Early Validation of SoCs Security Architecture Against Timing Flows Using SystemC-based VPs

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

***Abstract*—Modern *System-on-Chips* (SoCs) have been increasingly deployed in critical aspects of our lives. As a consequence, they have access to a large number of secret assets that must be protected against unauthorized access. In order to provide sound security guarantees, an SoC typically has a security architecture as authentication mechanisms to control the access of different *Intellectual Properties* (IPs) to secret assets. Since the SoC's security architecture cannot be changed after production, it is of utmost importance to detect any security flaws in the design phase. Moreover, to prevent costly fixes in later stages, security validation should start as early as possible.**

In this paper, we propose a novel approach to validate the security architecture of a given SoC against timing flows using SystemC-based *Virtual Prototype* (VP) and static information flow tracking technique at the system level. Experimental results on two real-world VP-based SoCs demonstrate the scalability and applicability of the proposed approach in identifying timing flows.

I. Introduction

The increasing deployment of computing devices in a large number of highly personalized activities and critical aspects of our lives (e.g., medical devices, automobiles, flight control, and banking systems) has raised their requirements on security significantly. Modern computing systems are typically implemented as system-on-chip (SoC) designs, namely a single integrated circuit containing the system functionality [1]. An SoC design involves the composition of a large number of *Intellectual Properties* (IPs) such as memories, processing units, I/O interfaces, and other various hardware accelerators (e.g., hardware encryption units). These IP blocks are integrated through a number of on-chip interconnects (buses) to implement the system functionality. Since data (including secure assets) in such a system is transferred via the shared interconnects across different IPs, access control or information flow requirements are defined by a collection of security policies. The policies specify the conditions under which a secret asset can be accessed at any point in the system execution. Thus, an SoC needs a security architecture [2] to ensure that the system enforces and manages these policies e.g., a mechanism of authentication or managing access to shared resources.

A common property that often needs to be guaranteed in these systems is non-interference [3], where certain parts of the system should never interfere with other parts. However, guaranteeing non-interference in a given SoC is a non-trivial and crucial task as depending on security architecture (the access control policies) implemented in the SoC, information can flow through difficult-to-detect side channels. The IPs through which secret data leaks are called side channels, and attacks exploiting this information are called side-channel attacks. Among the existing side-channel security attacks, timing-based attacks are more interesting for attackers as they only need to measure the completion time of the victim process without physical access to the design. Thus, attackers can access secret data at a very low cost and effort.

Information Flow Tracking (IFT) [3] has been shown as a powerful technique to help mitigate security vulnerabilities that violate certain information flow policies and non-interference properties. IFT works by monitoring how information propagates through a system to see if secret information is leaking to an untrusted subsystem or to ensure that the integrity of a critical subsystem is not violated.

Since the cost of fixing any security flaws increases with the stage of development, the validation process should be performed as early as possible. For the early design entry, *Virtual Prototype* (VP) is being increasingly adopted by the semiconductor industry [4]. A VP is an abstract and executable software model that is typically implemented using SystemC [5] and its *Transaction Level Modeling* (TLM) [6] framework at the *Electronic System Level* (ESL) [7]–[9]. In comparison to the *Register Transfer Level* (RTL) designs, VPs provide designers with orders of magnitude faster simulation speed. By this means, a system can be implemented quickly and used as a reference model for lower levels of abstraction. Hence, VP-based security validation could be one promising direction to fix the security vulnerabilities in the SoCs before they are refined and to avoid costly design loops occur.

While there are some VP-based IFT security validation methods [10]–[12] at the ESL that focus on functional information flows (ensuring that data does not move among isolated IPs), the timing flows validation (certifying that the timing footprints of the isolated IPs do not form a communication channel) has not been considered yet.

In this paper, we focus on the security validation of SoCs security architecture (access control or information flow policies) against timing flows, in particular, helping system designers to detect non-interference property violations and to pinpoint security flaws caused by a poor security architecture implementation in the early stage of the SoC design process. We propose a VP-based security validation approach that uses the static IFT technique and consists of two main phases: 1) static data extraction, and 2) timing flow analysis. In the first phase, we build on the flexible Clang compiler [13]–[15]

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VerSys under contract no. 01IW19001, and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

to formally represent the behavior of a given VP-based SoC in terms of data and control flows w.r.t the given security properties. This is done by statically analyzing the *Abstract Syntax Tree* (AST) of the VP through several intermediate steps such as connectivity analysis, access control extraction, call-graph analysis, and data dependency analysis. In the second phase, we perform a static taint tracing and path analysis on the formal representation of the VP's behavior to identify all paths that violate the specified security properties. The violated paths are reported back to designers, allowing them to improve the security architecture of the SoC. The proposed approach is applied to two real-world VP-based SoCs namely LEON3-based SoCRocket VP [16] and RISC-V VP-based SoC [17] to show its scalability and applicability.

II. Related Works

Over the past few years, IFT techniques have been widely used to create secure systems by detecting security defects or enforcing security policies.

There exist several secure languages that provide designers with modeling provably secure hardware. Caisson [18], Sapper [19], SecVerilog [20], and VeriCoq-IFT [21] are hardware security design languages that allow designers to label and track information flow. For example, the Caisson [18] and Sapper [19] 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 methods enhance secure hardware design, their major drawbacks are new language familiarity and needing to redesign the entire hardware based on the new language syntax and semantics.

