Validation of SoCs Security Architecture: Challenges, Threats, and Methods

Mehran Goli

University of Bremen, Germany

DFKI Bremen, Germany

mehran@uni-Bremen.de

April 24-28, ICONS 2022

About Presenter

More Info

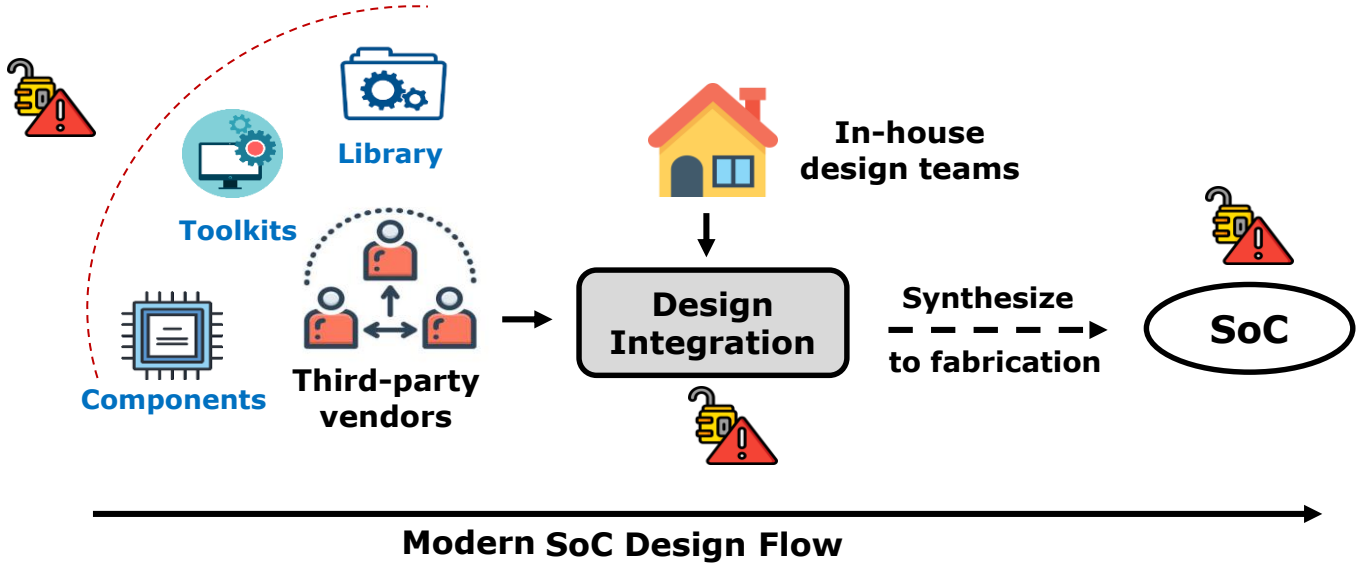
- **Mehran Goli** works as a senior researcher at the University of Bremen (Universität Bremen) and the German Research Center for Artificial Intelligence (DFKI) in Bremen, Germany.
- He holds a Ph.D. degree (Dr.-Ing.) in Computer Science from the University of Bremen, Germany (2019).
- **Dr. Goli** was a recipient of the **Best Paper Award** at the FDL conference in 2021, a nominee for the **GI Excellent Computer Science Dissertations award**, and a recipient of a Ph.D. scholarship award from the German Academic Exchange Service (DAAD).
- He is a member of program committees of DSD, FDL, ICONS, and CYBER conferences, a reviewer of IEEE TVLSI, ACM TECS journals, and an external reviewer of DATE, DAC, ICCAD, ETS, DSD, and RSP conferences.
- **Research interests:** system-level design, verification, security validation, and machine learning techniques for CAD.

Outline

- **Introduction**
  - SoC design flow and security issues
  - What are timing-based security attacks?
  - Why should the detection process be performed at ESL?
- **Security Threat Models**
  - Non-interference
  - Implicit and explicit flows
  - Timing-based data leakage threat
- **Vulnerability Detection Challenges**
- **Information Flows Detection Methodology**
  - Functional and timing flows
- **Conclusion**

Introduction

- **Modern SoCs ➡ notoriously insecure**
  - modern design flow **shifted from in-house** development of IPs **to the use** of existing **commercial IPs**
  - SoCs including several 3PIPs
  - 3PIP integrations can **manipulate or assist in manipulating** secret data

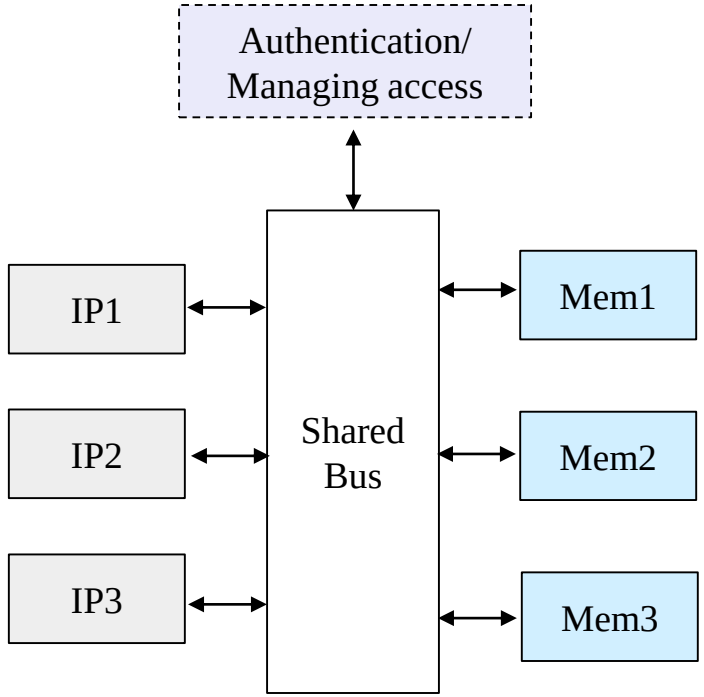


Introduction

- **Modern SoCs ➡ increasing deployed** in
  - highly personalized activities
  - critical aspects of our lives
- **SoC implemented as**
  - composition of **IPs** and **interconnects**
  - data including **secure assets** transferred

  via shared interconnects across different IPs
- To provide **sound security guarantees**
  - SoC has a **security architecture**

Specify the **conditions** under which a **secret asset can be accessed** at any point in the system execution.



