Clepsydra: Modeling Timing Flows in Hardware Designs

Armaiti Ardeshiricham, Wei Hu and Ryan Kastner
Department of Computer Science and Engineering
University of California, San Diego
{aardeshi, weh040, kastner}@ucsd.edu

***Abstract*—Emergence of side channel security attacks has challenged the classic assumptions regarding what data is publicly available. As demonstrated repeatedly, statistical analysis of information collected by measuring completion time of hardware designs can reveal confidential information. Even though timing-based side channel leakage can be easily exploited to breach data privacy, conventional hardware verification tools are not yet suited to assess these vulnerabilities. To acquaint the hardware design process with formal security evaluations, we introduce a model for tracking timing-based information flows through HDL codes. Based on this model, we have developed Clepsydra, a tool for automatically generating circuitry for tracking timing flows and generic logical flows within hardware designs in two distinct channels. The circuit generated by Clepsydra can be analyzed by EDA tools to detect timing leakage or formally prove constant execution time. We present proofs regarding soundness and precision of the proposed model along with results of employing Clepsydra to verify security properties on a variety of hardware units including crypto cores, bus architectures, caches and arithmetic modules.**

I. Introduction

Variations in the time taken by a computational unit to generate results form a leakage channel that carries information regarding the data being processed. Many implementations of cryptographic algorithms are shown to have a varying runtime based on the Boolean value of secret key. Thus, an attacker familiar with the underlying algorithm can leverage statistical methods to extract the key from timing measurements [1], [2]. In security critical applications, timing based attacks have targeted various hardware units such as caches [3], [4], shared buses [5], [6] and floating point arithmetic units [7], [8]. Being both inexpensive and pervasive, timing-based side channel attacks are attracting more attention. They can be launched at low cost since the attacker merely needs to measure the execution time of the victim process without physical access to the design. Moreover, any application encompassing data-dependent optimizations is susceptible to such attacks.

Most methods to protect against hardware based timing leakage rely on manual inspection of the HDL code, e.g., looking for sources of timing variation such as branches conditioned on secret values or data-dependent requests sent to shared resources. This can be a lengthy and cumbersome task, and it provides no formal guarantee regarding the design's security. Furthermore, such analysis only inspects the design with respect to already-known attack vectors and falls short in providing resilience against all possible timing based threats. Exhaustively testing the design to capture timing variations is also becoming impractical due to scale of modern chips. As the complexity and prevalence of hardware designs grow, so does the need for automatic and formal analysis of security properties. Over the past decade, multiple research solutions have been proposed for incorporating security analysis into traditional hardware verification tools. Many of the proposed techniques [9], [10], [11], [12] enable the designers to verify security properties regarding confidentiality, integrity, and non-interference based on the notion of information flow tracking (IFT).

By modeling how labeled data moves through a system, IFT tools indicate if sensitive information flows to any part of the design. However, this indication is limited to a binary decision as the nature of detected flows are not specified. More specifically, IFT techniques cannot segregate functional flows from timing based ones. This could be problematic in many security applications where functional flow is expected as it is protected by encryption, while timing flows shall be eliminated. For example, in a crypto core, functional flow from the key to the ciphertext is expected while the designer needs to ensure that the time taken for the ciphertext to become available does not depend on the value of the secret key. Furthermore, it might be the case that the output is not directly accessible by untrusted parties but its timing footprint is public. For instance, completion time of a shared floating point arithmetic unit can reveal information regarding its input values, even if the output itself is hidden. Thus, to assess the security of the design with respect to side channel attacks, timing flows should be distinguishable from functional ones.

In this work, we show how timing flows can be precisely modeled, and introduce an IFT technique for capturing timing leakage of hardware designs. Our model is based on detecting and propagating sources of potential timing variations in the code by inspecting interfaces of design's registers. We introduce Clepsydra, which automatically generates the logic required for tracking timing flows and logical flows in arbitrary HDL codes. The logic generated by Clepsydra can be processed by conventional EDA tools in order to analyze timing properties of the design under test. As this logic is generated after statically analyzing all the execution paths in the design, it does not rely on activating the worst case execution path of the design in order to expose timing variation during verification.

Clepsydra is easily adoptable in the hardware design flow. It does not employ any additional HDL language features. And while it generates synthesizable logic that could be used for runtime detection, we envision its usage primarily during design time, when the Clepsydra logic is analyzed by EDA tools to verify the existence (or lack thereof) of timing-based properties. The Clepsydra logic is only used for analysis and is discarded before manufacturing, thus it imposes no additional runtime overhead. We show how to use Clepsydra to detect timing leakage in various existing architectures, or prove that they have constant execution time. More specifically, this paper provides the following contributions:

- Modeling timing-based information flows in hardware designs;
- Developing Clepsydra for automatic generation of digital logic for testing timing behaviour of hardware designs;
- Analyzing timing-based security properties of various hardware architectures using Clepsydra.

The rest of this paper is organized as follows. Section II summarizes how IFT-based methods are employed for hardware security analysis, and how we aim to improve them. Our proposed model for tracking timing flows is introduced in Sections III. In Section IV we elaborate implementation of Clepsydra, and present the experimental results

gathered from using Clepsydra to analyze security properties of various architectures in Section V. We provide a brief summary of related work in Section VI, and conclude the work in Section VII.

II. Background & Motivation

In this section we elaborate how IFT-based techniques enable security analysis, and argue why current tools are inadequate for establishing formal guarantees of timing based properties. We further point out how employing a more meticulous model for segregating different forms of logical flows can resolve this issue.

A. Security Properties

Isolation of different logical components is a primary security property that hardware designers seek to provide. Two major security properties can be enforced through isolation:

- **Confidentiality**: Preventing untrusted parties from observing secret information by isolating the units which process secret data. For example, in a cryptographic hardware we want to ensure that the secret key does not leak to public outputs as a result of design flaws, hardware Trojans or side channel leakage.
- **Integrity**: Preventing unauthorized parties from modifying sensitive information. For instance, the registers storing cryptographic keys should only be accessible by trusted sources.

