COSSEA: Context-based SoC Security Enforcement Architecture

Carsten Heinz, Andreas Koch
Embedded Systems and Applications Group
TU Darmstadt
Darmstadt, Germany
{heinz, koch}@esa.tu-darmstadt.de

***Abstract*—Modern embedded System-on-Chips (SoCs) increasingly rely on third-party Intellectual Property (IP) blocks to realize complex designs while meeting cost and time budgets. However, this IP integration from many vendors can adversely affect the SoC's security when even a single IP originates from a malicious actor and then threatens the integrity of the *entire* SoC.**

In this work, we propose COSSEA, a global, context-based security architecture, to control communication on the SoC at a fine-grained level. COSSEA isolates IP from the central system interconnect with different access permissions based on dynamically adaptable context descriptions. A global state-machine coordinates the transition between these security contexts. COSSEA can scale to thousands of contexts using features such as a parametrizable policy directory.

At run-time, the entire security architecture is managed *independently* of any CPU or peripheral, reducing the attack surface. COSSEA is already set up to be securely controlled by a hardware root-of-trust using authenticated and encrypted messages.

We evaluate COSSEA for both FPGA and ASIC and observe hardware overheads of less than 2% for small SoCs typical of MCU-level embedded systems.

***Index Terms*—Hardware Security, Embedded Systems, System-on-Chip, ASIC, IP Blocks**

I. Introduction

In our increasingly interconnected world, embedded systems have become omnipresent in various applications. From consumer devices to safety-critical industrial or automotive systems, System-on-Chip (SoC) architectures have become essential.

At the same time, the widespread adoption increases the potential attack surface and the possible impact of exploiting such systems. Therefore, safeguarding the integrity of SoCs against malicious threats has never been more pressing.

This paper presents COSSEA, a context-based hardware security layer for small embedded SoCs that allows fine-grained isolation of peripheral components within the SoC. As a separate hardware component, it provides reliable security guarantees independent of the firmware executed on the SoC's CPU. In combination with a real-time operating system (RTOS), COSSEA can even provide isolation between tasks.

This research was funded by the German Federal Ministry for Education and Research (BMBF) in project 16ME0233 (VE-Jupiter). The authors would like to thank Markus Scheck and Yannick Lavan for providing their Bluespec System Verilog wrapper for Synopsys SRAM macros.

II. Threat Model

In this work, we assume that the communication network of the SoC is trustworthy. Furthermore, we assume the COSSEA security infrastructure is not manipulated during manufacturing.

The first attack scenario involves an untrusted, malicious IP with bus-master access to the communication network integrated into an SoC. This IP could potentially exfiltrate and manipulate data from memory. Furthermore, it could instrument another IP, such as a DMA engine, to access memory on its behalf.

In the second attack scenario, a malicious task within the RTOS could access peripherals and memory intended only for other tasks in a small embedded system. Even an MPU would not protect against this scenario if tasks are allowed to use DMA engines.

III. Related Work

Small embedded systems typically employ *Memory Protection Units* (MPU) to restrict tasks running on the CPU to access regions in the physical memory. The MPU resides in the CPU and, thus, does *not* provide any protection from peripherals.

In the open instruction set architecture RISC-V, such an MPU is specified as a *physical memory protection* (PMP) [1]. A draft specification proposal exists to extend the PMP to cover the entire SoC, called *IOPMP*, which has already been integrated into SoCs [2], [3]. While these systems protect the central interconnect from malicious peripherals, the CPU configures the IOPMP's access rules. In COSSEA, the hardware module is completely separate from the CPU, making it resilient against privilege escalation attacks in the firmware. Furthermore, COSSEA reduces the attack surface by splitting permissions into a sequence of smaller contexts. To realize this with an IOPMP, the *CPU* would have to update the configuration from software, changing one permission entry at a time and thus carrying a high latency overhead.

A commercial solution for embedded Arm Cortex-M SoCs is Arm TrustZone [4]. In this system, peripherals are divided into different security levels. Peripherals within the same security level can access each other and its associated memory regions. However, this approach does not protect against a malicious peripheral within a security level.

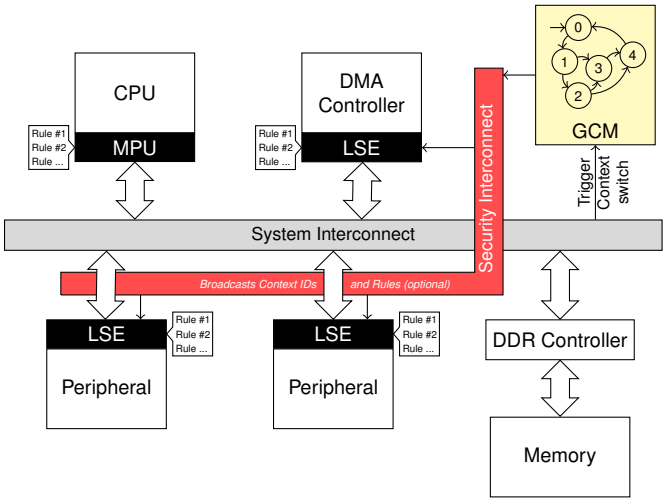

Fig. 1. Sample SoC with COSSEA security overlay.