Introduction

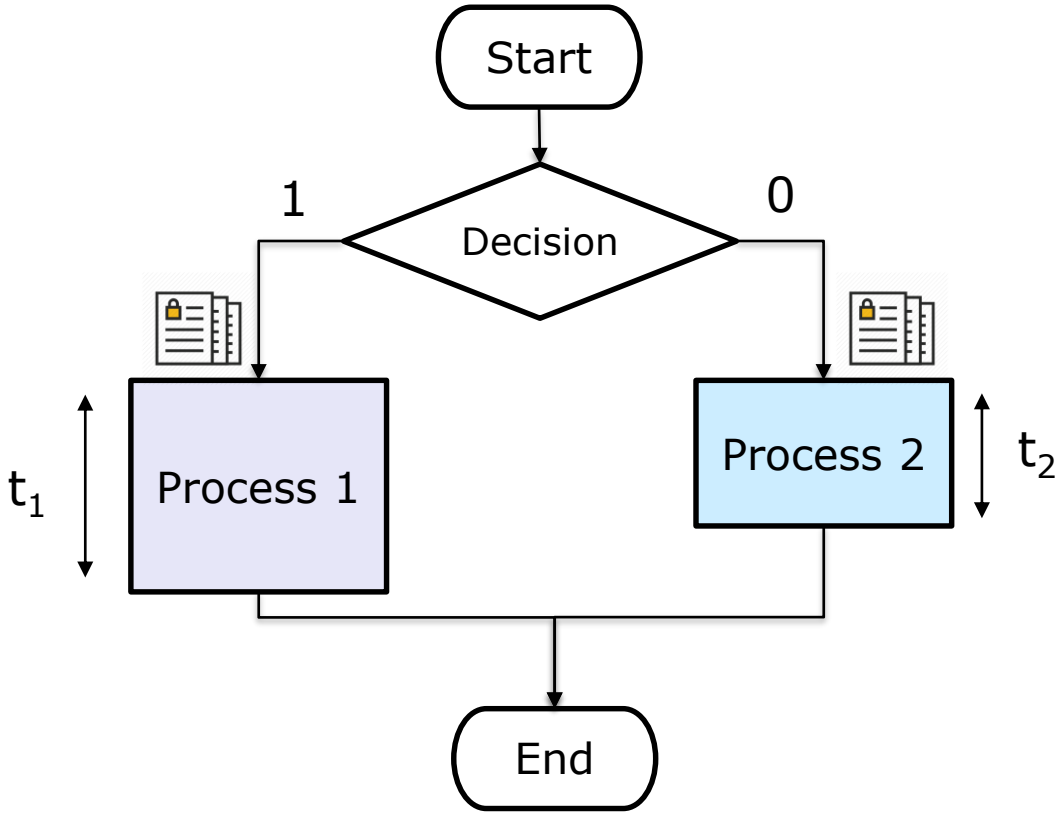
- **SoC security validation** ➡ **utmost importance**
- **Non-interference**
  - **Idea:** certain parts of the system (secure zones) should never interfere with other parts (insecure zones)
- **Guaranteeing non-interference** ➡ **non-trivial** and **crucial** task
  - depending on the SoC security architecture
  - information can flow through difficult-to-detect side channels
- **Timing-based attacks** ➡ interesting for attackers
  - as they **only need to measure the execution time** of the victim process **without physical access** to the design
- **Access secret data** at a **very low cost** and **effort**

Introduction - Functional Flow

- **IP isolation technique** ➡ w.r.t **non-interference**
  - certain parts of the system (secure zones) should never interfere with other parts (insecure zones)
  - for **SoC security validation**, a common property needs to be checked is non-interference
- **Information Flow Tracking (IFT)** ➡ **promising solution**
  - powerful technique to help mitigate security vulnerabilities that violate certain information flow policies
  - **Idea:** monitoring how information propagates through a system to see if secret information is leaking

Introduction - Timing Flow

- What are **timing-based security attacks**?

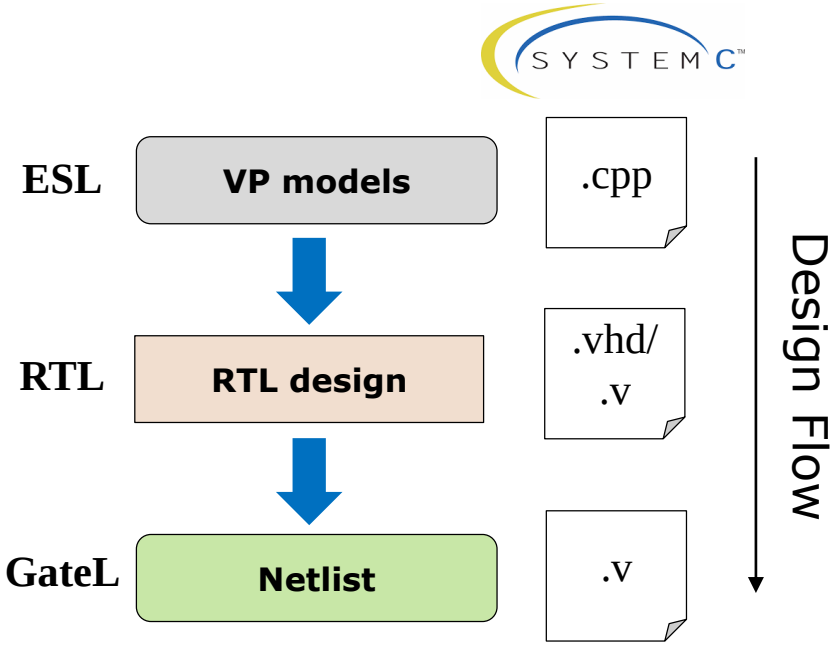


The time taken by a (computational) modules to generate the results may be different **($t_1 \neq t_2$)** regarding the data being processed.

Attackers who are familiar with the underlying algorithms use statistical approaches to extract the key by **measuring the execution time**.

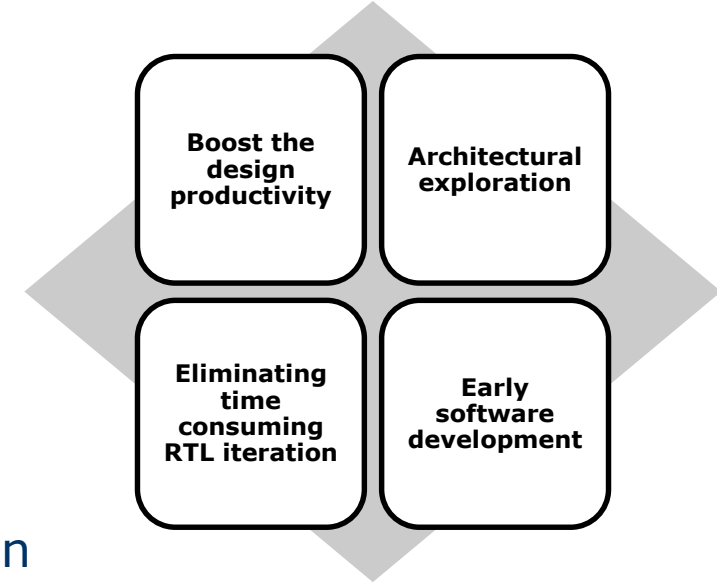
Introduction – Why at ESL?

