Early SoC Security Validation by VP-based Static Information Flow Analysis

Muhammad Hassan[1,2] Vladimir Herdt[2] Hoang M. Le[2] Daniel Große[1,2] Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
muhammad.hassan@dfki.de {vherdt,hle,grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—Security is one of the most burning issues in embedded system design nowadays. The majority of strategies to secure embedded systems are being implemented in software. However, a potential hardware backdoor that allows unprivileged software access to confidential data will render even the perfectly secure software useless. As the underlying SoC cannot be patched after deployment, it is very critical to detect and correct SoC hardware security issues in the design phase. To prevent costly fixes in later stages, security validation should start as early as possible. In this paper, we propose a novel approach to SoC security validation at the system level using *Virtual Prototypes* (VP). At the heart of the approach is a scalable static information flow analysis that can detect potential security breaches such as data leakage and untrusted access; confidentiality and integrity issues, respectively. We demonstrate the applicability of the approach on real-world VPs.**

I. Introduction

The increasing functionality and connectivity of embedded devices such as in the Internet-of-Things have raised their requirements on security significantly. This is due to the increasing amount of sensitive information and personal data being stored in those devices as well as the security-critical functions they perform. Secure embedded systems cannot be achieved by focusing on just the software or the hardware part, but rather require holistic approaches. As the number of vulnerabilities in software/firmware historically dominates, security strategies and validation approaches for embedded software have received considerably more attention and also benefited from traditional software security research. Formally verified operating systems (e.g. seL4) and compilers (e.g. CompCert) or mature information flow tracking approaches are just to name a few. However, hardware security is at least equally important, since a potential hardware backdoor that allows unprivileged software access to confidential data will render all software/OS-level protection mechanisms useless. Consider for example a recent discovered exploitable bug [1] in the Actel ProASIC3 FPGA, which is claimed to be one of the most secure devices in the industry and being used in military and other critical application domains. A backdoor via JTAG allows an attacker to get hold of cryptography keys as well as other data from the FPGA.

Furthermore, fixing hardware security issues after deployment is always associated with very high cost. The reason is that the underlying SoC cannot be patched after production. Therefore, it is very critical for the embedded industry to move rapidly from considering security as an afterthought to integrating it in the SoC design process. Still, current practice and research on SoC security validation focuses mostly on the *Register-Transfer Level* (RTL) and below. Many approaches for Trojan/backdoor identification and IC counterfeit detection exist. Verification of hardware security architectures (e.g. ARM TrustZone) and security modules starts only with the availability of RTL designs. While these steps are indispensable, ample opportunities to save time and cost by prioritizing security earlier at the system level have not been yet fully exploited. One of the promising directions is to leverage *Virtual Prototypes* (VPs) that are now an established industry practice for early software development and early software/hardware validation. VP models, typically written in SystemC [2], [3] using *Transaction Level Modeling* (TLM) techniques, are abstract, executable models of underlying SoC hardware components and available much earlier than RTL. Thus, VP-based security validation would enable to detect and correct many SoC hardware security issues very early.

Information Flow Tracking (IFT) has become one of the key techniques in security research. The basic idea is to control how (labeled) information is propagated by the system under consideration. This tracking allows to enforce policies for secure information flow such as confidentiality and integrity. Both dynamic and static approaches have been proposed (see e.g. [4] for a survey on IFT approaches for software security). For hardware designs, researchers have also started to apply IFT in the last decade [5], [6], [7]. More recently, approaches for RTL have been emerging [8], [9], [10]. At the system level using VPs, IFT has to the best of our knowledge not yet been considered.

In this paper, we present a novel *VP-based IFT* approach, which is to the best of our knowledge *the first of its kind*. Essentially, our approach operates directly on the SystemC VP models by combining several passes of static analysis. The main difficulties to be overcome here are to deal with the challenging language C++, which SystemC is based on, as well as the specific semantics of TLM on-chip communication via *TLM-2.0 payload*. To this end, we build on the flexible compiler infrastructure provided by LLVM/Clang to perform in interleaved manner connectivity analysis, access control extraction, call-graph analysis, data flow analysis and static taint tracking to identify static paths that violate specified

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project CONVERS under contract no. 16ES0656, the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1, and the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

secure information flow properties. These potential vulnerable paths are reported back to user for further inspection. Our static analysis is *sound*, i.e. it never misses a violating path if such exists.

The remainder of the paper is structured as follows. The paper starts with introducing the literature review in Section II. Then, the overview of our proposed approach with a motivating example is presented in Section III. The proposed methodology for static information flow analysis is introduced and explained in Section IV. Experiments on a real-world VP to show the efficacy of our approach are presented in Section V, followed by limitations in Section VI. Finally, the paper is concluded in Section VII.

II. Related Work

The information flow properties used in this paper are inspired by [11]. This work verifies these properties on firmware using symbolic execution. Such an analysis can conceptually also be applied to SystemC VPs, however, applying symbolic execution to SystemC VP models is very challenging and a satisfactory solution still has to be researched. Our static analysis is a more lightweight solution, which expectedly also scales better on real-world VPs.

