A Framework for Design, Verification, and Management of SoC Access Control Systems

Francesco Restuccia, Andres Meza, Jason Oberg, and Ryan Kastner

Abstract—System-on-chip (SoC) architectures are a heterogeneous mix of microprocessors, custom accelerators, memories, interfaces, peripherals, and other resources. These resources communicate using complex on-chip interconnect networks that attempt to quickly and efficiently arbitrate memory transactions whose behaviors can vary drastically depending on the current mode of operation and system operating state. Security- and safety-critical applications require access control policies that define how these resources interact to ensure that malicious and unsafe behaviors do not occur. AKER is a design and verification framework for on-chip access control. The core of AKER is the access control wrapper (ACW) – a high-performance yet efficient hardware module that dynamically arbitrates on-chip communications. AKER distributes ACWs across the SoC and programs them to perform local access control. AKER provides a firmware generation tool and a property-driven security verification methodology to ensure that the ACWs are properly integrated and configured. AKER security verification confirms that the ACW behaves properly at IP level. It verifies the hardware root of trust firmware configures the ACW correctly. And it evaluates system-level security threats due to interactions between shared resources. AKER is experimentally validated on a Xilinx UltraScale+ programmable SoC. Additionally, an AKER access control system is integrated into the OpenPULP multicore architecture that uses OpenTitan hardware root-of-trust for firmware configuration.

Index Terms—Access control systems, System-on-chip architectures, Security verification, Safety-critical, Security-critical.

1 Introduction

Modern system-on-chip (SoC) architectures use complex interconnect networks to provide low latency, high bandwidth communication between microprocessors, hardware accelerators, memories, I/Os, and other on-chip resources. On-chip communication protocols are often subject to safety and security considerations given the increasing requirements to provide mechanisms for safe and secure computation and data storage. Safety considerations translate into tight constraints on latency, throughput, and resource consumption [1], [2]. Security considerations require guarantees on the confidentiality and integrity of the data. The access control policy defines the ability to access the shared on-chip resources. The access control policy can be dynamic as access control will change over the SoC lifecycle. It also varies depending on the system operating mode. For example, the access control policy can varies drastically depending on whether the system is booting, performing secure operations, operating in debug mode, or in return merchandise authorization (RMA) mode. Access control plays such a critical role in the safety and security of the SoC. As evidence, 5 out of 12 of the 2021 CWE Most Important Hardware Weaknesses are related to access control systems [3]. Such vulnerabilities are extremely dangerous as they provide opportunity for low-level system access. If properly exploited, they can endanger the confidentiality, integrity, and availability of the data and computation. Moreover, they can be challenging to patch – at best a firmware rewrite can be sufficient; at worst, hardware features must be disabled or a chip re-manufacturing is required.

Francesco Restuccia, Andres Meza, and Ryan Kastner are with the University of California San Diego; Jason Oberg is with Cycuity.
E-mail: {frestuccia, anmeza, kastner}@ucsd.edu, jason@cycuity.com

AKER is a framework that aids in the design, verification, and integration of SoC access control systems. AKER is built upon a flexible access control wrapper (ACW) that efficiently monitors on-chip communication requests at the source. AKER provides techniques to automate the integration of the ACWs into an efficient distributed SoC access control mechanism. AKER automatically generates firmware to configure, monitor, and control the ACWs. AKER automates the mapping of SoC access control policies onto the distributed ACWs. AKER provides an extensive security verification methodology that assesses the SoC access control at the IP-, firmware-, and system-level. AKER's major contributions include:

Security Verification: AKER provides a property-driven security verification methodology [4] to provides assurance about the secure operation of AKER-based access control systems. The security verification is done at the *IP* level, the *firmware* level, and the *system* level.

Secure firmware generation: AKER integrates with the OpenTitan Hardware Root of Trust. Moreover, it provides a tool for spotting ill-specified or incorrect access control policies potentially causing illegal data leaks and for the automatic generation of secure firmware for the management of an AKER access control system.

Flexibility: AKER complies with the AXI standard and is completely transparent to the system – no internal knowledge or modifications are required to controllers, peripherals, and interconnect to integrate an AKER access control system. AKER allows static or dynamic management of the access control policy by the HWRoT, providing high flexibility to cope with the complex life-cycle of modern SoCs. The integration of an AKER access control system has been experimentally demonstrated on an FPGA SoC architecture and on the OpenPULP [5].

Efficient performance and resource usage: AKER filters illegal transactions before entering the interconnect – illegal requests are never allowed to reach the network. This avoids any identification issues and prevents system-level interference generated by illegal transactions (e.g., DoS attacks, see Section 5.1). AKER access control systems have a very low impact on system performance ($< 1\%$ in the tested scenarios). The resource usage of an AKER-based access control system is minimal and is configurable to be tailored to the SoC and use case.
Open-Source: The AKER framework is openly released [6]. This provides a solid base for design and security verification extensions.

This article expands upon the initial work [7] to include automated firmware generation and verification. Additionally, it proposes system-level security verification techniques to assess the safety and security of the system access policy.

2 SYSTEM-ON-CHIP ACCESS CONTROL

A System-on-Chip (SoC) architecture can be modeled as a set of *controller* devices accessing one or multiple *peripheral* devices.[1] Examples of controllers are processors, hardware accelerators, and other IP cores. Examples of peripherals are DRAM memory controllers, ROM, GPIOs, IP core control and status registers (CSRs), and on-chip memories. Controllers and peripherals communicate using memory-mapped accesses as defined by AXI [8], TileLink [9] or some other on-chip communication protocol, which provides asymmetric, flexible interfaces for targeting high-performance communications. This enables autonomous and concurrent communications with the shared peripheral resources of the SoC. Controllers can have different rights in accessing the shared resources – in security-critical systems, an *access control system* manages and arbitrates the access to the on-chip resources. The access control system plays a critical role in maintaining the integrity, privacy, and availability of the SoC. The access control system uses an *access control policy* that specifies which requests are allowable at a given time. It is crucial that the access control system properly enforce the access control policy while limiting its impact on performance and area.

2.1 SoC Architecture

Figure 1 depicts a generic SoC architecture consisting of controllers C and peripherals P. Each controller ($C_1, \ldots, C_N$) exports a manager (M) interface that accesses a shared on-chip bus. The L peripherals P ($P_1, \ldots, P_L$) each export a subordinate (S) interface that service accesses from the controllers. Controllers and peripherals are interconnected via an interconnect I that arbitrates the accesses of the controllers to the peripherals. A controller C_i performs a transaction with peripheral P_j by issuing an address request through its manager interface. The interconnect collects the request and routes it to the destination peripheral P_j. P_j is accessible via its *peripheral address region*, which is a unique

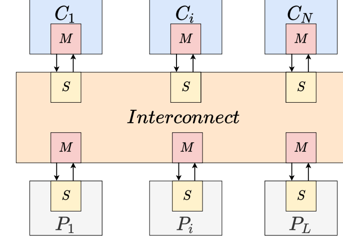

Fig. 1: The sample System-on-chip architecture deploying N controller modules (C) and L peripheral modules (P).

set of contiguous addresses assigned to P_j. The transaction is eventually 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). AKER uses the AMBA AXI on-chip interface specification to define the manager and subordinate interfaces. The AXI standard uses a multi-channel communication interface consisting of five independent channels: AR channel (address read), AW channel (address write), R channel (data read), W channel (data write), and B channel (write response). We focus on AXI in this work, but the proposed techniques can easily be modified for other communication protocols including TileLink and Open Core Protocol (OCP).

2.2 SoC Access Control Policy

The SoC access control policy specifies the allowable transactions between controller and peripheral resources. This is derived from an access control list that articulates the permissions of the controllers to access the peripherals. For example, an access control list would state that controller C_1 has read-only permission to peripheral P_2, C_2 has read/write access to peripheral P_1, and C_2 has read-only permission to P_2. We assume that a memory map is provided describing how the memory space is divided between the peripherals. This allows to map the local access control list to contiguous address ranges that can be checked using an execution monitor [10]. The goal is to create a separation kernel – "an environment which is indistinguishable from that provided by a physically distributed system" [11].