Several IFT-based methods have been developed for hardware trustworthiness, targeting the RTL designs. *Proof-Carrying Hardware* (PCH) [22], [23] verifies the equivalence between the design specification and its implementation using run-time *Combinational Equivalence Checking* (CEC). However, converting RTL code to a formal representation and developing proofs for security properties, requires additional knowledge of formal methods, theorem proving environments, and proof-writing. This makes PCH-based methods very tedious and time-consuming which adopting them need a lot of manual effort. RTLIFT [24] 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, all the aforementioned methods are only applicable at RTL and do not support SystemC constructs.

At the ESL, there are only a few works [10]–[12] on security validation of VP-based SoCs. These methods use the IFT technique to validate a given VP-based SoC 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 flows. The method presented in [25] introduces a timing flow analysis technique to validate SystemC HLS designs. Similarly, [26] detects and reports information flow violation in accelerator designs implemented in C language. However, these methods are limited to single IP blocks and do not support TLM-2.0 constructs.

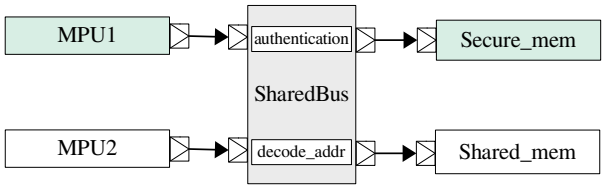

Fig. 1: Architecture of motivating example.

III. Timing Flow Threats and Motivating Example

In this section, we first describe the threats of timing flow in VP-based SoCs at the ESL. Second, using a motivating example, we explain how timing variations are generated and represented at the VP level and what are the necessary conditions for blocking them.

A. Timing Flow Threats

In an SoC, transporting (secret) data among different IP components is typically performed through shared interconnects (buses). For a given VP-based SoC, TLM communication by means of transactions is used to model transporting data among IP blocks at the ESL. Since all data (transactions) is transported through shared interconnects, designers implement different access control or information flow policies (security architecture) to protect the secret data from unauthorized access. A common property that often needs to be checked in these systems is non-interference [27]. In this context, there are three general security properties:

- Confidentiality: an IP creates an unwanted information flow from a target IP in retrieving secret data which this IP is not allowed to access,
- Integrity: an IP presents itself as a different IP to create an information flow to some target IP to modify some data, and
- Availability: an IP may use some shared resource to the extent that other IPs cannot use it

With respect to the notion of non-interference, secure assets can be inferred through two general classes of information flow: explicit and implicit. Explicit information flows result from two modules directly communicating. Implicit information flows are very subtle and generally leak secret data through behavior. In a given SoC, typical implicit information flows show up in the form of timing, where information can be extracted from the time difference of operations.

Assume that part of the interconnect module in an SoC that is responsible for the authentication (controlling access to secure parts) implemented in such a way that the time takes for performing the authentication process (e.g, checking key) is dependent on the key value of the incoming transactions. In this case, an unauthorized IP (attacker) can extract sensitive data (i.e., the key) by observing and analyzing the variations in the completion time of the authentication unit. Once the key is leaked, the unauthorized IP can use it to read (confidentiality) or modify (integrity) secret data stored in the secured memory.

Another source of timing flows in an SoC with a shared interconnect arises when IP blocks that are supposed to be isolated can covertly communicate by modulating the access patterns to a shared resource (e.g., memory) and affecting

```
1 struct MPU1 : sc_core::sc_module {
2  /*...*/
3  void thread_process(){
4  /*...*/
5  key_extension* ext = new key_extension;
6  ext->sec_key = "A1B2C3D4";
7  trans.set_extension(ext);
8  /*...*/
9  socket->b_transport(*trans, delay);
10 /*...*/};
11 struct SimpleBus : sc_core::sc_module {
12 const char* Skey = "A1B2C3D4";
13 tlm_utils::simple_initiator_socket_tagged<SimpleBus>*
       init_socket[2];
14 tlm_utils::simple_target_socket_tagged<SimpleBus>*
       targ_socket[2];
15  /*..*/
16 int decode_addr(sc_dt::uint64 addr, sc_dt::uint64&
       masked_addr) {
17 unsigned int id = static_cast<unsigned int>((addr >> 8)
        & 0x3);
18 masked_addr = addr & 0xFF;
19 return id;}
20  /*..*/
21 bool authentication (char* trans_key) {
22 size_t i = 0;
23 while (i < strlen(Skey)){
24  if (trans_key[i] != Skey[i])
25   return false;
26  i++;}
27 return true;}
28   /*..*/
29 virtual void b_transport( tlm::tlm_generic_payload&
       trans, sc_time& delay ){
30  sc_dt::uint64 address = trans.get_address();
31  sc_dt::uint64 masked_address;
32  unsigned int mem_id = decode_addr(address,
       masked_address);
33  security_extension* ext = new security_extension;
34  ext = trans.get_extension();
35  bool permission = authentication(ext->key);
36  trans.set_address( masked_address);
37 if (mem_id == 0)
38  if (permission == 1)
39   (*init_socket[0])->b_transport(trans, delay);
40 else if (mem_id == 1)
41  (*init_socket[1])->b_transport(trans, delay);
42  /*..*/};
```

Fig. 2: A part of the motivating example source code.

the time when other IP blocks can use the same resource. Moreover, if the access control policy implemented e.g., based on priority-based messaging, an unauthorized IP (attacker) can generate transactions with high priority to access the shared resource and block the other transactions generated by authorized IPs (availability).

