Chapter 16
Integrating Information Flow Tracking into High-Level Synthesis Design Flow

Wei Hu, Armaiti Ardeshiricham, Lingjuan Wu, and Ryan Kastner

16.1 Introduction

High-level synthesis (HLS) enables hardware designers to write an untimed circuit description allowing them to focus on architectural design optimizations like pipelining, task level parallelism, and array partitioning [1]. By removing the burden of describing cycle accurate behaviors, HLS designers can perform a more comprehensive design space exploration to better find an architecture that meets the desired power, performance, and area (PPA) constraints.

While HLS tools are effective at exploring tradeoffs and optimizations related to PPA, security has largely been an after-thought. The emergence of hardware security flaws and threats [2–5] has brought a demand for hardware security verification tools. Due to the high cost (or even technical impossibility) to patch hardware security vulnerabilities after chip fabrication, identifying and eliminating security flaws in the early design phase is crucial.

Information flow tracking (IFT) is a fundamental technique for hardware security verification [6–8]. IFT allows the designer to verify security properties related to confidentiality, integrity, and availability. An important first step is to create security enhanced circuit models for accurate description of security-related design behaviors and formal verification of security properties [9–12]. Some recent

W. Hu
Northwestern Polytechnical University, Xi'an, China
e-mail: weihu@nwpu.edu.cn

A. Ardeshiricham · R. Kastner (✉)
UC San Diego, La Jolla, CA, USA
e-mail: aardeshi@ucsd.edu; kastner@ucsd.edu

L. Wu
Huazhong Agricultural University, Wuhan, China
e-mail: wulj@mail.hzau.edu.cn

S. Katkoori, S. A. Islam (eds.), *Behavioral Synthesis for Hardware Security*,
https://doi.org/10.1007/978-3-030-78841-4_16

works [13–15] incorporate security models in HLS to allow automated synthesis of secure hardware accelerators. Ideally, these security models can be seamlessly integrated into the standard EDA flow in order to allow security to be verified alongside traditional design constraints during design space exploration [16, 17] without incurring additional design burdens (e.g., learning new design languages and tools) on hardware designers. However, securing hardware accelerators is still a significant challenge for HLS [18].

We aim to answer the question of "how do we best integrate security into HLS hardware design flow?" To better understand this question, we describe a security aware HLS flow that integrates into a property driven hardware security verification flow [6]. We enhance the HLS design flow by integrating information flow tracking to allow the verification of security properties. We discuss the value of performing information flow security verification at the register transfer level (RTL); we develop precise IFT methods; we present fine-granularity information flow model formalizations, and we illustrate how hardware security properties related to confidentiality, integrity, isolation, constant time, and malicious design modification could be formally verified using standard EDA verification tools. Specifically, we make the following contributions:

1. Proposing a method to enhance the backend of HLS by employing information flow security verification using standard EDA tools;
2. Developing precise hardware IFT methods at the register transfer level and deriving security enhanced circuit model formalizations in a standard HDL;
3. Presenting experimental results to demonstrate the effectiveness of our security verification techniques in proving hardware security properties and identifying security vulnerabilities.

The remainder of this chapter is organized as follows. Section 16.2 provides a background discussion on how to integrate security into the HLS design flow. We lay out a basic methodology for security enhanced HLS design flow. Section 16.3 illustrates how hardware security properties can be modeled and verified from the perspective of information flow—a frequently used technique for enforcing security in hardware designs. In Sect. 16.4, we elaborate on our efforts in developing standardized hardware security models for security verification at the RTL. Section 16.5 presents experimental results to demonstrate the effectiveness of our security verification solution in identifying security flaws. We conclude the book chapter in Sect. 16.6.

16.2 Background

HLS allows designers to specify circuits in a more abstract manner. The key feature is that HLS specifications are *algorithmic* or *untimed*—designers do not need to describe circuit behaviors on a cycle-by-cycle basis. This allows them to more efficiently build and verify their hardware designs.

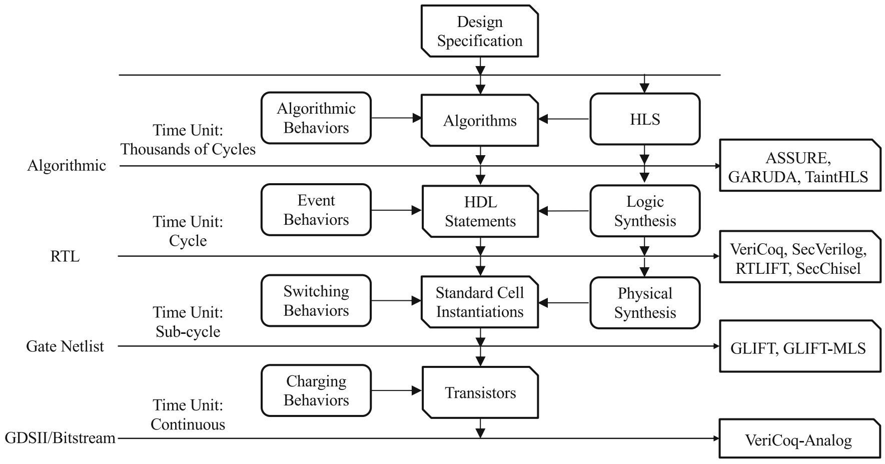


Fig. 16.1 Hardware IFT at different levels of abstraction

An HLS designer provides a specification in a high-level language (e.g., C, C++, SystemC). The designer then goes through an optimization process to create different architectures, and hopefully find one that suits their needs with respect to the traditional circuit metrics of performance, power, and area. This process involves adding HLS optimizations involving pipelining, data partitioning, and data representation to find a final design that best fits their design goals. HLS tools translate the high-level algorithmic specification into a *register transfer level (RTL)*. RTL provides more details about the functionality of the circuit. In particular, it has cycle accurate behaviors. This RTL is then gradually translated down to lower abstraction levels and eventually a physical layout (GDSII).

Figure 16.1 shows the hardware design flow using HLS as a design entry point. At higher of abstraction, we use models of computation that describe hardware design behaviors, e.g., algorithms and functions. While these high-level entities are more concise, they abstract away a huge amount of information that is needed for the final hardware implementation. As the synthesis process proceeds to RTL and gate level, the hardware design is represented using more concrete circuit models such as function units and standard macro cells. The timing behavior of hardware designs also becomes more accurate as the design description is refined; RTL provides cycle level accuracy while gate level and lower abstractions yield sub-cycle timing information. Additionally, area models are better understood as more details about gate sizes, their locations, and wire lengths become available.

In a similar sense, hardware security verification can be performed at different abstraction levels. There are projects that perform verification on algorithmic specifications [13–15]. Other hardware security techniques work at RTL [11, 12, 19–21]. Some techniques perform verification at the gate level [9, 22] and even some take into account analog characteristics [23].

It is important to select the right abstraction when employing secure information flow analysis for hardware design [24]. *The key question that we aim to answer is: "What is the best level of abstraction for information flow analysis for an HLS design flow?"* To try to better understand these tradeoffs, we perform a comparison of hardware IFT techniques. We focus on the HLS, RTL, and gate levels which are commonly considered in the hardware security verification flow.

At the algorithmic level, the hardware design is described using highly abstractive design models. As a result, we need to make very conservative assumptions about design behaviors and employ conservative rules for security label propagation. In addition, it is difficult to model timing flows (see Sect. 16.3) due to the lack of accurate timing information. Additionally, distinguishing implicit flows from explicit ones is a challenging task at this level. The effect of conditional operations can spread across a wide range of operations, which are hard to track. The benefit of modeling IFT at HLS is that the verification is typically faster than that at lower levels due to the simplified circuit models and utilization of conservative label propagation rules.