- **Detection process ➡ as early as possible**
  - cost of fixing any security flaws **increases** with the stage of development
- For the early design entry
  - **Virtual Prototype (VP)** increasingly adopted by the semiconductor industry
  - abstract and executable software model
  - typically implemented using **SystemC TLM-2.0**
- **VPs** in comparison to **RTL designs**
  - **significantly faster** simulation speed
  - much **earlier available**

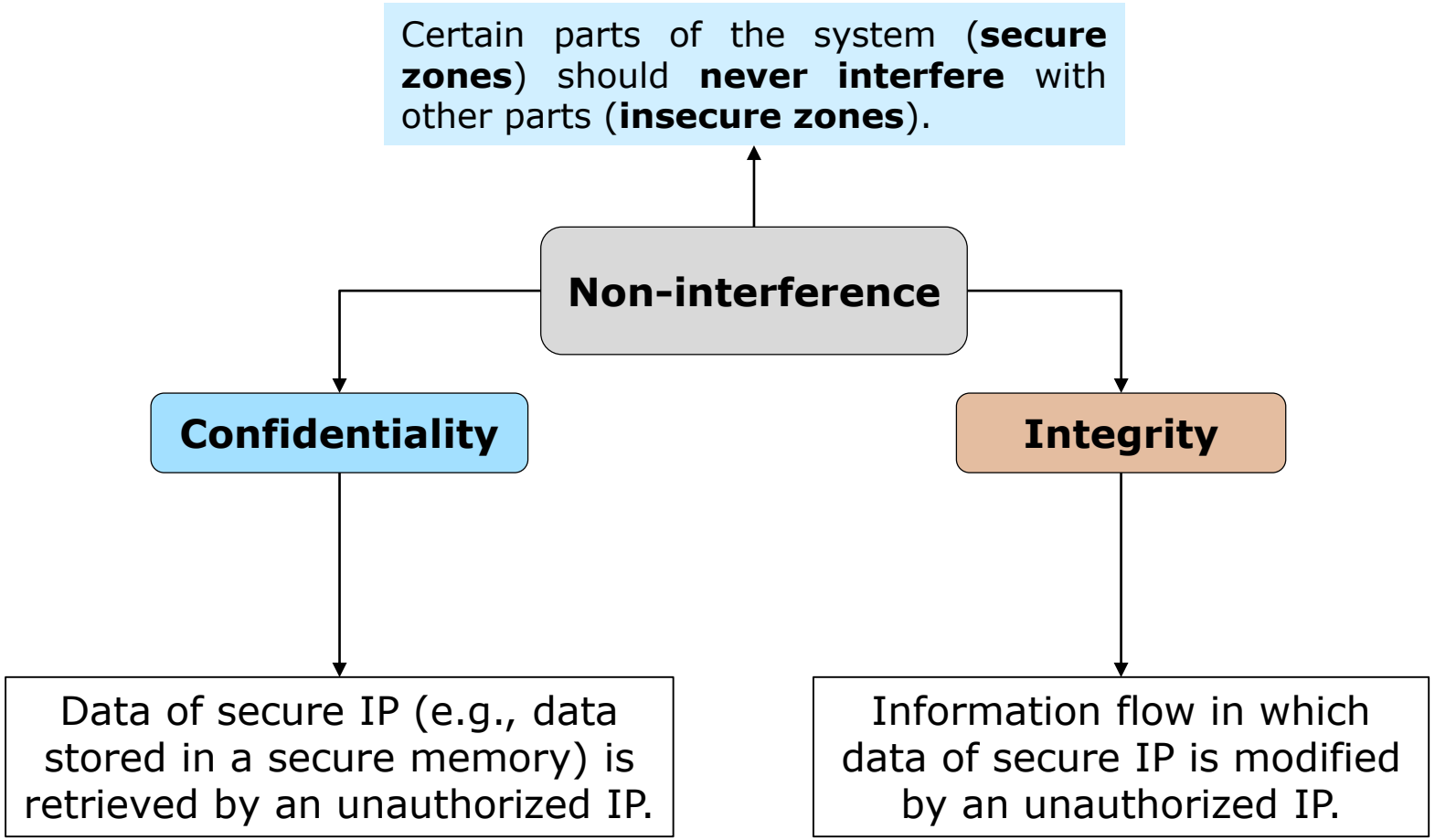


Introduction - Why at ESL?

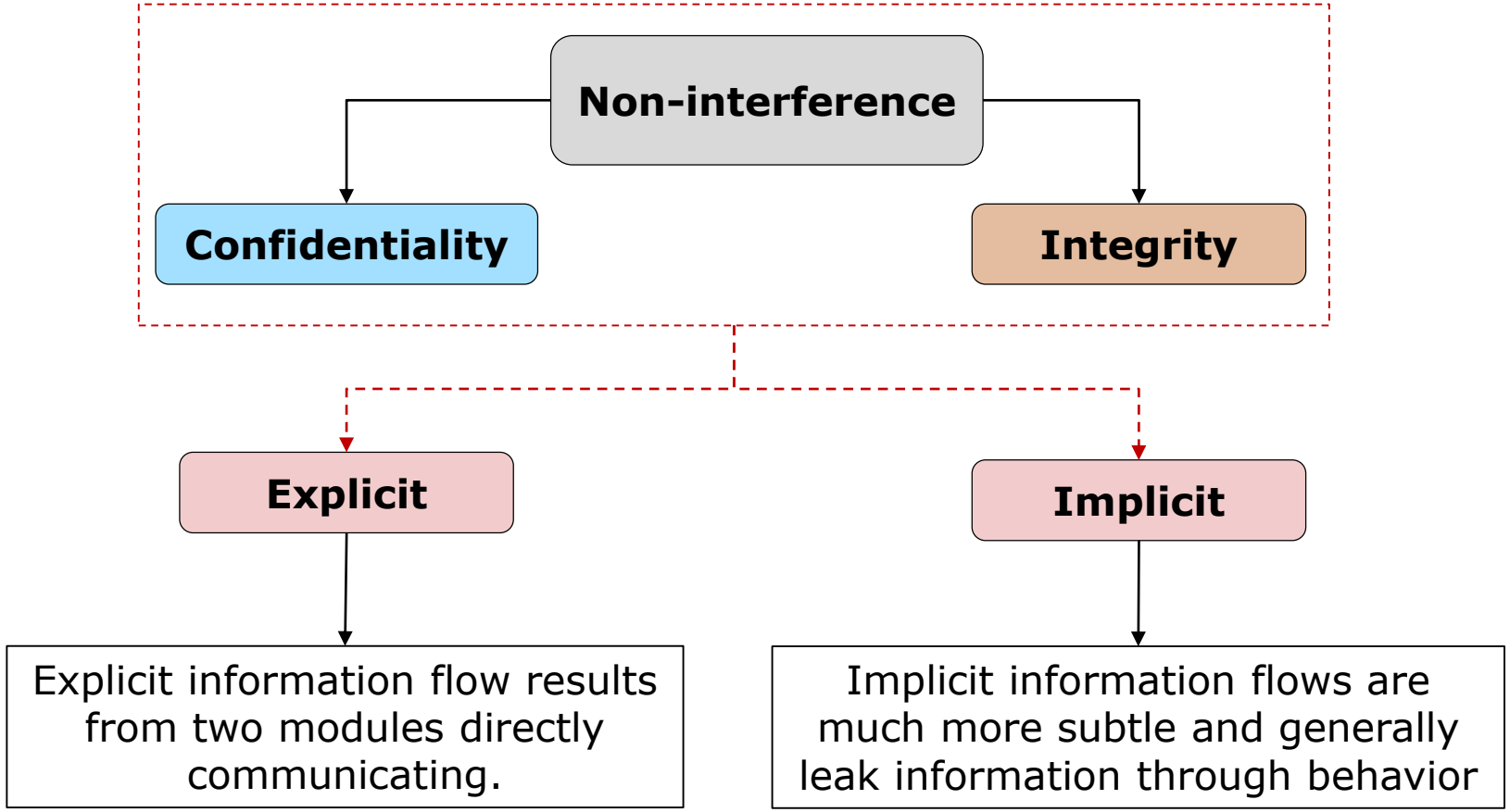
- This leads designers to use **VPs** for
  - architectural exploration
  - performance analysis
  - early software development
  - overall, **reference models** for lower levels of abstraction
- **VP-based security validation** ➡ **promising direction**
  - to fix security vulnerabilities in SoCs before they are refined
  - to **avoid costly design loops** occur