2.3 Threat Model

One or more controllers attempt to perform *illegal* transactions, i.e., they violate the SoC access control policy that is provided by the system designer. The illegal requests may be done maliciously. Or they could be unintentional, e.g., due to a poorly programmed or misconfigured controller IP core. The security threats are related to integrity (e.g., an untrusted controller can modify the control or status registers of a peripheral), confidentiality (e.g., keys are leaked to non-secure memory space), and system availability (e.g., a denial-of-service attack). It is assumed that the functionalities of I_{AXI} and of the P are trustworthy and implemented correctly. The MITRE consortium defines a set of common hardware weaknesses (CWEs) [12]. Many hardware CWEs are related to SoC access control. It is assumed that the attacker has full knowledge of these types of weaknesses. We formally specify the vulnerabilities that we consider

1. The terminology controller/peripheral is used in the paper to describe the system-level interactions between the IP cores. When specifically referring to the AXI standard, we use the terms manager (M) and subordinate (S).

as properties during our security verification process (see Section 3.3, Section 4.1, and Section 4.3). We map these vulnerabilities to MITRE CWEs. We verify the properties using a property-driven hardware security methodology [4].

2.4 SoC Access Control System

Now that we have defined the SoC architecture, access control policy, and the threat model, we address the implementation of the access control system. An important question is where to implement the access control mechanisms. The primary alternatives include access control enforcement at the peripherals, in the interconnect, or at the controllers (leveraging a centralized specialized resource). We discuss the tradeoffs between each of these options, which also serves as motivation for the AKER access control system.

2.4.1 Access Control in the Interconnect

An access control policy can be enforced in the interconnect by implementing only selected physical connections within the crossbar. The configuration of the AXI interconnect is statically configured at design time only with the physical connections allowed by the policy between controllers and peripherals. This completely removes the ability to communicate between controllers and peripherals.

Limitations: A hard-coded access control policy prohibits any sort of dynamic access control policy. This limits the viability of this architecture. Modern SoCs have complex lifecycles and access control policies change over their lifetimes – moving from manufacturing, to test, to system integration, and ultimately put into use. Each phase has different policies with respect to accessing on-chip resources. Additionally, safety- and security-critical systems operate in different modes (e.g., secure and non-secure) and those have different access control policies. A static access control policy would not work in these scenarios.

2.4.2 Access Control at the Peripherals

The access control policy can be enforced at the peripherals. Such an access control mechanism analyzes read and write address requests that arrive at a peripheral and rejects those requests that violate the access control policy. SPE [13] and SECA [14] are two examples that perform access control at the peripheral. The access control policy is enforced in a decentralized manner using logic local to each peripheral that analyzes the validity of the requests. The local access control policy can be programmable to provide the flexibility to handle dynamic policies. In such cases, the local access control policy requires a secure configuration stage.

Limitations: Performing access control at the peripheral requires that the transactions are securely authenticated and attributable to the requesting controller. Unfortunately, the AXI standard [8] does not provide built-in support for such authentication. A common workaround leverages the AXI ID signals for source identification [15]. However, the IDs were designed to facilitate parallel requests by allowing a manager to issue parallel address requests using multiple ID values. Moreover, the AXI standard does not specify a mechanism to address the integrity of the ID [8]. Thus, ID manipulation is possible during request propagation, adding uncertainty about the source of the address requests. Additional complications arise due to the fact that AXI requires that any illegal request received by a peripheral to be properly terminated with an AXI-compliant error. It is used to avoid network locks. Thus, illegal requests will occupy on-chip bandwidth on the initial transaction and due to the corresponding error message from the peripheral. This causes interference with legitimate transactions and could potentially lead to a denial-of-service attack.

2.4.3 Access Control at the Controllers

Access control systems can be also deployed at controllers. As controllers are considered untrusted entities, the controller cannot be trusted in enforcing an access control policy on its own issued transactions. For such a reason, some solutions have been proposed based on wrapping each controller with a *smart wrapper* module, which interfaces with a centralized access control *central policy engine* deployed in the system. The smart wrappers communicate with the central engine any time a transaction is issued by a controller. The central engine is responsible for authenticating any address request issued in the system. E-IIPS [16] and RSPE [17] are two examples of such centralized access control systems. Such solutions cut off any illegal transaction at the source, with clear benefits on system performance and security.

Limitations: Every decision is made by the central security policy engine, which requires that all of the transactions from every controller are routed to the central policy engine. The engine must address all of the requests in low latency and high-throughput manner so that it does not induce a performance bottleneck in the on-chip communications. This can be challenging in high-performance or latency-critical applications.

In the following section, we introduce AKER access control systems. Our solution enforces the access control system at the controllers and overcome the limitations of the previously proposed solutions.

3 AKER ACCESS CONTROL WRAPPER

AKER is a design and verification framework for developing high-performance, flexible, and verifiably secure SoC access control systems. An AKER access control system consists of a distributed set of ACWs that locally monitor the memory transactions from the controllers. It includes efficient logic for checking the validity of the controller's transactions with a local access control policy. The ACW decouples a controller on any illegal request. The controller can be readmitted to the SoC interconnect by the trusted entity. The local access control policies are configurable by the trusted entity using firmware automatically generated by AKER. AKER includes security verification techniques to ensure its correct integration, configuration, and functionality. AKER provides security property templates matching the CWEs related to access control systems. These properties are crucial for property-driven hardware security verification [4]. The properties cover a range of confidentiality and integrity vulnerabilities at the IP, firmware, and system levels. It is worth noting that our framework for security verification is conceived to be extendable with any other security property required to be verified in the system under analysis. AKER builds on three pillars: **(i)** the Access Control Wrapper

(ACW) – a high-performance, programmable module monitoring the requests issued from a controller (Section 3.1); **(ii)** the automated firmware generation tool for securely programming the ACWs (Section 4.2) and **(iii)** the IP-level (Section 3.3), Firmware-level (Section 4.1), and System-level (Section 4.3) property-driven security verification.

3.1 Internals of the Access Control Wrapper

The Access Control Wrapper (ACW) is a configurable hardware module designed to transparently monitor an AXI-compliant controller. An ACW filters memory transactions from a controller, rejecting those that violate the access control policy while allowing legal transactions to pass through. Figure 2 shows an example of an AKER-based access control system. An access control wrapper ACW_i is situated between the controller C_i's M interface and the interconnect I_{AXI}. The ACW monitors the transactions from its controller immediately stopping any transaction that violates the access control policy. The ACW stores a local access control policy describing that controller's allowable memory address requests. The ACW exposes an S interface that is used to interact with ACW, e.g., to program its local access control policy and query its control and status registers. This S interface is connected to a *trusted entity* (TE) through a separate secure bus – this ensures exclusive access to the ACWs from the TE and avoids misconfiguration of the ACWs as long as the TE is not compromised. AKER uses the OpenTitan hardware root of trust as its trusted entity though this could be changed to another root of trust with necessary modifications. The ACW has an interrupt line to indicate any illegal accesses. This is connected to the TE.

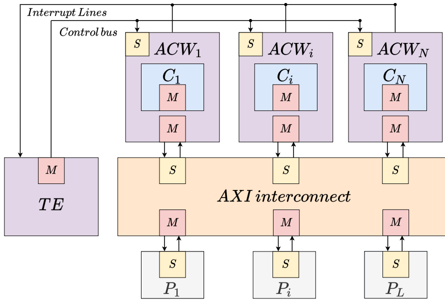

Fig. 2: A multi-controller, multi-peripheral SoC communication architecture using an AKER access control system. Legal controller transactions are transmitted to the interconnect while the illegal ones are blocked at the source by the ACW.

Each ACW_i holds a local access control policy $LACP_i$, which is configured and maintained by TE. The $LACP_i$ describes the address regions that can be accessed by C_i – it defines n_r regions for read operations and n_w regions for write operation. Each address request issued by C_i is checked against the configuration of the $LACP_i$; if the request is fully contained in one of the $LACP_i$'s address regions, the request is considered *legal* and propagated to I_{AXI}. n_r and n_w impact the resource consumption of the ACW module. The ACW design is developed to allow easy customization to match the requirements of the SoC. To maximize the performance, a request is checked in parallel with all of the regions. This means that the latency introduced by each ACW is independent of n_r and n_w and constant. The ACW has three operating modes:

1) Reset Mode: The initial state for ACW_i while it awaits a valid $LACP_i$ configuration. Any request issued by C_i is blocked in reset mode. Once $LACP_i$ is configured, ACW_i moves to supervising mode.

2) Supervising Mode: The normal operating mode of the ACW_i. Transactions issued by C_i are compared against the configured $LACP_i$ – legal transactions are propagated to I_{AXI}; illegal transactions are blocked and not propagated to I_{AXI}. An illegal transaction moves ACW_i into *Decouple Mode*.

3) Decouple Mode: All transactions from C_i are blocked from the interconnect. The ACW goes in this mode when C_i attempts an illegal transaction. The ACW_i saves diagnostic information about the illegal request into its anomalies registers and notifies the TE by raising an interrupt.