Performing information flow analysis at the RTL allows analysis that requires timing accurate behaviors. This includes timing side channels, which would not be exposed when considering IFT using an algorithmic (HLS) abstraction. Additionally, interactions related to sharing resources are also apparent at the RTL but are not visible at higher abstraction levels. This includes shared registers, shared functional units, memories, and interfaces. Understanding the implications of resource sharing is a particularly challenging aspect of hardware security and performing IFT at the algorithmic level would abstract away some important aspects related to this.

Moving to the gate level for IFT analysis provides some additional benefits since more details of the circuit are available. Sub-cycle timing and switching behaviors are better understood which can provide better analysis of security concerns related to timing or switching. Unfortunately, as the hardware design refines, it is translated into a significantly larger number of gates. This poses big challenges in scalability of IFT techniques and security verification performance. It is generally known to the EDA community that RTL verification is much easier and faster than gate-level verification methods [25]. That is an important reason why recent hardware IFT techniques have gravitated towards the RTL for better performance and scalability.

Clearly there are tradeoffs for performing analysis at these different abstraction levels. The lack of details about cycle level timing and resource sharing severely limit the type of information flow analysis that can be done using an algorithmic description. This requires the IFT logic to be very conservative which forces the designer to be overly conservative in their security decisions. This points to the need to perform the analysis at a lower level of abstraction.

Gate-level analysis has sub-cycle timing information and a clear notion of resource sharing. Additionally, the information flow analysis is simplified in some regards as the basic units are logical operations; analyzing flows of information is a lot easier when computation is broken done at this level of granularity [22]. Yet, some higher-level information about control is lost in the translation to gates. For example, it is hard to differentiate between control and data flow since everything

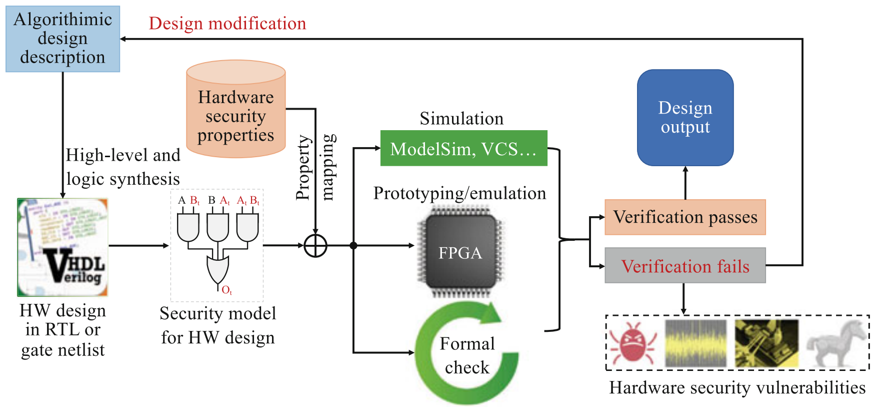

Algorithimic design description
Design modification
Hardware security properties
Property mapping
Simulation
ModelSim, VCS…
Design output
High-level and logic synthesis
Prototyping/emulation
FPGA
Verification passes
Verification fails
HW design in RTL or gate netlist
Security model for HW design
Formal check
Hardware security vulnerabilities

Fig. 16.2 Secure hardware design flow that employs information flow security verification for secure vulnerability detection

is just Boolean gates. This means that timing flows cannot be separated from functional flows [20, 26].

We argue that IFT analysis performed at the RTL provides the best tradeoff between scalability and ability to differentiate different flows of information including timing side channels, vulnerabilities related to resource sharing, and those related to control and data path interactions. We make a case for this in subsequent sections. But before we do that, we need to describe the hardware security verification process and some background on information flow analysis.

Figure 16.2 describes a general framework for integrating information flow analysis into an HLS security verification flow, which follows a property based approach to hardware security [6]. The framework takes an algorithmic hardware description and a set of security properties. The description is synthesized in a typical fashion using HLS, logic, and physical synthesis. The key question that we consider is when to generate the security models and perform the security analysis. Our experiments and discussions attempt to understand value and tradeoffs of performing this analysis at different levels.

We use an IFT analysis method that generates a security model for analysis. The security model is derived from the original circuit; it is fully synthesizable, but separate circuit, that can be analyzed using existing EDA verification tools. The security model is used to verify information flow security properties specified using standard property specification languages such as SystemVerilog Assertion (SVA). Since both the formal circuit model and security properties are described with standard HDLs, the verification process can be performed through simulation, FPGA prototyping or emulation as well as formal proof. If the hardware design adheres to all desired security properties, it is ready for design output. Otherwise, the security verification fails, indicating potential existence of unintentional design flaws or intended malicious design modifications. In such a case, the design process should iterate until the design passes security verification. The security model is not required to be added to the final circuit though it can be if one desires run-time security violation checking.

16.3 Hardware Information Flow Tracking

Information can flow through hardware designs in a variety of different ways. This includes *logical flows* and *physical flows*. This book chapter primarily aims to understand the flow of information in hardware designs during the early design phase. Thus, we only account for the logical information flows.

Logical information flows can be further classified into *explicit* and *implicit*. Explicit information flows happen when data is explicitly assigned to a destination operand while implicit flows usually occur when some operands are conditionally updated. The following code snippet illustrates the difference between these two types of information flows.

```
1:  key_hash := hash(key)
2:  if(key_hash == CORRECT_HASH_RECORD)
3:      unlock := 1
4:  else
5:      unlock := 0
6:  end if
```

In this example, there is an explicit flow from *key* (or more accurately) *hash*(*key*) to *key_hash* resulting from the first explicit assign statement (*line* 1). There is also a piece of implicit information flow from *key_hash* to *unlock* even if *key_hash* is not directly assigned to *unlock* (*lines* 2–5). This is due to the fact that by observing the status of *unlock*, we can learn if the *key_hash* matches the record.

From the example, explicit flows are easy to capture while implicit flows are more difficult to determine. An even more subtle case of implicit flow is *timing flow*, which is caused by conditional updates of stateful elements [20]. The following code snippet shows a case of timing flow caused by fast paths (*lines* 1–2 and *lines* 3–4) in the exponentiation unit. An unauthorized process may be able to infer the exponent by observing the amount of time take to calculate the exponentiation.

```
1:  if(exponent == 0)
2:      power := 1
3:  else if(exponent == 1)
4:      power := base
5:  else
6:      power := exp(base, exponent)
7:  end if
```

From the above examples, information flows in hardware design can lead to leakage of protected information and cause security violation. In order to understand

and further prevent such leakage, we need to specify and enforce information flow security properties, which will be discussed in the following subsection.

16.3.1 Information Flow Security Properties

Undesired flows of information could violate different security policies such as *confidentiality*, *integrity*, and *isolation*. For instance, in the *key verifier* example from the previous subsection, flow of secret information from the *key_hash* variable to the *unlock* signal violates the confidentiality property if the adversary has access to the value of *unlock* signal. To detect such violations, information flow security properties are added to the design in forms of logical assertions. An assertion in the form of $assert(key \not\rightarrow unlock)$ detects the confidentiality breach in the *key verifier* example. Here, the "$\not\rightarrow$" operator indicates the absence of information flows from the left-hand side variable to the right-hand side variable.

Information flow properties capture security relevant design behavior which cannot be expressed by existing property specification languages for functional verification. In the following we review the major security policies which are specified using the model of information flow.