With respect to hardware IFT, it is worth mentioning the *language-level* idea pursued by e.g. Sapper [8] or SecVerilog [9]. They extend the type systems of existing languages at the RTL abstraction to include security labels. Then, information flow properties can be verified statically by performing type checking. Impressive results based on this idea have been achieved recently in [12]. The approach described there is capable to verify a simplified implementation of TrustZone written in SecVerilog. While such language-level approach is very amenable for static IFT, it puts a burden on users to add appropriate security labels as well as to understand their semantics in combination with the type system, basically requiring users to learn a new language.

A more practical approach is to perform IFT (with some trade-offs) on hardware designs written in existing languages widely used in the industry. GLIFT [6], [7] applies IFT on gate-level descriptions and recently, RTLIFT [10] on RTL descriptions. These approaches are suited for individual IP cores and circuits. Our approach, on the other hand, operates on the higher level of abstraction of SystemC TLM-2.0 and targets information flow at system level between IP components of a SoC.

As mentioned earlier, our approach is to the best of our knowledge the first attempt to leverage VP models for system-level IFT. We believe this new promising line of research will prove to be fruitful for making security validation truly a cross-cutting and cross-level activity in the embedded design flow.

III. Approach Overview

In this section we give a high-level overview of our approach starting with the threat model we consider. Then, we discuss a motivating example and show the overall workflow of the approach.

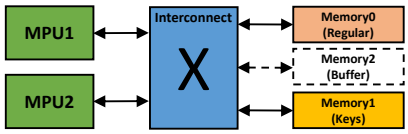

Fig. 1. Motivating Example

A. Threat Model

Considering a SoC, we want to protect assets like for instance: cryptographic keys, digital certificates, signatures, classified text, authentication data from id sensors or register settings. Transporting the sensitive data is performed through buses between the different IP components of the SoC. Hence, for our proposed VP-based IFT approach the analysis of the information flow using TLM communication between the IP blocks is of particular interest. In this context, 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), availability (an IP may use some shared resource to the extent that other IPs cannot use that shared resource) and authenticity (an IP is actually the rightful initiator of the transaction) are the general security concerns. We focus only on confidentiality and integrity.

B. Motivating Example

We present here a SystemC TLM-2.0 example (Fig. 2) that will be used to showcase the main ideas of our approach throughout this paper. For brevity, we refrain from giving a proper introduction to SystemC. The SystemC TLM-2.0 constructs and semantics necessary to understand the example will be explained as needed. The example presents a simplified SoC consisting of a regular MPU (Microprocessor) (MPU1), a trusted MPU (MPU2), a regular memory (Memory0) and a confidential memory with security keys (Memory1) as shown in Fig. 1. The buffer (Memory2) is initially not available. The modules are connected to an interconnect which routes transactions where the MPUs act as initiator and the memories as target. The communication uses a 32-bit address mode as follows: 1) bits 0 to 7 - local address inside a memory; 2) bits 8 to 15 - memory address; 3) bits 16 to 23 - MPU ID; 4) bits 24 to 31 - unused. Their behavior is implemented in thread functions (MPU1: `thread_proc()` Line 3 - MPU2: `thread_proc()` Line 16), and *b_transport* functions (Interconnect: `b_transport()` Line 26 - Memory: `b_transport()` Line 46).

The MPUs execute instructions that initiate TLM-2.0 transactions (i.e. read or write) to the memory. In this illustrative example, the actual instruction behavior is abstracted away as we only focus on the communication. The Interconnect receives the transaction and checks the address generated by MPUs (Line 27, to Line 35), accordingly routes the TLM 2.0 transaction to the corresponding memory. The memory receives the transaction, checks the *cmd* from transaction, and writes to (Line 56) or reads from (Line 54) the memory.

