VIP-VP: Early Validation of SoCs Information Flow Policies using SystemC-based Virtual Prototypes

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*—The emergence of *Virtual Prototypes* (VPs) at the *Electronic System Level* (ESL) has played a major role in modernizing the System-on-Chips (SoCs) design flow to raise design productivity and reduce time-to-market constraint. Leveraging VPs and extending their use-cases for early security validation are shown as a promising direction. As the cost of fixing any security flaws increases with the stage of development, VP-based security validation can significantly avoid costly iterations.**

In this paper, we propose VIP-VP, a novel VP-based dynamic information flow analysis approach at the ESL.VIP-VP enables designers to validate the information flow policies of a given VP-based SoC against security threat models, such as information leakage (confidentiality) and unauthorized access to data in a memory (integrity). Experimental results including a real-world VP-based SoC demonstrate the scalability and applicability of the proposed approach.

I. Introduction

In order to reduce the design costs and meet the time-to-market constraint, the modern *System-on-Chip* (SoC) design flow has shifted from in-house development of *Intellectual Properties* (IPs) to the use and reuse of existing commercial IPs. By this, designers often leverage libraries, toolkits, and components from third-party vendors. This decentralization also raises the concern that legitimate commercial off-the-shelf IPs may manipulate or assist in manipulating secret data in such a way that their users do not expect. As a result, modern SoCs are notoriously insecure where third-party IPs, in particular, can be used as a vehicle for malice. To overcome this problem and provide strong security for the entire SoC, designers take advantage of the IP isolation technique based on the notion of non-interference. This defines the information flow policies in hardware which is based on the fact that for a given SoC, certain parts of the system (considered as secure zones) should never interfere with other parts (insecure zones). Therefore, for security validation of a given SoC, a common property that needs to be checked is non-interference.

Information Flow Tracking (IFT) [1] has been shown as a powerful technique to help mitigate security vulnerabilities that violate certain information flow policies and non-interference properties (i.e., confidentiality and integrity). 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 by an insecure one.

Since the cost of fixing any security flaws (e.g., information flow policies) 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. A VP is an abstract and executable software model that is typically implemented using SystemC [2], [3] and its *Transaction Level Modeling* (TLM) [4] framework at the *Electronic System Level* (ESL) [5]–[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.

IFT methods have been developed to validate hardware designs [10]–[13], or software models through the use of source-level instrumentation [14], [15] and binary instrumentation [16]. Most of the works for security validation of digital hardware designs are only applicable at the abstraction levels of RTL and below. At the ESL, there exist only a few works [17]–[19] on security validation of VP-based SoCs. While the results of their analysis are complementary to our approach, they have some limitations in terms of scalability issues [18], inability to identify leakage flow in the case of dynamic variables [19], and low degree of automation [17].

In this paper, we focus on the security validation of VP-based SoCs, in particular, helping system designers to detect non-interference property violations and to pinpoint security flaws caused by poor information flow policies in the early stage of the SoC design process. We present VIP-VP, an Early **V**alidation of SoCs **I**nformation Flow **P**olicies using SystemC-based **V**irtual **P**rototypes. VIP-VP is based on the dynamic IFT technique to identify security flaws of VP-based SoCs and consists of three main phases: 1) information extraction, 2) property generation, and 3) security validation. In the first phase, we build on the flexible Clang compiler [20] to statically analyze the *Abstract Syntax Tree* (AST) of a given SystemC-based VP and generate an instrumented version of the VP for transactions extraction at run-time. In the second

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.

978-1-6654-1825-6/21/$31.00 ©2021 IEEE

phase, the extracted transactions which represent the entire simulation behavior of the VP are translated into a set of transaction flows. In the third phase, the information flow policies which are given by designers are translated into set of properties, then the translated run-time behavior of the VP (transaction flows) is validated against the generated properties.

The proposed approach is applied to two VP-based SoCs including the real-world LEON3-based SoCRocket VP [21] to show its scalability and applicability. VIP-VP is automated, fast, and does not rely on any commercial tool for its analysis.

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 [22], Sapper [23], SecVerilog [24], and VeriCoq-IFT [25] are hardware security design languages that allow designers to label and track information flow. For example, Caisson [22] and Sapper [23] 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) [26], [27] 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 [10] 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 is limited to single IP cores. Moreover, all the aforementioned methods are only applicable at RTL and do not support SystemC constructs.

The existing verification methods [28]–[30] at the ESL are not able to detect security threat models as the design functionality (and its related protocol rules) is not affected. At the ESL, there are only a few works [17]–[19] on security validation of VP-based SoCs. The method in [19] presents a static VP-based IFT solution for security validation of given SoC. However, the main drawback of the static approaches is that if the address of transactions is defined at run-time, e.g., generated either explicitly by initiator modules (based on some dynamic computation) or implicitly by its running software, they are not able to detect this security violation.

