Safety Verification of Third-Party Hardware Modules via Information Flow Tracking

Andres Meza*, Francesco Restuccia*, Ryan Kastner*, and Jason Oberg†
*University of California San Diego
†Tortuga Logic Inc, San José

***Abstract*—Modern System-on-Chip (SoC) architectures are heterogeneous consisting of hundreds of IP cores and shared on-chip resources. On-chip communication uses simple and efficient standards like AMBA AXI and TileLink. These communication standards focus on performance and are often underspecified with respect to safety and security. If used incorrectly, this opens the door for nefarious behaviors, which are especially dangerous in mission-critical applications that have tight constraints on safety and security. These behaviors are compliant with the on-chip communication standard and therefore can be difficult to capture in a standard verification flow. This paper describes how to use Information Flow Tracking (IFT) to verify the safety of bus interactions among on-chip hardware resources. Our methodology is integrated into a safety verification flow leveraging Tortuga Logic Radix-S IFT tool.**

I. Introduction

Mission-critical systems increasingly rely on system-on-chip (SoC) architectures to deliver high performance and meet real-time constraints. For example, deep neural networks are commonly executed on custom on-chip hardware accelerators that perform object detection, image classification, and other critical computing tasks.

Due to the critical nature of the operating tasks, mission-critical systems have strict safety and security requirements. Among them, *timing predictability* is crucial – the system must be able to correctly operate its critical functionaries within a given *deadline*. Breaking such timing constraints can cause dramatic consequences.

Heterogeneous systems are composed of multiple specialized modules. It is common practice to integrate third-party modules alongside in-house modules to obtain a platform with the desired functionality. Communication among these modules and all of the on-chip resources is facilitated by a system interconnect implementing an on-chip communication protocol. In this paper, we use AMBA AXI due to its widespread usage throughout the industry. A similar analysis is possible with other on-chip communication protocols, which we leave as future work.

Generally, each module, be it a third-party module or an in-house developed module, utilizes the same communication protocol as the system interconnect to communicate with the rest of the system. Thus, the role of the system integrator is to verify that all modules, especially third-party modules, strictly adhere to the requirements set out in the standard for that protocol.

However, even if every module passes the aforementioned verification, this does not mean that the system is free of communication-related vulnerabilities. Previous works have demonstrated how a lack of specification in the on-chip protocol definition can be exploited by hardware modules to generate conditions endangering the execution of the entire system [1]–[3]. Such conditions must be avoided in any mission-critical system.

We explore the use of hardware information flow tracking tools to verify the safety requirements of on-chip communication. Hardware information flow tracking (IFT) is a verification technique that enables the tracking of information as it propagates through the hardware [4]. Hardware IFT techniques have been developed and popularized to identify security vulnerabilities in hardware modules throughout the development lifecycle [5]–[8]. Tortuga Logic Radix-S is a simulation-based IFT tool for security verification [9]. We use Radix-S for our experiments, but note that our techniques generalize to other commercial formal IFT tools.

We demonstrate our safety verification methodology to verify an AXI bus stall issue caused by a lack of specification in the AMBA AXI standard [1]. Our verification is performed on a design using AXI-compliant hardware modules implemented on a Xilinx FPGA multi-core SoC platform. Although we focus on a specific issue of the AMBA AXI standard in this paper, our methodology can be extended for the verification of other weaknesses or vulnerabilities related to AMBA AXI and, eventually, to other popular on-chip communication standards (TileLink, Wishbone).

In the next section, we introduce an example SoC architecture and describe the AXI bus stall problem. Section III describes our safety verification methodology and demonstrates its effectiveness for the AXI bus stall problem. We conclude in Section IV and provide some directions for future work.

II. Motivations and Background

The AXI standard leaves great flexibility in the definition of bus transactions. If not properly managed, such flexibility has been demonstrated to be the source of unpredictable behavior ranging from uneven bandwidth distribution [2] to complete system deadlocks [1]. This section briefly introduces the architecture under analysis and the safety issue under consideration.

A. Sample SoC architecture

A typical System-on-Chip (SoC) architecture is composed of a set of controller devices C (e.g., processors, hardware accelerators, DMAs, etc.) sharing a set of peripheral devices P (e.g., memory controllers, GPIOs, etc.). Controllers and peripherals communicate through a system interconnect. We assume that the system interconnect is based on the AMBA AXI standard [10]. A generic SoC architecture deploying N controllers $(C_1, \ldots, C_N)$ and L peripherals $(P_1, \ldots, P_L)$ is depicted in Figure 1. Each of the controllers $(C_1, \ldots, C_N)$ exports a manager (M) interface. Each of the peripherals exports a subordinate (S) interface. The AXI interconnect I_{AXI} arbitrates the access of the controllers to the peripherals.