- **Confidentiality:** The confidentiality property analyzes the relation between secret data and publicly observable ports. To preserve confidentiality, we need to constrain the flow of information from data objects which contain secret information. For instance, in a cryptographic core, confidentiality is stated as $assert(key \not\rightarrow pub)$, where pub represents public ports that are not encrypted.
- **Integrity:** The Integrity property is the dual of confidentiality and refers to the information flow from untrusted data to critical components in the design. For instance, to preserve integrity of the program counter (PC) register in a processor, there should be no flow of information from the public inputs such as the Ethernet port to the PC. This property can be specified as $Ethernet\_port \not\rightarrow PC$.
- **Isolation:** The isolation property denotes eliminating information flow between two entities. As an example, consider a SoC where the AES core and the IIR filter should be isolated. This expectation is modeled by $assert(AES\_out \not\rightarrow IIR\_in \;\&\&\; IIR\_out \not\rightarrow AES\_in)$. Note that isolation is a two-way policy and is enforced on both cores.
- **Timing Side Channels:** Information flow leakage through timing side channel can be used to break confidentiality even in cases where the secret data is not directly readable. Hence, to avoid timing side channel attacks, we need to eliminate timing leakage. For instance, to detect timing leakage in a cryptographic core, we need to specify that the secret key does not flow to the cipher via timing channels. This is stated as $assert(key \not\rightarrow_{time} cipher)$. Here, "$\not\rightarrow_{time}$" operator represents absence of timing flows.
- **Hardware Trojans:** Information flow properties can detect certain class of hardware Trojans where a malicious circuitry is inserted to generate unauthorized

flow of information [27]. For example, in the Trust-Hub benchmarks [28], hardware Trojans are added to cryptographic cores to transfer the secret key to the output "$Antena$". The information flow property to detect these Trojans can be formulated as $assert(key \not\rightarrow Antena)$.

The information flow security policies can be translated to a set of SystemVerilog assertions written over the instrumented design. More specifically, an IFT policy modeled as $assert(A \not\rightarrow B)$ is translated to the following properties, where A_t and B_t are the security labels of A and B, respectively.

```
assume (A_t == 1);
assume (B_t == 0);
```

In this example we assume both A and B have single bit labels. A timing side channel policy modeled as $assert(A \not\rightarrow_{time} B)$ is verified using the following properties, where B_time is the timing security label of B.

```
assume (A_t == 1);
assume (B_time == 0);
```

16.3.2 Fundamentals of Hardware IFT

Hardware IFT is a commonly used technique for measuring the flow of information in circuit designs. The core idea behind hardware IFT is to associate data objects in the hardware design with a label for encoding security attribute, e.g., sensitive information can be labeled as `secret` while information from an open computing environment should be marked as `untrusted`. These meta data will be processed along with the data objects to determine the security attribute of the outputs. The output label will be updated according to the flow of information. Specifically, an input flows to the output *if and only if* the input has an influence on the value of the output. In this case, the security label of the input will be involved in determining the security label of the output. Thus, information flows can be measured by observing the relation between the input and output security labels. Figure 16.3 uses streaming cipher as an example to illustrate how hardware IFT is performed.

Apart from the *original logic* that XORs the plaintext (i.e., m_i) and key (i.e., k_i) streams shown in Fig. 16.3a, we need to associate the inputs with security labels, i.e., m_t_i and k_t_i, respectively. Additional *IFT logic* is then instantiated to process these metadata and calculate the output security label as shown in Fig. 16.3b. The way in which the IFT logic is implemented depends on the label propagation policy employed. This example simply takes the logical OR (i.e., the upper bound) of input security labels as the security label for the output. There are more complex but also more precise label propagation policies as we will illustrate in Sect. 16.4. Before that, we briefly cover the fundamental aspects of hardware IFT.

It is possible to succinctly describe the flow relationships on the Boolean operations. IFT logic formalizations and generation algorithms lay the theoretic

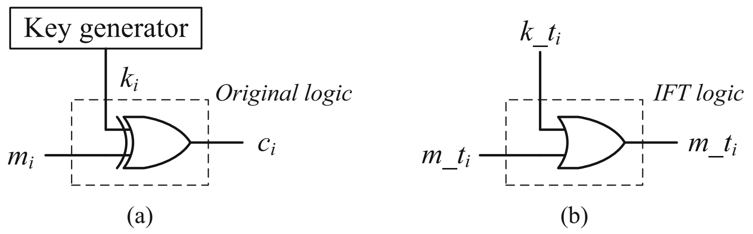


Fig. 16.3 Hardware information flow tracking uses IFT logic for label propagation. (**a**) The data encryption logic for stream ciphers. (**b**) The IFT logic for the encryption operator

Table 16.1 IFT logic for Boolean operations

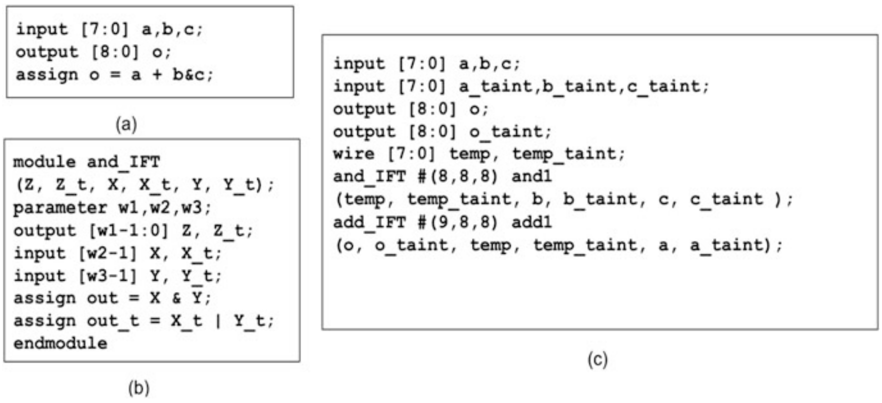
Gate	Boolean function	IFT logic
AND-2	$f = g \cdot h$	$\mathcal{L}(f) = g \cdot \mathcal{L}(h) + h \cdot \mathcal{L}(g) + \mathcal{L}(g) \cdot \mathcal{L}(h)$
OR-2	$f = g + h$	$\mathcal{L}(f) = \overline{g} \cdot \mathcal{L}(h) + \overline{h} \cdot \mathcal{L}(g) + \mathcal{L}(g) \cdot \mathcal{L}(h)$
XOR-2	$f = g \oplus h$	$\mathcal{L}(f) = \mathcal{L}(g) + \mathcal{L}(h)$
INV	$f = \overline{g}$	$\mathcal{L}(f) = \mathcal{L}(g)$
AND-N	$f = f_1 \cdot f_2 \cdots f_n$	$\mathcal{L}(f) = \prod_{i=1}^{n}(f_i + \mathcal{L}(f_i)) - f$
OR-N	$f = f_1 + f_2 + \cdots + f_n$	$\mathcal{L}(f) = \prod_{i=1}^{n}(\overline{f_i} + \mathcal{L}(f_i)) - \overline{f}$
XOR-N	$f = f_1 \oplus f_2 \oplus \cdots \oplus f_n$	$\mathcal{L}(f) = \sum_{i=1}^{n} \mathcal{L}(f_i)$

groundwork for hardware IFT [29]. Table 16.1 shows the flow tracking logic for Boolean operations where $\mathcal{L}(\cdot)$ denotes the function for calculating security label, sum represents logical OR while product means logical AND. The minus operator means excluding that term from the equation.

The IFT logic shown in Table 16.1 can be extended to Boolean operations of variable widths, e.g., using the generate feature of HDL. In this way, we construct an IFT library for deriving IFT methods as we will introduce in the following section. The role of the IFT library in creating IFT logic for large circuits is similar to technology library in technology mapping.