Readmission Policy: An ACW_i relies on the TE for readmission. This ensures that TE can take appropriate actions on C_i before safely readmitting the module in the system. The TE can retrieve and analyze the diagnostic information saved by ACW_i via the ACW S interface. It can perform recovery operations on C_i (such as resetting, reconfiguring, or even reprogramming C_i) before switching back to *Supervising Mode* and thereby readmitting C_i to communicate with peripherals. If the TE determines that the illegal request is the result of a permanent fault, it can keep the ACW_i in *Decouple Mode*, thus permanently disconnecting C_i from the system. These steps are performed by the HWRoT leveraging the functionalities provided by the AKER SFG tool described in Section 4.2. Figure 3 depicts the

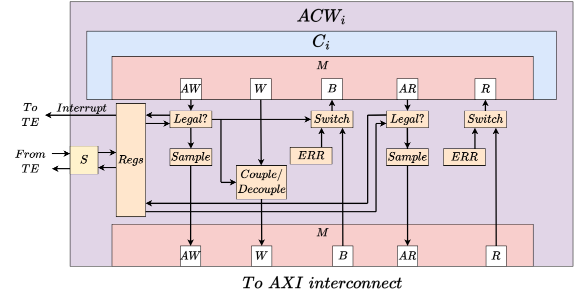

Fig. 3: The architecture of ACW_i: C_i is the controller module. $Regs$ are the configuration registers holding the $LACP_i$. The AXI S interface is managed by the HWRoT.

internals of ACW_i. The ACW is compatible with any AXI-compliant IP core. The ACW interfaces directly to the five AXI channels (AW, W, B, AR, and R) from the controller. The ACW inspects all memory transactions with the local access policy stored in ACW control registers ($Regs$). A legal transaction will pass through to the AXI Interconnect as normal. An illegal request is blocked, and the ACW decouples the controller's AXI M interface from the SoC AXI Interconnect. When C_i issues a read request AR through its M interface, ACW_i performs the following behaviors:

Address Check: AR is checked against the $LACP_i$. The address of AR is compared against each of the allowable read regions.

Legal Request: When AR is fully included in at least one of the allowed read regions of $LACP_i$, it is propagated to I_{AXI}.

Illegal Request: If AR is not fully included in any of the read regions stored in $LACP_i$, AR is denied

to reach I_{AXI}. ACW_i saves internally information regarding the illegal request AR. ACW_i sends an AXI-compliant error to C_i, notifies the TE about the illegal transaction, and switches into *Decouple Mode*. Previous outstanding transactions: any legal outstanding transaction initiated before an illegal transaction are completed normally.

When C_i issues a write request AW, the ACW_i behaves similarly as the read request, but uses AW instead of AR, and checks the address request with the $LACP_i$ write regions instead of the read regions. The AXI standard states that transactions cannot be aborted, thus C_i expects to provide its write data after it issues an illegal request. The write data is stored internally in an ACW_i status register and not transmitted to I_{AXI}. This can be inspected by the TE for readmission of the controller. ACW_i and acknowledges C_i with an AXI-compliant error.

3.2 ACW extension to other protocols

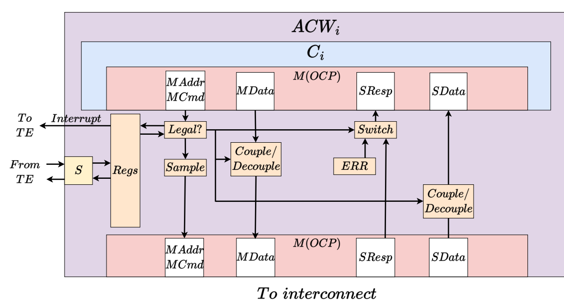

Fig. 4: The architecture of ACW_i ported to the OCP protocol.

Given its wide implementation in modern commercial platforms, we opted for developing our solution targeting the AMBA AXI protocol. Nevertheless, our solution can be ported to other popular on-chip protocols, such as the Open Core Protocol (OCP) [18] or SiFive TileLink [9]. The porting process starts with the identification of the signals dedicated to the propagation of transaction requests, read data, write data, and transaction responses. To make an example, differently from AXI, OCP shares common signals for the propagation of the read and write requests issued by C_i – $MAddr$ is the signal designated to propagate the target address while $MCmd$ propagates the type of the requested transaction (read or write). The $MData$ signal is the signal designated for the propagation of the write data provided by C_i. The signals $SResp$ and $SData$ are designated for the propagation of the request's responses and of the requested read data, respectively. Figure 4 depicts the internals of the ACW ported to operate following the OCP protocol.

3.3 IP-level Security Verification

IP-level security verification validates that the access control wrapper adheres to properties related to proper decoupling, the integrity controller's operation, and confidentiality of the configuration data. This section describes the AKER IP-level verification process focusing on the interactions between the controller C, the ACW module wrapping C, and a peripheral P. We aim to assure ACW performs its access control operations as intended using the six-step CWE-IFT verification process [19].

1) Create the Threat Model: The first step in the verification process develops the threat model. This assumes that the controller C is untrusted, while the ACW and P are trusted. The threat model evaluates the scenario where C can illegally transmit information to a peripheral P, i.e., the ACW is not correctly implementing its local access control policy.

2) Identify the Assets: Assets correspond to important information, resources, registers, and other components to be protected. Assets are used to refine the threat model to specific properties, e.g., AKER uses the assets in its templates to automatically generate security properties. The IP-level assets are the five AXI channels connecting the ACW to C and P and the signals forming the anomaly registers and configuration. These assets are referred to as the M AXI group and config/control group, respectively, due to the similarity in their security requirements.

3) Identify the Potential Weaknesses: Weaknesses are mechanisms that introduce security vulnerabilities related to the defined assets and the threat model. Identifying these weaknesses requires an understanding of the design specification, the hardware implementation, and the threat model. As mentioned in Section 2.3, Mitre's CWE database contains many CWEs (i.e., weaknesses) related to SoC access control which we assume to be known by the attacker. Due to the implementation-agnostic and broad nature of CWEs, we manually review every access-control-related CWE in order to determine if the weakness it describes is applicable to **(1)** our design/implementation and **(2)** the assets we are protecting at the current level of verification. Although this process may seem costly with respect to manual effort, it provides structure to what would otherwise be an ad-hoc weakness identification process. In other words, weaknesses are difficult to systematically identify because their presence in a design is typically unknown and/or accidental. By manually reviewing CWEs, we increase the likelihood of considering and identifying potential weaknesses in our design that we may not have considered without referring to the CWE database. It is worth mentioning that hardware CWEs are a relatively new addition to the CWE database so it is possible for a designer-identified weakness to not yet have a corresponding CWE. For this reason, we also rely on security analysis guided by the design specification (instead of CWEs) in order to identify potential weaknesses in our design implementation. Using the process described above, we identify 17 CWEs related to IP-level verification:

Relevant CWEs: 1220, 1221, 1244, 1258, 1259, 1264, 1266, 1267, 1268, 1269, 1270, 1271, 1272, 1274, 1280, 1282, 1326

These are divided into two groups: **(1)** CWEs related to M AXI including the read and write access points available to C (the AXI channels). **(2)** CWEs related to the configuration registers (e.g., storing the $LACP$ of the ACW), to the anomaly registers storing metadata on illegal requests, and to the logic checking the legality of the transactions.

4) Define the Security Requirements: Security requirements articulate how a weakness translates into a failure mechanism. Consider the M AXI group of assets. The security requirements addressing the existence and the content of information flows between C, the ACW, and P. The ACW

modulates the interactions between C and P. Thus, whenever C and P communicate, an information flow between C and the ACW and between the ACW and P occurs. The source of such flows commands their allowable behaviors. Information flows between C and P, must occur only when a legal transaction has been issued by C and the ACW is in supervising mode. In all of the other instances, the only allowable information flows are those where the source is the ACW, the destination is either C or P, and the content of the flow does not deviate from the default AXI values.

> **Requirement 1**: C cannot receive/send data from/to P which originates while the ACW is in reset mode.

We develop security requirements for registers and signals in the config/control group of assets identified in Step 2. Several CWEs (e.g., 1258, 1266, 1269, and 1271) [12] focus on the failure to properly initialize, set, and clear the contents of security-critical registers/signals (especially on transitions between system states/modes) – the config/control group of requirements dictates what content is appropriate for registers/signals given the ACW's current operation mode.

> **Requirement 2**: The configuration/anomaly registers are cleared and set to their default values while the ACW is actively being reset.