A more formal representation of the information flow policies of the SoC will be given later in Section IV-A1. The

```
1  struct MPU1: sc_module {
2   //...
3   void thread_proc() {
4     tlm::tlm_generic_payload* trans = new
          tlm::tlm_generic_payload;
5     //...
6     trans->set_data_ptr(reinterpret_cast<unsigned
          char*>(&data));
7     //...
8     socket->b_transport(*trans, delay);
9     //...
10  }
11  int data;
12 };
13
14 struct MPU2 : sc_module {
15  //...
16  void thread_proc(){
17    //...
18  }
19  //...
20 };
21
22 struct Interconnect : sc_module {
23  SC_HAS_PROCESS(Interconnect);
24  Interconnect(sc_module_name name) { ... }
25  //...
26  void b_transport(int id, tlm::tlm_generic_payload&
       trans, sc_time& delay) {
27    masked_addrs = trans.get_address();
28    mpu_nr = (masked_addrs >>16) & 0xFF;
29    address =  masked_addrs & 0xFF;
30    mem_nr = (masked_addrs >> 8) & 0x3;
31    if ( (mpu_nr == 1) && (mem_nr == 0) )
32      (*initiator_socket[0])->b_transport(trans,delay);
33    else if ( (mpu_nr == 2) &&  (mem_nr == 1) )
34     (*initiator_socket[1])->b_transport(trans,delay);
35    else
36     trans.set_response_status( tlm::TLM_OK_RESPONSE
          );
37  }
38  sc_dt::uint64 addrs;
39  sc_dt::uint64 masked_address;
40  unsigned int mem_nr;
41  int mpu_nr;
42 };
43
44 struct Memory : sc_module {
45  //...
46  virtual void b_transport (tlm::tlm_generic_payload&
       trans, sc_time& delay) {
47    tlm::tlm_command cmd = trans.get_command();
48    sc_dt::uint64 adr = trans.get_address();
49    unsigned char* ptr = trans.get_data_ptr();
50    unsigned int   len = trans.get_data_length();
51    unsigned char* byt = trans.get_byte_enable_ptr();
52    unsigned int   wid = trans.get_streaming_width();
53    //...
54    if ( cmd == tlm::TLM_READ_COMMAND )
55     memcpy(ptr, &mem[adr], len);
56    else if ( cmd == tlm::TLM_WRITE_COMMAND )
57     memcpy(&mem[adr], ptr, len);
58    //...
59    trans.set_response_status( tlm::TLM_OK_RESPONSE );
60  }
61  int mem[MEMORY_SIZE];
62 };
```

Fig. 2. SystemC TLM 2.0 example with two MPUs and two memories

intuition here is that MPU1 should not be able to access Memory1 which holds the security keys. The access control policies that are implemented in the Interconnect to enforce this are as follows:

1) if ((mpu_nr == 1) && (mem_nr == 0))
2) if ((mpu_nr == 2) && (mem_nr == 1))

Such access control policies are commonly implemented in components with routing functions. Intuitively, the access control policies, when properly implemented, would be enough to achieve the isolation of the untrusted MPU1. This will be confirmed later by our analysis as detailed in Section IV-C. Now consider a new scenario that a designer decides to add an additional buffer (Memory2), shared by both MPUs, for buffering data for overall performance enhancement. However, with the shared buffer indirect information flow between Memory1 and MPU1 exists. A software adversary can use MPU2 to read the confidential keys from Memory1 and write them into the buffer. Then MPU1 can read the keys from the buffer. Such indirect information flow (via another IP) is not trivial to detect, especially without an automated analysis like our proposed VP-based IFT approach.

C. Overall Workflow

The overall workflow of our data flow driven information flow approach for SystemC TLM 2.0 is shown in Fig. 3. The approach is based on a static analysis, hence it needs to be only run once to validate the information flow security properties. Essentially, our approach uses data flow analysis to perform static taint analysis, and combines this information with system binding information and call-graphs to validate the security properties defined for a SoC.

First the *security properties* are read, followed by identification of source and sink (destination) of each property to be verified in different paths. From the elaboration phase of the system, connectivity of VPs is identified (*binding information*). This helps in identifying how data flows through the system (see Section IV-A2).

It is followed by *access control information* extraction (see Section IV-A3). It identifies the set of all access control policies implemented inside a system. In the next step construction of call-graphs is performed to be used in analysis at the end.

The *data flow analysis* identifies the set of all data flow values computed at different points in a system. Due to static nature, the analysis computes an over-approximation of the data flow values. This data flow information is extended to perform taint analysis between source and sink of each property. All the tainted variables from the source are identified wrt. SystemC.

In the next step, all the information from aforementioned steps is evaluated and combined to obtain the final access paths and extended paths between source and sink for each property. Both the paths are then filtered using predicates to ensure the validation of property. Essentially, the result shows which security properties have been satisfied and which are not satisfied at the end. A property is satisfied iff one of the three following conditions is satisfied : 1) There exists no access paths or extended access paths between a source and sink. 2) There exists no access paths or extended access paths between any tainted variable from source and sink. 3) The predicates of the property fail completely.

Our approach identifies the failing paths for each unsatisfied security property, and allows the verification engineer to focus his efforts to improve the design by improving either access control policies or information flow policies.

In the following we detail the ingredients of our approach as well as demonstrate them using the motivating example.