16.4 Register Transfer Level Information Flow Tracking

RTLIFT software accepts a hardware design implemented in the Verilog language and generates functionally equivalent Verilog code which is instrumented with information flow tracking logic. The outputted code is described at the same abstraction level as the input design. This is achieved by defining label propagation rules for RTL language constructs eliminating the need to synthesize the input design to a netlist. The code generated by RTLIFT can be analyzed by standard EDA verification tools and allows leveraging decades of research on functional testing to assess security properties of hardware designs. If the instrumented design passes



```
input [7:0] a,b,c;
output [8:0] o;
assign o = a + b&c;
```

(a)

```
module and_IFT
(Z, Z_t, X, X_t, Y, Y_t);
parameter w1,w2,w3;
output [w1-1:0] Z, Z_t;
input [w2-1] X, X_t;
input [w3-1] Y, Y_t;
assign out = X & Y;
assign out_t = X_t | Y_t;
endmodule
```

(b)

```
input [7:0] a,b,c;
input [7:0] a_taint,b_taint,c_taint;
output [8:0] o;
output [8:0] o_taint;
wire [7:0] temp, temp_taint;
and_IFT #(8,8,8) and1
(temp, temp_taint, b, b_taint, c, c_taint );
add_IFT #(9,8,8) add1
(o, o_taint, temp, temp_taint, a, a_taint);
```

(c)

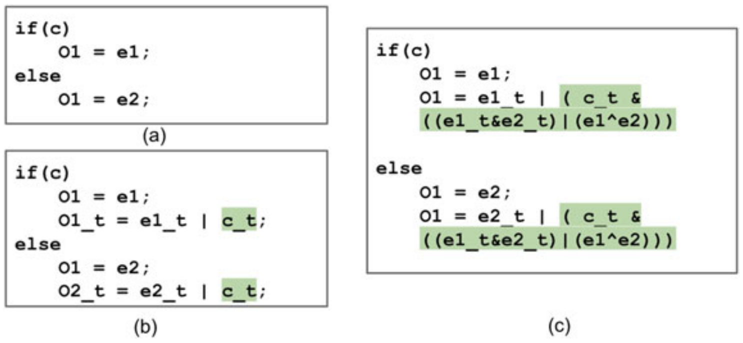
Fig. 16.4 Tracking explicit flows via RTLIFT. (**a**) Input Verilog code. (**b**) RTLIFT library for *and* operation. (**c**) Instrumented Verilog code

the security verification, the original code can be used for fabrication. Otherwise, the original code should be modified, instrumented, and verified again.

RTLIFT enables tracking both explicit and implicit flows. Flow tracking starts by extending each design component (e.g., wires and registers in a given Verilog code) with a label that carries out information regarding the security properties of the data. After extending the variables with security labels, every HDL operation is replaced with an IFT-enhanced operation. An IFT-enhanced operation is functionally equivalent to the original operation, but it also includes the logic for tracking explicit flows through that operation. The IFT-enhanced operations are defined in RTLIFT library for all valid Verilog operations. Figure 16.4 shows an example of tracking explicit flows at RTL.

Tracking only explicit flows might inaccurately report the absence of information flow by ignoring existence of implicit flows through conditional statements. To capture these flows, we need to obtain a list of variables which affect the execution of each statement. The logic for tracking the implicit flows can be generated using this list with different levels of precision. To implement a conservative IFT approach, any usage of tainted conditions should yield a tainted output. To have more precise flow tracking, we need to figure out if different outcomes are possible for the right-hand side of an assignment, assuming the conditions were flipped. Using this approach, we can track the implicit flow through each conditional statement by modeling it as a multiplexer.

To illustrate the idea, we show implicit flow tracking for a simple example shown in Fig. 16.5. Here, `e1_t` and `e2_t` represent the explicit flows from expressions `e1` and `e2`. The imprecise approach, as shown in Fig. 16.5b, marks the output of a conditional statement as tainted whenever the condition is tainted. The precise IFT logic specifies that information flows from the condition signal to the output only in cases where the condition is tainted while both inputs are tainted or they have different Boolean values.



(a)
```
if(c)
    O1 = e1;
else
    O1 = e2;
```

(b)
```
if(c)
    O1 = e1;
    O1_t = e1_t | c_t;
else
    O1 = e2;
    O2_t = e2_t | c_t;
```

(c)
```
if(c)
    O1 = e1;
    O1 = e1_t | ( c_t &
    ((e1_t&e2_t)|(e1^e2)))

else
    O1 = e2;
    O1 = e2_t | ( c_t &
    ((e1_t&e2_t)|(e1^e2)))
```

Fig. 16.5 Tracking implicit flows via RTLIFT. (**a**) Input Verilog code. (**b**) Verilog code instrumented with imprecise logic. (**c**) Verilog code instrumented with precise logic. Tracking logic for implicit flows are highlighted

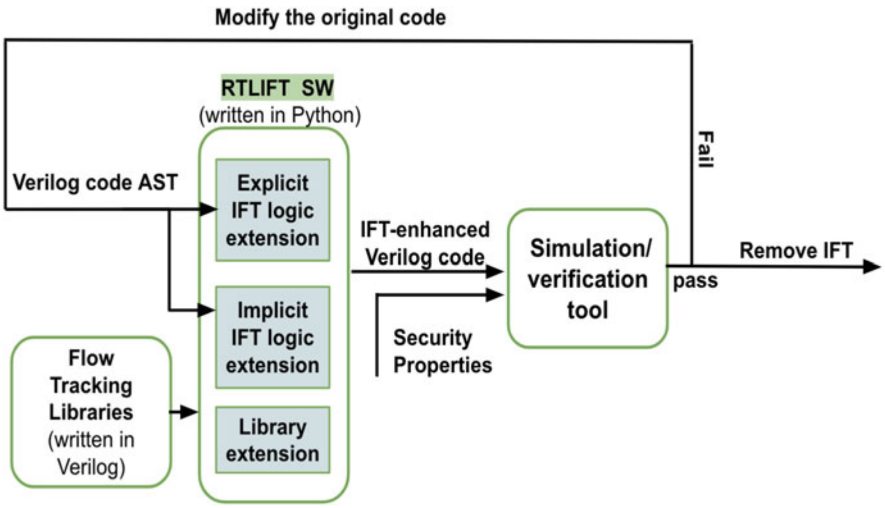

Modify the original code
RTLIFT SW
(written in Python)
Verilog code AST
Explicit IFT logic extension
Implicit IFT logic extension
Library extension
Flow Tracking Libraries (written in Verilog)
IFT-enhanced Verilog code
Security Properties
Simulation/ verification tool
Fail
pass
Remove IFT

Fig. 16.6 AST based RTLIFT overview

Figure 16.6 gives an overview of an Abstract Syntax Tree (AST) Based RTLIFT tool. The IFT instrumentation is done by analyzing the data flow graph of the input design. The data flow graph is collected by using *Yosys frontend compiler* [30] to transform the code to its AST representation. RTLIFT analyzes the node of each assignment statement via in-order traversal. And for each operation, it adds a module from the IFT library. RTLIFT considers a single bit label for each variable bit. Here, a high value indicates either sensitive or untrusted value, depending on the property to be verified.

The AST generated by high-level HDL frontends are further converted into the RTLIL main internal data format for further design optimization in Yosys. Afterwards, the RTLIL representation can be converted into various formats including

Verilog and ILANG. We provide another RTL hardware IFT method that targets RTLIL to leverage the synthesis optimizations in Yosys.