In order to provide sound security guarantees, information flow must be analyzed through both data channels (also known as functional channels) and timing channels. The former ensures that data does not move among isolated components, while the latter certifies that the timing footprints of the isolated entities do not form a communication channel.

B. IFT & Hardware Security Verification

IFT techniques provide a systematic approach for verifying security properties related to integrity and confidentiality. This works by assigning security labels to different signals and tracking how these labels propagate through the system. Different security properties can be tested by defining the input labels and inspecting the output labels. Precision of an IFT technique, i.e. how closely the reported flows resemble the actual flows, is directly affected by the label propagation rules.

If the label propagation rules are not comprehensive enough to capture all forms of digital flows, the design might be inaccurately marked as secure. Information can flow in both explicit and implicit ways. In the explicit form, information flows to the output of an operation which is processing sensitive data. More subtly, data that controls conditional statements can implicitly affects the results. For instance, in the code `if(c) then x = y + z;` signal `x` is explicitly affected by signals `y` and `z`, and implicitly by signal `c`. For an IFT technique to be sound and free from false negatives, it should be capable of tracking both implicit and explicit flows.

Furthermore, label propagation rules should detect cases where flow of information is blocked. For example, if certain bits of the secret data is ANDed with zero, there will be no flow from those bits to the output of the AND gate. However, a conservative tracking rule, which assigns the highest security labels of the inputs to the output, marks all the output bits as sensitive. In contrast, a precise IFT tool, built upon stricter rules which take into account Boolean values of the operands and the operation functionality, can recognize absence of flows and avoid certain false positives [13].

C. Isolating Timing Flows

Existing IFT techniques track both *functional flows* and *timing flows* using the same set of labels and propagation rules. Thus, when a flow is detected, whether it is a functional flow or a timing flow, remains unknown. However, different applications necessitate different forms of isolation. For instance, both timing and functional isolation should be guaranteed when a cache is shared among mutually untrusting processes. But secure implementation of a cryptographic algorithm only requires elimination of timing channels as functional flows are protected by encryption. This property cannot be tested using IFT techniques which capture all forms of logical flows through a single set of labels. As the cipher is always affected by the secret key through functional flows, its security label will be raised to the security label of the key, independent of existence of timing flows. This significantly limits employment of IFT techniques for security analysis as similar scenarios happen in many applications where functional flows are inevitable but timing based flows should be eliminated.

Since conventional IFT techniques are designed for tracking all forms of logical flows, employing them to detect only timing flows results in a considerable number of false positives. As timing flows are a subset of information flows; a set of stricter propagation rules can be designed that work on a separate set of labels and track only timing flows while ignoring functional ones. In the next section we introduce a set of rules for detecting only timing flows and tracking them through the system.

III. Modeling Timing Flows

Timing flows exist from inputs to outputs of a circuit if the time that is taken for the outputs to become available depends on the Boolean values of the inputs. These flows can be exploited if the input signals causing them contain secret information, and the completion time of the unit can be measured by an untrusted party. For instance, consider a division unit, as shown in Fig 1, implemented via consecutive subtraction of the divisor from the dividend. The execution time of this algorithm depends on the input values as the number of subtractions is not fixed. This indicates that even if the Boolean value of the quotient is undisclosed, evaluating the execution time reveals information regarding the inputs. In this section we discuss how timing variations are represented in digital circuits, and develop a formal model for capturing them.

A. Characterizing Timing Flows

Completion time of a design is defined by the time when its output is updated to its final value. If no timing flow exists from input `X` to output `Y`, the time taken for `Y` to reach its final value should be constant as `X` changes. Thus, in order to detect timing flows, we need to determine whether or not the updates made to the outputs occur at constant timesteps. This can be addressed by detecting variations in the update time of all design variables, and tracking them to the final outputs. We discuss how this can be done for any arbitrary digital circuit by answering three questions: How are timing flows generated from a set of sensitive inputs? How does the flow propagate once generated? And lastly, what are the necessary conditions for blocking the flow of timing information and enforcing constant time execution? Since we are interested in detecting timing variations in terms of clock cycles, we need to analyze the design's registers and the signals which control them.

Generation of Timing Variation: Design's registers are written to by a set of data signals which are multiplexed by controllers at each cycle. Considering a register where none of its data or control

```
Module div (a, b, clk, go, q);
input [31:0] a, b;
input clk, go;
output [31:0] q;
reg [9:0] counter;
always @(posedge clk)
  if (done & go)  // reset…
  else if( !done)
     Decrease counter
     Shift the divisor
     if (dividend >= divisor)
        Update temp_quotient
        Subtract divisor from dividend

  if(counter==0)
     Write temp_quotient to out
     Set done

 // counter implementation
```

Fig. 1. Division algorithm based consecutive subtraction. The red box shows generation of timing variations, and the green box depicts their blockage.

signals has timing variation but might contain sensitive data, we want to figure out the circumstances under which timing variations occur at the register's output. If the register is definitely updated at each clock cycle, there will be no timing variations. However, if occurrence of updates are tentative, i.e. there is a degree of freedom for the register to hold its current value or get a new value, timing variation could occur. If the controller signal which is deciding the occurrence of the update is sensitive, the resulting timing variation will contain sensitive information as well.

Going back to the division example in Fig 1, the updates made to the register `temp_quotient` are conditioned on the input. Thus, based on the Boolean values of the input signals, this register might get its final value at different times. In Theorems 1 and 2, we show that detecting conditional updates caused by sensitive data soundly captures all timing flows while discarding functional-only ones.

Propagation and Blockage of Timing Variation: If any of the data or control signals of a register has cycle level variations, the variation can flow through the register. While simply propagating these flows soundly exposes all timing variations, it overestimates the flow when mitigation techniques are implemented to eliminate the variations. In other words, we need to be able to detect situations where timing variations are not observable at a register's output even though they are present at its input.