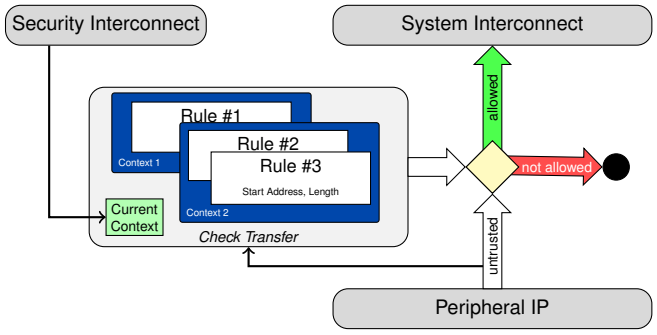

Fig. 2. Internal architecture of a Local Security Enforcement unit, based on [6]. The current context enables the rules contained in the corresponding context.

Other, more fine-grained security solutions have been presented. A multi-level security system [5] is implemented on a Zynq FPGA and IP Firewalls enforce access permissions. Similar to COSSEA, a central policy server contains the firewall rules. DD-MPU [6], as a decentralized security architecture, has IP firewalls that monitor the associated peripherals' configuration interface. Based on the extracted information, the rules in the firewall are updated. This allows for fast updates of rules but lacks any global coordination. SECA [7] is a security architecture that utilizes finite-state machines to enforce the correct behavior of applications. While SECA relies on fixed FSMs known at design time, COSSEA extends the concept and allows *runtime* configurable state machines.

IV. COSSEA Overview

To achieve security guarantees against the threat model, COSSEA adds a security overlay to the existing SoC communication infrastructure. A Local Security Enforcement (LSE) unit is added to each bus master of the untrusted IPs. In addition, a Global Context Management (GCM) unit coordinates the current context of LSE units.

The LSE units prevent prohibited transfers from (potentially malicious) IPs from entering the SoC's communication infrastructure. Thus, the communication infrastructure transports only transfers allowed within the current security context.

A. Local Security Enforcement (LSE)

The LSE is responsible for protecting the communication infrastructure of an SoC from malicious IPs. For this, it monitors transfers on the bus-master interface of the IP and can block forbidden transfers. The decision is based on a rule set stored inside the LSE. This rule set contains a list of address ranges that are allowed to be accessed or modified by this IP. Each rule has a context identifier associated with it. Only rules matching the current active context identifier are enabled.

An LSE protects the SoC against malicious IP at a fine-grained level. However, an LSE alone does not have a global view across the entire SoC, and thus, it is hard to coordinate multiple LSEs to allow, for example, access to a memory region shared between multiple IPs.

B. Global Context Management (GCM)

The global context management configures the currently active context identifier at the LSEs. A finite state machine specifies the allowed transitions between context identifiers. This approach allows various control flow sequences and *security automata* to be modeled and enforced. Furthermore, it can extend permissions related to execution modes or task switches on the CPU core across the entire SoC.

The transition to the next state is initiated by the application running on the CPU, e.g., as part of the RTOS. The state machine only allows approved transitions, but still, an attacker could try to benefit from switching to consecutive contexts. Since the permissions of individual contexts are minimal, exploiting this is more difficult than without COSSEA. As the protection of COSSEA is targeted towards malicious peripherals, we only allow context switches originating from the CPU. Protection against a malicious firmware can be achieved by other means, e.g., an MPU can further restrict the firmware to only allow context switches by the *machine* mode. Furthermore, control-flow integrity of a firmware running on the CPU can be enforced by an hardware monitor, such as DExIE [8].

Example: In an SoC with a DMA controller and two peripherals A and B, which both read data provided by the DMA controller, we can use different contexts to isolate memory. The first context permits memory regions reserved for peripheral A and a second context for the memory regions reserved for peripheral B. In this way, the DMA controller can only write to the memory allocated for the relevant peripheral. An attacker thus cannot misuse the DMA engine to leak data between the peripherals.

V. Implementation

A. Local Security Enforcement (LSE)

The LSE monitors and intercepts transfers between an IP and the central communication infrastructure. It also provides communication interface endpoints for the IP and the main

infrastructure. For allowed communication transfers, the LSE forwards transfers transparently. If a transfer is not allowed, the LSE prohibits it, so it is not passed on to the communication infrastructure.