In [18], a dynamic VP-based IFT method is introduced which overcomes the limitation of the static approaches. The

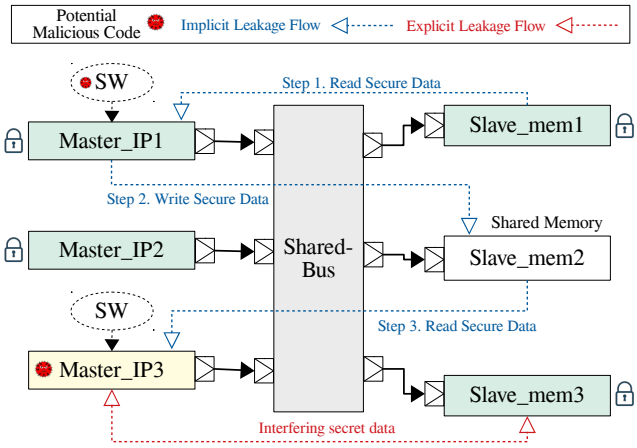

Fig. 1: Example of explicit and implicit information flows.

method is based on the *GNU Debugger* (GDB) to extract the run-time behavior of a given VP and validate it against a set of security properties. As the method takes advantage of the GDB to monitor the run-time behavior, the required time for its analysis can be huge. It means that if the detection of security flaws needs a long run-time trace of the VP's behavior, the method may fail.

Recently, [17] proposes a dynamic IFT of software binaries targeting embedded systems. The method works by integrating a dynamic IFT engine in combination with the security policy into a given VP of the embedded system. However, this integration requires lots of manual effort that must be performed by designers before using the approach (e.g., adapting the memory interface and replacing the register type). Moreover, as the method is built upon the assumption that the hardware IP is secure, it cannot detect information leakage flow if the SoC includes e.g., a synthesizable IP (integrated with the SoC using a TLM-2.0 interface) containing malicious parts to exploit the confidential data.

The method in [31] introduces a timing flow analysis technique to validate SystemC HLS designs against timing-based side channel attacks. However, it does not support TLM-2.0 constructs.

III. SystemC and TLM Framework

SystemC is a C++-based system level design language, providing designers with an event-driven simulation kernel. SystemC is considered as a de-facto standard for modeling hardware/software co-design and creating VPs using its TLM framework at the ESL. TLM-2.0 (as the current standard) introduces the transaction concept allowing designers to describe a model in terms of abstract communications (supporting both *Loosely-timed* (LT), and *Approximately-timed* (AT) models) using the base protocol, standard interfaces (e.g., *b_transport* and *nb_transport*), initiator and target sockets, the generic payload, and the utilities. A transaction is a data structure (i.e., a C++ object), consisting of several attributes such as data, address, and command (e.g., read or write). Transactions are transmitted through TLM modules using function calls. TLM modules may be implemented as initiators, interconnects, or

targets. An initiator module initiates new transactions through the initiator socket, an interconnect acts as a transaction router and forwards the incoming transactions without modifying them. The target module is the endpoint for the transactions and responds to the incoming transactions.

IV. Threat Models and information flow

The threat models of leaking information with respect to the notion of non-interference can be divided into two common categories: confidentiality and integrity. The former refers to information flow wherein data of secure IP (e.g., data stored in a secure memory) is retrieved by an unauthorized IP. The latter refers to information flow in which data of secure IP is modified by an unauthorized IP.

With respect to the above threat models, secure assets can be inferred through two general classes of information flow in a given VP-based SoC: explicit and implicit. Explicit information flow results from two modules directly communicating. For example, an explicit flow can occur between an initiator module and a target module that are directly exchanging data through a shared interconnect. Fig.1 shows such a scenario where module *Master_IP3* access memory *Slave_mem3* through the shared interconnect *Shared-Bus*. Implicit information flows are much more subtle and generally leak information through behavior. For example, an implicit flow can occur between two modules where one belongs to the trusted zone and the other to the untrusted zone through shared memory. In Fig. 1, an implicit information flow causes sensitive data to be read from the secure memory *Slave_mem1* (step 1) by the trustworthy initiator module *Master_IP1* and then written to the shared memory *Slave_mem2* (step 2) which potentially is accessible by initiator module *Master_IP3* which belongs to untrusted zone (step 3).

In both cases, the unwanted leakage flow can occur based on one of the following security scenarios (or a combination of them):

- The SoC includes a synthesizable IP core purchased from an untrusted third-party vendor and integrated with the shared interconnect using a TLM-2.0 interface. The IP may contain malicious part to exploit the confidential data e.g., the explicit leakage flow in Fig. 1.
- Malicious software running on the (trustworthy) hardware IP may exploit hardware backdoors to cause malfunctions or leak secret data. The first two steps of implicit information flow in Fig. 1 show such a threat model where an adversary software running on the secure module *Master_IP1* controls the IP to read secret data from *Slave_mem1* and write it to the shared memory *Slave_mem2*. In this case, the stored secret data in the shared memory can be accessible by unauthorized IPs.
- An incorrect initialization (either by an adversary involved in the SoC design process or unintentionally) of the SoC firmware (e.g., memory configuration file) can cause an unauthorized IP to access the sensitive data stored in the secure memory.
- The existing SoC is extended or modified but its information flow policies are not updated. Especially in case that the design team decides to improve the existing SoC by adding new IPs (e.g., an accelerator, a processor or a memory), the initial security policies may not be sufficient to protect the sensitive data against leakage. For example, designers add *Slave_mem2* as shared memory to increase the overall performance of the system. However, after this modification, they did not update the information flow policies of the SoC.