5) Specify the Security Properties: In order to verify a security requirement, it must be manually converted into a formally specified security property that uses explicit values, design signals, and operators to form an evaluable expression. Rather than specifying nearly identical security properties for each design signal that should adhere to a given security requirement, AKER provides a property generation framework that automatically generates these specific properties given a single security property template with placeholder signals and a list of target design signals. For the security requirements relevant to the M AXI group (Requirement 1), the security property templates are primarily information flow tracking IFT properties. IFT properties enable us to tag information from a particular source signal and track it as it flows through our system [20]. For example, Security Requirement 1 can be formalized using a property template that fails if any information originating from C during active reset flows to P.

```
'signal_from_C'      //source
 when (ARESETN == 0) //tagging condition
 =/=>                //no-flow operator
'signal_to_P'        //destination
```

Note that the property involves the no-flow operator (=/=>) which indicates noninterference between the source and destination [21]. Hardware information flow properties are a type of hyperproperty [22] that require specialized verification tools [23]. IFT properties are more challenging to verify than trace properties. Trace properties are stated over a set of traces and are commonly used in functional verification. Hyperproperties are stated over sets of traces and are useful for proving noninterference – a crucial aspect of information flow analysis [24]. For the security requirements relevant to the config/control group (Requirement 2), the security property templates are primarily trace properties that indicate the exact values of a specific signal/register under various conditions. For example, the security requirement is formalized using the following template which fails if the configuration/anomaly registers do not contain their default values after being reset.

```
'reg' == 'dflt_val'
 unless
 (ARESETN != 0 && 'acw_w/r_state' != 2'b00)
```

In total, we develop eighteen security property templates for verifying the security of the ACW, which are expanded to hundreds of individual properties in Step 6. Eleven of these templates are related to information flow and seven templates relate to trace properties.

6) Verify the Security Properties: The final step verifies the properties using off-the-shelf hardware security verification tools. The eighteen security property templates created in Step 4 and the assets identified in Step 5 are used to automatically generate 316 security properties which are comprised of 164 information flow properties and 152 trace properties. The verification uses a configurable AXI DMA module acting as controller C. C is wrapped with an ACW. A top testbench module mimics the behavior of the peripheral P. The testbench iterates through resets and configurations of the ACW and the DMA with the goal of switching the ACW between operative modes to provide adequate coverage of the necessary conditions for all of the security properties. Each property is written as an assertion using the Cycuity Security Rules. We use Cycuity Radix-S [25] to generate a security model from the security rules and the ACW Verilog. The security model is simulated with the testbench. Radix-S reports how many times each individual property assertion fails along with the time at which each failure occurs.

4 AKER FIRMWARE AND SYSTEM INTEGRATION

AKER uses a distributed set of ACWs that locally enforce memory access control policies at each controller. The ACWs require that their local access control policy be set by a trusted entity (TE). The interactions between the ACWs and the trusted entity must be verified and secure. Moreover, the firmware for configuring the ACWs running in the trusted entity plays a crucial role in the system security – a misconfiguration can compromise the SoC's confidentiality, integrity, and availability. A *Hardware Root of Trust* (HWRoT) is a special system resource that performs security-critical tasks required for safe and secure operation. The HWRoT stores and maintains cryptographic keys. It helps authenticate the validity of SoC data and code. And generally ensures the SoC boots and operates in a safe and secure manner. AKER uses the OpenTitan [26] as its trusted entity for configuration and management. The highly-critical operations performed by an HWRoT require security verification. OpenTitan includes an extensive design verification including testbenches, self-checking agents, and assertions. In Section 4.1, we provide a further degree of security to the system, performing a property-based security verification for the integration of the OpenTitan as the AKER trusted entity. Following, in Section 4.2, we present the AKER techniques for analyzing the security of an access control policy provided by a system integrator and automatically generating secure firmware for the configuration and management of the AKER access control system from the HWRoT. Finally, in Section 4.2, we provide a security verification to assess the

system-level interactions between the ACWs and the trusted entity after firmware generation.

4.1 Firmware-level Security Verification

We leverage the same six-step process proposed in Section 3.3 for validating the interactions between the ACWs and the trusted entity. . In this specific case, we are concerned about four entities: a controller C, an ACW (wrapping C), a peripheral P, and the trusted entity TE.

1) Create the Threat Model: the ACW, TE, and P are considered trusted. C is untrusted – its possible ability to communicate with P via the ACW in a manner that does not adhere to the access control policy is a system threat.

2) Identify the Assets: the assets are additional to the assets from the IP level. The identified additional assets are the design signals composing the AXI channels of the ACW's S interface for configuring the ACWs from the trusted entity and the interrupt lines of the ACW connected to the TE.

3) Identify the Potential Weaknesses: seven relevant CWEs have been identified. These expand the potential weaknesses in the config/control group from Section 3.3. Such additional potential weaknesses include the configuration ports of the ACW, which allow the configuration of the $LACP$ and the two interrupt lines of the ACW connected to TE. The CWEs and weaknesses identified in the IP level security verification are still relevant for this scenario. However, as these have already been examined, we focus here on the weaknesses related to the interactions between the ACW and the TE.

Relevant CWEs: 276, 1191, 1193, 1262, 1283, 1290, 1292

4) Define the Security Requirements: the security requirements address the existence and the content of information flows between the TE and the anomaly and configuration registers of the ACW through the ACW S port. As TE is in charge of configuring the $LACP$ of the ACW, it should be the only source of information flows modifying the configuration registers. Also, the anomaly registers metadata of the ACW should not be able to be modified by the TE.

Requirement 3: The configuration/anomaly registers contain the default values until they are modified by the TE (config.) and/or ACW (illegal req. metadata tracking).

Concerning the interrupt lines of the ACW, the security requirements address the signal values. The TE should be alerted by the corresponding ACW whenever an illegal transaction happens, by driving the interrupt line.

Requirement 4: An interrupt to TE is generated after the ACW detects an illegal request.

5) Specify the Security Properties: the requirements relevant to the TE and the configuration and anomaly registers are primarily information flow tracking IFT properties. To make an example, Requirement 3 can be formalized using the following template – such a template fails in case any unauthorized source modifies the configuration registers and anomaly registers after a reset has been triggered.

```
'unauthorized_signal'        //source
 when ('reg' == 'dflt_val') //tagging cond.
 =/=>                        //no-flow op.
'reg'                        //destination
 unless ('reg' == 'dflt_val')
```

The security property templates we specified for the security requirements of the interrupts are trace properties – requirement 4 is formalized using a specification failing if the interrupt line does not assume the proper value given the operative mode of the ACWs, as following described.

```
'INTR_LINE_W/R' == 1
 unless ('acw_w/r_state' != 2'b10)
```

Totally, for the verification of the security in firmware-level interactions between the ACW and the TE we developed four security property templates. Three of such templates are IFT properties. The remaining one is a trace property.

6) Verify the Security Properties: we use the security property templates for the assets identified in Step 2 and crafted in Step 5 to automatically generate 1,438 security properties. Such properties are split into 1,436 IFT properties and 2 trace properties. The setup for verifying such properties resembles the one introduced for the IP level security verification in Section 3.3, but for the additional presence of the trusted entity.

4.2 Secure Firmware Generation

The AKER Secure Firmware Generator (SFG) supports the integration of an AKER access control system into an SoC. The Secure Firmware Generator provides a driver for the ACW, it generates firmware for the secure setup and management of an AKER access control system, and it analyzes the access control policy for data leaks caused by improper configurations. The SFG tool inputs two files: **(a)** the *SysArch* file, defining the system architecture (i.e., controllers, peripherals, and associated address regions) and **(ii)** the *AkerACP* file, defining the SoC access control policy (static or dynamic) – our repository reports instructions and templates for the definition of these files [6].

AKER allows for *static* or *dynamic* access control policy in the AkerACP file. In static scenarios, the policy comprises a single AKER *access control map*, that is, a valid configuration for the AKER access control system defining the read/write access rights of any controller to the peripherals. In a dynamic scenario, multiple AKER access control maps can be defined. Consider a system having N controllers $C = C_1; C_2; \dots; C_N$ and L peripherals $P = P_1; P_2; \dots; P_L$ (as described by Figure 2). To keep a compact notation, we assume that each peripheral has a single address region, that is, a contiguous set of addresses associated with each peripheral (see Section 2.1). The set of the address regions is denoted as $R = R_1; R_2; \dots; R_M$, so that R_1 is the address region associated with P_1 and so on. The tool can manage scenarios where the peripheral requires multiple address ranges having different read/write rights. For the sake of generality, we assume that the interconnect permits connections among all of the controllers and peripherals though the tool can be extended to consider constrained scenarios.

