ATLaS: Automatic Detection of Timing-based Information Leakage Flows for SystemC HLS Designs

Mehran Goli
Institute of Computer Science, University of Bremen
Cyber-Physical Systems, DFKI GmbH
28359 Bremen, Germany
mehran@uni-bremen.de

Rolf Drechsler
Institute of Computer Science, University of Bremen
Cyber-Physical Systems, DFKI GmbH
28359 Bremen, Germany
drechsler@uni-bremen.de

ABSTRACT

In order to meet the time-to-market constraint, *High-level Synthesis* (HLS) is being increasingly adopted by the semiconductor industry. HLS designs, which can be automatically translated into the *Register Transfer Level* (RTL), are typically written in SystemC at the *Electronic System Level* (ESL). Timing-based information leakage and its countermeasures, while well-known at RTL and below, have not been yet considered for HLS. The paper makes a contribution to this emerging research area by proposing ATLaS, a novel timing-based information leakage flows detection approach for SystemC HLS designs. The efficiency of our approach in identifying timing channels for SystemC HLS designs is demonstrated on two security-critical architectures which are shared interconnect and crypto core.

ACM Reference Format:
Mehran Goli and Rolf Drechsler. 2021. ATLaS: Automatic Detection of Timing-based Information Leakage Flows for SystemC HLS Designs. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394885.3431591

1 INTRODUCTION

The emergence of side-channel security attacks has disputed the validity of the classic assumptions about what data can be publicly available. Among the existing side-channel security attacks, timing-based attacks are more interesting for attackers as they only need to measure the execution time of the victim process without physical access to the design. Thus, attackers can access secret data at a very low cost and effort. The security critical modules of a given design that can be vulnerable against timing-based attacks are various hardware modules such as crypto cores [5], shared interconnects, and arithmetic modules [2, 8].

The threat of information leakage by means of timing-based side-channel attacks is based on the idea that, in a given design, the time taken by a (computational) module to generate the final results may be different regarding the data being processed. These time variations can form a leakage channel that carries the sensitive information regarding the data being processed and opens doors for attackers to access sensitive data. Many implementations of cryptographic algorithms are demonstrated to have a different execution time based on the value of the secret key. Thus, attackers who are familiar with the underlying algorithms can take advantage of statistical approaches to extract the key by measuring the execution time [18].

Since the cost of fixing any security flaws (e.g., timing-based information flows) increases with the stage of development, the detection process should be performed before production/manufacturing and ideally as early as possible. *High Level Synthesis* (HLS) has recently emerged as an alternative design entry to the *Register Transfer Level* (RTL). HLS designs, usually developed using SystemC language [1] at the *Electronic System Level* (ESL) [10, 11, 16]. HLS designs can be automatically synthesized into RTL and the quality of the generated RTL models is mostly comparable to hand-written RTL for the same functionality with much shorter development time [7, 13, 20]. Due to the flexibility in generating multiple variants of the same design, more and more third-party *Intellectual Properties* (IPs) are expected to be delivered as SystemC HLS designs. As a consequence of this decentralization, modern *System-on-Chips* (SoCs) are notoriously insecure where third-party IPs, in particular, can be used as a vehicle for exploiting the secret data.

To protect against timing-based attacks, manual analysis of the source code, e.g., by looking for sources of timing variation such as branches conditioned on secret values or data-dependent requests sent to shared resources is not sufficient. This can be a very time-consuming and error-prone task that requires lots of manual effort by designers. Moreover, testing the design to capture timing variations is also becoming impractical due to the scale of modern SoCs. As the complexity of hardware designs increase, the need for automatic analysis of security threats is inevitable. Hence, alternative approaches have been developed in the last decade [3, 21, 24, 25, 28]. However, these approaches are only applicable at the abstraction levels of RTL and below (e.g., gate or transistor level).