As the VP is used as a reference model, the access control or information flow policies (security architecture) of the SoC are translated into lower levels of abstraction in the SoC design flow. Thus, the abstract timing model of the SoC's security architecture at the VP level is mapped onto the cycle-accurate model where the completion time of operations depends on clock cycles or their latency. Hence, the main goal of our VP-based timing flows analysis is to early detect the potential timing-based security flaws in a given SoC caused by a weak implementation of the security architecture.

B. Motivating Example

We present here a simple VP-based SoC implemented in SystemC TLM-2.0 (Fig. 1) that will be used to showcase the main ideas of our approach throughout this paper. The VP consists of a trusted microprocessor (*MPU1*), a regular microprocessor (*MPU2*), a regular memory (*Regular_mem*), and a confidential memory (*Secure_mem*). The modules are connected to the shared interconnect *SharedBus* (which routes transactions) where the MPUs act as initiators and the memories as targets. Fig. 2 shows a part of the VP source code including the *MPU1* (Lines 1 to 10) and *SharedBus* (Lines 11 to 42) modules. The communication uses a 16-bit address mode where bits 0 to 7 are used for local address inside memory and bits 8 to 15 for memory address. The MPUs execute instructions that initiate transactions (i.e., read or write) to access memories. For a transaction generated by an initiator module, the *SharedBus* module receives the transaction and checks its address attribute to route it to the corresponding memory. As illustrated in Fig. 2, this process is implemented as *decode_addr* unit in the interconnect (Lines 16 to 19). Moreover, the *SharedBus* module contains an *authentication* unite (Lines 21 to 27) which is implemented as the access control policy of the SoC to authenticate any access of initiator modules to the secured memory *Secure_mem*. The authentication process is performed by comparing the authentication key (shortened for simplicity) *Skey* (Line 12) which is only available for the *SharedBus* and the authorized initiator module (*MPU1*). In order to access the secured memory *Secure_mem*, an initiator module needs to create transactions that contain the memory address and the authentication key. The address filed of transactions contains the memory address of *Secure_mem* while the authentication key is stored in their extension filed. In the case that both conditions (memory address and authentication) are satisfied (Lines 37 to 39), the initiator module is allowed to access the secured memory.

Now consider the scenario that the authentication algorithm is implemented as a loop over all characters of the authentication key (Lines 21 to 27). Once the two keys differ in a character, the comparison function returns with *false*, and when only all characters are identical, is *true* returned. In this case, as long as the characters in both *trans_key* and *Skey* (Line 24) are equal, the next character is compared. As soon as one differs, the function returns. Since each additional comparison takes extra time, an unauthorized IP (*MPU2*) can take advantage of this time difference to brute-force the character (by generating transactions) for each position one at a time. In comparison to a regular brute-force attack, this requires an effort that is linear in the length of the authentication key instead of exponential.

A possible solution to block this timing-based information leakage flow is to fully control the update on the result of the *authentication* unit with a non-sensitive variable. Fig. 3 shows a safe implementation of the *authentication* unit where the key comparison is always performed for the total length of the secret key and is not dependent on the value of *trans_key*. In this case, the final result is fully controlled by a loop condition (Fig. 3, Line 4) with non-sensitive variables *i* and *Skey*. Thus the final result *flag* is generated at constant time steps.

As it has been proven in [28], detecting conditional updates

```
1 bool authentication_blockage (char* trans_key) {
2  size_t i = 0;
3  bool flag = 1;
4  while (i < strlen(Skey)){
5   if (trans_key[i] != Skey[i])
6    flag = false;
7   i++;}
8  return flag;}
```

Fig. 3: Safe implementation of the authentication unit of the *SharedBus* module.

caused by sensitive data captures all timing flows. Thus, in order to detect timing flows in a given VP-based SoC, we need to determine whether or not the updates made to the results of the access control or information flow policies implemented in the SoC occur at constant time steps. This can be addressed by detecting variations in the update time of the transaction's attributes and the corresponding variables in the VP labeled as sensitive and tracking them to the final results that supposed to be generated in constant time.

IV. Timing-based Flows Detection Methodology

The overall workflow of the proposed approach is illustrated in Fig. 4, consisting of two main phases which are 1) static data extraction and 2) timing flow analysis.

In the first phase, we take advantage of the Clang compiler to analyze the AST (generated by Clang) of VP to formally represent its behavior in terms of *Correlation Graph* (CG) and *Control Flow Graph* (CFG) w.r.t the given security properties. The security properties are defined by users and consist of two main elements which are 1) the *source* with *High Security* (HS) tag and 2) the *sink* that must be generated in *Constant Time* (CT) or be isolated.

In the second phase, we take advantage of the aforementioned data structures (i.e., CG and CFG) to perform an information flow analysis based on the security properties to identify all timing-based flows from *source* to *sink*.

The violated paths are reported back to designers, allowing them to improve the security architecture of the SoC.

A. Static Data Extraction

In this section, we first describe how the security properties for our timing flow analysis are defined. Then, we explain how the data and control flows of a given VP-based SoC are extracted w.r.t the defined security properties.