The AKER SFG tool is comprised of three modules: **(i)** the ACW *driver* template, **(ii)** the *firmware generator* tool, and **(iii)** the *data leak analysis* tool. The ACW driver template sketches the functionalities required for the management of an ACW. The template driver is leveraged by the firmware generator tool to automatically generate the firmware code for the secure setup and maintenance of the selected access control policy. For best compatibility, we developed the template

driver in the C language and following the OpenTitan guidelines [26]. The description of the workflow of the SFG tool for static and dynamic policies is reported next.

Static Policy Management: As a first step, the SFG tool parses the static M^S access control map from the AkerACP file. M^S can allow information leaks due to *intra-map* data flows caused by an unintended proxy or intermediary in the system, also known as *confused deputy* (e.g., see CWE 441). In such a scenario, information flow is caused by unintentional data movement operated by a controller. To explain the problem, consider a simple architecture involving two controllers C_1 and C_2 and two peripherals P_1 and P_{prot} (a protected peripheral storing critical data). Consider now the following policy defined in M^S: **(i)** C_1 has write access rights to R_1 and read access rights to the protected region R_{prot}; **(ii)** C_2 has read access rights to R_1, but it is forbidden to access the protected region R_{prot} (for both read and write). Since C_1 has read access to R_{prot} and write access to R_1, C_1 can potentially operate a data flow from R_{prot} to R_1. Once the protected data are stored in R_1, they could be easily accessed by C_2. This would create a security-critical data flow from R_{prot} to C_2 breaking the access control policy defined in M^S. To avoid such scenarios, the AKER SFG tool implements an analyzer able to capture the possible data flows of M^S in a generic architecture. If any flows are discovered during analysis, the AKER SFG tool generates a report for the system integrator containing the source and the destination for each potential flow. The AKER SFG tool also automatically proposes countermeasures to remove the illegal flows. The generic algorithm we developed to find all of the potential illegal data flows is reported below.

Algorithm 1: The algorithm reporting all of the possible data leaks due to intra-map flows.

```
Input: System architecture and static access control policy M^S
Output: Set of possible data leaks due to intra-map flows
for C_i ∈ C do
    for C_j ∈ C, j ≠ i do
        F^S_{i,j} = M^S_{i,W} ∩ M^S_{j,R}
        L^S_{i,j} = M^S_{i,R} \ (M^S_{i,R} ∩ M^S_{j,R})
        if F^S_{i,j} ≠ ∅ and L^S_{i,j} ≠ ∅ then
            for P_K ∈ L^S_{i,j} do
                report P_K ==> C_j through F^S_{i,j}
            end
        else
            no flows for (C_i, C_j)
        end
    end
end
```

At first, the algorithm sets two controllers (C_i and C_j). For such controllers, $M^S_{i,W}$ is the write access control policy for C_i, defining which peripheral can be written by C_i. Dually, $M^S_{j,R}$ is the read access control policy for C_j, defining which peripheral can be read by C_j. Then, the set $F^S_{i,j}$ is computed – this is the set of buffers providing a channel through which illegal data flow can happen. Following, the set $L^S_{i,j}$ is computed. $L^S_{i,j}$ contains the address regions corresponding to peripherals that are readable by C_i but not by C_j – the data in such regions could illegally flow to C_j through the buffers in $F^S_{i,j}$. At this point, if both $F^S_{i,j}$ and $L^S_{i,j}$ are not empty, illegal data flow can happen from each region in $L^S_{i,j}$ to C_j though any region in $F^S_{i,j}$. The sets $F^S_{i,j}$ and $L^S_{i,j}$ are reported to the system integrator, which can prevent illegal flows by applying the countermeasures proposed by the SFG tool in order to make at least one of the sets $F^S_{i,j}$ or $L^S_{i,j}$ empty. However, this step would require modifications to the access control policy. To provide the best flexibility, we leave the final decision on the modification to operate on M^S to the system integrator – they can decide to fully or partially apply the proposed countermeasures solving the illegal flows or accept the risk and keep M^S with no modifications. The algorithm is iterated for each tuple of controllers. Finally, the SFG tool fills the AKER template driver according to the improved definition of M^S approved by the system integrator and generates the firmware for the secure setup of the ACWs in the AKER access control system from the HWRoT.

Dynamic policy management: Following the requirements of modern applications, the AKER framework has been developed to allow the definition of dynamic access control policies. Such policies are particularly handy in systems dealing with multiple operative modes (i.e., normal mode, debug mode, secure boot, etc.), each requiring to enforce different access rights on peripherals. In such a scenario, consider a system having Q operative modes. The access control policy defined by the system integrator comprises Q access control maps $M^D = M_1, M_2, \ldots, M_Q$, each associated with one of the system modes. The HWRoT is in charge of setting up the proper map according to the running operative mode and securely switching the policy at any mode transition of the system.

As explained for the static scenario, also in this case intra-map illegal flows can happen. Thus, the first step performed by the SFG tool is to check and report intra-map flows for each of the maps $\in M$. Again, the system integrator has flexibility in applying the proposed countermeasures. However, dynamic scenarios can generate *inter-map* leaks – such leaks take advantage of the transition from one map to another. To explain this issue, consider a simple scenario (as in the example of the previous section) with two controllers (C_1 and C_2) and two peripherals (P_1 and P_{prot}). Assume that the system has two operative modes, each associated with an access control map (M_1 and M_2, respectively). M_1 defines the following rights: **(i)** C_1 has write access to R_1 (associated with P_1) and read access to R_{prot}, **(ii)** C_2 is forbidden to access either R_1 and R_{prot}, both in read and write. M_2 defines the following rights: **(i)** C_2 has read access to R_1 but it is forbidden to access R_{prot}, both in read and write, **(ii)** C_1 is forbidden to access R_1 and R_{prot}.

By applying the algorithm proposed for static policy, both M_1 and M_2 are exempt from intra-map flows. However, consider the transition between M_1 to M_2. When M_1 is the running access control map, C_1 could read data from R_{prot} and write them to R_1. After, whenever the system mode changes (and thus M_2 is loaded in the access control system), P_1 may still retain data moved from R_{prot} by C_1, which become accessible by C_2 when M_2 is the running policy. This creates an *inter-map* security flow breaking the access control policy. The proposed algorithm to spot all of the possible illegal data leaks generated by inter-map flows in a generic system is following reported. At first, a double loop selects two of the access control maps M_x and M_y – this is to check the illegal inter-map flows in the transition from M_x to M_y. Following, a double loop

Algorithm 2: The algorithm computing the possible data flow due to inter-map data leaks.

Input: System architecture and dynamic access control policy M^D
Output: Set of possible data leaks due to inter-map data leak
for $M_x \in M^D$ **do**
  for $M_y \in M^D; x \neq j$ **do**
    for $C_i \in C$ **do**
      for $C_j \in C; j \neq i$ **do**
        $F_{i,j}^{x->y} = M_{i,W}^{x} \setminus M_{j,R}^{y}$
        $L_{i,j}^{x->y} = M_{i,R}^{x} \cap (M_{i,R}^{x} \setminus M_{j,R}^{y})$
        if $F_{i,j}^{x->y} \neq \emptyset$ *and* $L_{i,j}^{x->y} \neq \emptyset$ **then**
          for $P_K \in L_{i,j}^{x->y}$ **do**
            report $P_K ==> C_j$
              through $F_{i,j}^{x->y}$
          end
        else
          no flows for (C_i, C_j) when $M^x \quad > M^y$
        end
      end
    end
  end
end