In this paper, we focus on the detection of timing channels in SystemC HLS designs. We present **ATLaS**, an **A**utomatic Detection of **T**iming-based Information **Lea**kage Flows for **S**ystemC HLS Designs. ATLaS is based on the *Information Flow Tracking* (IFT) technique for capturing timing flows of HLS designs and consists of two main phases: 1) information extraction and 2) timing-based data flow analysis. In the first phase, we build on the flexible Clang compiler [19] to statically analyze the *Abstract Syntax Tree* (AST) of a given SystemC design, to extract the data path and control flow of the design, and to formally represent them in a set of well-structured formats. In the second phase, we perform a data flow

ASPDAC '21, January 18–21, 2021, Tokyo, Japan

ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3431591

```
1  SC_MODULE (Divider){
2   sc_in<bool>          clk, rst, ready;
3   sc_in<sc_uint<8>>    divisor, dividend;
4   sc_out<sc_uint<8>>   quotient, residue;
5   sc_int<8> counter = 256;
6   void run();
7   SC_CTOR (Divider){
8   SC_METHOD(run);
9    reset_signal_is(rst,true);
10   sensitive << clk.pos();}
11  };
12 void Divider::run(){
13  sc_uint<8> quotient_temp, dividend_temp;
14  /*...*/
15  if (ready.read() && (!rst.read())){
16   dividend_temp = dividend.read();
17  if (dividend_temp >= divisor.read()){
18   dividend_temp = dividend_temp-divisor.read();
19   quotient_temp = quotient_temp + 1;}
20  /*...*/
21  if (counter == 0){
22   quotient.write(quotient_temp);
23   remainder.write(dividend_temp);}
24  else
25   counter = counter-1;
26  /*...*/ }
```

Figure 1: Part of the *simple-divider* design's source code.

analysis on the extracted formal representation of the design's behavior based on the security properties to identify all timing-based information flows from the sensitive inputs to the final outputs. The potentially vulnerable paths are reported back to designers for further inspection. Our static analysis is sound, i.e., it never misses a timing channel if such exists.

2 RELATED WORKS

Over the last decade, IFT has been widely used in the field of security for hardware systems.

Several secure languages have been developed to provide designers with modeling provably secure hardware. Caisson [22], Sapper [21], SecVerilog [28], and VeriCoq-IFT [6] are hardware security design languages, enabling designers to label and track information flow. For example, the Caisson [22] and Sapper [21] are both FSM based languages that have been developed by combining domain-specific abstractions common to hardware design and type-based techniques used in secure programming languages. Although the aforementioned tools facilitate secure hardware design, their major disadvantages are new language familiarity and needing to redesign the entire hardware using the new language.

The method in [4] gives the flexibility to define both implicit and explicit flows. It encodes security attributes into the design for formal verification of hardware security properties. However, the method cannot separate functional flows from timing-based ones. This could be problematic in many security applications where the functional flow is expected as it is protected by encryption, while timing flows must be eliminated. Clepsydra [3] uses the IFT technique for capturing timing leakage of hardware designs. To do this, it automatically generates the logic required for tracking timing flows and logical flows in arbitrary HDL codes at RTL. However, both methods are only applicable at RTL and do not support SystemC constructs.

At ESL, [12, 17, 26] use the IFT method to validate SystemC-based designs against the security violations related to the confidentiality and integrity threat models. While these methods are able to analyze the functional information flows (i.e., information does not move among isolated IPs), they cannot detect any timing flow. Moreover, they are not able to analyze SystemC HLS designs as they only focus on the subset of SystemC where the timing and precise functionality are abstract and not fixed. The existing verification methods [9, 14, 15] at ESL are not able to detect this security threat model as the design functionality (and its related protocol rules) is not affected.

To the best of our knowledge, ATLaS is the first timing-based information leakage flows detection approach at ESL that can verify SystemC HLS designs against the timing-based IFT security properties and report back the vulnerable paths to designers. It is automated, fast, non-intrusive, and does not rely on any commercial tool for its analysis.

3 TIMING-BASED INFORMATION LEAKAGE THREAT

For a given SystemC design, timing-based information flows exist from inputs to outputs of the design if the completion time of the outputs depends on the values of the design's inputs. These leakage flows can be exploited if the input signals on which the outputs are dependent contain sensitive data, and the completion time of the module can be measured by an unauthorized party. The module's completion time is the time when its output is updated to its final value. If the intermediate signals or variables are updated at each clock cycle, there will be no timing variations. On the other hand, if there is a degree of freedom for the variables to hold their current value or receive a new value, timing variation can occur.