IV. Data Flow Driven Information Flow Analysis

A. Information Extraction

1) Information Flow Property Specification: Information flow can be used to model various security properties i.e.,

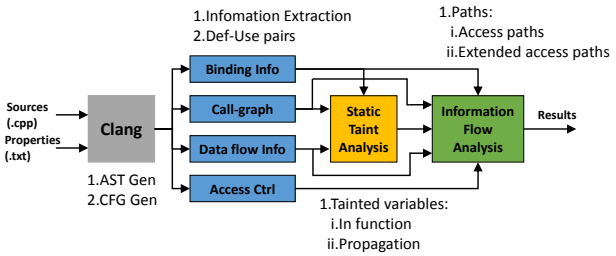

Fig. 3. Information flow analysis overview

confidentiality, integrity, availability, isolation, and hardware trojans etc. Our approach focuses only on two widely used security properties; confidentiality and integrity based on the principle of non-interference. An information flow property defines a data flow relation among two entities *source*, and *sink*, where the data is allowed or disallowed to flow under some certain conditions. These properties need to be defined in a way to capture the complete flow without missing out on any information. For SystemC TLM 2.0, the classical property specification does not work as the classical specification always considers an input port and an output port of the system as untrusted/trusted information sources. Whereas in SystemC TLM 2.0, the VPs interact through transactions, hence, the property specifications and definitions need to adapt. Therefore, we define an information flow property as a tuple *(source, sourcep, op, sink, sinkp)* which depending on the property being defined (confidentiality/integrity), comprises of the following elements:

1) *source* - The generation point of a transaction / variable inside a VP.
2) *sourcep* - a predicate associated with *source* which specifies under what condition is the data valid at *source*
3) *op* - operator like *no flow*.
4) *sink* - The point in program where the *source* assigns some value.
5) *sinkp* - a predicate associated with *sink* which specifies under what condition is the data valid at *sink*

Information is said to flow between *source* and *sink* when there is a static path between the two entities, and *sourcep* predicate and *sinkp* predicate are satisfied. Whereas, information is said not to flow when there is no static path even when *sourcep* and *sinkp* are satisfied. This could be due to several underlying implementation issues.

For our analysis, we only focus on *op no flow*, which inhibits the flow of information between *source* and *sink* to make the VP secure. Also, other *op* like *flow* can be defined as a dual of *no flow* for every case.

2) VP Binding Information: In a SoC, VPs are connected to each other in a certain way which affects how information is propagated. Our analysis defines a connection of two VPs when their corresponding sockets (*initiator_socket* and *target_socket*) are connected using SystemC TLM 2.0 *b_transport* function call. The way these functions are registered, and bound during elaboration phase is vital because each VP in TLM 2.0 setting contains a *b_transport* function. If binding information is not available, the connectivity of modules cannot be identified statically (before execution). Also, without this information, a *b_transport* function call from a wrong VP can be analyzed because of similar function name, resulting in further over-approximation. This binding information helps in constructing call-graph (see Section IV-A4).

3) Access Control Extraction: Our analysis extracts all the access control conditions from each function of a VP. An access control condition is defined as a condition in *if-else* control flow structure, or in *while* and *for* loops. Our analysis extracts this information without expanding any functions and stores it. This is useful for the analysis in that it tells which VPs are allowed to access which VPs in a complete SoC. Access control conditions can be based on VP id, VP address, or the socket id etc. Generally, this information is embedded in Interconnects which act as transaction routers in SystemC TLM 2.0.

4) Call-Graph Construction: Our static analysis performs call-graph construction once in the start. Call-sites are not expanded at this stage i.e. before data flow analysis. Also, there are a lot of SystemC specific function calls which are added to the call-graph but they are never expanded. For the commonly occurring system function calls, their behavior is already defined inside the analysis, for e.g., *memcpy()* etc. This call-graph is used to guide the analysis to propagate the information to the correct function. Hence, it uses binding information from Section IV-A2 to correctly identify the function call from the correct VP.

B. Static Analysis

1) Data Flow Analysis: A data-flow analysis algorithm takes as input the SoC under test to compute test objectives for each VP (i.e., def-use pairs). A *reaching uses* procedure - an instance of data flow analysis techniques is used to identify test objectives (i.e., def-use pairs) for a VP, which actually answers such a question: for each variable defined, which uses can potentially use the values?

Our data flow analysis is inspired from [13] but we do not use the associations as defined. Rather, we only define definition-use (*def-use*) pairs according to their classification to help in our analysis. Like the *def-use* pairs across threads. Our static analysis defines a def-use pair as an ordered triple (x, d, u) such that d is a statement where variable x is defined and u is a statement where x is used. Furthermore, there is a static path from d to u in the program without re-definition of x in-between. Please note, there is a static path from every context switch statement from one thread to the start of a transition of every other thread. A transition starts at the beginning of a thread and right after a context switch. Based on this general observation, we define a *du-path* as a static path between d and u without re-definition of x.

Similarly, we classify event-based synchronization of SystemC by means of the wait/notify function. This can also be considered as a data flow relation . The wait can be considered a definition which suspends the active thread, while the notify is considered a use.

The analysis identifies all possible *def-use* pairs of a VP by performing intra-function analysis, and inter-thread analysis. Inter-function analysis is deliberately not performed as it will be compensated for during the taint analysis.