In division example in Fig 1, instead of directly writing the `temp_quotient` to the output, a wait period is taken before updating the output value. If the wait period is longer than the worst case execution time, the output gets its update at constant time steps. We will show in Theorem 3 that if there exists a non-sensitive control signal which *fully controls* the occurrence of updates to a register, it can block flow of timing variation from input to output of the register. *Fully controlling* control signal implies that the register gets a new value if and only if the controller gets a new value. Thus, the timing signature of the register output is identical to the control signal (with a single cycle delay), and is independent of its input. Implementing this policy reduces the number of false positives to some extent without imposing any false negative. In the division example, if the counter is large enough, the output value changes immediately after the `done` condition is updated, and keeps its old value while `done` does not change. Hence, all the variations to the final output are controlled by the `done` signal which is non-sensitive, indicating constant execution time with respect to inputs.

B. Theorems & Proofs

In the rest of this section we formally define IFT concepts and prove the claims we made earlier. We show that our model soundly discovers all potential timing channels by proving that detecting tentative updates of design's registers is adequate for exposing all timing variations, and the presence of non-sensitive fully controlling signals eliminates existing timing variations. We also prove that our model ignores functional-only flows and thus is more precise for analyzing timing-based properties compared to IFT techniques which capture all logical flows.

Definition 1. An **event** e over data set Y and time values T is shown as the tuple $e = (y, t)$ for $y \in Y$ and $t \in T$, where y and t can be retrieved by functions $val(e)$ and $time(e)$. If y is an $n-$dimensional vector, and t the number of clock ticks that has past, the inputs or outputs of a design with n ports can be represented by event e.

Definition 2. Trace $A(Y, n)$ represents n events $\{e_i\}_{i=1}^{n}$ over data set Y, which are ordered by time: $time(e_i) = time(e_{i+1}) + 1$.

Definition 3. For any trace $A(Y, n)$, its **distinct trace** d(A) is defined as the the longest sub-trace of A, where consecutive events have different values, and for any two consecutive events in A such that $val(e_i) \neq val(e_{i-1})$, e_i is in $d(A)$.

For example, for trace $A = \{(10, 1), (10, 2), (20, 3), (20, 4)\}$, its distinct trace is $d(A) = \{(10, 1), (20, 3)\}$ since the values only change at clock cycle 1 and 3.

Definition 4. Traces $A(X, k)$ and $A'(X, k)$ are **value preserving** with respect to set I if the only difference between their corresponding events e_i and e'_i is in the j-th element of the value vector such that $j \in I$.

In IFT analysis, we are interested in the effects of a set of sensitive variables by testing the design with respect to input traces which only differ in the sensitive inputs. This idea can be modeled by using value preserving traces where I is the set of sensitive inputs.

Definition 5. Output of an FSM F is **completely controlled** by input J if the FSM output is updated if and only if input J is updated.

Definition 6. For any set of wires W, **sensitivity label set** W_s and **timing label set** W_t indicate if W carries sensitive information or timing variation, respectively.

Definition 7. In a sequential circuit represented by the FSM $F = (X, Y, S, s_0, \delta, \alpha)$, a **functional-only flow** from a set of sensitive inputs I exists if there exist two value preserving (with respect to I) input traces $A(X, k)$ and $A'(X, k)$ such that when fed to the FSM, the timesteps of the distinct traces of the outputs are equivalent, while the values of corresponding events varies. Stated formally: if $B = \alpha(A, s_0)$ and $B' = \alpha(A', s_0)$, then:

$$\forall e_i, e'_i \in d(B), d(B')\ time(e_i) = time(e'_i) \text{ and}$$

$$\exists e_j, e'_j \in d(B), \in d(B') \text{ such that } val(e_j) \neq val(e'_j)$$

Definition 8. In a sequential circuit represented by the FSM $F = (X, Y, S, s_0, \delta, \alpha)$ a **timing flow** from a set of sensitive inputs I exists if there exist two value preserving (with respect to I) input traces $A(X, k)$ and $A'(X, k)$ such that when fed to the FSM, the timestep of the distinct traces of the outputs are not equivalent. Stated formally, if $B = \alpha(A, s_0)$ and $B' = \alpha(A', s_0)$, then:

$$\exists e_j, e'_j \in d(B), \in d(B') \text{ such that } time(e_j) \neq time(e'_j)$$

Definition 9. For a combinational logic function $f : X \rightarrow Y$ its **flow tracking** function $f_s : X \times X_s \rightarrow Y_s$ determines whether or not sensitive inputs affect the outputs. If $f(x_1, ..., x_n) = (y_1, ..., y_m)$ then $f_s(x_1, ..., x_n, x_{1_s}, ..., x_{n_s}) = (y_{1_s}, ..., y_{m_s})$, where if set of sensitive inputs $\{x_j | x_{j_s} = 1\}$ can affect value of y_i then $y_{i_s} = 1$ indicating **information flow** exists from the sensitive inputs to output y_i.

Definition 10. For a sequential logic function $f : X \times S \rightarrow Y$, where X, S, Y are the inputs, states, and the outputs, the **time tracking** function $f_t : X \times X_s \times X_t \times S \times S_s \times S_t \rightarrow Y_t$ determines if a set of inputs tainted with sensitive information or timing variation can affect timing variations of the output. If $f(x_1, x_2, ..., x_n, s_1, s_2, ..., s_l) = (y_1, y_2, ..., y_m)$ then $f_t\ (x_1, ...,\ x_n,\ x_{1_s}, ...,\ x_{n_s},\ x_{1_t}, ...,\ x_{n_t}, s_1, ...,\ s_l,\ s_{1_s},\ ...,\ s_{l_s},\ s_{1_t}, ...,\ s_{l_t}) = (y_{1_t}, ...,\ y_{m_t})$, where if a set of tainted inputs $\{x_j | x_{j_s} \vee x_{j_t} = 1\}$ can affect whether or not state s_i is updated then $s_{i_t} = 1$ and we say **timing flow** exists from the tainted inputs to output s_i.