For example, consider the *simple-divider* design in Fig. 1, illustrating a sequential divider module implemented in SystemC. Please note that lines 5, 21, 24, and 25 are initially not available. The signals *divisor* and *dividend* are inputs of the *Divider* module while the *quotient* and *residue* signals are the final outputs.

Now consider the security scenario that *divisor* and *dividend* inputs contain secret data and must be protected against unauthorized access. Since the underlying algorithm for the *Divider* module is based on comparison and subtraction (lines 17–19), the updates made to the intermediate variables *dividend_temp* and *quotient_temp* are conditioned on the sensitive inputs. Therefore, based on the values of the input signals, these variables (which are directly connected to the final outputs) may receive their final values at different times. Thus, an attacker who is familiar with the underlying algorithm can take advantage of this timing variation to exploit secret data.

A possible solution to block this timing-based information leakage flow is to *fully* control the update of the final outputs with a non-sensitive variable. In this example, the *counter* variable (lines 21–25) adds a waiting period to the updating process of the final outputs. Thus, in the case that the waiting period is longer than the worst-case execution time, the output gets its update at constant time steps.

As it has been proven in [3], detecting conditional updates caused by sensitive data captures all timing flows. Thus, in order to detect timing flows, we need to determine whether or not the updates

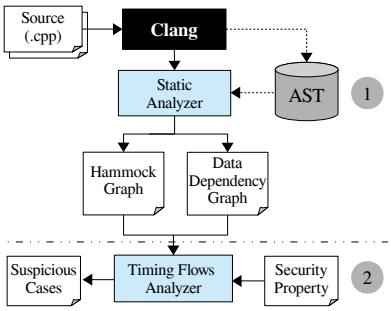

Figure 2: Methodology overview.

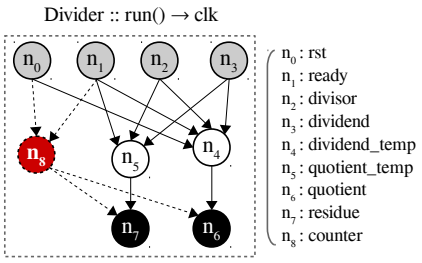

Figure 3: Part of the generated DDG of the *simple_divider*.

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.

4 TIMING-BASED FLOWS DETECTION METHODOLOGY

The overall workflow of ATLaS is shown in Fig. 2, consisting of two main phases which are 1) information extraction and 2) timing-based data flow analysis.

In the first phase, the AST of a given SystemC HLS design is analyzed to extract the data path and control flow of the design and formally represent them in terms of *Data Dependency Graph* (DDG) and *Hammock Graph* (HG). In the second phase, we take advantage of the aforementioned data structures (i.e., DDG and HG) to perform a data flow analysis based on the security properties and identify all timing-based information leakage flows from the sensitive inputs to the final outputs. The security properties are given by designers and specify conditions on the final outputs w.r.t to the sensitivity level of the design's inputs.

Since the proposed approach is based on a static analysis, it needs to be run only once to detect all timing-based flow security violations.

4.1 Information Extraction

As mentioned earlier, the necessary condition to form timing variation is the existence of intermediate variables or signals which their update process depends on secret data. In order to identify these cases, we need to determine all variables of the design that have the flexibility of selecting between receiving new values and holding their current values. Since the impact of sensitive data on computation results can be through both the data path and the control flow of the design, we need to analyze these parts of the design to detect all information leakage flows. Hence, the first step is to extract the data path and control flow of SystemC HLS designs.

To formally represent the data path and control flow of a given SystemC design, we consider two data structures which are DDG and HG, respectively. The DDG describes the relation of different design's variables (including all modules' signals, ports, and global and local variables). The formal definition of DDG is as follows:

Definition 1. *A Data Dependency Graph (DDG) is a structure* (N, E, Z)*, where* N *is a set of nodes,* E *is a set of edges, and* $Z \subseteq N$ *is set of output variables. The edge from node* X *to node* Y *shows that* Y *is dependent to* X*.*

To build the DDG of a given SystemC design, a static analysis on the AST of the design is performed. First, all variables of the design are extracted and tokenized by a unique string including the module, function (for local variable), and variable name. In order to know how different variables affect each other, a recursive analysis is performed on the AST from the points where updates on variables occur, i.e., computational statements. The computational statements are usually defined as assignments in the design. Moreover, due to the SystemC structure, it is also possible to assign a computation of some variables to the output port of a module using the *write()* member function of the output ports. In the case of an assignment statement, the left-hand side operand can be either a global or local variable (signal). The right-hand side operand (or input of the *write()* member function of the output ports) can be a function call or an input port. If a statement includes a function (or SystemC process) call, we recursively extract the relation of the function variables with the left-hand side variable or module port.