The RTLIL defines design objects such as *module*, *cell*, *wire*, *process*, and *memory*. For the *module* and *wire* objects, we only need to extend the original design variable list with security labels. We also assign one-bit security label for each data bit. Similarly, the IFT logic for *cell* design objects can also be created by mapping these cells into a standard IFT logic library. The *process* design objects represent the *if–then–else* and *case* statements. These are the most challenging steps in RTL tracking logic generation. We take an approach similar to AST based RTLIFT to create IFT logic for processes.

16.5 Experimental Results

This section presents experimental results to better understand the tradeoffs of when to perform IFT analysis. We start with some concrete design examples to demonstrate the effectiveness of our information flow security verification method in detecting design flaw, timing channel and malicious design modification in Sect. 16.5.1. Section 16.5.2 performs an information flow security verification performance analysis. Section 16.5.3 performs an analysis of hardware IFT logic generated from different RTL intermediate representations and gate-level netlist in terms of complexity and precision. In Sect. 16.5.4, we demonstrate our design methodology that integrates IFT into HLS flow for hardware security verification using a RSA example.

16.5.1 Security Verification Results

16.5.1.1 Design Flaw

We use a Present encryption core from Opencores [31] to demonstrate how our security verification method can be used to detect design flaws that can cause security issues. Figure 16.7 shows the architecture of the core.

We create RTL IFT logic for the core using the IFT logic generation method introduced in Sect. 16.4 and label the lowest plaintext bit (i.e., *message*[0]) as `secret`. We then formally verify the diffusion security property stating that *each plaintext bit should affect (or flow to) multiple bits in the ciphertext*. The property can be formalized as follows:

```
assume message_t = 64'h01
assume key_t = 80'h00
assert cipher_t[0] == cipher_t[1]
```

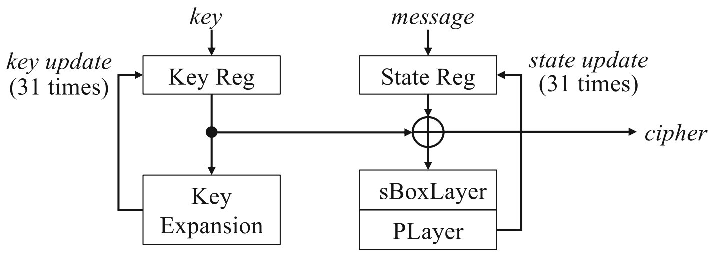


Fig. 16.7 A Present cipher core from Opencores [31]

The security property requires that when the lowest plaintext bit flows to the lowest ciphertext bit (i.e., *cipher*[0]), it should also flow to the second bit of the ciphertext (i.e., *cipher*[1]). Or the lowest plaintext bit should have not yet flowed to either of these two ciphertext bits, e.g., after core reset.

We translate the security property into security constraints and formally verify if the Present core adheres to such a property using *Mentor Graphics Questa Formal*. Formal proof result indicates that the security property can be violated under certain conditions. The counter example returned by the formal verification tool shows that the security property fails to hold right after the *message* is loaded into the state register. This is because the state and key registers are shared and updated during different rounds of encryption. The $key \oplus state$ is assigned directly to *cipher* at all times. Loading the state register will allow *message* to flow to *cipher*, rendering the security labels of *cipher*[0] and *cipher*[1] logical `1` and `0`, respectively.

In this example, our security verification method has successfully identified the design flaw that feeds intermediate encryption results to the observable ciphertext port.

16.5.1.2 Timing Channel

Timing variations in hardware designs have been repeatedly exploited by attackers to break software implementations of ciphers such as RSA and AES. Many of these timing side channel attacks target timing variations through caches and cipher implementations that use pre-computed values that are indexed based on the value of the secret key [32, 33]. In such scenarios, the attacker can collect information regarding the secret key by extracting the cache access pattern of the process running the encryption. To mitigate such attacks, several cache architectures have been developed to eliminate the correlation between the index value of sensitive cache accesses and the time that it takes for the cache to retrieve data in later cycles.

We use hardware IFT to show existence of timing flows in a traditional cache implementation (i.e., with no mitigation technique in place) and the random permutation cache (RPcahce) introduced in [34]. To write the security properties, we consider two processes with isolated address spaces that share the cache. We mark indexes of accesses made by one process (with pid i) as tainted and check if the data read by the other process (with pid j) contains timing variation. The IFT properties for detecting timing flows in an instrumented cache are written as follows.

```
if (pid == i)
  assume(index_t == 16'hFFFF);
if (pid == j)
  assert(data_rd_proc_time == 32'b0);
```

The IFT verification fails for the conventional cache as expected. This is due to the fact that the address which is used to access the cache (i.e., *index_t*) influences the data which is being evicted from the cache. Once process j accesses the evicted data, this access takes longer and is distinguishable through repeated measurements.

We next test the RPcache that eliminates any relation between the cache collisions by randomly permuting the mapping of memory to cache addresses, and randomly choosing a cache line for eviction. This randomization disables the attacker from observing the victim's cache patterns. The IFT verification passes for the RPcache assuming that the random number generator is untainted.

16.5.1.3 Stealthy Hardware Trojan

We use a satisfiability Trojan example proposed by Hu et al. [35] to demonstrate how our security verification method can be used to detect malicious design modifications. The Trojan uses a signal pair that cannot be logical `1` at the same time from AES S-Box as Trojan trigger and adds two multiplexers to multiplex the AES key to the ciphertext output port, as shown in Fig. 16.8. This Trojan design will be activated when both signals are logical `1` and thus the Trojan will never be activated under normal operation. As a consequence, the Trojan cannot be detected using functional testing or even formal equivalence checking.

We also use the method introduced in Sect. 16.4 to create RTL IFT logic for the AES design and label the *key* as `secret`. We declassify at the last add round key operation (i.e., *cipher*) and manually set *cipher* to `unclassified`. This declassification operation is generally regarded as safe and thus allowed. We then

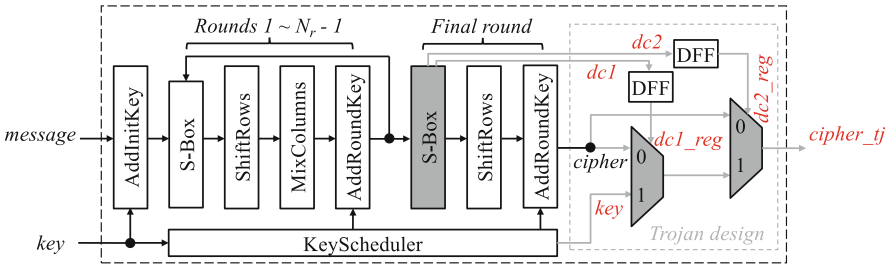


Fig. 16.8 A satisfiability hardware Trojan that leaks the AES key [35]

formally prove that the public output port of the AES core should not take a `secret` label after the declassification operation. We specify the following security property for this proof:

```
assume key_t = {128{1'b1}}
assume cipher_t = 128'h00
assert cipher_tj == 128'h00
```

We then formally prove the above security property under the open source Yosys proof tool. Proof result indicates that the security property does not hold under certain conditions. The counter example returned by Yosys shows that the $cipher\_tj$ can be non-zero when $dc1$ and $dc2$ are both logical `1`. Thus, the formal security verification has precisely captured the trigger condition of the Trojan.

16.5.2 Verification Performance Analysis

We use several design examples and benchmarks for verification performance analysis. We use the IFT logic generated by our RTLIFT tool as the security verification model and verify information flow security properties on these security models. In our test, we use the *SAT* solver in *Yosys* to prove security properties. Table 16.2 shows the security properties proved and verification performance results.