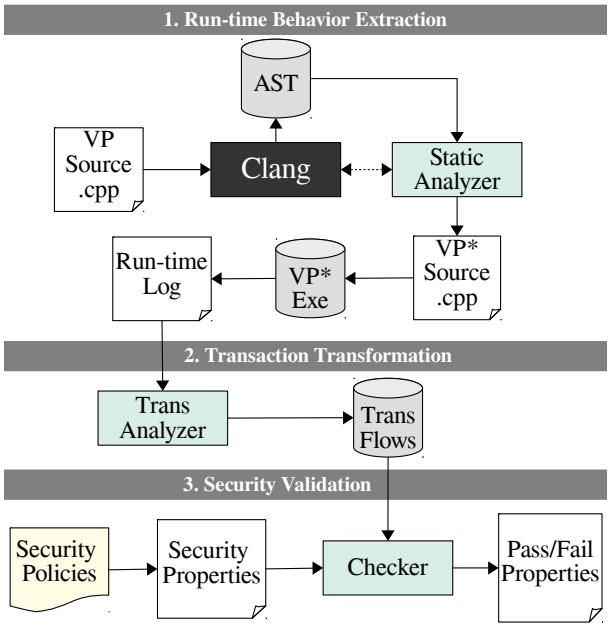

Fig. 2: VIP-VP methodology overview.

The lack of well-implemented information flow (or access control) policies makes the system susceptible to vulnerability. For a given SoC, information flow policies specify the authorized information flows in the system and how the different combination of labeled data is computed when the data propagates through the system. A strong security policy in the SoC could prevent the access of unauthorized IPs to sensitive data stored in secure memories and block leakage flows. Therefore, for a given SoC, the information flow policy validation is of utmost importance and must be performed to ensure that the secure assets cannot be inferred either through implicit or explicit information flows.

V. Information flow policy Validation Methodology

The overall workflow of VIP-VP is illustrated in Fig. 2, consisting of three main phases which are 1) run-time behavior extraction and 2) transaction transformation, and 3) security validation.

In the first phase, we take advantage of the Clang compiler to analyze the AST (generated by Clang) of VP to generate an instrumented version of the VP source code for tracing transactions at run-time (inspired by [32]). Next, a post-execution analysis is performed on the extracted transactions to transform them into a set of transaction flows which represents the entire simulation behavior of the VP in a big-picture view.

Each transaction flow identifies the flow of information from the *source* (where a transaction is created by an initiator IP) to the *sink* (where the transaction is received by a target IP). In the third phase, first, the information flow policies of design are translated into a set of security properties w.r.t the explicit and implicit information flow models. Second, the generated transaction flows are validated against the generated security properties to ensure that there is no flow between an unauthorized source and a secure sink. The violated properties are reported back to designers, allowing them to improve the information flow policies of the SoC.

A. Run-time Behavior Extraction

In a given VP-based SoC, transporting the (sensitive) data is performed through interconnects between different IP modules by means of TLM transactions. Therefore, analyzing the flow of information in the VP and monitoring its simulation behavior can be done by extracting all transactions. In order to trace a given VP's transactions, we take advantage of the Clang compiler to generate an instrumented version of the VP source code. To do this, the *Static Analyzer* module (Fig. 2–phase 1) analyzes the AST of VP (which is generated by Clang from its source code) in two steps. First, the information related to the VP's structure is extracted by visiting the relevant node in the AST. The static information that needs to be extracted is as follows:

- The name of TLM IPs and their member functions and processes to which a transaction object is referenced.
- Type of each TLM IP which is identified by analyzing its socket(s) type. Due to the TLM-2.0 rules an initiator module only has initiator socket(s), a target module only has target socket(s) and an interconnect module has both types of socket.

In the second step, we take advantage of the extracted information to automatically generate an instrumented version of the VP source code, including the *Trans_Recorder* statements. The *Trans_Recorder* statement consists of the instructions to extract run time information that is needed to trace flow of transactions. This information is the reference address of transactions(consider as transactions ID), the run-time value of their attributes (address and command), and related parameters such as phase (e.g., BEGIN_REQ) and functions' return status (e.g., TLM_COMPLETED). The simulation time is also extracted to determine the time at which the transaction is processed and the orders of different IPs communication. The order of communication is required to detect the implicit flow in the security validation phase. To trace the flow of transactions when they are transmitted through different IPs, the *Trans_Recorder* statements must be inserted into the locations where 1) the transaction is defined (e.g., as function arguments or local variables within the function's body) and 2) the transaction object is used as an input argument for transport interface (e.g. *b_transport*) calls.