After analyzing the computational statements and extracting the relation of their input operands with their result variable, the analysis is performed for all compound statements (e.g., *if-else*, *for-loop*, or *while* statements) in which these computational statements are defined. All variables of the compound statements which consist of a computation statement are also added into the dependency list of the result variable of the computational statement. Please note that multiple occurrences of the same variable are represented with a single node in the DDG in order to reduce the size of graph as much as possible. It should be taken into account that in our framework, we keep the correspondence between nodes in the DDG and variables in the statements of the SystemC code so when a node in DDG is selected, the related statement in the SystemC code is easily determined.

For example, Fig. 3 shows a part of the generated DDG of the *simple-divider* design. In this graph, the nodes without input arrows (the gray nodes) show the primary inputs while the nodes without output arrows (the black nodes) indicate the primary outputs of the design. It also shows that all nodes (and the corresponding design's variable) belong to the method *run* of module *Divider* which is controlled by *clk*. Node n_8 is initially not available and is added into the graph when the blockage mechanism is used where the final results are fully controlled by the *counter* variable.

In order to know how different statements (data and control flow) of a given SystemC design are related to each other, the

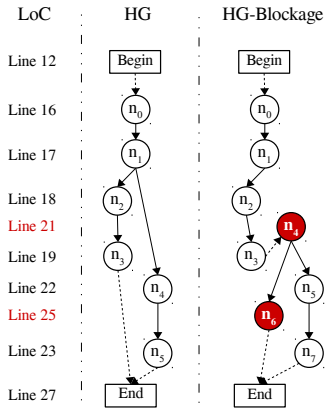

Figure 4: Part of the generated HG of the *simple_divider* design without and with blockage mechanism.

HG representation of the design is automatically generated from the AST of the SystemC design. The formal definition of HG data structure is as follows:

Definition 2. *A Hammock Graph (HG) is a structure* (N, E, n_0,-n_e)*, where N is a set of nodes, E is a set of edges in an $N \times N$ processing. n_0 is the initial node and n_e is the end node. If* (n, m) $\in E$ *then n is an immediate predecessor of m, and m is an immediate successor of n. There is a path from n_0 to all other nodes in N. From all nodes of N, excluding n_e, there is a path to n_e.*

The HG graph is generated by analyzing the AST of a given SystemC HLS design. To do this, we visit all nodes in the AST which are related to statements of the design. This includes both computational and control flow statements. As the top-level entities of a SystemC design are modules and global functions, the first entry point of performing statements dependency analysis is to find the node of the aforementioned entities. Then, a *Depth-First Search* (DFS) algorithm is performed within the top-level entities to visit all nodes of the statement's type in the AST. We take advantage of modules binding information to extract the connection of modules within the design. Moreover, function calls within the modules process are extracted by visiting the relevant nodes in the AST to understand how lower hierarchies in a module (i.e., local function and process) are connected to each other. Please note that each statement of the design is tokenized by the line of code where the statement is defined.

For example, Fig. 4 shows a part of the generated HG and HG-Blockage (including the blockage mechanism) of the *simple-divider* design. Each node of the HG represents a statement of the design and is tokenized by the corresponding line of code (LoC). As illustrated in this figure, in HG, nodes n_4 and n_5 (the final outputs) are controlled by node n_1 (a condition that has sensitive data) which causes timing flows. On the other hand, for HG-Blockage, nodes n_5 and n_7 (the final outputs) are fully controlled by node n_4 which does not contain any sensitive data.

In the next, by taking advantage of DDG and HG, we introduce an algorithm to detect all potential timing flows based on given security properties.