sets two controllers C_i and C_j under analysis. For such controllers, $M_{i,W}^{x}$ is the write access control policy for C_i, defining which peripheral can be legally written by C_i according to the map M^x. Dually, $M_{j,R}^{y}$ is the read access control policy for C_j, defining which peripheral can be legally read by C_j according to the map M^y. The set $F_{i,j}^{x \quad >y}$ is computed. This set contains the buffer regions that can allow the illegal data flow while transitioning from M^x to M^y. Following, the set $L_{i,j}^{x \quad >y}$ is populated. This set contains the read regions through which data could be leaked from the regions accessible by C_i in M_1, but not accessible by the C_j either in M_1 and M_2. At this point, if both the sets $F_{i,j}^{x \quad >y}$ and $L_{i,j}^{x \quad >y}$ are not empty, illegal data flow can happen from each region $\in L_{i,j}^{x \quad >y}$ to C_j through each region $\in F_{i,j}^{x \quad >y}$ when transitioning from M_1 to M_2. Again, the system integrator is warned by the AKER SFG tool on the possible illegal data flows and countermeasures. The flows are reported with source and destinations. The system integrator can prevent any illegal flow to happen making at least one of the set $F_{i,j}^{x \quad >y}$ or $L_{i,j}^{x \quad >y}$ empty. The AKER SFG tool provides guidelines to achieve this goal and, for best flexibility, leave the final decision to the system integrator. However, differently from static scenarios, in dynamic scenarios, illegal flows can be avoided using also a different strategy. The transitions among the maps are operated by the HWRoT. Knowing the regions acting as the buffer area for the data leaks to happen among two generic maps M^x and M^y, the HWRoT could avoid any possible data leak by wiping such buffer areas before transitioning to M_y. As reported in the algorithm, such regions are the one in $F_{i,j}^{x \quad >y}$. To this aim, we provide in the AKER SFG tool an optional feature automatically generating the firmware code for wiping the selected buffer areas before transitioning to a new map. It is worth mentioning, however, that such an operation can generate data losses. The area to be wiped are reported to the system integrator, which must carefully evaluate the system requirements. Again, to provide maximum flexibility, we leave the final decision to the system integrator, which decides whether to implement the wiping functionalities, change the access control policy, or accepting the risk generated by intra-map and/or inter-map flows. To consider all possible flows among controllers and transitions among maps, the algorithm is iterated for each tuple of controllers and for each tuple of maps. To make the algorithm more efficient, the system integrator could define a set of possible transitions of the system.

Again, the SFG tool fills the AKER template driver according to the improved definition of M^D and the wiping functionalities (if requested). The firmware is then generated for the secure setup and dynamic management of the ACWs in the AKER access control system from the HWRoT.

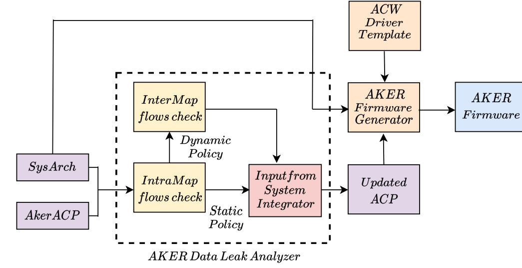

Fig. 5: The AKER Secure Firmware Generator workflow (simplified).

Sample workflow example: We report following an example describing the sample workflow of the SFG tool, summarized in Figure 5. The system integrator inputs the SFG tool with the SysArch and the AkerACP files. The files are analyzed by the AKER data leak analyzer – if the selected policy is static, the tool checks the presence of intra-map flows applying Algorithm 1. If the defined policy is dynamic, besides looking for intra-map flows, the tool checks also the presence of inter-map flows by applying Algorithm 2. The discovered flows are prompted to the system integrator, together with the suggested guidelines to secure the selected access control policy. The tool outputs the updated Access Control Policy considering the determination of the system integrator. The updated policy is taken in input by the AKER firmware generator, together with the ACW Driver Template and the SysArch file, to automatically generate the firmware code for the management of the target AKER access control system. Finally, the SFG tool outputs the AKER firmware, which should be integrated into the firmware code of the TE implementing all of the other functionalities required in the target system.

4.3 System-level security verification

We leverage the proposed verification process for validating the security of the interactions between multiple controllers (each wrapped by an ACW) and the shared resources available in the system-on-chip for firmware configurations generated by the AKER SFG tool. For this scenario, we consider eleven entities: two controllers (C_1 and C_2), two ACWs (ACW_1 and ACW_2), wrapping C_1 and C_2, respectively, three peripherals P_1, P_2, and P_3, the AXI interconnect, and the trusted entity TE (see Figure 2 considering $N = 2$ and $L = 3$). In the considered scenario, the local access control policy setup in ACW_1 ($LACP_1$) states that C_1 can read from all regions of $S_1^R = \{P_1, P_2\}$ and write to all regions of $S_1^W = \{P_1\}$. The $LACP_2$ of ACW_2 states that C_2 can read from all regions of $S_2^R = \{P_3\}$ and write to all regions of $S_2^W = \{P_2, P_3\}$.

1) Create the Threat Model: it is assumed that the ACWs, the peripherals, and the trusted entity are trusted. C_1 and C_2 are considered untrusted. The threat model focuses on the ability of the generic controller C_i to communicate with the generic peripheral P_k (through the ACW) in a manner that does not adhere to the $LACP$.

2) Identify the Assets: the identified assets relevant for this verification level are the AXI signals connecting the generic ACW_i to the generic C_i and all of the signals within the unauthorized regions for each C_i.

3) Identify the Potential Weaknesses: three additional relevant CWEs have been identified. These further expand the potential weaknesses defined in the IP-level and firmware-level verification steps. As this validation is at the system level, the additional weaknesses consider the configuration of the generic local access control policy $LACP_i$ for the generic ACW_i, as it relates the generic C_i sharing resources with the generic C_j.

Relevant CWEs: 441, 1189, 1260

4) Define the Security Requirements: we develop security requirements addressing the existence and the content of information flows between the generic tuple (C_i,P_k), according to $LACP_i$. We consider that $LACP_1$ (for ACW_1) states that no information should flow between C_1 and P_3, while $LACP_2$ (for ACW_2) states that no information should flow between C_2 and P_1.

Requirement 5: Any C cannot receive/send data from/to any region not contained within its ACW's LACP.

5) Specify the Security Properties: in this case, all of the specified security property templates are IFT properties. To make an example, the send aspect of Requirement 5 is formalized by the following template, which fails if any information originating from the generic controller C_i flows to any unauthorized region.

```
'sig_from_C'    //source (always tagged)
 =/=>
'unauthorized'  //destination
```

We develop two IFT security property templates in total for verifying the security of the interactions between the multiple controllers and shared resources in the system.

6) Verify the Security Properties: the assets identified in step 2 and the security property templates created in step 5 are used to generate automatically 76 IFT properties. The verification setup, in this case, builds upon the firmware-level verification proposed in Section 4.1, by adding one additional controller module (wrapped by a corresponding ACW), three memory modules (serving as peripherals), and the AXI interconnect connecting controllers to peripherals.

4.4 Discussion on the requirements

In Section 3.3, 4.1, and 4.3, we provided an overview of the six-step security verification we performed at the IP-level, firmware-level and system-level, respectively. Although each level of verification has a different focus, the overall goal of our security verification effort is to verify that the access control provided by an AKER access control system meets certain security requirements. These security requirements, and their associated security properties, relate to the confidentiality, integrity, isolation, and value of the signals/registers which enable configuration and communication in an AKER access control system. Across the aforementioned verification sections, we illustrate our process using five security requirements and their respective security property templates. However, it is important to note that we define/specify a total of twenty-one security requirements/property templates in our security verification framework in order to adequately cover the potential weaknesses identified at each level of verification. When expanded using our verification framework, these twenty-one security property templates automatically generate a total 1,805 security properties with 1,661 being IFT properties and the remaining 144 being trace properties (see Step 6 of each verification section for more information). If needed, our security verification framework enables system integrators to easily customize or extend the twenty-one default requirements/property templates to address any specific concerns depending on the specific functionalities of their target controllers/system.

5 EXPERIMENTAL EVALUATION

This section presents the experimental evaluation we conducted to evaluate the performance and area usage of the AKER access control system and assess its functional and security correctness. First (Section 5.1), we compare the performance and area usage of an AKER access control system against two popular methods for implementing on-chip access control introduced in Section 2.4. The measurements are conducted on realistic designs deployed on a commercial FPGA SoC platform. Following (Section 5.2), we integrate and validate the proper functionalities of an AKER access control system on the OpenPULP architecture [5] as a case-study. Finally, we leverage the automatic tools proposed in Section 5.3 for generating the firmware code running in the HWRoT for setting up the AKER access control system in the OpenPULP case-study.