Theorem 1. *The time tracking logic F_t of FSM F captures timing flows of the FSM.*

Proof. To prove this theorem we show that the existence of a timing flow reduces to variations in occurrence of updates to the output, and therefore is captured by F_t.

If a timing flow exists with respect to the set of tainted inputs I, based on Definition 8 there exist value preserving traces $A(X, k), A'(X, k)$ such that :

$$\text{if } B = \alpha(A),\ d(B) = (e_1, e_2, ..., e_m)$$

$$\text{and } B' = \alpha(A'), d(B') = (e'_1, e'_2, ..., e'_m) \text{ then :}$$

$$\exists j \in [1 : m] \text{ such that } time(e_j) \neq time(e'_j)$$

Consider n to be the smallest index such that $time(e_n) \neq time(e'_n)$. Without loss of generality we can assume that $time(e_n) = t_n$ and $time(e'_n) = t_n + d,\ d > 0$. Basically, we are assuming n to be the time when the new value of trace B' appears with delay d compared to trace B. We can write the elements of these two traces up to the nth element:

$$d(B) = (v_1, t_1),\ (v_2, t_2),\ ...,\ (v_{n-1}, t_{n-1}),\ (v_n, t_n)$$

$$d(B') = (v'_1, t_1),\ (v'_2, t_2),\ ...,\ (v'_{n-1}, t_{n-1}),\ (v'_n, t_n + d)$$

$$\forall i \in [2 : n] :\ v_i \neq v_{i-1} and\ v'_i \neq v'_{i-1} \text{ Based on Definition3.}$$

The following observations can be made based on the above traces:

$$(a) : (v_n, t_n) \text{ and } (v_{n-1}, t_n - 1) \in B$$

$$(b) : (v'_{n-1}, t_n) \text{ and } (v'_{n-1}, t_n - 1) \in B'$$

From (a) we can infer that value of trace B is updated at time t_n from v_{n-1} to v_n while equation (b) shows that value of trace B' is not updated at time t_n and is equal to v'_{n-1}

By Definition 4, all input events remain the same $\forall i \notin I$, meaning that the only difference between them is the sensitive inputs. Thus, the difference in the update to the output is caused by the set of sensitive inputs and is captured by F_t based on Definition 10. □

Theorem 2. *The time tracking logic F_t of FSM F does not capture functional-only flows of the FSM.*

Proof. We prove this theorem by showing that the existence of functional only flows will not impose any variations on the occurrence of updates to the output, and thus will not be captured by F_t.

If a functional-only flow exists with respect to the set of sensitive inputs I, then based on Definition 7 there exists value preserving traces $A(X, k), A'(X, k)$ such that:

$$\text{if } B = \alpha(A), d(B) = (e_1, e_2, ..., e_m) \text{ and}$$

$$B' = \alpha(A'), d(B') = (e'_1, e'_2, ..., e'_m) \text{ then :}$$

$$(1)\ \forall i \in [1 : m] :\ time(e_i) = time(e'_i)$$

$$(2)\ \exists j \in [1 : m] :\ \text{such that } val(e_j) \neq val(e'_j)$$

We claim that there is no time t_n such that the value of one of the traces is updated while the other one is not. Without loss of generality, we show that there is no time t_n where the value of B is updated but the value of B' remains the same. We prove this via proof by contradiction.

Contradiction hypothesis: At time t_n, trace B is updated while trace B' holds its value.

Let v_n and v_{n-1} be the values of trace B at times t_n and $t_n - 1$ respectively. Based on the contradiction hypothesis, (v_n, t_n) is an event in B. Hence, $d(B)$ contains an event e_i such that $time(e_i) = t_n$. Let us assume that e_i is the ith element of $d(B)$. Similarly, assume values of trace B' in times t_n and $t_n - 1$ are v'_n and v'_{n-1}. Based on the contradiction hypothesis we know (v'_n, t_n) is not an event in $d(B)$ since B is not updated at this time. Thus, $d(B')$ does not have any event e'_j which timestep is t_n. Hence, if we pick the ith element of $d(B')$, called e'_i, then $time(e'_i) \neq t_n$. So:

$$\exists i\ in [1 : m] \text{ such that } time(e_i) \neq time(e'_i)$$

This is contradictory to the definition of functional-only flows since there could be no time t_n where the values of one of the traces is updated while the other one is not. Based on Definition 10 this is not captured by FSM_t. □

Theorem 3. *If FSM F is completely controlled by input J such that $J \notin I$, then no timing variation is observable at the output of FSM F as a result of processing traces which are value preserving with respect to set I.*

Proof. We will prove this theorem via proof by contradiction.

Contradiction hypothesis: there exist value preserving (with respect to I) traces $A(X, k)$ and $A'(X, k)$ which impose timing flow at the output of FSM F which is completely controlled by input $J \notin I$.

Based on Definition 8:

$$\text{let } B = \alpha(A, s_0),\ \text{and } d(B) = \{e_1, e_2, ..., e_m\}$$

$$\text{let } B' = \alpha(A', s_0),\ \text{and } d(B') = \{e'_1, e'_2, ..., e'_m\}$$

$$\exists i \in [1, m] \text{ such that } time(e_i) \neq time(e'_i)$$

let n be the smallest index in the above equation such that $time(e_i) \neq time(e'_i)$. Without loss of generality we can assume $time(e_i) = t_n$ and $time(e'_i) = t_n + d$. We can write elements of $d(B)$ and $d(B')$ up to the nth element:

$$d(B) = (v_1, t_1), (v_2, t_2), ..., (v_{n-1}, t_{n-1}), (v_n, t_n)$$

$$d(B') = (v'_1, t_1), (v'_2, t_2), ..., (v'_{n-1}, t_{n-1}), (v'_n, t_n + d)$$

Using Definition 3, we make the following observations:

$$(a) : (v_n, t_n) \text{ and } (v_{n-1}, t_n - 1) \in B$$

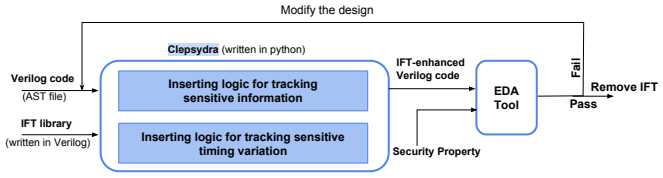

Fig. 2. Clepsydra overview

$$(b) : (v'_{n-1}, t_n) \text{ and } (v'_{n-1}, t_n - 1) \in B'$$

The above equations specifies that trace B has been updated at time t_n while B' is not updated. Lets denote the sub-trace of the fully controlling input J from traces A and A' with j and j', respectively. Based on Definition 5, equation (a) indicates that input j is updated at time t_n, and from (b) we know input j' is not updated at this time. This is a contradiction as the only difference between input traces A and A' are with respect to set I, and since $J \notin I$ then j and j' should be identical. □

IV. Clepsydra Implementation

In this section, we describe implementation details of Clepsydra. As shown in Fig 2, the input to Clepsydra is a hardware design described by its abstract syntax tree (AST) which is obtained by parsing its HDL representation. As output, Clepsydra generates a synthesizable Verilog code which has all the functionalities specified in the original design, alongside the complementary logic for propagating both timing based and generic information flows from design inputs to its outputs. The tracking logic is realized in two steps: 1) extending each variable in the design with labels *sensitivity level* and *timing level* which indicate if the variable carries sensitive information or timing variation, respectively; and 2) inserting logic for updating these labels as their corresponding variables change. The code generated by Clepsydra is then given to EDA tools for security analysis. Security properties are assessed by specifying labels of the input variables and observing the output labels after simulation, formal verification or emulation. If the output labels comply with the designers' intention, the tracking logic is discarded and the original design can be used for fabrication. In case of violating the security properties, the original design should be modified, fed to Clepsydra, and retested.

Clepsydra enables analyzing timing behavior of a design with respect to any arbitrary subset of its inputs which are marked as sensitive. This facilitates modeling a variety of security properties. For example, constant time execution can be tested by marking all the inputs as sensitive. But in many scenarios we are only interested in constant execution time with respect to certain inputs. For instance, when a cache is shared between mutually untrusting processes, timing variations caused by accesses from sensitive data is exploitable. However, variations due to cache conflicts on non-sensitive data are not valuable to the adversary. indiscriminately eliminating all timing variations results in disabling the cache as a whole. Moreover, many mitigation techniques are based on randomizing timing variations. To differentiate benign variations from sensitive ones, we should inspect the source of the variations. This is done by tracking sensitive data throughout the circuit, and extracting the sensitive timing variations from them.

A. Tracking Sensitive Information

Sensitive information affects computation result through both the data path and the control path, creating explicit and implicit flows. To

Input : Verilog code AST file
Output: IFT-enhanced Verilog code

Preprocessing:
1 **for** *each register r* **do**
2 | n: number of the paths to r;
3 | m: number of the controllers of r;
4 | **if** $n \neq 2^m$ **then** r_bal=0;
5 | **else** r_bal=1;
end
6 **for** *each conditional assignment a* **do**
7 | traverse CFG;
8 | a_con = list of controllers;
end
Logic insertion:
9 **for** *each variable* $x[n:0]$ **do**
10 | define $x\_s[n:0]$, $x\_t[n:0]$;
end
11 **for** *each DFG operation* $a = b\ op\ c$; **do**
12 | instantiate IFT-enhanced operation:
13 | $op\_IFT\ (a,\ a\_s,\ b,\ b\_s,\ c,\ c\_s)$;
14 | insert time tracking logic:
15 | $a\_t = b\_t \mid c\_t$;
end
16 **for** *each controller c* **do**
17 | insert buffer $c\_buf <= c$;
end
18 **for** *each conditional assignment* $A <= B$ **do**
19 | insert implicit IFT logic:
20 | $A\_s <= B\_s \mid c\_s;\ \forall c \in A\_con.$
21 | $A\_t <= (B\_s\ \&\ !A\_bal)$;
22 | $\mid (B\_t\ \&\ !(c$ is non-sensitive and fully controlling);
end

Algorithm 1: Tracking logic generation

detect explicit flows, Clepsydra replaces each data path operation with an IFT-enhanced version of it which is available as a Verilog module in a predesigned IFT library (lines 11-13 of Algorithm 1). Each IFT-enhanced operation receives the original inputs of the operation along with their sensitivity labels, and computes the outputs of the operation as well as their sensitivity labels. A simple example of replacing an add operation with an IFT-enhanced module is shown in the first line of Fig 3. Various complexity-precision trade-offs for the tracking logic can be explored by modifying the label propagation rules of the IFT-enhanced modules [13].

To track whether or not an assignment is implicitly affected by sensitive data, we need to figure out if its execution depends on any sensitive variable. To do so, Clepsydra extracts the design's control flow graph from its AST representation, and constructs a list of control signals for each conditional assignment (lines 6-8 of Algorithm 1). Next, based on the variables in the list and their sensitivity labels the logic for tracking the implicit flow is generated and added to the explicit flow tracking logic (lines 18-20 of Algorithm 1).