2) Static Taint Analysis: Static taint analysis identifies how a single VP or a variable inside a VP affects or taints other VPs inside a system. The core idea behind the taint analysis wrt. SystemC TLM 2.0 is that any VP inside the SoC when generates a transaction, all the entities and paths it takes to reach the destination entity are also compromised. Because if the generated transaction is malicious, the malicious value can propagate throughout the system and at the end leak some data. Not only does the transaction leaks the data, but all the variables in between also become potential security risks. Static taint analysis may be viewed as a conservative approximation of the full verification of non-interference or the more general concept of secure information flow. Because information flow in a system cannot be verified by examining a single execution trace of that system, the results of taint analysis will necessarily reflect approximate information regarding the information flow characteristics of the SoC to which it is applied. Our taint analysis uses the data flow information (*def-use*) pairs from Section IV-B1, and combines it with security properties information from Section IV-A1.

The *source* of each property acts as a taint source, it can be a transaction generated in a VP, or it can be an internal variable or register. The *sink* in each property acts as the final destination where taint source propagates. We define a tainted variable as a variable that is affected by taint source directly or indirectly, for e.g., direct assignment from *source* or an assignment from a variable which was tainted by *source*. Hence, for a taint to propagate to *sink*, it is not necessary that the taint source itself has to propagate, rather any of its tainted variables can also propagate. Our static analysis builds a graph of all tainted variables belonging to one taint source. The transfer function for assignments taints the left-hand side if any of the operands on the right-hand side is tainted. Assignments to arrays and memories are treated conservatively by tainting the entire array/ memory. Because at static time we do not know the exact address location which will be accessed.

3) Information Flow Analysis: Our information flow analysis interleaves call-graph information, access control information, VP binding information, security properties and data flow analysis to detect vulnerable paths. We define two kinds of paths, 1) Access paths - a path between *source* and *sink*. 2) Extended access paths - combination of access paths.

Each access path is defined as a combination of nodes where each node represents an entity. A node can be from one of the four categories: 1) Taint source node - *tsrc*. 2) Function call node - *fcall*. 3) Access control node - *actrl*. 4) Taint sink node - *tsink*.

A taint source node always defines a starting point of an access path, and taint sink node always ends the access path. If there is no taint sink found (i.e., the information does not reach it) the access path is not created at all. A function call node is created when taint propagates to the next function, and an access control node is created when the path encounters access control condition. Each node carries with itself some useful information which helps in the analysis. For example, the node contains the taint source name, the calling function's name, the called function's name, their corresponding classes, and ids etc (see Fig. 4).

The analysis is done in three stages: 1) Forward analysis, 2) Backward analysis, 3) Predication. The forward analysis is carried out first, starting from first function where the taint source is defined. By definition of a taint source, we mean when the *source* is created (formal parameters of a function definition do not qualify for that). A taint source node is added as the starting point of the path. Then data flow analysis information i.e., def-use pairs information is used to deduce if taint sink lies in the same function. If it does, a taint sink node is added to the path, hence, an access path created between these two points. If the sink is not in the same function, the call-graph information is combined with data flow information to find the taint propagation. In case of a function call, a function call node is added to the path, and the actual parameters are mapped on to the formal parameters of the function followed by retrieving the data flow information of the callee. If an access control exists, a corresponding node is inserted in the path. Once the complete access paths for all the security properties are created, the backward analysis starts. During the creation of an access path, all the irrelevant information is abstracted away, i.e., if a function is being called without taint source / variables, or if there exists an access control not affecting taint source / variable. Because such information does not help in any way in information flow analysis.

The purpose of backward analysis is to detect if there is any other access path leading to the same taint sink. Our analysis uses forward analysis information and overlaps the access paths to check which access path ends at the sink. The starting point of the path can be any point, except the taint source. Because in that case a loop will be created. When the path is found, it is added to the path found by forward analysis. Then it is checked if this new path propagates this information anywhere else. If it propagates, the paths are added to the initial path and at the end, a complete extended access path is created. One thing to keep in mind, extended paths will always be equal to or greater than access paths, but never less. Because, essentially extended paths are a combination of access paths, and if there is only one combination possible for each path, then access paths equal extended paths.

In the last stage, predication is performed by applying the *sourcep* predicates and *destp* predicates for each property on to the extended paths found. A property is satisfied if the predicates are true and there is no static path between *source* and *sink*, or if the predicates are false.

C. Illustration

We use the same SystemC TLM 2.0 example as shown in Fig. 2 to illustrate our methodology. The information flow properties are deduced from the policies defined in Section III-B, i.e.:

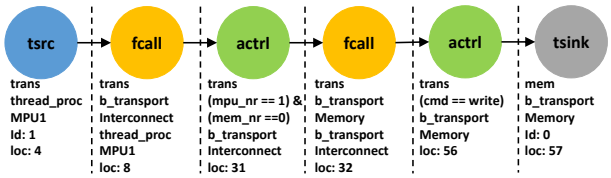

Fig. 4. One access path for example shown in Fig. 2

1) source: trans, sourcep: mpu_nr == 1, op: no flow, sink: mem, sinkp: mem_nr = 1
2) source: trans, sourcep: mpu_nr == 2, op: no flow, sink: mem, sinkp: mem_nr = 0

When the information flow analysis is executed, the first property is picked and all the paths with *trans* as *source* and *mem* as *sink* are searched. The availability of paths signify that there is a static path between a *source* and a *sink*, it can be within one function or spread over multiple functions, and it can be passing through various access control mechanisms. The analysis identifies four paths originating from MPU1 and four paths originating from MPU2. One of the paths originating from MPU1 and terminating at Memory0 is shown in Fig. 4.