Security Threat Models

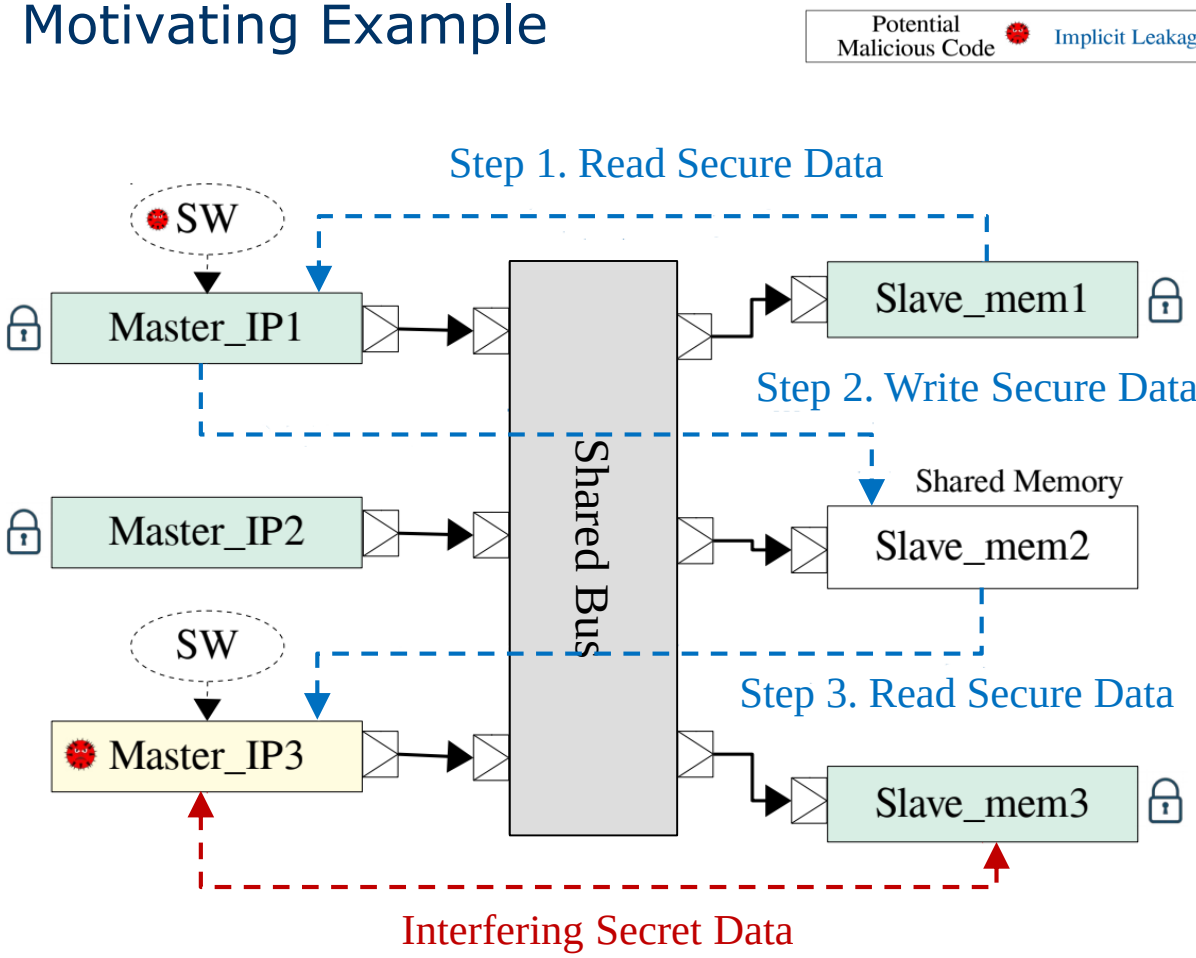


Security Threat Models



Security Threat Models - Functional Flow

- Motivating Example



In the case of **explicit information flow**: module *Master_IP3* access memory *Slave_mem3* through the shared interconnect *Shared-Bus*.
In the case of **implicit information flow**, an implicit 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).

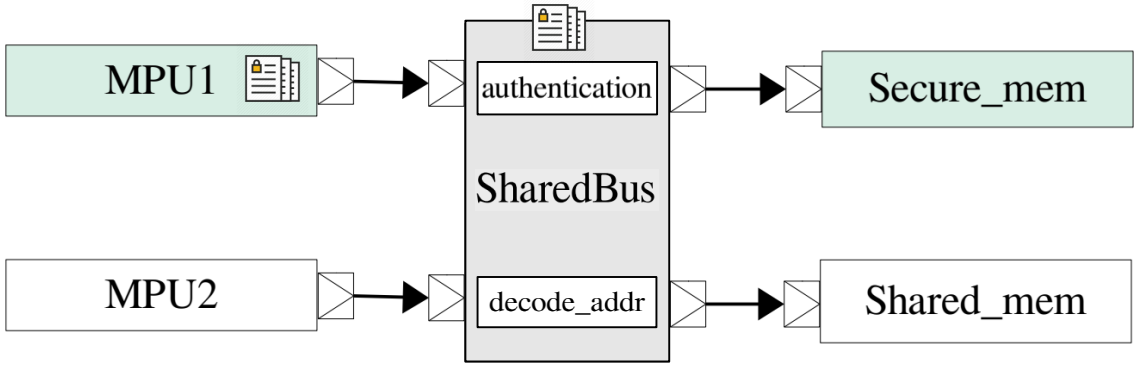
Security Threat Models - Functional Flow

- **Security Scenarios**
  - **Third-party IP** may contain **malicious** part to exploit the confidential data
  - **Malicious software** running on the (trustworthy) hardware IP may exploit hardware backdoors to cause malfunctions or leak secret data.
  - An **incorrect initialization** (either by an adversary involved in the SoC design process or unintentionally) of the SoC **firmware** (memory configuration file).
  - The existing **SoC is extended** or modified but its **information flow policies** are **not updated**.

Security Threat Models - Timing Flow

- Motivating Example

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



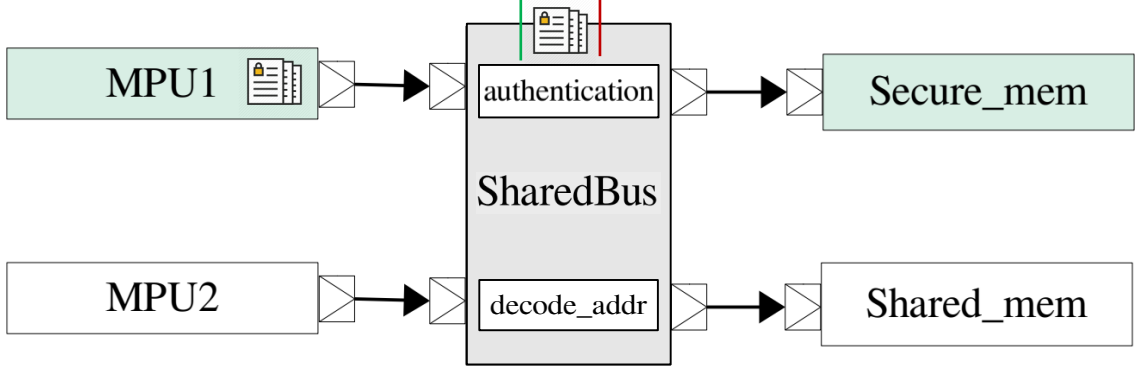

```
int decode_addr(sc_dt::uint64 addr, sc_dt::uint64&
     masked_addr) {
unsigned int id = static_cast<unsigned int>((addr >> 8)
      & 0x3);
masked_addr = addr & 0xFF;
return id;}
```