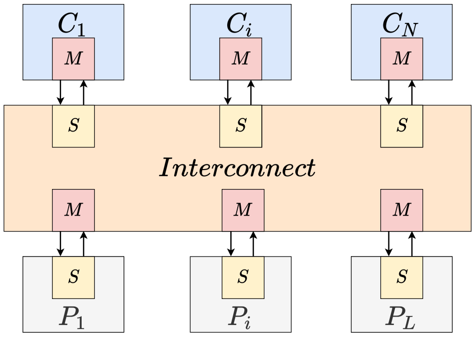

Fig. 1: The sample System-on-chip architecture deploying N controller modules (C) and L peripheral modules (P).

Bus transactions are issued by controllers and served by peripherals. An AXI bus transaction starts with the issue of an address request from the generic controller C_i. The AXI interconnect samples the address request and routes it to the destination peripheral P_j. The address request is served by P_j, which provides the requested read data (in the case of a read transaction) or accepts the provided write data and acknowledges with a write response (in the case of a write transaction).

B. The AXI bus stall problem

Consider a two-controller (C_0 and C_1), one peripheral (P_0) architecture (see Figure 1 with $N = 2$ and $L = 1$). To demonstrate the issue on real hardware, we consider the AXI SmartConnect [11] as the system interconnect, which is the state-of-the-art interconnect for Xilinx systems. AXI SmartConnect implements a round-robin arbitration to solve conflicts among controllers. We consider that each controller issues a single write request. This simple configuration is enough for showcasing the AXI bus stall problem. It is worth mentioning that the same considerations hold for architectures involving more controllers/peripherals and issuing multiple requests. The described system has been deployed on the Xilinx ZYNQ Ultrascale+ SoC platform, where C_0 and C_1 are two custom high-performance DMAs, while P_0 is the shared DRAM memory controller of the platform. The conditions generating the AXI bus stall problem are briefly described next. A full description of the AXI bus stall problem is reported in [1].

(1) Assume that C_0 issues a write request A_0 directed to P_0. Once issued, A_0 is sampled by I_{AXI} and routed to P_0. Similarly, assume that in the same time frame C_1 issues a write request A_1 directed to P_0.
(2) Assume that the round-robin arbitration at I_{AXI} is won by A_0 – this means that A_0 is propagated to P_0 by I_{AXI} before A_1.
(3) According to the AXI standard, after a write request is issued, the controller should provide the corresponding data to be written in the peripheral. However, AXI does not define any time limit for the controllers to provide the data words after a request has been granted.
(4) If C_0 delays the data provisioning for A_0, the service of A_1 is delayed – A_0 has been propagated first by I_{AXI} to reach P_0. This means that even if C_1 is ready to provide the data corresponding to its transaction A_1, such data cannot be propagated by I_{AXI} to P_0 until C_0 completes the provisioning of all of the data words corresponding to A_0 (data interleaving is forbidden in write transactions beginning in AMBA AXI4 [10]).

The delay possibly introduced by C_0 is fully compliant with the AXI standard. According to AXI, C_0 can delay its data provisioning for an unbounded time after booking the bus by issuing the address request. This means that C_0 can leverage this lack of specification provided by the standard to directly affect the availability of the shared peripheral P_0 to the other controllers in the system (in the specific case, C_1). A similar issue can happen for read transactions – the description is omitted for brevity.

III. Safety Verification Methodology

In the realm of security verification, many tools enable system integrators to specify security properties and then check if their system adheres to these properties. Some of these tools rely on formal methods in order to carry out this check while others rely on simulation-based methods. Due to the scaling issues associated with formal methods [4], the safety verification methodology we propose in this section relies on simulation-based tools. We leave the investigation of the trade-offs between formal methods and simulation-based methods for future works. In this section, we introduce a method capable of detecting the AXI bus stall problem described in Section II-B through the use of a commercial simulation-based IFT tool, supported by a custom-developed, parametrizable trigger module.

A. Addressing the AXI bus stall problem

At its core, the AXI bus stall problem described in Section II-B is caused by controllers not being constrained to provision data within a limited amount of time after booking the bus with a write request. In order to mitigate against this, system integrators need to verify that each controller provisions data within a limited amount of time. The acceptable amount of time varies depending on the constraints of the system. Once an integrator determines how much delay each controller can safely introduce into the system, they can follow the proposed methodology outlined in Section III-C to verify that each controller meets the appropriate delay requirement.