The starting node, i.e., *tsrc* contains the information that it represents variable *trans*, and its originating function is *thread_proc*, which belongs to class MPU1 with registered id 1 (id extracted from elaboration phase). Similarly, other nodes also contain useful information.

In the next step, the analysis picks first path and checks if the *sink* of first path overlaps with the *sink* of any other path, and finds that MPU2 access *mem*. So, this new path is appended to the actual path. In the third step, the analysis checks if the newly added path access *sink mem* at any other location in the system, and finds that MPU2 access *mem* in Memory1. It recursively checks for any other similar paths and does not find anything. The analysis finds all the extended paths in similar fashion and stores them. The same is done for second property. We get the following extended paths for first property:

1) MPU1 $\longrightarrow$ Memory0 $\longrightarrow$ MPU2 $\longrightarrow$ Memory1
2) MPU1 $\longrightarrow$ Memory1 $\longrightarrow$ MPU2 $\longrightarrow$ Memory0
3) MPU2 $\longrightarrow$ Memory0 $\longrightarrow$ MPU1 $\longrightarrow$ Memory1
4) MPU2 $\longrightarrow$ Memory1 $\longrightarrow$ MPU1 $\longrightarrow$ Memory0

Due to space limitation, each extended path is shown by the VP it originates from and where it propagates. The long arrows signifies flow of information between two VPs. This information flow could be passing through access control policies or it could be a direct path. For example, for the first extended path *trans* generates in MPU1, and it propagates to *mem* in Memory0. Then this *mem* in Memory0 can be accessed by MPU2's *trans*. Finally, *trans* in MPU2 can propagate to *mem* in Memory1. Essentially, each VP's name shown in extended path is an access path as shown in Fig. 4. If more VPs are connected in between *source* and *sink*, this extended path would also get long.

At the end predication is performed by applying source predicates *sourcep* and sink predicates *sinkp*. This process

```
1  ....
2  if ( (mpu_nr == 1) && (mem_nr == 0) )
3    (*initiator_socket[0])->b_transport(trans,delay);
4  else if ( (mpu_nr == 2) && (mem_nr == 1) )
5    (*initiator_socket[1])->b_transport(trans,delay);
6  else if ( mem_nr == 2 )
7    (*initiator_socket[2])->b_transport(trans,delay);
8  else
9    trans.set_response_status( tlm::TLM_OK_RESPONSE );
10 ....
```

Fig. 5. SystemC TLM 2.0 example with two MPUs and three memories

narrows down the number of paths as the predicates are applied. For example, initially first extended path related to first property is analyzed. The *sourcep* predicate states that the *trans* is from MPU1, hence using the information gathered in aforementioned sections, it is deduced that indeed this predicate is true. Because *sourcep* has been satisfied, the next predicate *sinkp* is checked, which states that *mem* should be in Memory1. The analysis does not find *mem* in MPU1. It proceeds further to Memory0 while passing through the access control policies. It checks if MPU1 is allowed to access Memory0 (Line 31 - Fig. 2), and finds that it is allowed. Then, the analysis deduces that Memory0 is not allowed to be accessed by MPU2 (Line 33 - Fig. 2), and returns false. When the access control condition is false, that means the information will not be allowed to flow further, and hence we stop searching in this path further. The second path is analyzed and again it is found that MPU1 is not allowed to access MEMORY1. Hence, there exists no path that allows MPU1 to access Memory1. In the similar fashion, property 2 is also satisfied. Hence, the the system is shown to be secure.

Now consider a new scenario that a designer decides to add an additional buffer (Memory2), shared by both MPUs, for buffering data for overall performance enhancement. The *b_transport()* function of Interconnect (Line 26 - Fig. 2) is updated with the following information as shown in Fig. 5. The rest of the function implementation remains the same.

When our analysis is executed once again, it detects that both the security properties fail now. Upon inspection of the extended paths, it is found that now MPU1 can access Memory1 using the shared *buffer - Memory2*. Similarly, MPU2 can access Memory0. This indirect information flow (via another IP) is not trivial to detect. To solve the problem, for example, another buffer should be added and the sharing of resources is prohibited using the *b_transport* code shown in Fig. 6 where each memory is given exclusive access. A more general solution is to add a memory management unit.

```
1  ....
2  if ( (mpu_nr == 1) && (mem_nr == 0) )
3    (*initiator_socket[0])->b_transport(trans,delay);
4  else if ( (mpu_nr == 2) &&  (mem_nr == 1) )
5    (*initiator_socket[1])->b_transport(trans,delay);
6  else if ( (mpu_nr == 1) && (mem_nr == 2) )
7    (*initiator_socket[2])->b_transport(trans,delay);
8  else if ( (mpu_nr == 2) && (mem_nr == 3) )
9    (*initiator_socket[3])->b_transport(trans,delay);
10 else
11   trans.set_response_status( tlm::TLM_OK_RESPONSE );
12 ....
```