For example, assume we want to trace the flow of transactions generated by the *Master_IP1* module of the SoC in Fig. 1. A part of the *Master_IP1* module is shown in Fig. 3 where the transaction object is used as an input

```
1 struct Master_IP1: sc_module {
2 tlm_utils::simple_initiator_socket<Master_IP1> socket;
3 void thread1(){
4 /*...*/
5  socket->b_transport (*trans, delay);
6  Fout << "Master_IP1::thread1::trans_ID = " << trans <<
       "data =" << trans->get_data_ptr << "cmd =" <<
       trans->get_command << "addr =" << trans->
       get_address << "sim_time ="<<sc_time_stamp() << "
       IP_instance_name" << this->name() << endl;
7 /*...*/ }
```

Fig. 3: Part of the instrumented code of the *Master_IP1*.

argument for the *b_transport* interface call (Line 5). Hence, the *Trans_Recorder* statement *Fout* (Line 6) is automatically generated and inserted into the new source code by the *Static Analyzer* module (Fig. 2–phase 1).

B. Transaction Transformation

After extracting the simulation behavior of a given VP-based SoC, the next step is to transform the extracted transactions into a well-structured format. Since the information flow policies of the VP are defined based on abstract communication between the initiator and the target modules, the extracted transactions should be transformed in such a way that describes the flow of data among different IPs w.r.t the abstract communication. Hence, a complete simulation behavior of the VP can be defined as a set of transaction flows S_{TF} where each TF shows a connection (i.e., the flow of data) between the *source* (initiator module) and *sink* (target module) as follows:

$$S_{TF} = \{TF_i \mid TF_i = \{source \rightarrow sink :: (TID, addr, cmd, ST)\}, 1 \leq i \leq n_{TF}\} \quad (1)$$

where *TID* denotes the transaction ID which is the transaction reference address and used to distinguish the generated transactions of different initiator modules. The *addr* and *cmd* depict the transaction address and command attributes. They are used to identify the type (read or write) and location of memory in which the *source* IP accesses the *sink* IP. The *ST* parameter represents the simulation time at which the transaction is transmitted. The n_{TF} parameter shows the number of generated transaction flows.

As illustrated in Fig. 2–phase 2, the *Trans Analyzer* module receives the *Run-time Log* file as an input and automatically generates a set of transaction flows based on the definition in (1). Each member of the S_{TF} stored in *Trans Flows* specifies an explicit flow of information between *source* and *sink*. The implicit information flow is not directly visible in S_{TF} as it is created based on a combination of explicit flows where an unauthorized initiator IP accesses the data of a secure target IP which is implicitly inaccessible.

For example, Fig. 4 shows a part of transaction flows of the VP-based SoC in Fig. 1 where TF1, TF2, ... and TF6 specify six explicit transaction flows. The combination of the transaction flows TF2, TF4, and TF6 shows an implicit flow of data between *Master_IP3* and *Slave_mem1*. In TF2, the *Master_IP1* generates a transaction to read data from secure

TF1: {Master_IP3 –> Slave_mem3 :: (0x442C011_1, 0x04, read, 10 ns)}
TF2: {Master_IP1 –> Slave_mem1 :: (0x475B02C_1, 0x92, read, 45 ns)}
TF3: {Master_IP2 –> Slave_mem3 :: (0x512DA09_1, 0x46, write, 85 ns)}
TF4: {Master_IP1 –> Slave_mem2 :: (0x475B02C_2, 0x0B, write, 140 ns)}
TF5: {Master_IP2 –> Slave_mem1 :: (0x512DA09_2, 0x04, write, 160 ns)}
TF6: {Master_IP3 –> Slave_mem2 :: (0x442C011_2, 0x0B, read, 210 ns)}

Fig. 4: Part of the transaction flows of SoC in Fig. 1.

memory *Slave_mem1*. In FT4, *Master_IP1* writes the read confidential data into shared memory *Slave_mem2*, and finally in TF6 the confidential data is read by *Master_IP3*. Therefore, the *Master_IP3* could access the confidential data by taking advantage of *Master_IP1*.

Therefore, the goal of the next phase is to generate security properties from information flow policies of a given SoC in such a way that detects all implicit and explicit leakage flows.

C. Security Validation

The first step of performing security validation is to translate the information flow policies of the SoC into a set of security properties. The information flow policies are a set of rules which ensure that no secret data is leaked through untrusted channels and typically written in a textbook specification (defined as reference model). Thus, designers use them to implement the access control policies of the interconnect module or other mechanisms to prevent information leakage. To generate security properties that validate the system against both the explicit and implicit leakage flows, we take the following information from designers as inputs.