B. Tracking Timing Flows

Clepsydra inserts logic components at each register interface to detect if any timing variation is generated from sensitive data, and whether or not existing variations from the register input flow to the register's output (lines 21-22 of Algorithm 1). As we proved earlier, the necessary condition for formation of timing variation is

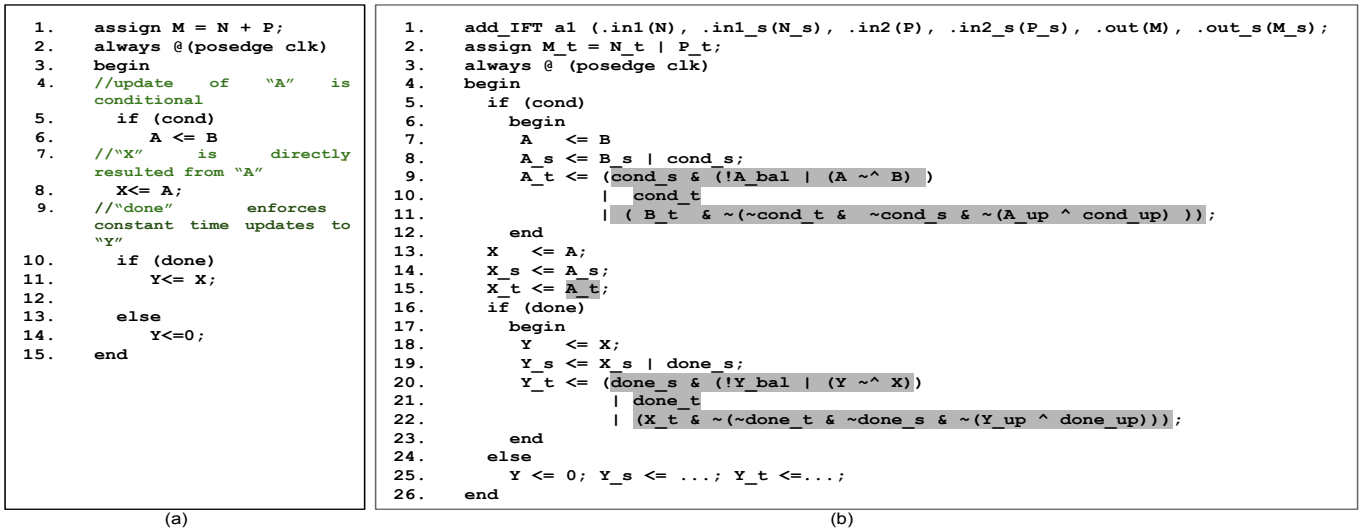

Fig. 3. (a) Original Verilog code, (b) IFT-enhanced Verilog code generated by Clepsydra

existence of a register which update depends on sensitive values. To identify these cases, we need to determine if a register has the flexibility of selecting between getting a new value and holding its current value. To examine this property for each register in the design, Clepsydra statically enumerates all the paths in which the register is written to, and compares it with the total number of paths that the controllers of that register can theoretically activate. If these two numbers are unequal, a bit which indicates the updates to the register are tentative is set (lines 1-5 of Algorithm 1). Such analysis on a Verilog code is relatively easy compared to software languages since multiple writes to a register are modeled as a single multiplexer with `n` data inputs and `m` control inputs. Tentative update scenarios happen if $n \neq 2^m$ which indicates that the multiplexer has direct feedback from its output to its own input. To illustrate this idea, consider the Verilog code written in Fig 3(a) and the IFT-enhanced Verilog code generated by Clepsydra in Fig 3(b). Highlighted parts in lines 9 and 20 show the logic responsible for detecting generation of timing flows. Values of `A_bal` and `Y_bal` are statically decided by Clepsydra after analyzing the branches in the original code. An XNOR function is also added to detect cases where the register gets its value from a different variable without actually getting updated. Even though such scenarios are rare in actual designs, the logic for detecting them is added to ensure capturing cases where tentative updates are disguised by renaming the variables.

Once generated, timing variations flow directly through the subsequent registers unless special mechanism for eliminating the variations is implemented. Register `X` in Fig 3 directly gets its value from register `A`, thus if any timing variation is present as the output of `A`, it will unconditionally flow to `X`. As shown in the second line of Fig 3(b), timing variation directly flows through combinational logic since we are interested in cycle level precision.

As we proved in the previous section if there exist any non-sensitive control signal which fully controls the updates to the register, it can block flow of timing information. To detect existence of fully controlling signals for conditional assignments to registers, Clepsydra inserts XOR gates for comparing occurrence of updates. This logic is shown in lines 11 and 22 of Fig 3(b). The XOR function indicates that updates of the output register and its controller are synchronous. And the inverters specify that the controller does not have any sensitive information. This logic is responsible for preventing overestimating the flow to some extent as depicted by the AND function. The logic behind `Y_up`, `A_up`, `B_up` and `done_up` are not shown in the figure for simplicity, but they are computed by XORing the current state with the next state. Since, the updates to the control signals are observable at the register output with one cycle delay, Clepsydra inserts buffers to store control values from the previous cycle in order to compute whether or not they have been updated in the previous cycle (lines 16-17 of Algorithm 1).

V. Experimental results

In this section we elaborate how various security properties are specified based on notion of IFT, and verified on Clepsydra logic. Table I lists the hardware designs we tested along with the assessed security properties. For each design, we briefly discuss the architectural features which create timing channels, the attack model for exploiting them, the existing mitigation techniques, and the results of our security analysis. For all of our experiments, we obtained the AST representation of plain Verilog code by parsing it using Yosis tool [14], and employed Clepsydra to generate tracking logic. On the IFT-enhanced code generated by Clepsydra, effect of input `X` on timing behaviour of output `Y` can be inspected by setting the input signal `X_s` as high, and observing the value of the output `Y_t` after simulation or formal verification.

A. Arithmetic Modules

For the first set of experiments, we sought proving constant time properties of arithmetic units, as variation in completion time of these units can be exploited to extract information regarding the input [8], [7]. We tested a fixed point math library from the Opencores website [15], which is supposed to run in constant time as claimed by its designers. In order to verify this claim, we marked data inputs of each unit as sensitive and observed the timing labels of the outputs.