Security Threat Models - Timing Flow

- Motivating Example

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

```
bool authentication (char* trans_key) {
 size_t i = 0;
while (i < strlen(Skey)){
 if (trans_key[i] != Skey[i])
  return false;
  i++;}
return true;}
```



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

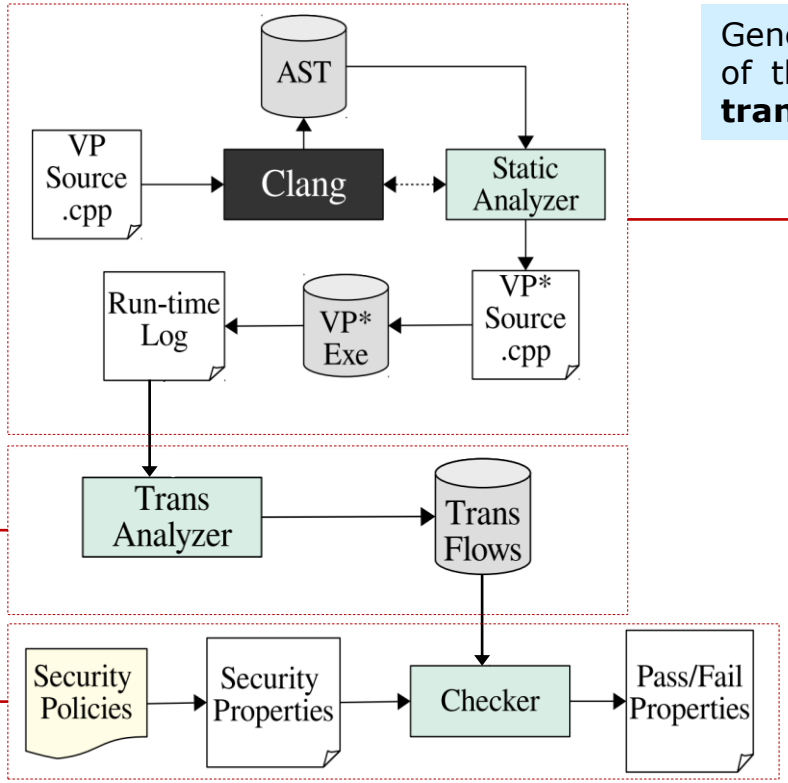
```
int decode_addr(sc_dt::uint64 addr, sc_dt::uint64&
     masked_addr) {
unsigned int id = static_cast<unsigned int>((addr >> 8)
      & 0x3);
masked_addr = addr & 0xFF;
return id;}
```

Vulnerability Detection Challenges

- **Manual analysis** of the source code
  - a very time-consuming and error-prone task
- **Testing the design** to capture timing variations
  - becoming impractical due to the scale of modern SoCs
- **Existing methods** are only applicable at RTL and below
  - do not support SystemC constructs, data types and semantics
- **IFT-based method at ESL**
  - only able to analyze the functional information flows
- **The existing verification methods at ESL**
  - are not able to detect security threat models
  - as the design functionality and protocol rules are not affected

Functional Flows Detection Methodology

1. **Run-time Behavior Extraction**
2. **Transaction Transformation**
3. **Security Validation**



Generating an instrumented version of the VP source code for **tracing transactions** at run-time.

Transforming the extracted **transactions** into a set of transaction flows.

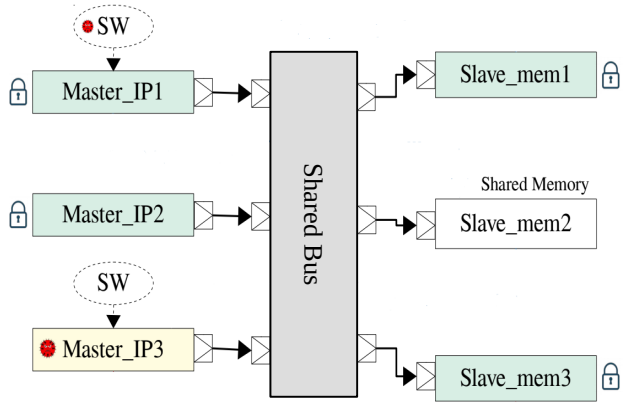
- **Translating** the **information flow policies** of design into a set of **security properties**.
- Generated **transaction flows** are **validated** against the generated **security properties**.

M. Goli and R. Drechsler, "VIP-VP: Early Validation of SoCs Information Flow Policies using SystemC-based Virtual Prototypes," **FDL**, pp. 1–8, **2021**.

Functional Flows Detection Methodology

- Motivating example
  - **Run-time Behavior Extraction**

The **Recorder statements** are defined based on a hierarchical structure where for **tracing transactions**,
- **reference address** of transactions,
- **root** and **instance name** of the **module**,
- **related parameters** such as phase,
- **run-time value** of their attributes.