Fig. 6. SystemC TLM 2.0 example with four memories

The information flow properties are also updated with two new properties:

1) source: trans, sourcep: mpu_nr == 1, op: no flow, sink: mem, sinkp: mem_nr = 3
2) source: trans, sourcep: mpu_nr == 2, op: no flow, sink: mem, sinkp: mem_nr = 2

After executing the analysis again, all the properties are found to be satisfied again. This example demonstrates the capability of our approach to detect *omission of access control policies*.

D. Implementation Details

In this section we describe the implementation details of our methodology briefly. The static information flow analysis is implemented using the LibTooling library for Clang compiler [14]. Clang generates an Abstract Syntax Tree (AST) of the SystemC TLM 2.0 source code. The AST is parsed to extract the required information to perform static analysis. The implementation parses the AST multiple times, and in each iteration extracts a different set of information. We next discuss important implementation details.

In the first iteration, the VP binding information is extracted by looking for the AST nodes related to elaboration phase. The binding of VPs is possible in different ways depending on the implementation of the SystemC design, i.e., using *bind* keyword or using AMBA binders.

In the next iterations, access control information and call-graph information is extracted using the aforementioned data extraction methods. The locations of *use* are also extracted to help with the analysis. Specially the bounds of access control information which define a context of program statements. All the extracted information is stored in data structures for performing data flow analysis, taint analysis, and information flow analysis.

V. Experimental Results

In this section we present a case study to demonstrate our information flow analysis approach for SystemC. We consider the LEON3-based VP SoCRocket [15] which has been modeled in SystemC TLM 2.0. The complete VP consists of more than 50,000 lines of code. The VP combines several IPs working together in master or slave mode. The VP is designed to be *bus-centric*, i.e. the IP cores are connected through an on-chip bus. The AMBA-2.0 AHB/APB (Advanced High-performance Bus / Advanced Peripheral Bus) bus is used as the common on-chip bus. In the VP, the LEON3 processor is directly connected with AHB as *AHBMaster* device, and AHB/APB Bridge is connected as a *AHBSlave* controller which controls the communication between AMBA AHB and AMBA APB devices. Various TLM IPs are connected to the AMBA APB bus like UART, GPTimer, and IRQMP. A memory controller is connected as *AHBSlave* with AHB bus which serves several memories. Essentially, AMBA-2.0 AHB/APB buses are complex interconnects which take in the transactions generated by the connected IPs, and forwards them to the intended IP based on the address and/or sockets. In the following we report results for three experiments integrating different IPs into SoCRocket.

A. CRYPTO AES IP

In the first case study, we consider the case of integrating two TLM IPs with SoCRocket: 1) A TLM IP *CRYPTO-AES*, a cryptographic hardware accelerator implementing AES-128 algorithm designed specifically to compute *cipher text* for the given *plain text* efficiently; 2) A secure memory IP *SEC-MEMORY* pre-loaded with cipher keys.

The *CRYPTO-AES* engine works in *Cipher Block Chaining (CBC)* mode, i.e. an *initialization vector*, and a *plaint text* are given as input to the IP, where *plain text* and *initialization vector* are XORed before being written on input of IP, *key* is read from *SEC-MEMORY*, and the IP generates a *cipher text*. This *cipher text* is fed back to the cryptographic engine as the new *initialization vector* for the next iteration. The IP *CRYPTO-AES* computes its round keys on the fly, instead of computing them beforehand and storing them in *SEC-MEMORY*. The LEON3 processor initializes *CRYPTO-AES* by configuring its configuration registers.

Because of the nature of *SEC-MEMORY*, i.e. it stores cryptographic keys, our information flow policy is that the LEON3 processor should not be allowed to read (confidentiality) or write (integrity) these keys from *SEC-MEMORY*. Hence, the following security property covering both confidentiality and integrity is derived:

1) source: trans, sourcep: IP_ADDR == 0x1, op: no flow, sink: memory_buffer, sinkp: MEMORY_ADDR == 0x4

IP_ADDR = 0x1 refers to the address of LEON3 and *MEMORY_ADDR = 0x4* to the address of *SEC-MEMORY*. With our approach we observed that the security property failed. Our methodology was able to report information flow between LEON3 and *SEC-MEMORY* through the *debug* interface. Normally, the *debug* interface is constrained to output limited information or dummy information in case of cryptographic algorithms, but it was not the case. We fixed the failing path by restricting the *debug* interface access to *CRYPTO-AES* only. The property was satisfied on next analysis run. Sometimes, these intentional (supposedly non-malicious) flaws can occur in hardware designs, specially when the design team includes undocumented features for assistance in testing. These flaws can be exploited to get access to trusted data. The analysis reported a computation of 37 access paths, and 79 extended access paths. In these paths, only two extended paths (one for read access, and one for write access to debug interface) allowed the information flow while the others did not. The analysis took 51.63 seconds to report the results. 0% false positives were reported because of concrete path conditions.