- List of secure IPs including both the initiator (*source*) and target (*sink*) modules:

$$SIP_{source} = \{IP_1, IP_2, ..., IP_n\}$$
$$SIP_{sink} = \{IP_1, IP_2, ..., IP_m\} \quad (2)$$

- List of forbidden information flows between $source_i$ and $sink_j$ derived by the information flow policies of the SoC:

$$Forbid_{flow} = \{source_i \ \rightarrow \ sink_j \ (addr\_range) :: no \ flow \ \} \quad (3)$$

Each member of $forbid_{flow}$ in (3) specifies that there must be no flow of information between $source_i$ and $sink_j$. The $addr\_rang$ parameter is optional and use when a certain part (address range) of the $sink_j$ is secured. While this specification, itself, can be considered as an explicit security property, it cannot be used directly to check the implicit flow of information. To cover explicit flows, we need to generate properties that check no unauthorized IPs take advantage of authorized IPs to access confidential data in secure memories (the threat model illustrated in Fig. 1). Hence, we proposed Algorithm 1 to automatically generate the implicit security properties based on SIP_{source}, SIP_{sink}, and $forbid_{flow}$. For each forbidden flow f in $forbid_{flow}$, the algorithm generates three transaction flows TF_t, TF_{t+1}, and TF_{t+2} which are required to shape an implicit channel (Lines 1-6). The index parameter t depicts the order in which the explicit transaction flows must occur. The TF_t property specifies a transaction flow where a secure IP reads secret data from the secure memory (*sink*) specified in f (Line 3). The TF_{t+1} shows a transaction flow where the secure IP writes the secret data in an unauthorized memory (Line 5). The TF_{t+2} describes the transaction flow where the unauthorized IP (*source*) specified in f read the secret data from the unauthorized memory (Line 6).

For example, consider the VP-based SoC in Fig. 1 where designers want to check there is no flow of information between *Master_IP3* and secure memory *Slave_mem1*. The inputs of the algorithm are as follows:

$$SIP_{source} = \{Master\_IP1, Master\_IP2\}$$
$$SIP_{sink} = \{Slave\_mem1, Slave\_mem3\}$$
$$Forbid_{flow} = \{Master\_IP3 \ \rightarrow \ Slave\_mem1 :: no \ flow \ \} \quad (4)$$

Based on the Algorithm 1, the following implicit security properties are generated.

$$\begin{aligned} ISP = \{p_1 = \{&Master\_IP1 \rightarrow Slave\_mem1 : read\}, \\ &\{Master\_IP1 \rightarrow Slave\_mem2 : write\}, \\ &\{Master\_IP3 \rightarrow Slave\_mem2 : read\}, \\ p_2 = \{&Master\_IP2 \rightarrow Slave\_mem1 : read\}, \\ &\{Master\_IP2 \rightarrow Slave\_mem2 : write\}, \\ &\{Master\_IP3 \rightarrow Slave\_mem2 : read\}\} \end{aligned} \quad (5)$$

The p_1 and p_2 properties in (5) ensure that the *Master_IP3* does not take advantage of authorized *Master_IP1* or *Master_IP2* to access confidential data in secure memory *Slave_mem1* via shared memory *Slave_mem2*.

In the next step, the transaction flows are validated against the generated security properties by the *Checker* module (Fig. 2–phase 3). To do this, each forbidden flow in the $Forbid_{flow}$ is considered as an explicit property and the S_{TF} of the SoC is traversed to find information flow policies violation (i.e, the existence of an explicit information leakage flow). Regarding the implicit information leakage flows, for each property p in *ISP* the S_{TF} is traversed to find the sequence of transaction flows (TF_t, TF_{t+1}, TF_{t+2}) specified in the property p. If these three sequences are found, the property is violated and the corresponding transaction flows are reported back to designers. In addition to this scenario, there might be a case that only the first two sequences of the property are violated. It means that an authorized initiator IP reads sensitive data from a secure memory and then writes the data to an unauthorized memory. We consider this case as a suspect case as it may lead to information leakage. Thus, the corresponding transaction flows are also reported back to designers.

Coming back to the VP-based SoC in Fig. 1, analyzing the transaction flows (Fig. 4) by *Checker* (Fig. 2–phase 3) module shows that the first property p_1 in the ISP is violated. The transaction flows which reported back to designers are TF_2, TF_4, and TF_6.

VI. Experimental Results

The proposed approach was evaluated by two VP-based SoCs. The experiments cover both the generality and scalability of the proposed approach. The former refers to the information flow policies validation of designs implementing various aspects of the TLM-2.0 standard (core-interfaces, the

Algorithm 1 Timing Flow Analyzer

Input: SIP_{source}, SIP_{sink}, $forbid_{flow}$
Output: Implicit security properties ISP

for each flow $f \in forbid_{flow}$ **do**
  for each $S_{ip} \in SIP_{source}$ **do**
    $TF_t \leftarrow (S_{ip} \rightarrow sink :: read)$
    for each $IS_{mem} \notin SIP_{sink}$ **do**
      $TF_{t+1} \leftarrow (S_{ip} \rightarrow IS_{mem} :: write)$
      $TF_{t+2} \leftarrow (source \rightarrow IS_{mem} :: read)$
      $P_i \leftarrow \{TF_t, TF_{t+1}, TF_{t+2}\}$
      $ISP \leftarrow P_i$
      $i \leftarrow i + 1$

base protocol, and coding styles) [33]. The latter refers to the information flow policies validation of a real-word VP-based SoC [21].