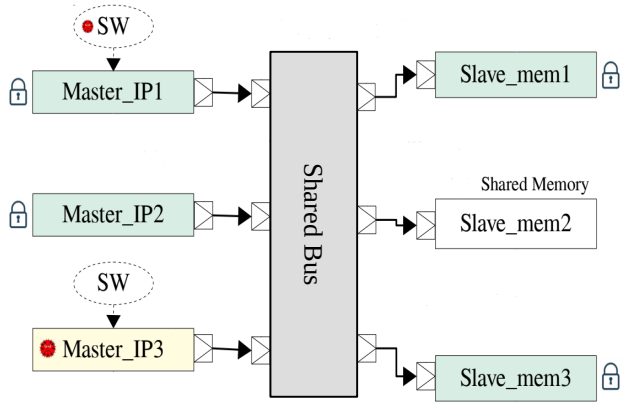

```
1 struct Master_IP1: sc_module {
2  tlm_utils::simple_initiator_socket<Master_IP1> socket;
3  void thread1(){
4  /*...*/
5   socket->b_transport (*trans, delay);
6   Fout << "Master_IP1::thread1::trans_ID = " << trans <<
        "data =" << trans->get_data_ptr << "cmd =" <<
        trans->get_command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

Assume we want to trace the flow of transactions generated by the **Master_IP1** module of the SoC. A part of the **Master_IP1** module is shown in this slide where the transaction object is used as an input argument for the **b_transport** interface call.

Functional Flows Detection Methodology

- Motivating example
  - **Transaction Transformation**

A complete **simulation behavior** of the VP can be defined as a set of **transaction flows.**



Transaction is created by an initiator IP

Transaction is received by a target IP

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

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)}

Combination of transaction flows *TF2*, *TF4*, and *TF6* shows an implicit flow of data between **Master_IP3** and **Slave_mem1.**

Implicit flow

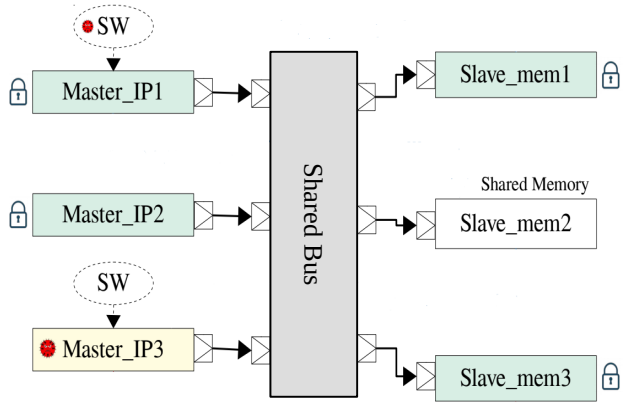
This is a part of transaction flows where *TF1* to *TF6* specify six explicit transaction flows. For example, *TF1* shows that a transaction is generated by ***Master_IP3*** to access data in memory ***Slave_mem3***. Here we can also see the details of the flow.

Functional Flows Detection Methodology

- Motivating example
  - **Security Validation**

List of secure IPs
- initiator (source)
- target (sink)

$$SIP_{source} = \{IP_1, IP_2, ..., IP_n\}$$
$$SIP_{sink} = \{IP_1, IP_2, ..., IP_m\}$$



List of forbidden information flows between **source** and **sink**

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

Explicit flows

Functional Flows Detection Methodology

- Motivating example
  - **Security Validation**

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**
      $TF_{t+1} \leftarrow (S_{ip} \rightarrow IS_{mem} :: write)$
      $TF_{t+2} \leftarrow (source \rightarrow IS_{mem} :: read)$
      $P_i \leftarrow \{TF_t, TF_{t+1}, TF_{t+2}\}$
      $ISP \leftarrow P_i$
      $i \leftarrow i + 1$

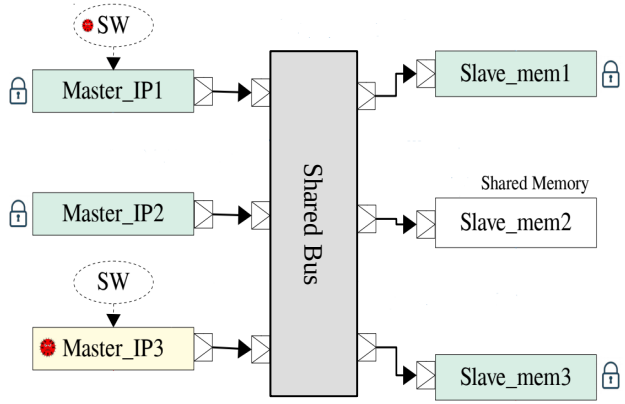
Implicit flows

Three transaction flows required to shape an **implicit channel**

Secure IP **reads secret** data from the **secure memory** (sink).

Secure IP **writes** the **secret data** in an **unauthorized memory**.

Unauthorized IP (source) **read** the **secret data** from the unauthorized memory.



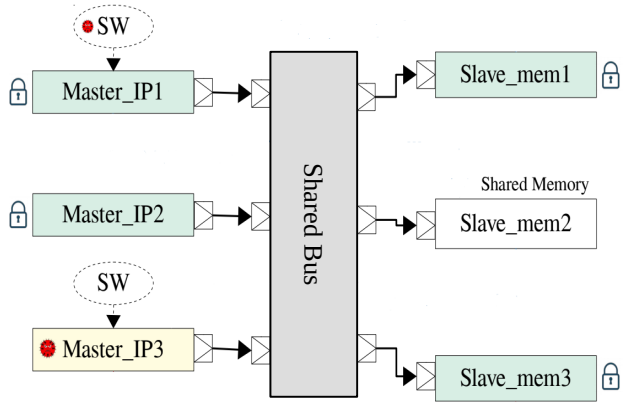
The algorithm take as inputs the set of secure initiator IPs, secure target IPs, and *forbid_flow*. Then, for each forbidden flow *f*, it generates Three transaction flows required to shape an implicit channel. The ***TFt*** property specifies a transaction flow where a secure IP reads secret data from the secure memory (in sink) specified in *f*. The ***TFt+1*** shows a transaction flow where the secure IP writes the secret data in an unauthorized memory. The ***TFt+2*** describes the transaction flow where the unauthorized IP (source) specified in *f* read the secret data from the unauthorized memory.

Functional Flows Detection Methodology

- Motivating example
  - **Security Validation**

From specification

$$\begin{cases} SIP_{source} = \{Master\_IP1, Master\_IP2\} \\ SIP_{sink} = \{Slave\_mem1, Slave\_mem3\} \\ Forbid_{flow} = \{Master\_IP3 \rightarrow Slave\_mem1 :: no\ flow\} \end{cases}$$

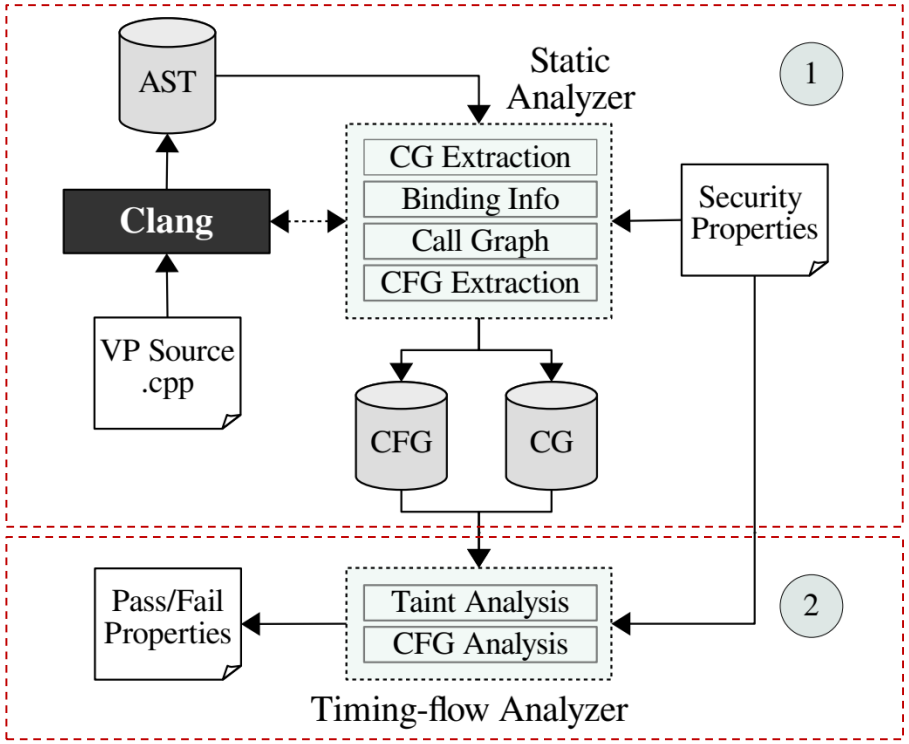


$$\begin{aligned} ISP = \{p_1 = \{&Master\_IP1 \rightarrow Slave\_mem1 : read\}, \\ &\{Master\_IP1 \rightarrow Slave\_mem2 : write\}, \\ &\{Master\_IP3 \rightarrow Slave\_mem2 : read\}, \\ p_2 = \{&Master\_IP2 \rightarrow Slave\_mem1 : read\}, \\ &\{Master\_IP2 \rightarrow Slave\_mem2 : write\}, \\ &\{Master\_IP3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

The **p1** and **p2** properties 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**.

Timing Flows Detection Methodology

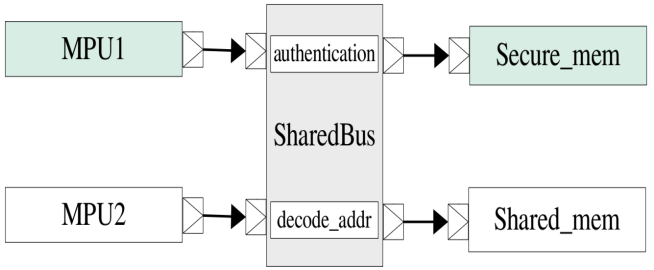
1. **Static Data Extraction**
   - formally represening VP's behavior
   - CG and CFG
2. **Timing Flow Analysis**
   - static taint tracing and path analysis



