
16-bit adder	16441632	603	16524855	556	7.9 %
32-bit adder	32385907	1215	32549597	1243	15.6%
8-bit multiplier	15029971	847	15310281	1759	26.7%
16-bit multiplier	31816947	2078	32200870?	7647	36.6%
4-way case	849874	70	883810	54	3.2%
8-way case	869915	226	958070	129	8.4%
16-way case	869799	199	997874	289	12.2%

the value of `Co` can be changed if we have control over more than one of the inputs, or we have control over only one input but the other two inputs are not equal. Next, we have employed the IF-enhanced full adder to design a ripple carry adder, and then an IFT-enhance multiplier is built from the adder. As it can be seen from table II, generating IFT logic at a higher level of abstraction can reduce false positives rate for both data path and control path unit.

VI. Conclusion

This paper presents RTLIFT for precisely measuring all digital flows through RTL designs in order to formally prove security properties related to integrity, confidentiality and logic side channels. RTLIFT can be directly applied on HDL codes and easily integrated into the hardware design flow through automated IFT logic augmentation. Furthermore, it enables designers to tradeoff between the complexity and precision of the IFT logic for data path elements and control path logic separately allowing for fast property-specific verification. Experimental results show that generating the IFT logic at a higher level of abstraction can increase the IFT precision and improve the performance of security verification.

References

[1] Qin, Feng, et al. "Lift: A low-overhead practical information flow tracking system for detecting security attacks." *MICRO, 39th Annual International Symposium on.*IEEE, 2006.

[2] Vachharajani, Neil, et al. "RIFLE: An architectural framework for user-centric information-flow security." *MICRO, 37th International Symposium.* IEEE, 2004.

[3] Crandall JR, Chong FT. "Minos: Control data attack prevention orthogonal to memory model." *MICRO, 37th International Symposium.* IEEE, 2004.

[4] Dalton, Michael, et al. "Raksha: a flexible information flow architecture for software security." *ACM SIGARCH Computer Architecture News.* ACM, 2007.

[5] Venkataramani, Guru, et al. "Flexitaint: A programmable accelerator for dynamic taint propagation." *High Performance Computer Architecture, HPCA, IEEE 14th International Symposium on.*, 2008.

[6] Kannan H, Dalton M, Kozyrakis C. "Decoupling dynamic information flow tracking with a dedicated coprocessor." *Dependable Systems & Networks, DSN. IEEE/IFIP International Conference on.*, 2009.

[7] Tiwari, Mohit, et al. "Complete information flow tracking from the gates up." *ACM Sigplan Notices.* ACM, 2009.

[8] Tiwari, Mohit, et al. "Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security." *Computer Architecture (ISCA), 38th Annual International Symposium on.* IEEE, 2011.

[9] Li, Xun, et al. "Caisson: a hardware description language for secure information flow." *ACM SIGPLAN Notices,2011.*

[10] Li, Xun, et al. "Sapper: A language for hardware-level security policy enforcement." *ACM SIGARCH Computer Architecture News.*, 2014.

[11] Bidmeshki, Mohammad-Mahdi, and Yiorgos Makris. "Toward automatic proof generation for information flow policies in third-party hardware IP." Hardware Oriented Security and Trust (HOST), International Symposium on. IEEE, 2015.

[12] Zhang, Danfeng, et al. "A Hardware Design Language for Timing-Sensitive Information-Flow Security." *ASPLOS. 2015.*

[13] SecVerilog documentation: http://www.cs.cornell.edu/projects/secverilog/

[14] Denning, Dorothy E. et al.,"Certification of programs for secure information flow." Communications of the ACM,1977.

[15] Hu, Wei, et al. "Detecting Hardware Trojans with Gate-Level Information-Flow Tracking." Computer 49.8 (2016): 32-40.

[16] Oberg, Jason. (2014). "Testing Hardware Security Properties and Identifying Timing Channels." UC San Diego: Computer science.

[17] Yosys Open Synthesis Suite: http://www.clifford.at/yosys/

[18] http://opencores.org/projects