4.2 Timing-based Data Flow Analysis

We propose Algorithm 1 to detect all potential timing-based information leakage flows in a given SystemC HLS design. In order to illustrate each part of the algorithm, we use the motivating example (Fig. 1) where no blockage mechanism is used. The algorithm receives DDG, HG, and a set security properties as inputs and returns the list of nodes in HG (and the corresponding lines of code) which cause timing flow violations. Each security property consists of two main elements which are 1) the inputs with *High Security* (HS) tag and 2) the outputs that must be generated in *Constant Time* (CT). Thus, a security property *SP* is defined as follows:

$$SP = \{(SI, SO) \mid SI \leftarrow \{in1 = HS, \ldots\}, SO \leftarrow \{out1 = CT, \ldots\}\} \quad (1)$$

For example, in the case of the *simple-divider* design, the security property is defined as follows:

$$SP = \{(SI, SO) \mid SI \leftarrow \{divisor = HS, dividend = HS\}, \\ SO \leftarrow \{quotiont = CT, remainder = CT\}\} \quad (2)$$

From the SP, lists of the secure inputs and the outputs that must be generated in constant time are extracted. To know whether or not a variable is affected by secure inputs, a forward tracing is performed on the DDG from the corresponding secure input node to an output node with a CT tag. All nodes in this trace that are related to the secure input get the HS tag and are added into the sensitive list of secure inputs SL_{SI} (lines 5–6). Moreover, as the output variables may receive their final values through the intermediates variables, a backward tracing is performed on the DDG to extract the variables of assignment statements which are implicitly or explicitly related to the outputs with CT tag. These nodes (and the corresponding variables of the design) are added into the sensitive list of secure outputs SL_{SO} (lines 7–8). For example, in the case of the motivating example, the SL_{SI} and SL_{SO} of the design after tracing its DDG (Fig. 3) are $\{n_2, n_3, n_4, n_5\}$ and $\{n_7, n_6, n_4, n_5\}$, respectively.

In the next step (lines 9–22), the HG of the design is analyzed to find all sensitive control signals (which are in SL_{SI}) that control the occurrence of updates on the variables with CT tag (which are in *SO* and SL_{SO}), causing flows of timing variation from secure inputs to outputs of the design. To do this, each condition node type of the HG (e.g., *if-else*) is visited and its control variables n_{ctrl} and child nodes (which are not condition nodes type) are extracted (lines 9–12). If the intersection of the extracted control variables of the condition node n_{ctrl} and the sensitive list of secure inputs SL_{SI} is not empty, further analysis is performed on the child nodes n_{childs} of the condition node n (lines 13–18). The goal of this analysis is to find the assignment statements which their left-hand side variables are in the secure outputs list *SO* (case of explicit flow) or in the sensitive list of the secure outputs SL_{SO} (case of implicit flow). If there is at least one child node which meets the first condition (line 15), a timing flow exists in the design. Thus, the nodes in the HG including the condition and assignment (and the corresponding lines of code in the design source code) are stored in *TFV* and reported to designers. On the other hand, if the second condition occurs (line 17), there might be a timing flow in the design. To know whether the suspicious cases are real leakage flows, we need to find a condition which its control signals are not in SL_{SI} and it fully controls the updates on variables in *SO* (lines 19–22). Fully

Algorithm 1 Timing Flow Analyzer

Require: Security property *SP, DDG, HG*
Ensure: Timing Flows Violation TFV, $TFV_{suspicious}$
1: Secure Inputs $SI \leftarrow$ Extract from (SP)
2: Secure Outputs $SO \leftarrow$ Extract from (SP)
3: Sensitive List Secure Inputs $SL_{SI} \leftarrow SI$
4: Sensitive List Secure Outputs $SL_{SO} \leftarrow \emptyset$
5: **for each** *secure input* $S_i \in SI$ **do**
6: $SL_{SI} \leftarrow$ ForwardTraverse (DDG, S_i, SO)
7: **for each** *secure output* $S_o \in SO$ **do**
8: $SL_{SO} \leftarrow$ BackwardTraverse (DDG, S_o, SI)
9: **for each** node $n \in HG$ **do**
10: **if** $n.type()$ == cond **then**
11: $n_{ctrl} \leftarrow$ Extract list of controllers from n
12: $n_{childs} \leftarrow$ Extract child elements of n
13: **if** $n_{ctrl} \cap SL_{SI} \neq \emptyset$ **then**
14: **for each** child $c \in n_{childs}$ **do**
15: **if** ($c.type() \neq$ cond) && (vars in $c \in SO$) **then**
16: $TFV \leftarrow (n, c)$
17: **else if** ($c.type() \neq$ cond) && (vars in $c \in SL_{SO}$) **then**
18: $TFV_{suspicious} \leftarrow (n, c)$
19: **else**
20: **for each** child $c \in n_{childs}$ **do**
21: **if** ($c.type() \neq$ cond) && (vars in $c \in SO$) **then**
22: $FC_{flag} \leftarrow 1$
23: **if** ($TFV == \emptyset$) && ($FC_{flag} == 0$) **then**
24: $TFV \leftarrow TFV_{suspicious}$
25: **return** TFV, $TFV_{suspicious}$