M. Goli and R. Drechsler, "Early Validation of SoCs Security Architecture Against Timing Flows Using SystemC-based VPs," **ICCAD**, pp. 1–8, **2021**.
M. Goli and R. Drechsler, "ATLaS: Automatic Detection of Timing-based Information Leakage Flows for SystemC HLS Designs," **ASP-DAC**, pp. 67–72, **2021**.

Timing Flows Detection Methodology

- Motivating example
  - **Security Properties Definition**



Part of design with **High Security** (HS) tag

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

Under which **condition** is the data valid at **source** and **sink**.

Part of design in which the **time** taken for it to reach its final value must be **Constant** (CT) as the source changes.

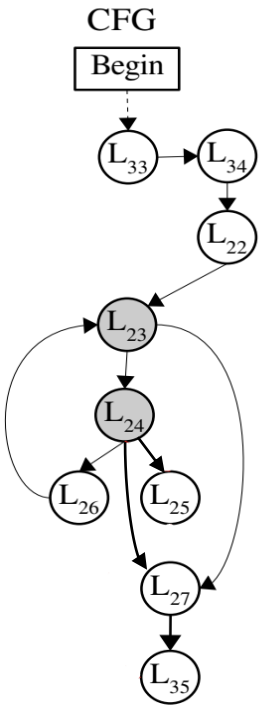
$$SP = \{P_1 : (source, \emptyset, sink, c_{snk}) \mid source \leftarrow MPU1 :: thread\_process() : sec\_key, \; sink \leftarrow SharedBus :: b\_transport() : permission \; c_{snk} \leftarrow SharedBus :: b\_transport() : mem\_id = 0\}$$

In the security property **P1**, variable **sec_key** is an attribute of the transaction which is defined in its extension filed to hold the authentication data for all generated transactions by the **MPU1** module in the **thread_process().** The **permission** variable belongs to the access control policy of the **SharedBus** in its **b_transport** function and holds the authentication result. The property ensures that the **permission** variable must obtain the result in constant time as **sec_key** changes.

Timing Flows Detection Methodology

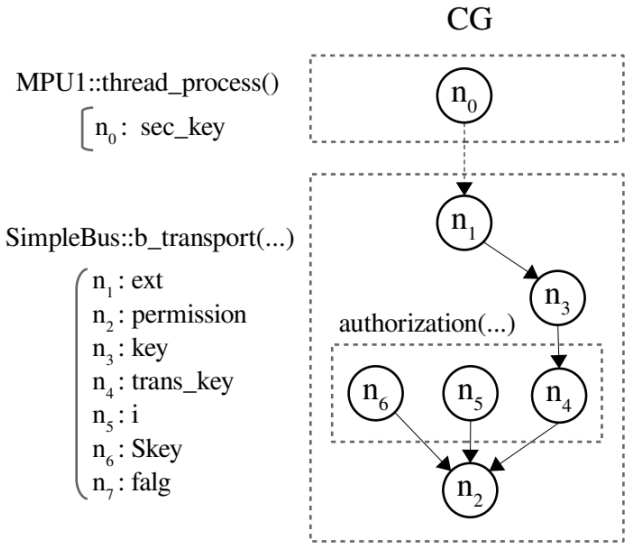
- Motivating example
  - **Static Data Extraction**

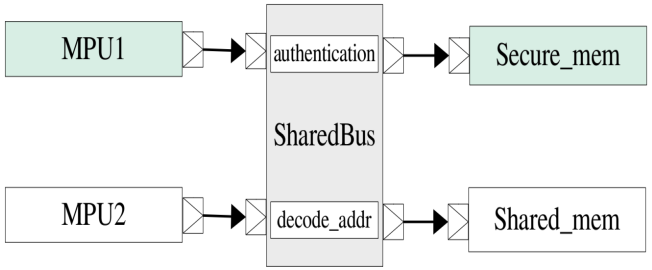
Data Flow Graph



In order **to know whether conditional updates caused by sensitive data**, we need to extract the control flow of a given VP. A part of the generated CFG of the motivating example w.r.t security property. The gray nodes show the control flow statements (meaning condition node type like, *if-else*). The white nodes indicate the computational statements.

Data Dependency Graph





A part of the generated CG of the motivating example w.r.t security property shown in the previous slide. Each node of the CG is a transaction's attribute, its related parameter or a variable of the VP which is tokenized by the name of module and function (for local variable) to which the transaction or variable belongs. The dot-box in the CG shows the function calls graph, started from initiator module MPU1 by calling **thread_process()** and goes through the **b_transport()** function of the **SharedBus**. Thus, this graph identifies how the source (node **n0**) is connected to sink (node **n2**) through the intermediate variables.

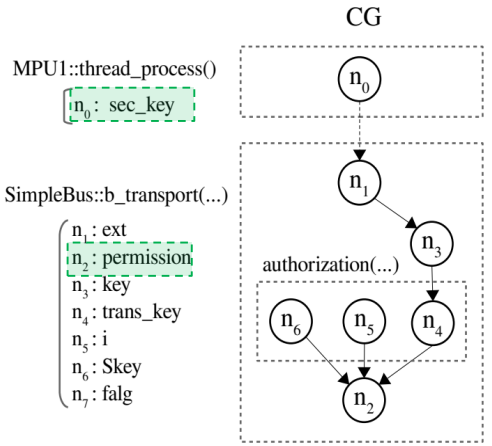
Timing Flows Detection Methodology

- Motivating example
  - **Timing-based Data Flow Analysis**

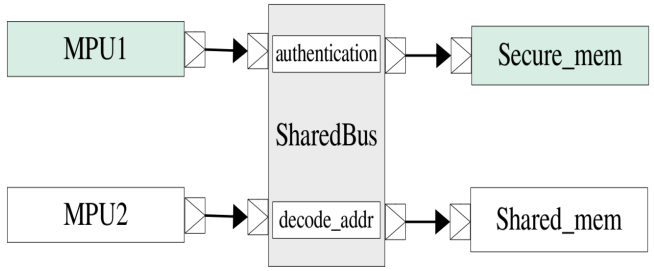
Input: P_i, CG, CFG
Output: Timing Flow TF

$L_{st} \leftarrow$ ForwardTraverse $(source, CG)$
for each node $n \in CFG$ **do**
  if $n.type() ==$ condition **then**
    $n_{ctrl} \leftarrow$ Extract list of variables from n
    if $n_{ctrl} \cap L_{st} \neq \emptyset$ **then**
      $L_{path} \leftarrow$ DFS $(n, CFG, sink)$
      for each path $p \in L_{path}$ **do**
        for each node $n_p \in p$ **do**
          if $n_p.type() == condition$ **then**
            remove(p, L_{path})
      if $L_{path} \neq \emptyset$ **then**
        for each node $n_p \in p$ **do**
          if $sink \in n_p$ **then**
            $TF \leftarrow (n, n_p)$

$\{n_1, n_3, n_4\}$



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



List of source taints of the VP after tracing its CG is **L_st =** {n1 , n3 , n4 }.

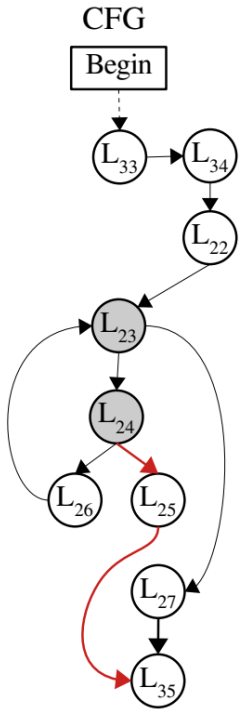
After generating a formal representation of a given VP-based SoC behavior, we perform a timing flow analysis to detect all conditional updates caused by the sensitive data. For each property **Pi**, a taint analysis is performed on the corresponding generated CG and CFG. The taint analysis identifies how the sensitive data (source) affects or taints other transactions and variables inside a system. First, the taint analysis is performed by a forward tracing on the **CG** from the source node to the sink node. All nodes in this trace that are related to the source get the **HS** tag and are added into the list of source taints **L_st**. In the next step, the CFG of the VP is analyzed to find all control statements including sensitive variables, transaction's attributes or its related parameters (stored in **L_st**) that control the occurrence of updates on **sink.**