The *Static Analyzer* module is implemented using the LibTooling library of the Clang compiler [20]. To access relevant nodes in the AST (generated by Clang) of a given VP, we use the primary node visitor *RecursiveASTVisitor* of Clang. The *Trans Analyzer*, *Checker*, and the algorithm to generate security properties of the second and third phases of VIP-VP are implemented in C++. 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 applied our information flow policies validation approach on the real-word LEON3-based VP SoCRocket [21] and consider the first security scenario introduced in Section IV where a third-party IP contains a malicious part to exploit the confidential data. The VP 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. The communication uses a 32-bit address mode where the 12 most significant bits are used to specify the memory address.

To model the above security scenario, we modified the SoC by integrating three TLM-2.0 IPs with its *AMBA-2.0 AHB* (i.e., Advanced High-performance Bus). These IPs are a synthesizable SystemC *AES_core* [34] used as a hardware accelerator to implement AES-128 encryption algorithm, and a secure memory *Slave_mem1* initialized by cryptography keys and plain texts, respectively. The *AES_core* works in cipher block chaining mode, i.e., an initialization vector, and a plaint text are given as inputs to the IP, and key is read from *Slave_mem1*, and the IP generates a ciphertext. The IP uses the generated ciphertext as the new initialization vector for the next iteration.

The design team purchases the *AES_core* (third-party IP) and integrates it with the AMBA-2.0 AHB using a TLM-2.0 interface. Note that the synthesizable IP in this case can be used to achive two goals: 1) to develop and test the functionality of VP when all IPs are available, and 2) to reuse it at the later stage of the SoC design flow (e.g., RTL). At the beginning of execution, all initiator modules read the memory configuration file to extract the range of memory addresses (defined in SoCRocket as *mem_addr*) that they are allowed to access. In the memory configuration file, the *Slave_mem1* is only accessible by *AES_core* with *mem_addr = 0xB00*. The *AHBMem* (default memory of the VP) connected to *AMBA-2.0 AHB* bus is shared between IPs such as *LEON3* processor and *ahbin* input device with *mem_addr = 0xA00*.

The security scenario is that the *AES_core* contains a malicious code that executes after every 5000 cryptography text generations. It writes into *AHBMem* the secret key that has already read from secure memory *Slave_mem1*. As all master IPs read the memory configuration file, the *AES_core* can extract the *mem_addr* of other memories for which it is not defined. Due to the nature of secure memory *Slave_mem1* (as it stores cryptographic keys), the information flow policies are that *LEON3* processor and *ahbin* are not allowed to read (confidentiality) or write (integrity) these keys from *Slave_mem1*. Thus, a part of the information flow policies of the VP are defined as follows:

$$\begin{aligned} &SIP_{source} = \{AES\_core\} \\ &SIP_{sink} = \{Slave\_mem1\} \\ &Forbid_{flow} = \{\{LEON3 \ \rightarrow \ Slave\_mem1 :: no\ flow\}, \\ &\qquad\qquad \{ahbin \ \rightarrow \ Slave\_mem1 :: no\ flow\}\} \end{aligned} \tag{6}$$

Based on the aforementioned description, VIP-VP generates two explicit and two implicit security properties. The results of information flow policies validation report no explicit flow but an implicit flow of information where *LEON3* potentially can access the secret keys stored in the secure memory *Slave_mem1* through the *AES_core* and *AHBMem*. The first transaction flows related to the violated property are as follows:

$$\begin{aligned} FT_{712} :\ &\{AES\_core \rightarrow Slave\_mem1 :: \\ &(0x20AC109\_1, 0xB0000010, read, 1050ns)\} \\ FT_{5401} :\ &\{AES\_core \rightarrow AHBMem :: \\ &(0x20AC109\_5004, 0xA0000A02, write, 11750ns)\} \\ FT_{6721} :\ &\{LEON3 \rightarrow AHBMem :: \\ &(0x1DBCD00\_4026, 0xB0000F90, read, 13950ns)\} \end{aligned} \tag{7}$$

The main reason for this security vulnerability is the weak information flow policies of AMBA-2.0 AHB. The only policy implemented in AMBA-2.0 AHB is that for receiving transactions generated by master IPs, it checks whether the transactions' address is in the range of memory addresses. We fixed this security flaw in AMBA-2.0 AHB by adding access control policies that restrict the access of unauthorized master IPs (i.e., *LEON3* and *ahbin*) to the secure memory (*Slave_mem1*) as well as blocking the flow of transactions from authorized master IPs (i.e. *AES_core*) to unauthorized memories (i.e. *AHBMem*). All properties were satisfied on the next analysis run. For this experiment, the number of extracted transactions is 60,000 and the whole analysis took about 48.3 seconds to report the results. We also applied the security validation method in [18] to the SoCRocket VP with the same security scenario, however, due to a large number of transaction flows, the method failed to validate the VP in four hours (considered as time-out).