controlling control signal implies that variables in *SO* getting a new value if and only if the controllers get a new value. If there is no such a condition in the design and also no direct leakage flow exists, the suspicious case becomes a real leakage flow case, thus it is stored in *TFV* (lines 23–24).

Coming back to the motivating example, the condition type node in the HG of the design (Fig. 4) is n_1 whose its control signals are $n_{ctrl} = \{dividend\_temp, divisor\}$ and its child nodes are $n_{childs} = \{n_2, n_3, n_4, n_5\}$. Since the child nodes n_4 and n_5 are in the list of secure outputs *SO*, the first condition in Algorithm 1 occurs (line 15), indicating an explicit leakage flow in the design. Now consider the HG-Blockage of the design, the first node of condition type in this graph is n_1. Analyzing its child nodes $n_{childs} = \{n_2, n_3\}$ shows a suspicious case of leakage flow as variables *dividend_temp* and *quotient_temp* (corresponding to child nodes of n_1) are in the sensitive list of the secure outputs SL_{SO}. Thus, these nodes are stored in $TFV_{suspicious}$. By continuing the analysis, node n_4 is the next condition node type whose its control signal $n_{ctrl} = \{counter\}$ dose not have any intersection to the SL_{SI} (lines 19–22). Moreover, its child nodes list includes nodes n_5 and n_7 which are in the *SO*. Thus, this condition node (n_4) fully controls the secure outputs of the design and its condition signal is non-sensitive. In this case, there is no timing-based leakage flow in the design, however, the suspicious case is reported to designers for their information.

5 EXPERIMENTAL RESULTS

In this section, we elaborate on how various security properties are defined based on the notion of IFT and can be verified using ATLaS. The proposed approach was applied to two standard SystemC HLS designs which are shared interconnect (inspired by [27]) and RSA crypto core [23]. For each design, we briefly discuss the architectural features that cause timing flows, the attack model for exploiting, the possible mitigation technique, and the results of our security analysis. The *Static Analyzer* module is implemented in C++ using the LibTooling library of the Clang compiler [19]. To access relevant nodes in the AST (generated by Clang) of a given design, we use the primary node visitor *RecursiveASTVisitor* of Clang. The *Timing Flow Analyzer* module is implemented based on Algorithm 1 in C++. All the experiments were carried out on a PC equipped with 8 GB RAM and an Intel Core i7 CPU running at 2.4 GHz.

5.1 Shared Interconnect

In the first case study, we consider a common source of timing-based side-channel attacks in hardware designs when different IPs are connected to a shared interconnect (or bus). If the access control policy of the bus (the way IPs can access a shared resource e.g., memory) is not strong enough, a software adversary can use an IP to control the access patterns to a shared resource and impacting the time when other IPs can use the same resource. Such access control policies are commonly implemented in components with routing functions e.g., an *arbiter* module.

To model this security vulnerability, we have implemented a shared interconnect architecture inspired by the real-world AMBA-2.0 AHB bus in [27]. We have integrated three microprocessor units MPU1, MPU2 and MPU3 (act as master), two regular memories RgMem1 and RgMem2 (act as slave) and one shared memory ShMem (act as slave) to the interconnect module. To access the shared memory over the bus architecture, the master IP sends a request signal and waits for the arbiter to send back an acknowledge signal. The arbiter module was implemented based on the *Round Robbin* (RR) access policy, providing master IPs with an equal priority to access the shared memory.