The multiplication unit is based on accumulating partial products at each cycle. Thus, if the MSB bits of the multiplier are zero the result will be available faster since the partial products in the last cycles are zero. The output `ready` signal of the design is set after a counter reaches zero. After analyzing this design, we noticed that while the `ready` output is free from timing variations, the product result is not. This indicates that the result could potentially become available

TABLE I
SUMMARY OF THE DESIGNS AND SECURITY PROPERTIES TESTED USING CLEPSYDRA

Design	Security Property	Security Specification	Result
Division unit	Result is ready in constant time	set dividend_s, divisor_s=H, assert(quotient_t==L)	Proved
multiplication unit	Result is ready in constant time	set multiplier_s, multiplicand_s=H, assert(product_t==L)	Provable after debugging
cache	Isolation bw accesses to the same line	set index_s=H, assert(data_t==L)	violated
PLcache	Isolation bw accesses to the same line	set index_s=H, assert(data_t==L)	Provable if sensitive data is preloaded
RPcache	Isolation bw accesses to the same line	set index_s=H, assert(data_t==L)	Provable if RNG is secure
WISHBONE, round robin	Timing isolation bw cores	set request1_s=H, assert(ack2_t==L)	Violated
WISHBONE, TDMA	Timing isolation bw cores	set request1_s=H, assert(ack2_t==L)	Proved
WISHBONE, TDMA+group	Timing isolation bw cores	set request1_s=H, assert(ack2_t==L)	Provable bw different groups
AES	Cipher is ready in constant time	set key_s, plaintext_s=H, assert(cipher_t==L)	Proved
RSA	Cipher is ready in constant time	set key_s, plaintext_s=H, assert(cipher_t==L)	Violated

before the `ready` signal is raised. In order to eliminate this flow, we modified the design by adding a register which blocks the working result to write to the final output before the counter resets. After this modification, we could formally prove that the design runs in constant time using Questa Formal Verification tool. The division unit, similar to the example we had throughout the paper, is implemented by subsequently subtracting the divisor from the dividend. Similar to the multiplication unit, a wait state is responsible for enforcing constant time updates at the final outputs. This time no timing variation was detected by our analysis as all the output variables, including the result itself, are controlled by the wait state.

This set of experiments showed that our model is capable of isolating different forms of flows, and proving absence of timing flows while functional flows exist. Furthermore, it shows that the generated tracking logic is precise enough to detect cases where timing variations are eliminated by delaying all the updates as long as the worst case scenario.

B. Cache Implementations

Cache-based side channel attacks have been repeatedly employed to break software implementations of ciphers such as RSA and AES. These attacks target implementations which use pre-computed values that are stored in the cache and accessed based on the value of the secret key. Thus, an attacker who is capable of extracting the cache access pattern of the process running the encryption can deduce information regarding the key. Percival [4] has shown that an adversarial process sharing the cache with the OpenSSL implementation of the RSA cipher can retrieve cache access pattern of the victim process by inducing collisions in the cache. By remotely attacking AES implementation of the OpenSSL protocol, Bernstein [3] showed that timing channels can be exploited even when the cache is not shared with an untrusted process. In his attack, Bernstein exploited the cache collisions between different requests by the victim process itself to reveal the encryption key. While these attacks vary substantially in terms of implementation, they all exploit the timing variations from the cache collisions. Several cache designs have been proposed to bar index value of sensitive accesses to affect the time that it takes for the cache to retrieve data in later cycles. We have used Clepsydra to inspect timing flows in an unsecure cache and two secure architectures, PLcache and RPcache, introduced in [16]. To model timing leakage via external interference, we consider two processes with isolated address spaces sharing the same cache. Marking indexes of accesses made by one process as sensitive, we want to figure out if the data read by the other process contain timing variation. The internal interference scenario is modeled with a single process and inspecting if marking certain indexes as sensitive causes timing variation when the same lines are read with different tags.

PLcache eliminates leakage channel by letting processes to lock their data in the cache and disabling their eviction. Since sensitive data can no longer be evicted, it cannot affect the timing signature of the system. We implemented the PLcache and acquired its IFT-enhanced tracking logic from Clepsydra to test if this partitioning scheme eliminates the flow. Based on our analysis, if data with sensitive indexes is preloaded to the cache and locked, there will be no information leakage as the result of later accesses to the locked lines. However, this result is based on assuming that the preloading stage is not sensitive itself.

Next, we tested the RPcache which randomly permutes the mapping of memory to cache addresses to eliminate any exploitable relation between the collisions. When external interference between untrusting processes are detected, PRcache randomly chooses a cache line for eviction . Thus, the attacker cannot evict the victim's process sensitive information and observe whether or not that causes delay later on. In case of internal interference, collisions are handled by directly sending the data from the colliding access to the processor and randomly evicting another line. Our analysis showed that RP-cache eliminates timing variations assuming that the inputs to the random number generator are not sensitive.

C. Bus Architectures

Another source of timing channel in hardware designs arises when different units are connected over a shared bus. In such scenarios, cores that are supposed to be isolated can covertly communicate by modulating the access patterns to a shared resource and affecting the time when other cores can use the same resource. Using Clepsydra, we have inspected presence of timing flows when WISHBONE interconnect architecture is used to arbiter accesses on an SoC. To access a shared resource over WISHBONE, the master core sends a `request` signal, and waits for the arbiter to send back an `ack` signal. Timing channel between different cores can be assessed by marking the `request` signal sent by one core as sensitive and observing the timing label of the `ack` signal sent to the other core in later requests. We have tested this scenario for the original WISHBONE arbiter and two modified versions of it.

The original WISHBONE arbiter, implemented by the Opencores community [17], is based on a round robin algorithm. Our experiments revealed existence of timing channel between the cores connected over this architecture as the grant given to one core

depends on the former requests sent to the arbiter. In order to eliminate the channel, we replaced the round robin arbiter with a TDMA scheme and retested the design. In this case, no timing channel is detected as the grants are given based on a counter. In order to improve the efficiency of the arbiter, we tested a more flexible scenario where the cores are divided into two different groups. The cores within the same group are arbitrated based on a round robin algorithm, while the two groups are time multiplexed. This time our experiments showed that the two groups are isolated from each other while timing channel exists between elements of the same group.