To demonstrate the scalability of our approach, we repeated this experience for different benchmarks running on the *LEON3* processor of the VP. We also compared (Fig. 5) the required analysis time of the VIP-VP to the pure compilation

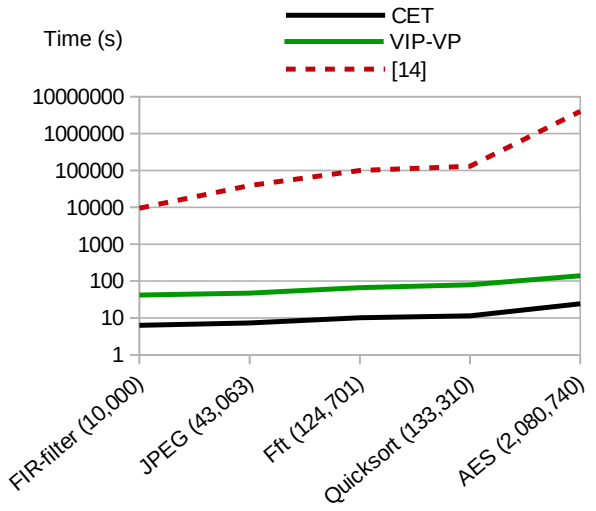

Fig. 5: Execution time of VIP-VP compared to [18] and CET of SoCRocket VP for variuse benchmarks running on LEON3 processor.

and execution time (CET) of the VP (as a baseline) and the method in [18]. As illustrated in Fig. 5, for each benchmark, the number of extracted transaction flows is presented in the parentheses. As can be seen in this figure, unlike [18], the execution time of the VIP-VP grows smoothly and similar to the CET when the software complexity increases. Since the time-out was set to four hours, an estimation of the execution time of [18] is shown in this figure. The estimation is obtained based on the execution time of [18] to analyze the SoCRocket VP for the running software on the LEON3 processor, generating up to 1000 transactions.

This also shows that VIP-VP can provide designers with a long run-time trace of a given VP's simulation behavior which is necessary to detect the threat of this security scenario as the malicious code in the IP may be activated under rare conditions.

B. Case Study 2: AES128-SoC VP

In the second experiment, a VP-based SoC is implemented by modification of the *AT-example* in [33]. The VP includes four initiator IPs specified by type A to D (*initA* to *initD*), an AT interconnect (AT-bus), five targets specified by type A to E (memA to memE) and a memory configuration file (specifying the memory address that each initiator is allowed to access). The difference between type A to D initiators and type A to E targets is based on various cases (9 of the 13 permitted phase transitions) of the TLM-2.0 base protocol [4]. The *AT-bus* is a generic interconnect that can support up to four initiator and eight target modules. The communication uses a 32-bit address mode as follows: 1) bits 0 to 11 – local address inside a memory, 2) bits 12 to 15 – memory address, 3) bits 16 to 23 – initiator ID, and 4) bits 24 to 31 – unused. The *initA* module executes the standard AES-128 encryption algorithm using the initialized keys and plain texts stored in secure memories *memA* and *memB*, respectively.

The security policies of the *AT-bus* are as follows:

- *memA* and *memB* are secure memories and only accessible by *initA*,
- *memC* and *memD* are regular memories and only accessible by *initB*, *initC* and *initD*.

Initially, memory *memE* is not available and memory configuration is defined based on the aforementioned security policies. At the beginning of execution, initiator modules read the memory configuration file to extract the range of memory addresses that they are allowed to access.

Now, consider the scenario that the design team decides to integrate *memE* with the *AT-bus* to increase the overall performance of the system. To use the new memory by other initiators, the memory configuration file needs to be modified. The expected update from the design team for memory configuration is as follows: in *memE*, memory blocks

- (0x000 to 0xBA4) are shared among *initB*, *initC* and *initD*,
- (0xBA5 to 0xDE6) are only accessible by *initA*, and
- (0xDE7 to 0xFFF) are shared between initC and initD.

To evaluate the quality of VIP-VP, we considered a combination of the third and fourth security scenarios in Section IV where the memory configuration file is incorrectly updated (either by a malicious insider on the design team or unintentionally) for the modified SoC as follows: in *memE*, memory blocks

- (0x000 to **0xBC4**) are shared among *initB*, *initC* and *initD*,
- (**0xBA5** to 0xDE6) are only accessible by *initA*, and
- (0xDE7 to 0xFFF) are shared between *initC* and *initD*.

The above incorrect update of the memory configuration file potentially enables the unauthorized IPs *initB*, *initC*, and *initD* to access a range of memory *memE* to which they are not allowed. Hence, a part of the information flow policies of the SoC is defined as follows:

$$
\begin{aligned}
&SIP_{source} = \{initA\}\\
&SIP_{sink} = \{memA, memB, memE(0xBA5 - 0xDE6)\}\\
&Forbid_{flow} = \{\{initB \rightarrow memA :: no\ flow\},\\
&\qquad\qquad \{initB \rightarrow memB :: no\ flow\},\\
&\qquad\qquad \{initB \rightarrow memE(0xBA5 - 0xDE6) :: no\ flow\},\\
&\qquad\qquad\qquad ...\\
&\qquad\qquad \{initD \rightarrow memE(0xBA5 - 0xDE6) :: no\ flow\}\}
\end{aligned}
\tag{8}
$$