5.1 Performance on FPGA SoC platforms

We developed a realistic architecture on a Xilinx Zynq Ultrascale+ FPGA SoC platform. The architecture is composed of three controller hardware accelerators C_1, C_2, and C_3 implemented in the FPGA fabric. C_1, C_2, and C_3 are connected to a Xilinx AXI SmartConnect [27]. The SmartConnect is in turn connected to the shared DRAM memory controller, accessible through an AXI S interface in the FPGA-PS interface [28]. This architecture mirror the one reported in Figure 1 (considering $N = 3$ and $L = 1$). C_1, C_2, and C_3 implement the functionalities of three separated high-performance DMA IPs. This enables easy configuration of the modules and can cover a wide range of communication behaviors. We deployed three different access control systems: in **Design (a)** we use the AXI SmartConnect (INTC) (Section 2.4.1), in **Design (b)** we leverage the Xilinx Memory Protection Unit (XMPU) [15] to implement the access control system in PS (Section 2.4.2), and in **Design (c)** we deploy an AKER access control system involving three ACW modules. In this experiment, the configuration of the modules is performed by

Functionality for policy definition	INTC	XMPU	AKER
Protect a limited set of predefined regions	Yes	Yes	Yes
Dynamic allocation of read/write regions	No	Yes	Yes
Definition of private read/write regions	No	No	Yes
Definition of read-only/write-only regions	No	No	Yes
Secure transactions identification	Yes	No	Yes

TABLE 1: Comparison of SoC access control systems. The INTC and the XMPU access control systems available exhibit limitations not seen in with AKER.

one of the processors of the platform, acting as the Trusted Entity. We deployed a custom, cycle-accurate timer in the FPGA fabric for obtaining high accuracy in the performance evaluations. A Xilinx System ILA [29] is also deployed to verify the proper behavior of the ACW modules. We synthesized the designs using Xilinx Vivado. A major limitation of the access control implemented Design (a) (leveraging the AXI interconnect) is that the access control policy is statically configured at implementation time and thus cannot be dynamically modified. Additionally, the Vivado synthesis tool assigns a predefined number of addressable regions for the controllers – no additional custom regions can be added. Also, such predefined regions have default full read/write permissions – no read-only regions can be defined. The XMPU integrated into the PS and leveraged in Design (b) for the implementation of the access control system allows defining 16 custom memory regions. The XMPU uses the workaround of the AXI ID signals described in Section 2.4.2 for the authentication of the address requests (see [15]). However, the implementation of the AXI SmartConnect used to connect the controllers to the PS does not propagate the AXI ID signals to the PS [27]. This means that, even forcing C_1, C_2, and C_3 to issue requests with unique IDs, the AXI SmartConnect propagates the requests to the PS with the same ID. Thus, the ID information is lost and therefore the XMPU cannot identify the source of the request. From the previous considerations, the XMPU cannot securely enforce any access control policy aiming at differentiating the requests issued by C_1, C_2, and C_3. It follows that the XMPU access control system could not enforce even a simple access control policy defining a private read/write buffer for each of the controllers deployed in the FPGA fabric. Again, in the XMPU the defined regions have default read and write privilege – they cannot be target for defining read-only regions. In Table 1, we summarize some of the essential features for the implementation of common security policies in modern access control systems and the limitations found in the available solutions. Following, we develop a simple access control policy compatible with the limitations of the access control systems of designs (a) and (b) to compare the performance and resource consumption of the systems.

5.1.1 Response times in isolation and under contention

This first experiment compares the performance impact of the three access control systems on the memory access time associated with C_1, C_2, and C_3. We setup a common forbidden region F in memory. This means that any controller C is forbidden to read or write to that region. The memory access times are evaluated in isolation and under contention. Figure 6(i) reports the memory access time in isolation. We activated C_1, C_2, and C_3 separately to access different amounts of data in a legal region of the memory. The results show similar performance in latency and throughput in all of the designs. This confirms that the impact on the performance of the one extra clock cycle introduced by an AKER access control system is negligible. In the next experiment, we test the contention generated

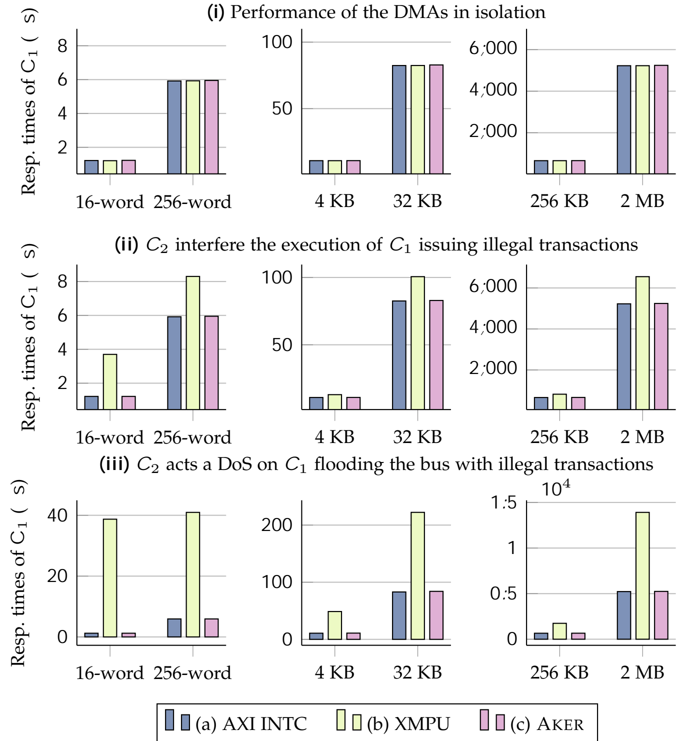

Fig. 6: Performance Evaluation: (i): the performance are similar in isolation. (ii): the performance differ in the presence of illegal requests. (iii): C_2 can perform a DoS attack to endanger the availability of the DRAM memory from C_1 in (b).

by illegal transactions on the three designs. The same configuration from the previous experiment is kept for C_1. We configure C_2 to attempt to concurrently access the forbidden region F (i.e., issuing illegal requests). C_2 is configured to model a controller stuck in trying to access the forbidden region due to a misconfiguration – C_2 issues a new illegal request once the access control system replies to the previous one with an error. As discussed in Section 2.4.2, the access control systems implemented at the peripheral (Design (b)) require to occupy time on the interconnect to abort the illegal requests. This impacts the performance of legal transactions. Figure 6(ii) reports the measured average access time for C_1. The results show that in Design (b) the interference generated by the illegal transactions issued by C_2 impacts the performance of C_1. In particular, the average measured response time increases by 203% on a 16-word transaction, changing from 1.22 s in Designs (a) and (c) to 3.7 s in Design (c). A lower impact has been measured for longer, consecutive accesses. However, we measured an impact of at least 20% on the average response times in all of the scenarios. Differently, Design (a) and (c) stop the illegal transactions of C_2 before entering the network. These results confirm how an AKER access control system is able to provide high performance (stopping the interference generated by any illegal transaction) while providing flexibility in the definition of the access control policy. In the third experiment, we test a denial of service scenario. We kept the same configuration of the previous experiment for C_1,

while we configure C_2 to flood the interconnect of illegal requests, leveraging the full throughput made available to C_2 by the AXI SmartConnect. In this case, C_2 mimics the behavior of a misconfigured or malicious high-throughput IP core. Figure 6(iii) reports the measured results, showing that the impact on the response times of C_1 in Design **(b)** is way higher than in experiment (ii). In particular, the measured average response time of a 16-word transaction issued by C_1 passes from 1.22 s of Designs **(a)** and **(c)** to 38.72 s in Design **(b)**. This corresponds to an increase of 3074%. As in the previous experiment, the impact decreases on longer and consecutive accesses. Nevertheless, in all of the tested cases, the impact is at least 165% on the nominal average response times. In other words, the response time of C_1 is more than double with respect to nominal conditions in all of the scenarios. This experiment shows how a misbehaving IP can create a denial of service when using Design **(b)** for access control. This issue can be critical in designs integrating software-configurable IPs – malicious software could exploit this issue to act Denial-of-Service of the memory or other resources to the other IPs integrated into the system. Indeed, even if detected at runtime, the access control system implemented in Design **(b)** does not provide any method to stop the flood of illegal transactions – according to the AXI standard [8], transactions cannot be aborted after being propagated. Thus, when the access control system is enforced at the peripherals (as in the case of the XMPU), any detected illegal request must be completed propagating faulty data to the requesting controller to keep the interconnect operational.

5.1.2 Resource consumption

Resources	PULP	4-regs ACW	16-regs ACW	PULP+ACWs
LUT	156937 (57%)	326 (0.1%)	730 (0.3%)	158421 (58%)
FF	53354 (10%)	358 (0.1%)	744 (0.1%)	54854 (10%)

TABLE 2: Resource consumption of the ACW module. The area impact of the ACW can be target according to the requirements of the target application.

Table 2 reports the resource consumption and occupation percentage on a Xilinx ZYNQ Ultrascale+ platform for: **(i)** the whole PULP SoC platform (Section 5.2), **(ii)** ACW deploying 4 regions (4-regs ACW), **(iii)** ACW deploying 16 regions (16-regs ACW), and **(iv)** two 16-region ACWs integrated in the PULP SoC (PULP+ACWs). The impact of the ACW modules on resource consumption is limited with respect to the whole cost of the PULP SoC – the resource cost of PULP+ACWs is around 1% higher with respect to PULP. Furthermore, the resource consumption of the ACW can be optimized to meet the requirements of a target application.