1) Security Properties Definition: The first step of our timing-based information flow analysis is to read the security properties defined by designers. Each property must contain two main elements which are *source* and *sink*. For a given design, the former contains the part of the design with high security tag while the latter refers to the part of the design in which the time taken for it to reach its final value must be constant as the *source* changes. These properties need to be defined in a way to capture the complete flow without missing out on any information. For a given VP-based SoC, the classical property specification does not work as it always considers an input port and an output port of the system as untrusted/trusted information sources. Whereas different IP

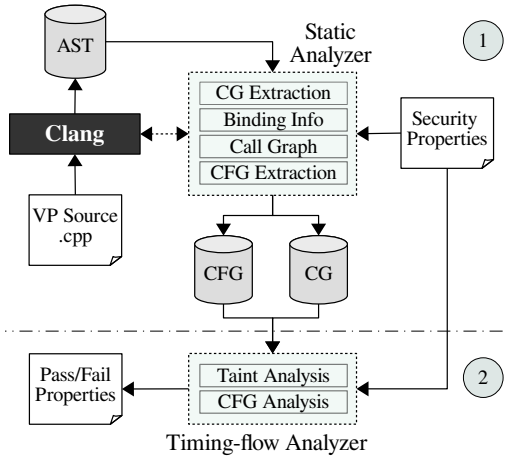

Fig. 4: Methodology overview.

blocks in a VP interact through transactions, hence, the property specifications and definitions need to adapt. Therefore, we define security properties as follows:

$$SP = \{P_i : (source, c_{src}, sink, c_{snk}) \mid source = HS, sink = CT\} \quad (1)$$

Please note that *source* can be a transaction generated in a VP, an attribute of the transaction or an internal variable related to the transaction. The *sink* in each property acts as the final destination. The c_{src} and c_{snk} parameters are predicates, describing under which condition is the data valid at *source* and *sink*, respectively. These parameters can be set to $\emptyset$, if *source* or *sink* is valid for all cases. The index i indicates the number of security properties in SP.

For example, in the case of the motivating example (Fig. 1), the security property is defined as follows:

$$\begin{aligned} SP = \{&P_1 : (source, \emptyset, sink, c_{snk}) \mid \\ &source \leftarrow MPU1 :: thread\_process() : sec\_key, \\ &sink \leftarrow SharedBus :: b\_transport() : permission \\ &c_{snk} \leftarrow SharedBus :: b\_transport() : mem\_id = 0\} \end{aligned} \quad (2)$$

In the security property P_1, variable sec_key is an attribute of the transaction which is defined in its extension filed to hold the authentication data for all generated transactions by the *MPU1* module in the *thread_process()*. The $permission$ variable belongs to the access control policy of the *SharedBus* in its *b_transport* function and holds the authentication result. The property ensures that the $permission$ variable must obtain the result in constant time as sec_key changes.

2) Creating Correlation Graph (CG): By analyzing the security properties, we build a correlation graph (CG) for each property. The CG is a data structure that describes the relation of different design variables from $source$ to $sink$ including all modules' ports, transactions, and global and local variables. The formal definition of the CG is as follows:

Definition 1. *A Correlation Graph (CG) is a structure* $(source, N, E, sink)$*, where* N *is a set of nodes,* E *is a set of edges, and* $source \in N$ *is the starting node while* $sink \in N$ *is the ending node. The edge from node* X *to node* Y *shows that* Y *is dependent to* X*.*

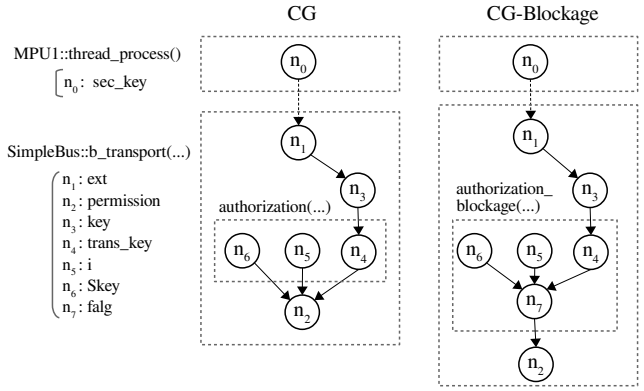

Fig. 5: A part of the generated CG of the motivating example.

In order to build the CG for each property in SP, a static analysis on the AST of the VP is performed. Since the first node of the CG is $source$, we perform a recursive analysis on the AST from the point where the $source$ is declared for the first time and extract all variables of the VP to reach the $sink$. The extracted variables are tokenized by a unique string including the name of module, function/process (for local variable), and variable. The recursive analysis includes visiting the computational, compound statements, and module binding information.

The computational statements are usually defined as assignments in the VP where updates on transactions' related parameters and variables occur. Due to the transaction construct, the update on a transaction's attributes is performed by calling its corresponding member function such as *set_address()* and *set_extension()*. Moreover, if a statement includes a function (or SystemC process) call, we recursively extract the relation of the function's variables with the left-hand side variable. Thus, in our analysis, we extract all the aforementioned constructs.

The compound statements are defined the control flow of the system and consist of blocks such as *if-else*, *for-loop*, or *while* statements. 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 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 CG in order to reduce the size of the graph as much as possible. It should be taken into account that in our framework, we keep the correspondence between nodes in the CG and variables in the statements of the VP code so when a node in CG is selected, the related statement in the VP code is easily determined.