Our information flow policies approach generates 27 security properties (nine explicit and 18 implicit) w.r.t (8). VIP-VP detects three explicit and two implicit security property violations. For this experiment, the number of extracted transactions is 33,000 and the whole analysis took about 21.7 seconds to report the results. As an instance, the first corresponding transaction flows of the violated explicit security property is as follows:

$$
\begin{aligned}
FT_{469} &: \{initB \rightarrow memE ::\\
&\quad (0x69C450\_0, 0x00025BA7, read, 660ns)\}\\
FT_{1093} &: \{initC \rightarrow memE ::\\
&\quad (0x6A9B20\_0, 0x00035BB1, read, 1150ns)\}\\
FT_{2721} &: \{initD \rightarrow memE ::\\
&\quad (0x6AC620\_0, 0x00045BB5, write, 3470ns)\}
\end{aligned}
\tag{9}
$$

The main reason for this security flaw is that after integrating the new memory (*memE*) with *AT-bus*, the interconnect information flow policies are not updated. A possible solution is that the *AT-bus* blocks transactions of unauthorized initiator IPs from accessing the secure memories. This can be done by checking the address of the received transactions that whether they satisfy the expected range of addresses declared in the memory configuration file. A more general solution is to add a memory management unit to the *AT-bus*.

In this experiment, we also applied the security validation method in [18] to the AES128-SoC VP with the same security scenario. However, due to a large number of transaction flows, the method failed to validate the VP in four hours (considered as time-out).

VII. Conclusion

In this paper, we proposed VIP-VP, an automated and fast VP-based information flow policies validation approach. At the heart of the approach is a dynamic IFT which is performed by automatically extracting the run-time simulation behavior (TLM transactions) of a given VP-based SoC. The extracted transactions and the design information flow policies are translated into a set of transaction flows and properties, respectively. The generated transaction flows are validated against the generated properties and potentially vulnerable flows are reported back to designers for further inspection. We have demonstrated the applicability and scalability of our approach on two VP-based SoCs including the real-world SoCRocket VP.

References

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

[2] A. S. Initiative., http://www.accellera.org/downloads/standards/systemc, 2016.

[3] 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.

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

[5] 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.

[6] M. Goli, J. Stoppe, and R. Drechsler, "Automated nonintrusive analysis of electronic system level designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 2, pp. 492–505, 2020.

[7] M. Goli and R. Drechsler, "Automated design understanding of SystemC-based virtual prototypes: Data extraction, analysis and visualization," in *IEEE Annual Symposium on VLSI (ISVLSI)*, 2020, pp. 188–193.

[8] ——, *Automated Analysis of Virtual Prototypes at the Electronic System Level: Design Understanding and Applications*. Springer Nature, 2020.

[9] ——, "Through the looking glass: Automated design understanding of SystemC-based VPs at the ESL," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2021.

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

[11] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "Theoretical fundamentals of gate level information flow tracking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 8, pp. 1128–1140, 2011.

[12] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *International conference on Architectural support for programming languages and operating systems*, 2009, pp. 109–120.

[13] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *ACM Sigplan Notices*, vol. 39, no. 11, pp. 85–96, 2004.

[14] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks." in *USENIX Security Symposium*, 2006, pp. 121–136.

[15] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *International Conference on Software Engineering*, 2009, pp. 474–484.

[16] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," in *International Conference on Virtual Execution Environments (VEE)*, 2012, pp. 121–132.

[17] P. Pieper, V. Herdt, D. Große, and R. Drechsler, "Dynamic information flow tracking for embedded binaries using systemc-based virtual prototypes," in *Design Automation Conference (DAC)*, 2020, pp. 1–6.

[18] 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.

[19] M. Hassan, V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Early SoC security validation by vp-based static information flow analysis," in *International Conference on Computer-Aided Design*, 2017, pp. 400–407.

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

[21] 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.

[22] 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," 2011, pp. 109–120.

[23] 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 *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 97–112.

[24] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 503–516.

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

[26] 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.

[27] 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.

[28] M. Goli, J. Stoppe, and R. Drechsler, "Automatic protocol compliance checking of SystemC TLM-2.0 simulation behavior using timed automata," in *International Conference on Computer Design (ICCD)*, 2017, pp. 377–384.

[29] ——, "Automatic equivalence checking for SystemC-TLM 2.0 models against their formal specifications," in *Design, Automation and Test in Europe (DATE)*, 2017.

[30] M. Goli and R. Drechsler, "Scalable simulation-based verification of SystemC-based virtual prototypes," in *EUROMICRO Symposium on Digital System Design (DSD)*, 2019, pp. 522–529.

[31] ——, "ATLaS: Automatic detection of timing-based information leakage flows for SystemC HLS designs," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021, pp. 67–72.

[32] M. Goli, M. Hassan, D. Große, and R. Drechsler, "Automated analysis of virtual prototypes at electronic system level," in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 2019, pp. 307–310.

[33] J. Aynsley, "TLM-2.0 base protocol checker," https://www.doulos.com/knowhow/systemc/tlm2, accessed: 2018-01-30.

[34] B. C. Schäfer and A. Mahapatra, "S2CBench: synthesizable SystemC benchmark suite for high-level synthesis," *IEEE Embed. Syst. Lett.*, vol. 6, no. 3, pp. 53–56, 2014.