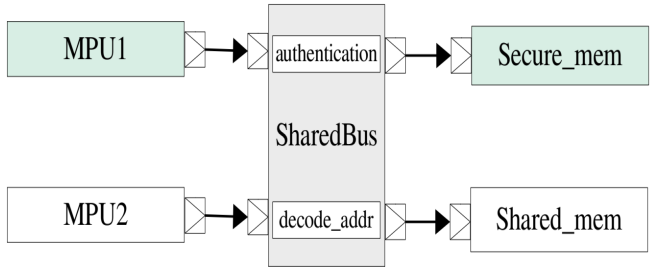
Timing Flows Detection Methodology

- Motivating example
  - **Timing-based Data Flow Analysis**

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

$L_{st} \leftarrow$ ForwardTraverse $(source, CG)$
for each node $n \in CFG$ **do**
  if $n.type() ==$ condition **then**
    $n_{ctrl} \leftarrow$ Extract list of variables from n
    if $n_{ctrl} \cap L_{st} \neq \emptyset$ **then**
      $L_{path} \leftarrow$ DFS $(n, CFG, sink)$
      for each path $p \in L_{path}$ **do**
        for each node $n_p \in p$ **do**
          if $n_p.type() == condition$ **then**
            remove(p, L_{path})
    if $L_{path} \neq \emptyset$ **then**
      for each node $n_p \in p$ **do**
        if $sink \in n_p$ **then**
          $TF \leftarrow (n, n_p)$





The first condition type **node in the CFG** of the VP is L_{23} whose control variables *{i, SKey}* are not in the list of source taints (*L_st*). Therefore, the analysis continues to the next condition node which is L_{24} whose control variables are *{trans_key, SKey}*. Since trans_key is in list of source taints, further analysis is performed on the child nodes of L_{24} . The result of DFS analysis shows that there are two paths p1 and **p2**. As **p2** includes a condition type node (*L23*), it is eliminated from L_{path}. Thus, **p1** is the only member of L_{path} whose L_{35} includes sink. In this case, L_{24} and path **p1** are stored in **TF** and reported back to designers.

$$p_1 = \{L_{24} \rightarrow L_{25} \rightarrow L_{35}\}$$

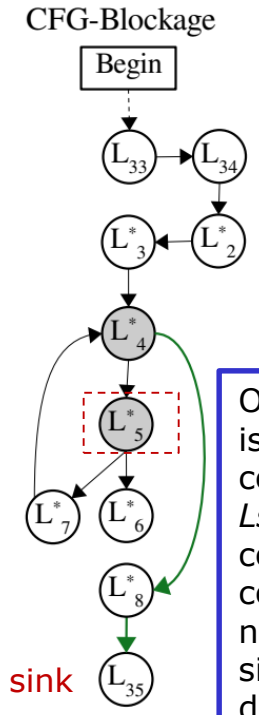
$$p_2 = \{L_{24} \rightarrow L_{26} \rightarrow L_{23} \rightarrow L_{27} \rightarrow L_{35}\}$$

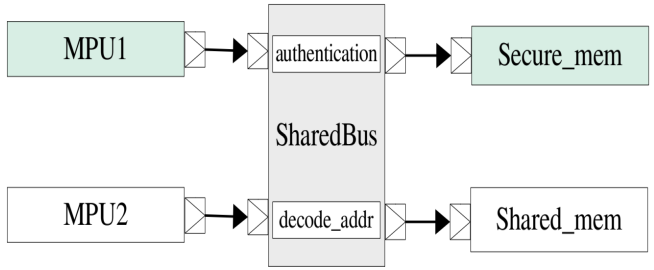
Timing Flows Detection Methodology

- Motivating example
  - **Timing-based Data Flow Analysis**

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

$L_{st} \leftarrow$ ForwardTraverse $(source, CG)$
for each node $n \in CFG$ **do**
  if $n.type() ==$ condition **then**
    $n_{ctrl} \leftarrow$ Extract list of variables from n
    if $n_{ctrl} \cap L_{st} \neq \emptyset$ **then**
      $L_{path} \leftarrow$ DFS $(n, CFG, sink)$
      for each path $p \in L_{path}$ **do**
        for each node $n_p \in p$ **do**
          if $n_p.type() == condition$ **then**
            remove(p, L_{path})
    if $L_{path} \neq \emptyset$ **then**
      for each node $n_p \in p$ **do**
        if $sink \in n_p$ **then**
          $TF \leftarrow (n, n_p)$





On the other hand, analyzing the CFG-Blockage shows that there is no timing flow in the VP as there is no explicit path from conditional node L^*_5 (which its control variable {*trans_key*} is in *Lst*) to the **sink**. The only available path (**p1**) is through the condition node L^*_4 which is eliminated from Lpath as includes a condition type node (L^*_4) . Therefore, the Lpath for this condition node (L^*_5) is empty. As we can see in this graph, the update on sink (node L_{35}) is fully controlled by the condition node L^*_4 which does not have any sensitive variables.

$$p_1 = \{L^*_5 \rightarrow L^*_7 \rightarrow L^*_4 \rightarrow L^*_8 \rightarrow L_{35}\}$$

Conclusion

- Validation of a given SoC's security architecture against **functional** and **timing flows are of utmost importance**.
- For functional flow analysis, we take advantage of a **dynamic information flow analysis,** performed by automatically extracting the run-time simulation behavior (TLM transactions) of VPs.
- In the case of timing flow analysis, at the heart of the approach is **a scalable static information flow analysis** that operates directly on the AST of SystemC VPs.
- We show how the analysis **formally represents the behavior of a given VP** in terms of data and control flows.
- The proposed methodologies are **automated**, **fast**, and **do not rely on any commercial tool** for their analysis.

Thank you!

Validation of SoCs Security Architecture: Challenges, Threats, and Methods

Mehran Goli

University of Bremen, Germany

DFKI Bremen, Germany

mehran@uni-Bremen.de

April 24-28, ICONS 2022