As an example, it takes 384.55 s to run formal verification and detect the Trojan for the example discussed in Sect. 16.5.1.3. From Table 16.2, RTLIFT provides an approach for constructing security models that allow security properties to be verified within acceptable amount of time.

16.5.3 Complexity and Precision Analysis

We use several IWLS benchmarks [37] for IFT complexity and precision analysis. We use the number of cells in synthesized IFT circuits as a measure for complexity while the number of simulated information flows as a measure of precision. We use combinational benchmarks in complexity and precision analysis to more accurately measure input-output flow relations, eliminating the complex flow relations over multiple clock cycles. Figure 16.9 shows our test flow as well as tools used to create different IFT logic circuits.

We use five different test flows for IFT logic generation. The *ABC-resyn2* flow first uses the *resyn2* script in *ABC* [38] to synthesize the benchmarks to gate-level netlists and then uses our GLIFT script to create GLIFT logic. The *ABC-dc* test flow uses the *resyn2rs*, *compress2rs*, *dc2*, *dch* and *mfs3* scripts to synthesize the design, which yields higher optimization effort and also enables don't care based optimization. The *AST2-conservative* and *AST2-precise* test flows first dump

Table 16.2 Proof time (*sec*) for verifying security properties on several design examples and benchmarks

Designs	Security properties	Proof time
AES-DPA [36]	assume key_t = {127{1'b1}, 1'b0}; assert	13.01
	cipher_t[0] == cipher_t[1]	
PresentEncryptor [31]	assume key_t = {79{1'b1}, 1'b0}; assert cipher_t[0] == cipher_t[1]	1.31
AES-T400 [28]	assume key_t = {128{1'b1}}; assert Antena_t == 1'b0	474.26
AES-T1700 [28]	assume key_t = {128{1'b1}}; assert Antena_t == 1'b0	703.76
RSA-T400 [28]	assume indata_t = 32'h0; assume indata_t = {32{1'b1}}; assert count_t[0] == 1'b0	4.59
AES-DC-TJ [35]	assume key_t = {128{1'b1}}; assume cipher_t = 128'h0; assert cipher_tj == 128'h0	384.55

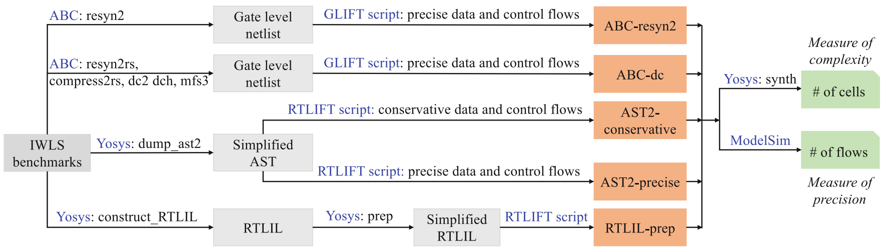


Fig. 16.9 Different test flows for IFT logic generation

simplified AST (i.e., -dump_ast2) using the Verilog frontend in Yosys and then create IFT logic from the AST extracted. The difference lies in that the *AST2-conservative* test flow employs a conservative policy to measure data and control flows while *AST2-precise* uses a precise one. The *RTLIL-prep* flow first constructs the RTLIL intermediate presentation for the benchmark and then invokes the *prep* synthesis script in Yosys to optimize the RTLIL representation. The IFT logic is generated from the optimized RTLIL.

After generating different versions of IFT circuits at the gate level and RTL for each benchmark, we use the *synth* script in *Yosys* [30] to synthesize these IFT logic circuits and report the number of cells in the synthesized IFT circuits. We also use *ModelSim* to test the IFT logic circuits under 2^{20} random test vectors and observe the total number of information flows. Table 16.3 shows the test results.

From Table 16.3, the *AST2-precise* and *RTLIL-prep* test flows perform identical optimizations and produce identical IFT logic. This is formally verified by equivalence checking of IFT logic circuits created by these two test flows. The *AST2-conservative* flow employs conservative policy to track data and control flows. It significantly reduces the complexity of IFT logic circuits at the side effect of a larger number of false positives in information flow measurement. These additional information will not actually happen. The *ABC-dc* test flow that performs don't care based logic optimization on the original design leads to smaller number of cells and information flows. By comparison of the *ABC-resyn* and *AST2-precise*/*RTLIL-prep* test flows, AST and RTLIL based optimizations tend to lead to larger number of cells while close number of information flows. The test results reveal the tradeoff between complexity and precision of hardware IFT.

For a better understanding, we visualize the number of cells and flows normalized to those for the *GLIFT-resy2* test flow. The normalized results are shown in Figs. 16.10 and 16.11, respectively.

Table 16.3 Precision and complexity of hardware IFT logic generated using different methods

	GLIFT-resyn2		GLIFT-dc		AST2-conservative		AST2-precise		RTLIL-prep	
Benchmarks	Cells	Flows	Cells	Flows	Cells	Flows	Cells	Flows	Cells	Flows
C1355	656	33546328	1398	33545816	334	33554368	3099	33546328	3101	33546328
C1908	1879	20954234	2028	21074948	425	26214350	2437	21074890	2434	21074890
C2670	3406	83584759	2891	83528247	921	97009773	3836	84786443	3820	83747198
C3540	6152	20982566	6324	21099557	1675	22864600	6205	20913238	6510	20913238
C432	1285	6552124	1219	6307165	353	7340006	1019	6553500	1008	6553500
C499	1625	33546253	2644	33545521	334	33554368	739	33546253	743	33546253
C5315	9224	92555711	9175	91789720	2498	112689371	9482	90829295	9522	90829295
C7552	7758	78802237	7426	78931128	2689	87655782	12661	79899472	12661	79899472
C880	2140	16049183	1909	16021046	523	25711253	1961	16011824	1972	16011824
frg2	5604	71020251	5312	70543284	3537	134929650	11081	70389291	11078	70389291
i10	12057	124753233	11910	128373586	3455	221522371	13438	121632233	13379	121632233
i5	1864	42869700	1566	42869700	712	68502275	2211	42869700	2221	42869700
i6	2731	54712363	2717	54712363	577	67152895	2379	54719267	2404	54719267
i7	3659	58060003	3297	58634475	922	68584377	3205	58134101	3189	58134101
i8	5474	69087423	5944	71595869	3257	84915387	10094	71866588	10077	71866588
i9	3460	60539033	3633	60326644	1060	66048068	3925	60573112	3925	60573112
k2	7845	23911899	6858	23630132	3637	45044927	13532	22074582	13296	22074582
too_large	26864	3127215	20924	3021917	23762	3145722	72424	1494259	72430	1494259
vda	3693	28572052	4024	26112629	1272	40758299	4999	27998546	4917	27998546
x1	4028	19340837	2285	18783256	3635	32155816	11026	17633240	11015	17633240
x4	4028	41280920	2102	41788569	1511	68221032	4614	42186099	4608	42186099