B. The Trigger Module

Our verification in Section III-C relies on a custom, parameterizable module (i.e., the trigger module) to track the state of the write transactions for a single controller. The inputs to this module are a signal specifying the maximum delay limit for the controller and the incoming and outgoing AXI signals of the controller's write-related channels. The output of this module is a signal indicating the state of the controller with respect to write transactions. Specifically, the module outputs whether the controller is in one of three states: (1) idle (i.e., not in a transaction), (2) in a transaction and provisioning data within the delay limit, or (3) in a transaction and provisioning data outside of the delay limit. With this module and the information it provides, system integrators can verify the safety of the controller using the approach described in the following section.

C. Simulation-Based IFT for Safety Verification

Our safety verification approach relies on simulation-based information flow tracking. This approach requires a design, a testbench, IFT properties, and an information flow tracking tool (e.g., Tortuga Logic's Radix-S). The following steps outline the flow of this verification approach when using Tortuga Logic's Radix-S.

1) Determine the Delay Limits: The first step in the safety verification approach determines the appropriate delay limit for every controller C_i in the system. A controller's delay limit is the maximum amount of delay (measured in clock cycles) the controller can safely introduce into the system. System integrators must determine this based on the constraints of their system. In hard real-time systems, this can be achieved by applying the results of worst-case analysis bounding the response time of the hardware modules deployed in the system [12]–[14]. In systems dealing with softer timing constraints, profiling and over-provisioning techniques can be evaluated as an alternative strategy.

2) Insert the Trigger Modules: The second step inserts a set of trigger modules T into the existing design. Since each trigger module T_i can only track the write transaction state of a single controller, system integrators should add a trigger module for every controller they wish to verify. Figure 2 depicts a generic SoC architecture deploying N controllers ($C_1, \ldots, C_N$) and L peripherals ($P_1, \ldots, P_L$) with the addition of K trigger modules ($T_1, \ldots, T_K$) for verification purposes (note that $0 \leq K \leq N$). Given the simplicity of the trigger module's design, we measured a minimal impact on the overall simulation time of the system. This means that system integrators could add a trigger module for every controller in their system (i.e., $K = N$) with minimal overhead.

3) Specify the Safety Properties: In order to verify that a controller meets a certain safety requirement (i.e., it does not introduce a delay larger than its delay limit determined in Step 1), the safety requirement must be represented as a formally specified and evaluable expression containing design signals, explicit values, and operators. We will refer to these expressions as safety properties, and each safety property will be specified as an IFT property. We use IFT properties because they enable us to track a signal as it propagates through a hardware design [4]. The following safety property template, based on the security property templates in [15], specifies that illegally provisioned write data (i.e., data provisioned after a delay limit as indicated by the output of a trigger module T) from some controller C should not flow to the system interconnect I_{AXI}. In other words, this property will track a controller's write data (i.e., `` `C_w_data` ``) when that controller's trigger module indicates that it is illegally provisioning data (i.e., `` `T_out` == 2`b11 ``), and if any of the illegally provisioned write data flow to the system interconnect (i.e., `` `I_AXI_w_data` ``), the property will fail.

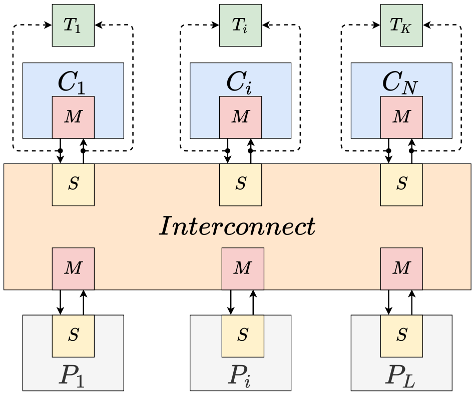

Fig. 2: The sample System-on-chip architecture deploying N controller modules (C) and L peripheral modules (P) with the addition of K trigger modules (T) for verification purposes (note that $0 \leq K \leq N$).

```
`C_w_data`            //source sig
  when
  (`T_out` == 2`b11)  //tracking condition
  =/=>                //no-flow operator
`I_AXI_w_data`        //destination sig
```

It should be noted that the hardware information flow property above makes use of the no-flow operator (=/=>) in order to indicate non-interference between the source signal and a destination signal [16]. Hardware information flow properties are a type of hyperproperty that are specified over sets of traces and are useful for proving a key aspect of information flow analysis (i.e., non-interference) [15].