In a VP-based SoC, transporting data is performed by means of transactions using SystemC TLM-2.0 interfaces (e.g. *b_transport*). In our analysis, we consider a connection between two IPs when their corresponding sockets (initiator socket and target socket) are connected using SystemC TLM-2.0 *b_transport* function call. In the initiator modules, the initiator socket calls the *b_transport* function to sent transactions, and the corresponding target socket in the shared interconnect for which the *b_transport* function is registered receives the transactions and after routing analysis sends them to the related target module. By this, the *b_transport* function is the common hub in the VP, connecting two IPs through initiator and target sockets. Thus, the binding information is extracted by building the call graph for the *b_transport* which is called by the initiator socket of initiator modules and goes through the interconnect to reach the target modules.

Fig.5 illustrates a part of the generated CG and CG-blockage of the motivating example w.r.t security property in (2). Each node of the CG is a transaction's attribute, its related parameter or a variable of the VP which is tokenized by the name of module and function (for local variable) to which the transaction or variable belongs. The dot-box in the CG shows the function calls graph, started from initiator module *MPU1* by calling *thread_process()* and goes through the *b_transport()* function of the *SharedBus*. Thus, this graph identifies how the *source* (node n_0) is connected to *sink* (node n_2) through the intermediate variables.

3) Creating Control Flow Graph (CFG): As mentioned earlier, the main reason to form timing flows is conditional updates caused by sensitive data. Using CG we can understand how sensitive data flows from *source* to *sink*. In order to know whether conditional updates caused by sensitive data, we need to extract the control flow of a given VP. The formal definition of CFG is as follows:

Definition 2. *A Control Flow Graph (CFG) models the flow of control between the basic blocks in a program. A CFG is a structure* (N, E) *where N is a set of nodes and E is a set of edges. Each node* $n \in N$ *corresponds to a basic block. Each edge* $e = (n_i, n_j) \in E$ *corresponds to a possible transfer of control from block* n_i *to block* n_j*.*

The CFG is generated by analyzing the AST of the VP. We visit all nodes in the AST which are related to both computational and control flow statements. To do this, a *Depth-First Search* (DFS) algorithm is performed within the top-level entity where the *source* is defined to visit all nodes of the statement's type (computational and control flow). We take advantage of modules binding information from CG to build the connection between two modules in CFG. 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.

Fig. 6 demonstrates a part of the generated CFG and CFG-Blockage of the motivating example (Fig. 1) w.r.t security property in (2). The gray nodes show the control flow statements (condition node type e.g., *if-else*) while the white nodes indicate the computational statements.

B. Timing Flow Analysis

After generating a formal representation of a given VP-based SoC behavior, we perform a timing flow analysis to detect all conditional updates caused by the sensitive data. Algorithm 1 shows this analysis where for a given security

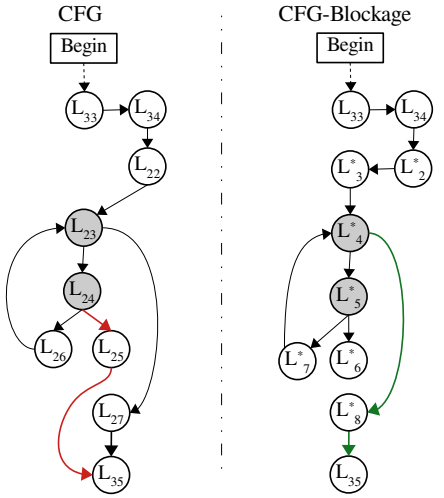

Fig. 6: A part of the generated CGF and CFG-blockage of the motivating example. L and L* indicate line of code in Fig. 2 and Fig. 3, respectively.

property P_i, a taint analysis is performed on the corresponding generated CG and CFG. The taint analysis identifies how the sensitive data (*source*) affects or taints other transactions and variables inside a system. The taint analysis is performed by a forward tracing on the CG from the *source* node to the *sink* node. All nodes in this trace that are related to the *source* get the HS tag and are added into the list of *source* taints L_{st} (Line 1). In the case of the motivating example, the L_{st} of the VP after tracing its CG (Fig. 5) is $\{n_1, n_3, n_4\}$.

In the next step (Lines 2–14), the CFG of the VP is analyzed to find all control statements including sensitive variables, transaction's attributes or its related parameters (stored in L_{st}) that control the occurrence of updates on *sink*. To do this, each condition node type of the CFG (e.g., *if-else*) is visited and its control variables n_{ctrl} are extracted (Lines 3–4). If the intersection of the extracted control variables of the condition node n_{ctrl} and L_{st} is not empty, further analysis is performed on the child nodes of the condition node n (Lines 5–14). The goal of this analysis is to find whether the condition node n which includes sensitive variable(s) controls the update on *sink*. To do this, we perform a DFS analysis on the child nodes of the condition node n to find a direct path from n to *sink* where no condition node is visited. The existence of a direct path indicates that condition node n has an explicit impact on the update of *sink*. If there is at least one path that meets this condition (Lines 11–14), a timing flow exists in the VP. Thus, the condition node and the related path (corresponding lines of code in the VP source code) are stored in TF and reported to designers.