In order to assess the security of the design with respect to timing-based side channel attacks, we have defined six security properties. They check whether access of a given MPU of the design to the shared memory ShMem is dependent on the other MPUs. For example, a security property to specify this condition is as follows:

$$SP = \{(SI, SO) \mid SI \leftarrow \{req1 = HS\}, SO \leftarrow \{ack2 = CT\}\} \quad (3)$$

The security property *SP* ensures that the acknowledge signal *ack2* sent by the arbiter to the *MPU2* module must not be dependent on the request signal *req1* of the *MPU1* module. As the policy of the bus to access a shared resource is based on the RR algorithm, our approach could detect the dependency between the *req1* and *ack2* signals. It means IPs that are supposed to be isolated can secretly communicate by controlling the access patterns to the shared memory and by affecting the time when other IPs can use the same resource.

In order to block this timing flow, we replaced the RR algorithm of the arbiter with a *Time Division Multiple Access* (TDMA) algorithm where the acknowledge signals are issued based on a counter. We have analyzed the interconnect architecture again, and in this case, no timing flow was detected as the acknowledge signals are fully controlled by a counter variable (which is non-sensitive). The analysis took 27.39 seconds to report the results.

5.2 RSA Crypto Core

In the second experiment, we applied our timing-based information leakage detection approach on the SystemC HLS implementation of the RSA algorithm, provided by [23]. We have defined the following security property to verify the design against the existence of any

timing channel.

$$SP = \{(SI, SO) \mid SI \leftarrow \{SKey = HS\}, SO \leftarrow \{msg = CT\}\} \quad (4)$$

The security property *SP* ensures that there must be no timing flow from the secret key *SKey* to the ciphertext *msg*. Using our detection algorithm, we could find a timing flow when messages are ciphered using the RSA secret key. The main reason for this vulnerability is an insecure implementation of the modular exponentiation step (module *modular_exp* in the design) in the RSA algorithm, where the duration of generating the ciphertext is linearly related to the number of '1' bits in the key. Although the number of '1' bits alone may not be sufficient to easily find the key, an attacker can perform a statistical correlation analysis of timing information to recover the key completely by repeating executions with the same key and different inputs.

In order to block this timing-based leakage flow, we have added a counter into the *modular_exp* of the design to eliminate the dependency of generating the ciphertext to the value of the secret key. The value of the counter has been defined based on two techniques which are delaying until the worst-case execution time and randomization. We have analyzed the RSA design again, and in this case, no timing flow was detected as the duration of generating the ciphertext is fully controlled by a counter variable which is non-sensitive. Although this countermeasure increases the execution time of the RSA design, it leads to a time-independent execution from the key. The analysis took 21.56 seconds to report the results and no suspicious cases were reported.

Please note that in crypto cores, functional flow from the secret key to the ciphertext exists (as the ciphertext is computed from the key), but it is protected by encryption. However, designers need to ensure that the time taken for the ciphertext to become available does not depend on the value of the secret key. The functional IFT-based methods cannot specify the nature of detected flows, thus, making designers unable to find the right mitigation technique. Our approach truly assesses the security of the design against the timing-based side-channel attacks.

6 CONCLUSIONS

In this paper, we presented ATLaS, the first timing-based information leakage flows detection approach for SystemC HLS designs at ESL. At the heart of the approach is a scalable static information flow analysis that operates directly on the SystemC models. The analysis formally represents the behavior (data path and control flow) of the system in terms of two well-structured graphs. Parts of the design that violate specified secure information flow properties are identified by analyzing the extracted graphs. These potentially vulnerable paths are reported back to designers for further inspection. We have demonstrated the applicability of our approach on two security-critical architectures.