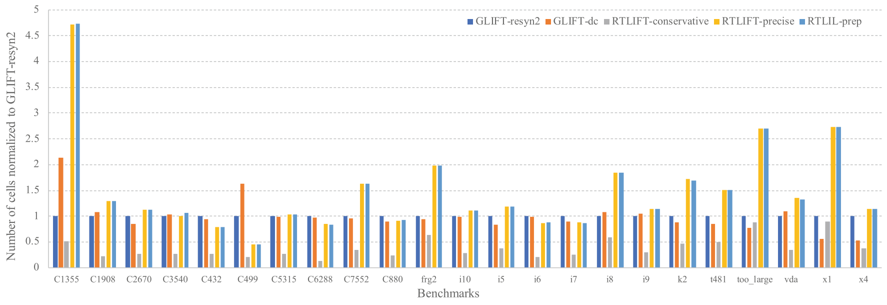


Fig. 16.10 The number of cells normalized to GLIFT-resyn2

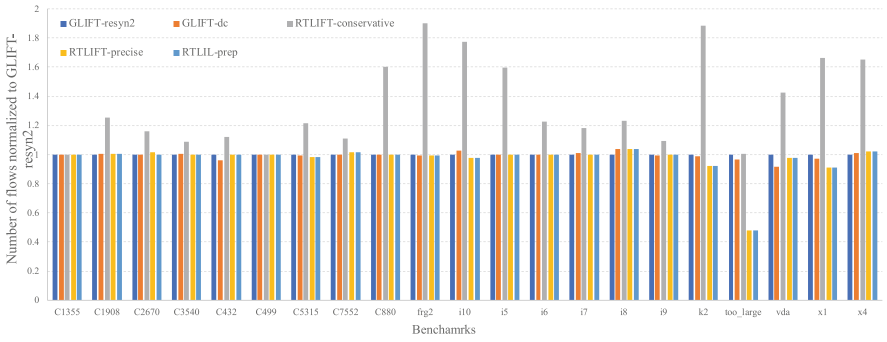
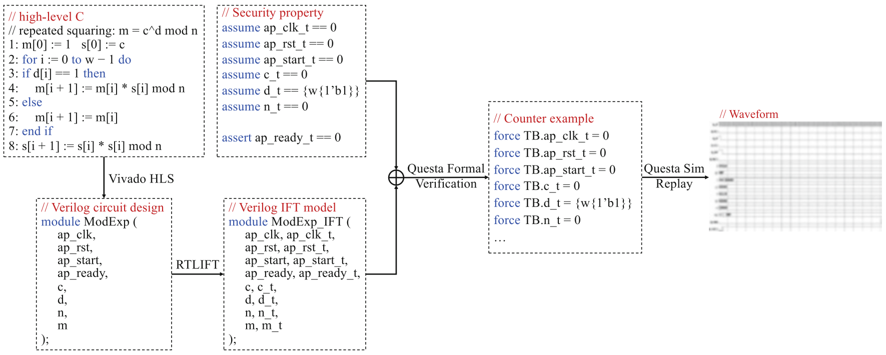


Fig. 16.11 The number of flows normalized to GLIFT-resyn2

16.5.4 An HLS Design and Verification Example

We use a RSA example to demonstrate our design methodology that integrates IFT into HLS flow for hardware security verification. Figure 16.12 shows our test flow. We implement a 32-bit instance of the right-to-left repeated squaring algorithm for calculating modular exponentiation (i.e., the basic operation of RSA) in HLS C. We use *Xilinx Vivado* to synthesize the C code into Verilog design. The resulting RSA design is then converted into IFT model using our RTLIFT tool.

The IFT model is combined with the following security property, which labels the secret exponent d as `high` while all other inputs as `low` and asserts that the encryption result ready output *ap_ready* should be `low`, to perform formal verification of the security property under *Mentor Graphics Questa Formal*. In our proof, we constrain the secret exponent d and modulus n to allow constant values so that the prover only needs to search on the message c, which minimizes the search state space and in turn accelerates the proof process. Such constraint should be applied since the RSA has its key generation rules and only allowed values can be used as legal RSA key pairs.



```
// high-level C
// repeated squaring: m = c^d mod n
1: m[0] := 1   s[0] := c
2: for i := 0 to w − 1 do
3: if d[i] == 1 then
4:     m[i + 1] := m[i] * s[i] mod n
5: else
6:     m[i + 1] := m[i]
7: end if
8: s[i + 1] := s[i] * s[i] mod n
```

```
// Security property
assume ap_clk_t == 0
assume ap_rst_t == 0
assume ap_start_t == 0
assume c_t == 0
assume d_t == {w{1'b1}}
assume n_t == 0

assert ap_ready_t == 0
```

```
// Verilog circuit design
module ModExp (
    ap_clk,
    ap_rst,
    ap_start,
    ap_ready,
    c,
    d,
    n,
    m
);
```

```
// Verilog IFT model
module ModExp_IFT (
    ap_clk, ap_clk_t,
    ap_rst, ap_rst_t,
    ap_start, ap_start_t,
    ap_ready, ap_ready_t,
    c, c_t,
    d, d_t,
    n, n_t,
    m, m_t
);
```

```
// Counter example
force TB.ap_clk_t = 0
force TB.ap_rst_t = 0
force TB.ap_start_t = 0
force TB.c_t = 0
force TB.d_t = {w{1'b1}}
force TB.n_t = 0
...
```

Fig. 16.12 An HLS design and verification example

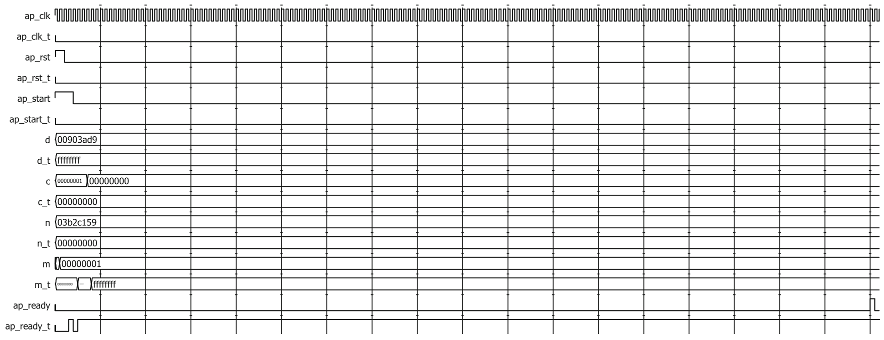


Fig. 16.13 Replay waveform of the counter example from RSA security verification

```
assume ap_clk_t == 0
assume ap_rst_t == 0
assume ap_start_t == 0
assume c_t == 0
assume d_t = {w{1'b1}}
assume n_t = 0

assert ap_ready_t == 0
```

Formal verification result shows that the specified security property can be violated and *Questa Formal* provides a counter example to show when such violation can happen. We replay the counter example under *Mentor Graphics Questa Sim* and the replay waveform is shown in Fig. 16.13.

From Fig. 16.13, the *ap_ready* output can be `high` (i.e., $ap\_ready\_t = 1$), indicating that it can contain information about the secret exponent d since we only labeled d as `high`. This is because the right-to-left repeated squaring RSA implementation contains a timing channel that leaks the secret exponent to *ap_ready* in that the exponent d is used to control a timing-unbalanced conditional branch, i.e., the *if-else* branch statement shown in the high-level C code in Fig. 16.12. This creates timing information flows from d to ap_ready since the secret exponent dominates the encryption time. Our design methodology has detected the timing channel in this RSA implementation using standard verification tools.

16.6 Conclusion

We introduce a methodology to integrate information flow analysis into the HLS design flow. We argue that the RTL provides an optimal place to perform information flow analysis. We describe RTLIFT—a precise information flow tracking method at the RTL for secure hardware design. We provide IFT logic formalization, information flow security property specification, and verification methodology. We demonstrate how our security verification method can be employed to enhance the EDA flow and identify hardware security vulnerabilities in the early design phase.

Acknowledgments This work was supported in part by the Natural Science Foundation of Shaanxi Province under Grant 2019JM-244, NSF award 1718586, and the Semiconductor Research Corporation Task 2770.001.