B. NFC IP

For the second case study, we integrate a Near-Field Communication (NFC) interface IP with SoCRocket. Essentially, it is a communication protocol that enables two devices to communicate when brought in close proximity. NFC is now widely used in smartphones as a mode of contactless payment system, for sharing files and photos between two

devices, and as e-ids (electronic ids) etc. Because of high-speed communication, the data is stored in memory using DMA (Direct Memory Access). Most current SoCs involve DMA to the memory through a dedicated DMA controller to reduce the workload on the processor cores.

Due to the nature of NFC IP and overall system security, the security policy states that system-specific addresses (could be pointing to configuration registers of SoC) in memory should not be accessed by the NFC IP (read or write). Therefore, we derive the following security property:

1) source: trans, sourcep: IP_ADDR == 0x5, op: no flow, sink: main_memory, sinkp: MEM_SPACE_ADDR < 0x000F0000

IP_ADDR = 0x5 refers to NFC IP in SoCRocket, whereas *MEM_SPACE_ADDR < 0x000F0000* classifies the system-specific addresses the NFC IP shall not access.

Our approach detects an access path between NFC and *main_memory* via DMA, thus the security property is violated. The reason is that the requested DMA address (from the NFC IP) is not checked against disallowed address ranges (missing bounds). The reported number of access paths, and extended access paths was 37 and 80, respectively. The analysis took 55.01 seconds to invalidate the property. 0% false positives were reported because of concrete path conditions. It clearly shows that IP development team and SoC integration team should collaborate actively to avoid such security vulnerabilities.

C. Smart Card Reader IP

For the third case study we integrate a smart card reader into the system. It reads the data from card and stores it in a secure section of on-chip memory (access control implemented by address range specification). Three security policies are specified: 1) The on-chip processor (LEON3) should not be allowed to read (confidentiality) / write (integrity) this secure portion of memory. 2) Smart card reader should not read (confidentiality) or write (integrity) SRAM. 3) Smart card reader should not read (confidentiality) or write (integrity) ROM (Read Only Memory - contains LEON3 configuration etc.).

After running our analysis, we observed that the first and second security properties are satisfied because of the strict access control policy implemented in the Memory Controller (*MCtrl*) in SoCRocket. But the third property fails. The *MCtrl* disallows regular ROM accesses by any IP other than LEON3, but this constraint is not present on DMA. This violation of the security property, reported by our analysis, could potentially be exploited by a hardware trojan to get DMA access for ROM. The reported number of access paths, and extended access paths was 37 and 83, respectively. The analysis took 74.35 seconds to validate all three properties. Due to the use of concrete path conditions, 0% false positives were reported.

VI. Limitations

Although our methodology is an overall sound analysis, it does share the inherent limitations that come with most other static-analysis tools due to over-approximation i.e., false positives. For example, during predication phase in analysis, the path predicates cannot be compared accurately if the access control policy is dependent on a dynamic variable, instead of a constant. In case of a dynamic variable, it always suggests that a path exists and shows it to the testing engineer. This over-approximation is still better than false negative. The analysis heavily relies on binding information, hence, if the binding information is given in an obscure way, it might be missed.

VII. Conclusion

In this paper we presented the first VP-based IFT approach for security validation. At the heart of the approach is a scalable static information flow analysis that operates directly on the SystemC VP models. The analysis performs in interleaved manner connectivity analysis, access control extraction, call-graph analysis, data flow analysis and static taint tracking to identify static paths that violate specified secure information flow properties. These potential vulnerable paths are reported back to user for further inspection. We have demonstrated the applicability of the approach on real-world VP SoCRocket.

References

[1] S. Skorobogatov and C. Woods, "Breakthrough silicon scanning discovers backdoor in military chip," in *CHES*, 2012, pp. 23–40.

[2] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.

[3] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[4] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," in *Software Safety and Security - Tools for Analysis and Verification*, 2012, pp. 319–347.

[5] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *ASPLOS*, 2004, pp. 85–96.

[6] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *ASPLOS*, 2009, pp. 109–120.

[7] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "Theoretical fundamentals of gate level information flow tracking," *TCAD*, vol. 30, no. 8, pp. 1128–1140, Aug 2011.

[8] 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," *SIGPLAN Not.*, vol. 49, no. 4, pp. 97–112, Feb. 2014.

[9] 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.

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

[11] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, "Verifying information flow properties of firmware using symbolic execution," in *DATE*, 2016, pp. 337–342.

[12] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a practical hardware security architecture through static information flow analysis," in *ASPLOS*, 2017, pp. 555–568.

[13] M. Hassan, V. Herdt, H. M. Le, M. Chen, D. Große, and R. Drechsler, "Data flow testing for virtual prototypes," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 380–385.

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

[15] 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.