ACKNOWLEDGMENTS

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SATiSFy under contract no. 16KIS0821K, the project VerSys under contract no. 01|W19001, and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

REFERENCES

[1] 2006. IEEE Standard SystemC Language Reference Manual. *IEEE Std 1666-2005*, 1–423.
[2] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *Symposium on Security and Privacy (SP)*. 623–639.
[3] Armaiti Ardeshiricham, Wei Hu, and Ryan Kastner. 2017. Clepsydra: Modeling timing flows in hardware designs. In *ICCAD*. 147–154.
[4] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *DATE*. 1691–1696.
[5] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).
[6] Mohammad-Mahdi Bidmeshki and Yiorgos Makris. 2015. Toward automatic proof generation for information flow policies in third-party hardware IP. In *HOST*. 163–168.
[7] Philippe Coussy, Andrés Takach, Michael McNamara, and Mike Meredith. 2010. An introduction to the SystemC synthesis subset standard. In *CODES+ISSS*. ACM, 183–184.
[8] Amin Ghasemazar, Mehran Goli, and Ali Afzali-Kusha. 2014. Embedded Complex Floating Point Hardware Accelerator. In *VLSI Design*. 318–323.
[9] Mehran Goli and Rolf Drechsler. 2019. Scalable Simulation-Based Verification of SystemC-Based Virtual Prototypes. In *DSD*. 522–529.
[10] Mehran Goli and Rolf Drechsler. 2020. *Automated Analysis of Virtual Prototypes at the Electronic System Level: Design Understanding and Applications*. Springer Nature.
[11] Mehran Goli, Muhammad Hassan, Daniel Große, and Rolf Drechsler. 2019. Automated Analysis of Virtual Prototypes at Electronic System Level. In *GLSVLSI*. ACM, 307–310.
[12] Mehran Goli, Muhammad Hassan, Daniel Große, and Rolf Drechsler. 2019. Security validation of VP-based SoCs using dynamic information flow tracking. *it-Information Technology* 61, 1 (2019), 45–58.
[13] Mehran Goli, Jannis Stoppe, and Rolf Drechsler. 2016. AIBA: an Automated Intra-Cycle Behavioral Analysis for SystemC-based Design Exploration. In *ICCD*. 360–363.
[14] Mehran Goli, Jannis Stoppe, and Rolf Drechsler. 2017. Automatic Equivalence Checking for SystemC-TLM 2.0 Models Against their Formal Specifications. In *DATE*.
[15] Mehran Goli, Jannis Stoppe, and Rolf Drechsler. 2017. Automatic Protocol Compliance Checking of SystemC TLM-2.0 Simulation Behavior Using Timed Automata. In *ICCD*. 377–384.
[16] Mehran Goli, Jannis Stoppe, and Rolf Drechsler. 2020. Automated Nonintrusive Analysis of Electronic System Level Designs. *IEEE Trans. on CAD of Integrated Circuits and Systems* 39, 2 (2020), 492–505.
[17] Muhammad Hassan, Vladimir Herdt, Hoang M. Le, Daniel Große, and Rolf Drechsler. 2017. Early SoC security validation by VP-based static information flow analysis. In *ICCAD*. 400–407.
[18] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*. 104–113.
[19] C. Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *BSD*. 1–2.
[20] Hoang M. Le, Daniel Große, Niklas Bruns, and Rolf Drechsler. 2019. Detection of Hardware Trojans in SystemC HLS Designs via Coverage-guided Fuzzing. In *DATE*. 602–605.
[21] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. 2014. Sapper: a language for hardware-level security policy enforcement. In *ASPLOS*. 97–112.
[22] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: A Hardware Description Language for Secure Information Flow. In *PLDI*. 109–120.
[23] Bin Lin and Fei Xie. 2018. SCBench: A benchmark design suite for SystemC verification and validation. In *ASP-DAC*. 440–445.
[24] Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. 2014. Leveraging Gate-Level Properties to Identify Hardware Timing Channels. *TCAD* 33, 9 (2014), 1288–1301.
[25] Jason Oberg, Timothy Sherwood, and Ryan Kastner. 2013. Eliminating Timing Information Flows in a Mix-Trusted System-on-Chip. *IEEE Des. Test* 30, 2 (2013), 55–62.
[26] Pascal Pieper, Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. Dynamic Information Flow Tracking for Embedded Binaries using SystemC-based Virtual Prototypes. In *DAC*. 1–6.
[27] Thomas Schuster, Rolf Meyer, Rainer Buchty, Luca Fossati, and Mladen Berekovic. 2014. SoCRocket - A virtual platform for the European Space Agency's SoC development. In *ReCoSoC*. 1–7, http://github.com/socrocket.
[28] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *ASPLOS*. 503–516.