In the case of our motivating example, the first condition type node in the CFG of the VP (Fig. 6) is L_{23} whose control variables are $\{i, SKey\}$ which are not in L_{st}. Therefore, the analysis continues to the next condition node which is L_{24} whose control variables are $\{trans\_key, SKey\}$. Since $trans\_key$ is in L_{st}, further analysis is performed on the child nodes of L_{24}. The result of DFS analysis shows that there are two paths $p_1 = \{L_{24} \rightarrow L_{25} \rightarrow L_{35}\}$ and

Algorithm 1 Timing-flow Analyzer

Input: P_i, CG, CFG
Output: Timing Flow TF

```
1:  L_st ← ForwardTraverse (source, CG)
2:  for each node n ∈ CFG do
3:      if n.type() == condition then
4:          n_ctrl ← Extract list of variables from n
5:          if n_ctrl ∩ L_st ≠ ∅ then
6:              L_path ← DFS (n, CFG, sink)
7:          for each path p ∈ L_path do
8:              for each node n_p ∈ p do
9:                  if n_p.type() == condition then
10:                     remove(p, L_path)
11:         if L_path ≠ ∅ then
12:             for each node n_p ∈ p do
13:                 if sink ∈ n_p then
14:                     TF ← (n, n_p)
15: return TF
```

$p_2 = \{L_{24} \rightarrow L_{26} \rightarrow L_{23} \rightarrow L_{27} \rightarrow L_{35}\}$. As p_2 includes a condition type node (L_{23}), it is eliminated from L_{path}. Thus, p_1 is the only member of L_{path} whose L_{35} includes *sink*. In this case, L_{24} and path p_1 are stored in TF and reported back to designers.

On the other hand, analyzing the CFG-Blockage shows that there is no timing flow in the VP as there is no explicit path from conditional node L^*_5 (which its control variable $\{trans\_key\}$ is in L_{st}) to the *sink*. The only available path ($p_1 = \{L^*_5 \rightarrow L^*_7 \rightarrow L^*_4 \rightarrow L^*_8 \rightarrow L_{35}\}$) is through the condition node L^*_4 which is eliminated from L_{st}. Thus, the L_{st} for this condition node (L^*_5) is empty. As illustrated in Fig. 6, the update on *sink* (node L_{35}) is fully controlled by the condition node L_{35} which does not have any sensitive variables.

C. Implementation Details

The *Static Analyzer* module is implemented using the LibTooling library of the Clang compiler [13]. To access relevant nodes in the AST (generated by Clang) of a given VP, we use the primary node visitor *RecursiveASTVisitor* of Clang. It provides designers with a recursive mechanism on the entire AST to visit each node based on the DFS algorithm. The *VisitCXXRecordDecl* (as SC_MODULE is defined based on *class* or *struct* in SystemC) and *VisitFunctionDecl* are used to find the deceleration nodes of modules and functions in the AST, respectively. The information (i.e., name and type) of modules' ports are extracted by accessing node type *FieldDecl*. The member functions of VP's modules are retrieved by visiting node type *CXXMethodDecl*. We extract the locations where transactions are defined as function arguments or local variables within the function's body by finding the node type *DeclRefExpr* in the AST. We take advantage of the node type *CXXMemberCallExpr* in the AST to extract the function calls and tracing the transaction object which is used as input arguments for the transport interfaces. The *Timing-flow Analyzer* is implemented in C++ based on Algorithm 1.

V. Experimental Results

The proposed approach was applied to two standard real-world VP-based SoCs namely LEON3-based SoCRocket VP [16] and RISC-V VP-based SoC [17]. For each case study, we briefly discuss the architectural features that cause timing flows, the attack model for exploiting, the possible mitigation technique, and the results of our timing flow analysis.

The analysis has been performed on a PC equipped with 24 GB RAM and an Intel core i7-8565U CPU running at 1.80 GHz.

A. Case Study 1: SoCRocket VP

In the first experiment, we consider a common source of timing flow in a given SoC (timing-based covert channel) where different IPs can access a shared resource (e.g., memory). Depending on the access control policies implemented in the SoC, timing flow between IPs when accessing the shared resource may exist. In this case, IPs that are supposed to be isolated can secretively communicate by modulating the access patterns to the shared resource and impacting the time when other IPs can use it.

In order to model this security scenario, we used the LEON3-based SoCRocket VP. The VP itself consists of more than 50,000 lines of code and several IPs working together in master or slave mode which are connected to the on-chip bus AMBA-2.0 AHB (Advanced High-performance Bus). The communication uses a 32-bit address mode where the 12 most significant bits are used to specify the memory address.

We modified the LEON3-based SoCRocket VP by integrating three TLM-2.0 IPs with its AMBA-2.0 AHB which are 1) two initiator modules *ahbin1* and *ahbin2* (act as master), and 2) one shared memory *ahbMem* (act as slave). We also implemented a *Memory Management Unit* (MMU) inside of the AMBA-2.0 AHB to control the access of master IPs to the shared memory. The MMU unit is based on the *Round Robbin* (RR) access policy, providing master IPs with an equal priority to access the shared memory *ahbMem*. To access the shared memory over the bus architecture, master IPs need to create transactions that contain the memory address and the access request. The address attribute of transactions contains the memory address of *ahbMem* while the access request is stored in their extension filed (*access_req*). When AMBA-2.0 AHB receives the first transaction of a given master IP that contains the address of *ahbMem* and its *access_req* is true, MMU grants access to the master IP for a specific quantum time *QT*. Thus, other IPs are not able to access the shared memory within *QT*.