4) Generate the IFT Models: The fourth step of the process uses Tortuga Logic's Radix-S tool to generate IFT models for every safety property specified in Step 3. The IFT model generated by Radix-S is a modified version of the design to be simulated that has been instrumented with additional logic in order to enable hardware information flow tracking of the design signals relevant to a particular IFT property. It should be noted that the IFT models generated by Radix-S are typically referred to as "security monitors" or "security models" but, for the sake of clarity, we refer to them as IFT models in this safety verification context.

5) Create Testbench: In the fifth step of the process system integrators create a testbench in order to drive the simulation of the design and IFT models. System integrators are likely to already have a testbench at their disposal for functional verification purposes. Functional testbenches can be reused for this safety verification but they may need to be extended or modified depending on how thoroughly they stimulate the controllers in the system. Interested readers can refer to [17] for methods of determining and increasing testbench coverage.

6) Verify Safety Properties via Simulation: The final step in the safety verification is to verify the specified safety properties via simulation. After the simulation has been completed, system integrators determine which, if any, properties failed and then adequately address the delay introduced by the controllers associated with those failing properties. In the event of a failed property during verification, system integrators should take appropriate countermeasures including but not limited to requesting a module redesign or sourcing alternative modules.

We tested our proposed methodology by using it to identify the AXI bus stall problem in a system integrating fully-compliant AXI modules. To this end, we leveraged the test setup described in Section II-B, modifying the DMA modules to introduce programmable bus stalls during write transactions. As expected, our proposed methodology was able to detect the bus stalls introduced by the DMAs – the safety verification failed any time a DMA module introduced a stall longer than the maximum allowable stalls parametrized in the specific instance of the verification.

IV. Conclusion

We proposed a safety verification methodology utilizing simulation-based information flow tracking for the purpose of verifying the safety of on-chip communication in hardware modules. We validated this methodology by using it to identify fully-compliant AXI controllers which introduced delays capable of causing the AXI bus stall problem via a write transaction.

While the safety verification methodology was focused on addressing the write case of the AXI bus stall problem, there are more safety vulnerabilities allowed for by the AMBA AXI standard that could be identified using a slightly modified version of this methodology. Some of these vulnerabilities include the read version of the AXI bus stall problem fully described in [1], the heterogeneous burst length problem described in [2], and other specific issues that can be generated by behaviors related to transactions IDs, memory protection, and memory buffering. Expanding the framework to consider additional vulnerabilities is a compelling future research direction.

Another interesting direction would explore how other verification techniques could be used to carry out safety verification. For instance, formal methods and standard simulation-based methods (without IFT) could be used to address the safety verification task presented in this paper, albeit with different sets of steps and safety properties. An in-depth analysis could provide valuable insight regarding the trade-offs (e.g., effort required by system integrators, verification time overhead, level of assurance, etc.) between such techniques for safety verification tasks on systems in real scenarios with multiple properties to be verified.

References

[1] F. Restuccia, A. Biondi, M. Marinoni, and G. Buttazzo, "Safely Preventing Unbounded Delays During Bus Transactions in FPGA-based SoC," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020.

[2] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo, "Is your bus arbiter really fair? restoring fairness in AXI interconnects for FPGA SoCs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, p. 51, 2019.

[3] F. Restuccia, A. Biondi, M. Marinoni, G. Cicero, and G. Buttazzo, "Axi hyperconnect: A predictable, hypervisor-level interconnect for hardware accelerators in fpga soc," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[4] W. Hu, A. Ardeshiricham, and R. Kastner, "Hardware information flow tracking," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–39, 2021.

[5] 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.

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

[7] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *Acm Sigplan Notices*, vol. 50, no. 4, pp. 503–516, 2015.

[8] C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni, "Tainthls: High-level synthesis for dynamic information flow tracking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 798–808, 2018.

[9] *The Tortuga Logic Radix-S offical website*, Tortuga Logic, https://tortugalogic.com/radix-s/.

[10] *AMBA AXI and ACE Protocol Specification*, ARM, 2011.

[11] *SmartConnect, LogiCORE IP Product Guide*, Xilinx, 2018, pG247.

[12] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo, "Modeling and Analysis of Bus Contention for Hardware Accelerators in FPGA SoCs," in *32st Euromicro Conference on Real-Time Systems (ECRTS 2020)*, 2020.

[13] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 239–252.

[14] M. Hassan and R. Pellizzoni, "Bounding DRAM interference in COTS heterogeneous MPSoCs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2323–2336, 2018.

[15] F. Restuccia, A. Meza, and R. Kastner, "Aker: A design and verification framework for safe and secure soc access control," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[16] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[17] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design Test of Computers*, vol. 18, no. 4, pp. 36–45, 2001.