References

1. Kastner, R., Matai, J., Neuendorffer, S.: Parallel programming for FPGAs (2018). Preprint, arXiv:1805.03648
2. Bulck, J.V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In: 2018 27th USENIX Security Symposium (USENIX Security 18), pp. 991–1008. USENIX Association, Baltimore, MD (2018). https://www.usenix.org/conference/usenixsecurity18/presentation/bulck
3. Weisse, O., Bulck, J.V., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T.F., Yarom, Y.: Foreshadow-NG: breaking the virtual memory abstraction with transient out-of-order execution (2018). https://foreshadowattack.eu/foreshadow-NG.pdf
4. Skorobogatov, S., Woods, C.: Breakthrough Silicon Scanning Discovers Backdoor in Military Chip, pp. 23–40. Springer, Heidelberg (2012)
5. Andreou, A., Bogdanov, A., Tischhauser, E.: Cache timing attacks on recent microarchitectures. In: 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 155–155 (2017)
6. Hu, W., Althoff, A., Ardeshiricham, A., Kastner, R.: Towards property driven hardware security. In: 2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV), pp. 51–56. IEEE, Piscataway (2016)
7. Hu, W., Ardeshiricham, A., Gobulukoglu, M.S., Wang, X., Kastner, R.: Property specific information flow analysis for hardware security verification. In: 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1–8 (2018)
8. Ma, H., He, J., Liu, Y., Zhao, Y., Jin, Y.: CAD4EM-P: security-driven placement tools for electromagnetic side channel protection. In: 2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST), pp. 1–6 (2019)

9. Tiwari, M., Wassel, H.M., Mazloom, B., Mysore, S., Chong, F.T., Sherwood, T.: Complete information flow tracking from the gates up. In: the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 109–120 (2009)
10. Bidmeshki, M., Makris, Y.: Toward automatic proof generation for information flow policies in third-party hardware IP. In: 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 163–168 (2015)
11. Zhang, D., Wang, Y., Suh, G.E., Myers, A.C.: A hardware design language for timing-sensitive information-flow security. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 503–516. ACM, New York, NY (2015)
12. Ardeshiricham, A., Hu, W., Marxen, J., Kastner, R.: Register transfer level information flow tracking for provably secure hardware design. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1691–1696 (2017)
13. Sefton, S., Siddiqui, T., Amour, N.S., Stewart, G., Kodi, A.K.: GARUDA: designing energy-efficient hardware monitors from high-level policies for secure information flow. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(11), 2509–2518 (2018)
14. Jiang, Z., Dai, S., Suh, G.E., Zhang, Z.: High-level synthesis with timing-sensitive information flow enforcement. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD), pp. 88:1–88:8. ACM, New York, NY (2018)
15. Pilato, C., Wu, K., Garg, S., Karri, R., Regazzoni, F.: TaintHLS: high-level synthesis for dynamic information flow tracking. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **38**(5), 798–808 (2019)
16. Ravi, P., Najm, Z., Bhasin, S., Khairallah, M., Gupta, S.S., Chattopadhyay, A.: Security is an architectural design constraint. Microprocess. Microsyst. **68**, 17–27 (2019)
17. Knechtel, J., Kavun, E.B., Regazzoni, F., Heuser, A., Chattopadhyay, A., Mukhopadhyay, D., Dey, S., Fei, Y., Belenky, Y., Levi, I., Güneysu, T., Schaumont, P., Polian, I.: Towards Secure Composition of Integrated Circuits and Electronic Systems: On the Role of EDA. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 508–513 (2020).
18. Pilato, C., Garg, S., Wu, K., Karri, R., Regazzoni, F.: Securing hardware accelerators: a new challenge for high-level synthesis. IEEE Embed. Syst. Lett. **10**(3), 77–80 (2018)
19. Deng, S., Gümüşoğlu, D., Xiong, W., Sari, S., Gener, Y.S., Lu, C., Demir, O., Szefer, J.: SecChisel framework for security verification of secure processor architectures. In: Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP), pp. 7:1–7:8. ACM, New York, NY (2019)
20. Ardeshiricham, A., Hu, W., Kastner, R.: Clepsydra: modeling timing flows in hardware designs. In: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 147–154 (2017)
21. Jin, Y., Guo, X., Dutta, R.G., Bidmeshki, M., Makris, Y.: Data secrecy protection through information flow tracking in proof-carrying hardware IP–Part I: framework fundamentals. IEEE Trans. Inf. Forensics Secur. **12**(10), 2416–2429 (2017)
22. Hu, W., Oberg, J., Irturk, A., Tiwari, M., Sherwood, T., Mu, D., Kastner, R.: Theoretical fundamentals of gate level information flow tracking. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **30**(8), 1128–1140 (2011)
23. Bidmeshki, M., Antonopoulos, A., Makris, Y.: Information flow tracking in analog/mixed-signal designs through proof-carrying hardware IP. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1703–1708 (2017)
24. Li, X., Tiwari, M., Hardekopf, B., Sherwood, T., Chong, F.T.: Secure information flow analysis for hardware design: using the right abstraction for the job. In: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS), pp. 8:1–8:7. ACM, New York, NY (2010)
25. Stroud, C.E., Wang, L.T., Chang, Y.W.: Introduction. In: Wang, L.T., Chang, Y.W., Cheng, K.T.T. (eds.) Electronic Design Automation, Chap. 1, pp. 1–38. Morgan Kaufmann, Boston (2009)

26. Oberg, J., Meiklejohn, S., Sherwood, T., Kastner, R.: Leveraging gate-level properties to identify hardware timing channels. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **33**(9), 1288–1301 (2014)
27. Hu, W., Mao, B., Oberg, J., Kastner, R.: Detecting hardware trojans with gate-level information-flow tracking. Computer **49**(8), 44–52 (2016)
28. Shakya, B., He, T., Salmani, H., Forte, D., Bhunia, S., Tehranipoor, M.: Benchmarking of hardware trojans and maliciously affected circuits. J. Hardware Syst. Secur. **1**, 85–102 (2017)
29. Hu, W., Oberg, J., Irturk, A., Tiwari, M., Sherwood, T., Mu, D., Kastner, R.: On the complexity of generating gate level information flow tracking logic. IEEE Trans. Inf. Forensics Secur. **7**(3), 1067–1080 (2012)
30. Wolf, C., Glaser, J.: Yosys - a free Verilog synthesis suite (2013). http://www.clifford.at/yosys/
31. Ameli, R.: Present Cipher Encryption IP Core (2011). https://opencores.org/ocsvn/present_encryptor/present_encryptor/trunk
32. Percival, C.: Cache missing for fun and profit. In: Proc. of BSDCan 2005 (2005)
33. Bernstein, D.J.: Cache-timing attacks on AES. VLSI Des. IEEE Comput. Soc. **51**(2), 218–221 (2005)
34. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side channel attacks. SIGARCH Comput. Archit. News **35**(2), 494–505 (2007)
35. Hu, W., Zhang, L., Ardeshiricham, A., Blackstone, J., Hou, B., Tai, Y., Kastner, R.: Why you should care about don't cares: exploiting internal don't care conditions for hardware trojans. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 707–713 (2017)
36. Satoh, A.: AES Encryption/Decryption Macro (2007). http://www.aoki.ecei.tohoku.ac.jp/crypto/
37. IWLS: IWLS Benchmarks Ver. 3.0 (2005). http://iwls.org/iwls2005/benchmarks.html
38. Berkeley Logic Synthesis and Verification Group: ABC: A System for Sequential Synthesis and Verification (2020). http://www.eecs.berkeley.edu/~alanmi/abc