The access control policy of the VP can be validated by defining a security property to check whether access of a given IP to the shared memory is dependent on the access request of other IPs. The security property is defined as follows:

$$\begin{aligned} SP = \{P_1 : (source, c_{src}, sink, c_{snk}) \mid \\ source \leftarrow ahbin1 :: gen\_frame() : access\_req, \\ c_{src} \leftarrow AHBCtrl :: b\_transport() : master\_id = 1, \\ sink \leftarrow AHBCtrl :: mmu() : access\_grnt2, \\ c_{snk} \leftarrow AHBCtrl :: b\_transport() : mem\_id = 2\} \end{aligned} \quad (3)$$

The security property P_1 ensures that the access grant $access\_grnt2$ which is issued by MMU of the AMBA-2.0 AHB for the $ahbin2$ must not be dependent on the $access\_req$ of the $ahbin1$ module. Based on the given security property, our timing flow validation approach could detect a path in the CFG of the VP where $access\_grnt2$ is a child node of a condition node (*else-if* statement) which its condition variable is in the list of source taint (L_{st}). This confirms the existence of a timing flow between $access\_req$ (source) and $access\_grnt2$ (sink) variables, thus, the corresponding lines of code in the MMU and path in the CFG are reported.

In order to block this timing flow, we replaced the RR algorithm of the MMU with a *Time Division Multiple Access* (TDMA) algorithm where the access grant is issued based on a *counter* which is a non-sensitive variable. We have analyzed the VP again, and in this case, no timing flow was detected as the grant access is fully controlled by a counter variable (which is non-sensitive). The analysis took 41.72 seconds to report the results.

B. Case Study 2: RISC-V VP

In the second experiment, we consider the case of a timing-based attack in a given SoC where secret data stored in the secured memory is protected by encryption and authentication process (which is commonly used in SoCs to maintain data privacy and integrity [29]). In the case that the authentication process is dependent on the sensitive data (e.g., secret key), the timing behavior may leak information about the sensitive data.

To model this security scenario, we used the open-source RISC-V VP-based SoC [17] that is implemented in SystemC TLM-2.0. The VP designed as an extensible and configurable platform around a RISC-V RV32IM CPU core with a generic bus system *SimpleBus*. The communication uses a 32-bit address mode where the 16 least significant bits are used to specify the memory address while the four most significant bits are dedicated to initiator module identifications.

We have modified the VP by integrating three TLM-2.0 IPs which are 1) two initiator modules *init1* and *init2* (act as masters) and 2) one secured memory *sec_mem* (act as a slave). The *init1* module contains an RSA unit to encrypt and decrypt secret data when writes to and reads from the secured memory *sec_mem*, respectively. To protect *sec_mem* from unauthorized access, an authentication process is performed on any incoming transactions to the *SimpleBus* as the access control policy. The authentication process is implemented in the *SimpleBus* based on the asymmetric key encryption technique using the RSA algorithm. To access *sec_mem*, the *init1* module needs to generate transactions whose data attribute is encrypted by the RSA unit, the address attribute includes the *sec_mem* memory address, and the authentication element is stored in its extension filed. The RSA unit of the *init1* module uses the private key to encrypt the transactions data, and the public key and transactions address to generate the authentication element. The private key is also shared with the *SimpleBus* to perform the authentication process by decrypting the authentication element of the incoming transactions and checking whether the decrypted authentication element is matched the address of transactions. In the case that these two elements are matched, *SimpleBus* routes the incoming transactions to the *sec_mem* memory.

The security property that must be ensured is that the time for the authentication process (as the access control policy of the SoC) in *SimpleBus* should not be dependent on the private key (sec_key). The security property is defined as follows:

$$\begin{aligned} SP = \{P_1 : (source, \emptyset, sink, \emptyset) \mid \\ source \leftarrow SimpleBus :: sec\_key, \\ sink \leftarrow SimpleBus :: authentication() : decrypt\_elmn\} \end{aligned} \quad (4)$$

The security property P_1 ensures that, for a given incoming transaction to the *SimpleBus*, the decryption process of its authentication element $decrypt\_elmn$ is not dependent on the private key sec_key which is shared with $init1$. Based on the given security property, our timing flow validation approach could detect a path in the CFG of the VP where in $modular\_exp$ function, the decryption process is controlled by a condition node (*if* statement) whose condition variable is in the list of source taint (L_{st}) and directly connected to the sec_key. This confirms the existence of a timing flow between sec_key (source) and $decrypt\_elmn$ (sink) variables, thus, the corresponding lines of code in the *SimpleBus* and path in the CFG are reported.

The main reason for this timing flow vulnerability is that the RSA decryption is implemented with the square and multiply algorithm to perform fast exponentiation (which is a common implementation). In this case, the bits in the private key are checked one by one, and a modulo operation is performed only when the bit is "1". It means that the required time to perform the RSA algorithm is directly dependent on the number of "1" bits, or the hamming weight of the private key sec_key. In the timing attack scenario, an IP (attacker) can continuously send transactions to *SimpleBus* and measure the time to finish those requests. It means that the attacker can estimate the number of "1" bits in the private key by simply measuring its own execution time. Once, the attacker could extract the private key, it can also access sensitive data in *sec_mem* memory.