5.2 PULP SoC Experiments

The PULP platform (Parallel Ultra-Low-Power) is an open-source computing platform comprising a multicore RISC-V processor. PULP is structured in two domains: an SoC domain and a Cluster domain – the SoC domain is in charge of performing the control and other high-level functions. The cluster leverages its eight RISC-V cores to perform hardware acceleration. The cluster and the SoC communicate through two communication pathways. One pathway enables the Cluster to access the SoC (i.e., the SoC is a Peripheral for the Cluster, which is the controller). Dually, the other pathway enables the SoC to access the Cluster (i.e., the Cluster is a Peripheral for the SoC, which is the controller). The communication pathways permit the fabric controller core (in the SoC) and the cores in the Cluster domain to exchange information and access a shared L2 memory. Additionally, the PULP's memory map defines memory regions for the Cluster subsystem, the ROM memory, the SoC peripherals subsystem, and the L2 memory. Such regions are fully addressable from any of the PULP's controllers (i.e., the fabric controller and the eight RISC-V cores). In the PULP platform, no default access control system is deployed for enforcing an access control policy on the transactions exchanged between the Cluster and the SoC. To provide such functionalities, we integrated an Aker access control system regulating the communication between the SoC domain and the cluster domain via two ACWs. Both of the pathways implement the AXI standard for data exchange. Thus, the process of wrapping their respective AXI M with the ACWs is straightforward. Once the integration is completed, we setup the Aker access control system to enforce an access control policy on the two pathways. The functionalities of the Aker access control system have been validated testing various ACW configurations through the official test simulations provided with the PULP platform. The test simulations run C programs on the fabric controller embedded in the SoC. Notably, some of these C programs make use of the PULP Cluster while others do not. As our baseline, we ran all of the test simulations on the default PULP platform, without the Aker access control system, in order to check their proper functionalities. Following, we configured the Aker access control system to allow any read/write transaction and verify that all of the tests run successfully as with the baseline. Thus, we configured the Aker access control system to block all read/write requests. As expected, we observed that the tests not making use of the PULP cluster run successfully, while the tests accessing the cluster hang waiting on responses from the decoupled domains. Finally, we setup a fine-grained access control policy to ensure that the Aker access control system is able to enforce realistic access control policies, different from the all-or-nothing approach leveraged in the initial validations. We verified the results of each configuration analyzing the testbench output logs and vcd/waveforms of the platforms (see Figure 7). The designs and testing frameworks are available in our repository [6]. In this experiment, we manually wrote the firmware setting up the Aker access control system. In the next section, we leverage the SFG tool proposed in Section 4.2 for the automatic generation of firmware code.

5.3 Secure firmware generation

In this section, we demonstrate the functionality of the Aker SFG tool by analyzing and fixing a dynamic access control policy for the PULP using the SFG tool. After this, we use the SFG tool to generate secure firmware and experimentally validate that this firmware works as expected on the PULP. It should be noted that the PULP's Aker access control system has two controllers (i.e., the Cluster and the SoC) and many peripherals (e.g., shared L2 memory, ROM, I/O interfaces, etc.) but, for the sake of clarity, we will only

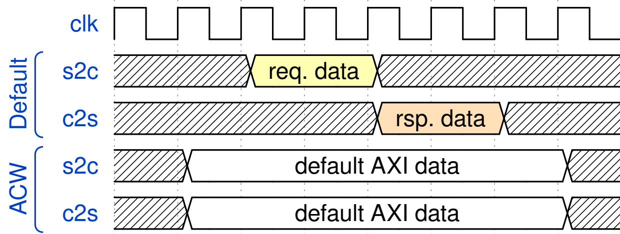

Fig. 7: s2c is the SoC to Cluster pathway. c2s is the Cluster to SoC pathway. The default group shows a portion of the execution of a C program using the the default OpenPulp. The ACW group shows the same portion of the execution with the addition of two ACWs configured to block all transactions.

refer to two of these peripherals in this section. The dynamic access control policy we implement builds around two operative modes we have added to the PULP: no_cluster mode and limited_cluster mode. no_cluster mode (associated with the AKER map M_1) defines the following rights: **(i)** *SoC* has read and write access to R_1 (associated with shared L2 memory) and read access to R_2 (associated with ROM), **(ii)** *Cluster* is forbidden to access either R_1 and R_2, both in read and write. This mode of operation is intended to be used to turn the shared L2 memory into a "private" memory for the SoC domain and thus improve its data integrity and confidentiality. limited_cluster mode (associated with AKER map M_2) defines the following rights: **(i)** *Cluster* has read and write access to R_1 but it is forbidden to access R_2, both in read and write, **(ii)** *SoC* has read access to R_1 and read access to R_2. This mode of operation is intended to be used to leverage the Cluster for hardware acceleration but in a limited capacity (i.e., the Cluster can only read from or write to the shared L2 memory). The pseudo-code of the firmware generated by the SFG tool is reported next.

```
...
void setup_M1() {
  //Configure ACW for SoC
  dif_ACW_init(ACW_for_SoC);
  ...
  //Configure ACW for Cluster
  dif_ACW_init(ACW_for_Cluster);
  ...
}
void setup_M2() {
  //Wipe region R1 for inter-map flow
  dif_wipe_region(R_1)

  //Configure ACW for SoC
  dif_ACW_init(ACW_for_SoC);
  ...
  //Configure ACW for Cluster
  dif_ACW_init(ACW_for_Cluster);
  ...
}
...
```

When the system architecture and the access control maps (i.e., M_1 and M_2) are passed to the AKER SFG tool, the tool reports that M_1 and M_2 are exempt from intra-map flows but the transition from M_1 (no_cluster mode) to M_2 (limited_cluster mode) has the possibility for an inter-map flow. Assuming R_1 is never cleared, the inter-map flow could happen if *SoC* reads data from R_2 and writes it to R_1 (during no_cluster mode), and then *Cluster* reads from R_1 (during limited_cluster mode). To deal with the inter-map flow, we follow the AKER SFG tool's proposed countermeasure which involves wiping R_1 before enabling access control policy M_2. We added this countermeasure to the generated secure firmware. The firmware is loaded into OpenTitan to program the ACWs in PULP and follow the same verification process described in 5.2 in order to check the correctness of the configuration done by the firmware.

6 Related Works

Multiple solutions have been proposed for enforcing access control in Network on Chip (NoC) architectures. Grammatikakis et al. proposed a NoC distributed firewall enforcing access control [30]. Fiorin et al. proposed multiple integration methodologies on NoC architectures [31]. Wassel et al. proposed a method to isolate users and effectively time-multiplex the access to the in the NoC [32]. Sepulveda et al. [33] proposed a methodology specifically aimed at the property-driven security verification of NoC routers. While AKER did not specifically focus on NoC architectures, our methodology could be extended to NoC architecture with some modifications. The research community spent considerable efforts to strengthen the security in shared bus architectures. Jacob et al. [34] showed how hardware vulnerabilities related to access control can be injected in real systems during the integration of third-party IP modules. A brief discussion of prevention techniques are provided without providing any specific solution. Restuccia et al. proposed solutions to enforce the security and the safety of AXI-based architectures, including an hypervisor-level interconnect for dynamic controllers management [35] and two methodologies preventing denial of service generated by misbehaving controllers [36], [37]. Tan et al. [38] and Siddiqui et al. [13] proposed two solutions implementing decentralized systems for the detection of anomalous conditions in hardware modules. Such solutions can help mitigating misbehaving conditions caused by hardware module. However, they do not implement dynamic access control systems functionalities. Oberg et al. [39] proposed a methodology for security verification of USB and I2C busses using information flow tracking on a time-division access control scheme. Huffmire et al. [40], [41] and Brunel et al. [42] described two mechanisms for securing off-chip memories. Cotret et al. [43] described a module for implementing a distributed firewall. This last has high performances impact (18% increase in latency), it lacks security verification, and lacks integration with modern HWRoTs for the management of secure configuration.

7 Conclusion

In this paper, we proposed the AKER framework. AKER builds upon three pillars: **(i)** the Access Control Wrapper (ACW), a universal module integrating with on-chip controllers, **(ii)** the AKER Secure Firmware generator tool, analyzing a selected access control policy and generating secure firmware, and **(iii)** the AKER extensive property-driven security verification, assessing the secure operation of an AKER access control system. We integrated an AKER access control system in real design and showed its limited impact on performance and resource consumption. The AKER framework is released in our open-source repository [6].