D. Crypto Cores

Lastly, we tested timing variation in hardware implementation of RSA and AES ciphers. Since the ciphertext is computed from the key, functional flow from the key to the cipher is inevitable. Thus, existing IFT tools cannot be leveraged to inspect existence of timing channels which are not intended by the designer. Here, we have tested existence of timing channel in two ciphers from the Trusthub benchmarks [18]. We have assessed timing flow from the secret key to the ciphertext by marking the key bits as sensitive and observing the timing label of the output. Using Questa Formal Verification tool we could prove that the AES core runs in constant time. However, for the RSA core, timing flow was detected from the secret key to the cipher as a result of insecure implementation of the modular exponentiation step. We have left comparing timing leakage of different RSA architectures and the effectiveness of the proposed mitigation techniques as our future work.

VI. Related Work

Over the past decade, multiple tools have been developed for tracking information flows through hardware designs. Tiwari et al [12] implemented a microprocessor which dynamically tracks information flows through the shadow logic added for each gate. Due to the huge overhead in terms of power, area and performance, gate level information flow tracking (GLIFT) has been used mostly for test and verification during design time [19]. Oberg et al [20] has proved that inserting shadow logic at the gate level detects all information flows including timing and functional flows.

At a higher level of abstraction, multiple secure HDL languages have been introduced to enable designing provably secure hardware. Caisson [21] and Sapper [22] are both FSM based languages extended with a type system. Using these languages, the designer defines a security label for each variable and restricts information flows by controlling the transactions between the states. SecVerilog [9] adds a type system to the Verilog language for representing different security levels. SecVerilog users are required to define a type for each variable in the design based on the property they wish to enforce. SecVerilog type system statically verifies that the user defined types and the IFT property comply. While these tools facilitate secure hardware design, they require redesigning the hardware using a new language. VeriCoq [23] automatically transfers HDL codes to Coq representation where security properties can be tested on annotated code. While all these methods employ conservative tracking rules by raising each signal's label to the highest label of its predecessor signals, RTLIFT [11] allows for more flexible flow tracking rules. Based on the precision level specified by the user, RTLIFT generates the flow tracking logic for the design under test, which can be used to analyze different security properties.

As stated earlier, all of the mentioned IFT tools track all logical information flows through a single channel and leave the nature of the detected flows as unspecified. In this work, we presented a method for tracking timing flows and generic information flows in two separate channels in order to enable analysis of a wider range of security properties.

VII. Conclusion

In this work we presented a model for tracking timing-based information flows in hardware designs. We formally proved that our model soundly captures all timing variations and precisely separates timing flows from logical flows. We introduced an IFT tool, Clepsydra, which automatically generates tracking logic based on the proposed model. Clepsydra facilitates hardware security verification by enhancing plain HDL codes with synthesizable logic on which variety of security properties can be tested using conventional EDA tools. In our experiments, we leveraged Clepsydra to detect timing channels in different architectures, and prove absence of timing leakage when mitigation techniques such as randomization, partitioning or delaying till the worst case execution are implemented.

VIII. Acknowledgement

This material is based upon work supported by the National Science Foundation under grant no. CNS-1527631 and CNS-1563767.

References

[1] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," *Advances in Cryptology - CRYPTO'96*.
[2] W. Schindler, "A timing attack against rsa with the chinese remainder theorem," in *International Workshop on Cryptographic Hardware and Embedded Systems*, 2000.
[3] D. J. Bernstein, "Cache-timing attacks on aes," 2005.
[4] C. Percival, "Cache missing for fun and profit," 2005.
[5] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of computer security*, 1992.
[6] J. Oberg *et al.*, "Information flow isolation in I2C and USB," in *Design Automation Conference (DAC)*, 2011.
[7] O. Aciicmez and J.-P. Seifert, "Cheap hardware parallelism implies cheap security," in *Fault Diagnosis and Tolerance in Cryptography. Workshop on*. IEEE, 2007.
[8] M. Andrysco *et al.*, "On subnormal floating point and abnormal timing," in *IEEE Symposium on Security and Privacy*, 2015.
[9] D. Zhang *et al.*, "A hardware design language for timing-sensitive information-flow security," in *ACM SIGARCH Computer Architecture News*, 2015.
[10] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *2012 IEEE 30th VLSI Test Symposium (VTS)*.
[11] A. Ardeshiricham *et al.*, "Register transfer level information flow tracking for provably secure hardware design," in *Proceedings of the 2017 Conference on Design, Automation & Test in Europe*.
[12] M. Tiwari *et al.*, "Complete information flow tracking from the gates up," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.
[13] W. Hu *et al.*, "Imprecise security: quality and complexity tradeoffs for hardware information flow tracking," in *Proceedings of the 35th International Conference on Computer-Aided Design*.
[14] "Yosys Open SYnthesis Suite," http://github.com/cliffordwolf/yosys.
[15] https://opencores.org/project,verilog_fixed_point_math_library.
[16] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ACM SIGARCH Computer Architecture News*, 2007.
[17] https://opencores.org/opencores,wishbone.
[18] https://www.trust-hub.org/.
[19] M. Tiwari *et al.*, "Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security," in *ACM SIGARCH Computer Architecture News*. ACM, 2011.
[20] J. Oberg *et al.*, "Leveraging gate-level properties to identify hardware timing channels," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.
[21] X. Li, M. Tiwari *et al.*, "Caisson: a hardware description language for secure information flow," in *ACM SIGPLAN Notices*.
[22] X. Li, V. Kashyap *et al.*, "Sapper: A language for hardware-level security policy enforcement," in *ACM SIGARCH Computer Architecture News*.
[23] M.-M. Bidmeshki and Y. Makris, "Toward automatic proof generation for information flow policies in third-party hardware ip," in *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*.