In order to block this timing-based leakage flow, we have added a *counter* variable into the $modular\_exp$ function of the RSA unit to eliminate the dependency of generating $decrypt\_elmn$ to the value of sec_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 access policy of the *SimpleBus* again, and in this case, no timing flow was detected as the decrypting process is fully controlled by the *counter* variable which is non-sensitive. For this experiment, the analysis took 34.09 seconds to report the results.

VI. Conclusion

In this paper, we proposed a novel VP-based IFT approach to validate a given SoC's security architecture against timing flows. The proposed approach contains a scalable static information flow analysis that operates directly on the AST representation of VPs. In the first phase, the behavior of a given VP is formally represented in terms of data and control flows w.r.t the given security properties. In the second phase, a static taint tracing and path analysis are performed on the formal representation of the VP's behavior to identify all paths that violate the given security properties. The violated paths are reported back to designers, allowing them to improve the security architecture of the SoC. We have demonstrated the applicability and scalability of our approach on two real-world VP-based SoCs. The proposed approach is automated, fast, non-intrusive, and does not rely on any commercial tool.

References

[1] S. Ray and J. Bhadra, "Security challenges in mobile and IoT systems," in *SOCC*, 2016, pp. 356–361.

[2] S. Ray and Y. Jin, "Security policy enforcement in modern SoC designs," in *ICCAD*, 2015, pp. 345–350.

[3] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," *Software safety and security*, vol. 33, pp. 319–347, 2012.

[4] M. Goli and R. Drechsler, "Automated design understanding of SystemC-based virtual prototypes: Data extraction, analysis and visualization," in *ISVLSI*, 2020, pp. 188–193.

[5] I. S. A. et al, "IEEE standard for standard SystemC language reference manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.

[6] J. Aynsley, Ed., *OSCI TLM-2.0 Language Reference Manual*. Open SystemC Initiative (OSCI)., 2009.

[7] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[8] M. Goli, J. Stoppe, and R. Drechsler, "Automated nonintrusive analysis of electronic system level designs," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 39, no. 2, pp. 492–505, 2020.

[9] M. Goli and R. Drechsler, *Automated Analysis of Virtual Prototypes at the Electronic System Level: Design Understanding and Applications*. Springer Nature, 2020.

[10] P. Pieper, V. Herdt, D. Große, and R. Drechsler, "Dynamic information flow tracking for embedded binaries using SystemC-based virtual prototypes," in *DAC*, 2020, pp. 1–6.

[11] M. Goli, M. Hassan, D. Große, and R. Drechsler, "Security validation of VP-based SoCs using dynamic information flow tracking," *it Inf. Technol.*, vol. 61, no. 1, pp. 45–58, 2019.

[12] M. Hassan, V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Early SoC security validation by VP-based static information flow analysis," in *ICCAD*, 2017, pp. 400–407.

[13] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *BSD*, 2008, pp. 1–2.

[14] M. Goli, M. Hassan, D. Große, and R. Drechsler, "Automated analysis of virtual prototypes at electronic system level," in *GLSVLSI*, 2019, pp. 307–310.

[15] M. Goli and R. Drechsler, "Through the looking glass: Automated design understanding of SystemC-based VPs at the ESL," *IEEE Trans. on CAD of Integrated Circuits and Systems*, (accepted) 2021.

[16] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, "Socrocket - A virtual platform for the european space agency's soc development," in *ReCoSoC*, 2014, pp. 1–7, http://github.com/socrocket.

[17] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *J. Syst. Archit.*, vol. 109, 2020.

[18] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," in *PLDI*, 2011, pp. 109–120.

[19] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: a language for hardware-level security policy enforcement," in *ASPLOS*, 2014, pp. 97–112.

[20] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *ASPLOS*, 2015, pp. 503–516.

[21] M.-M. Bidmeshki and Y. Makris, "Toward automatic proof generation for information flow policies in third-party hardware IP," in *HOST*, 2015, pp. 163–168.

[22] X. Guo, R. G. Dutta, and Y. Jin, "Eliminating the hardware-software boundary: A proof-carrying approach for trust evaluation on computer systems," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 2, pp. 405–417, 2017.

[23] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Trans. Inf. Forensics Secur.*, vol. 7, no. 1, pp. 25–40, 2012.

[24] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *DATE*, 2017, pp. 1691–1696.

[25] M. Goli and R. Drechsler, "ATLaS: Automatic detection of timing-based information leakage flows for SystemC HLS designs," in *ASP-DAC*. ACM, 2021, pp. 67–72.

[26] Z. Jiang, S. Dai, G. E. Suh, and Z. Zhang, "High-level synthesis with timing-sensitive information flow enforcement," in *ICCAD*, 2018, pp. 1–8.

[27] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Symposium on Security and Privacy*, 2012, pp. 11–20.

[28] A. Ardeshiricham, W. Hu, and R. Kastner, "Clepsydra: Modeling timing flows in hardware designs," in *ICCAD*, 2017, pp. 147–154.

[29] F. Hou, H. He, N. Xiao, F. Liu, and G. Zhong, "Efficient encryption-authentication of shared bus-memory in SMP system," in *CIT*, 2010, pp. 871–876.