The LSE stores a list of allowed transfer rules. Each rule consists of a context identifier, base and end address of the allowed memory range, and access type (read, write, or read/write). This information is stored in registers so that it can be modified during runtime. When the peripheral initiates a transfer, the LSE checks the transfer against the rules by comparing the transfer in parallel against all stored rules. When at least one rule of the current context matches, the transfer is allowed to access the system interconnect. The number of allowed rules is fixed at compile time. As each rule requires comparators at the width of the entire address, e.g., 32 bit, scaling the number of rules makes the design more hardware intensive. However, since we rely on multiple contexts to hold an application's entire ruleset, the number of rules in each context is relatively small, reducing the hardware cost of the comparators. To not restrict the LSE to a single usage scenario, the content of each rule can be configured during runtime. As this can introduce new security issues, we utilize a separate interconnect for configuration (see Section V-C).

B. Global Context Management (GCM)

The GCM coordinates the context transitions through a finite-state machine (FSM). Instead of an FSM fixed at design time (as in [7]), which would only support a single, specific application, COSSEA aims for wide applicability and implements a memory-backed state machine. We store the FSM as a transition table: The context identifier is used as the word address of the memory. Each word in the memory contains the context identifiers of succeeding states according to the FSM. As the context identifier addresses the memory, the bit-width of the identifier depends on the number of states. The GCM can be loaded with custom FSMs adapted for individual applications. At design time, only the maximum size of the FSM needs to be specified. We restrict each state to two transitions to reduce the required data for each state to be stored. Each entry thus has a data width of twice the context identifier. If more transitions are required, the FSM needs to be transformed into binary transitions. This requires additional state transitions but results in a more compact transition table.

A transition to the next state is signaled by an external signal, which also contains the binary value for selecting the next state. In our current implementation, this is controlled by the CPU. If the SoC has another trust anchor, such as a control-flow enforcement unit, it could easily be used for the state transitions in COSSEA.

After reading out the next context identifier, the GCM utilizes the security interconnect to broadcast the identifier.

C. Security Interconnect

We introduce an additional interconnect for the communication between the components of COSSEA. It is separated from the existing communication infrastructure of the SoC to reduce the attack surface.

The security interconnect serves the purpose of propagating the current context identifier and raising security exceptions.

With this limited function set, the implementation is very lightweight. The context identifier can be broadcasted by a 1-to-n interconnect. In a system with an 8 bit context identifier and a valid signal, only 9 signals must be routed from the GCM to the LSEs. Each LSE has a signal wire connected to the GCM to raise an exception.

Overall, the required number of signals is low compared to a typical system interconnect with 32 bit or 64 bit address and data channels.

The same security interconnect infrastructure can be utilized to initialize the rules stored in the LSEs at startup. Initialization is handled either by the SoC's core or by a root-of-trust (RoT) module for enhanced security. During system startup, the timing is not essential; thus, we can serialize the rules over the narrow interconnect.

D. Central Policy Directory (CPD)

The initial concept of COSSEA stores all rules locally in the LSEs. This leads to scaling limitations as more rules in an LSE require more resources and comparators. As the LSEs are connected to the communication infrastructure of the SoC, LSEs with a large number of rules can negatively impact the achievable frequency. In our evaluation, we have observed a negative impact on frequency with more than 16 rules. As a countermeasure, the LSE can also be configured to operate in a pipelined mode with an additional clock cycle, relaxing the critical path but adding latency to all memory transfers.

Thus, COSSEA has the option for a central policy directory in the GCM. The policies for all LSEs are stored in an SRAM inside the CPD. Only rules of the current context are loaded on-demand into the LSE. The transfer of rules utilizes the *Security Interconnect* with an increased data width. Instead of just transmitting the context identifier, it now transmits entire rules. A transmitted message now contains the rule with start and end address, context identifier, and destination LSE. For a system with 32 bit addresses, this results in a message size of around 80 bit, depending on the size of the context identifier and number of LSEs. The current implementation transmits the entire message in a single beat. For an improved routing, this could also be divided into multiple beats. As all LSEs receive the new context identifier at the same time, we have an atomic switch to the next context. In system interconnects without bursts, such as the tightly-coupled data memory protocol in PULPissimo evaluated in Section VI, the updated rule-set, activated by a context switch, is also immediately used. For other, burst based systems, active transfers complete under the old rules, before the new rule-set takes over.

For a context switch, an additional delay occurs when loading the rules first at the time the context switch is requested. To counteract the delay, COSSEA employs prefetching of rules. The GCM knows the next transitions and can push rules for the next context to the LSEs. Prefetching starts as soon as

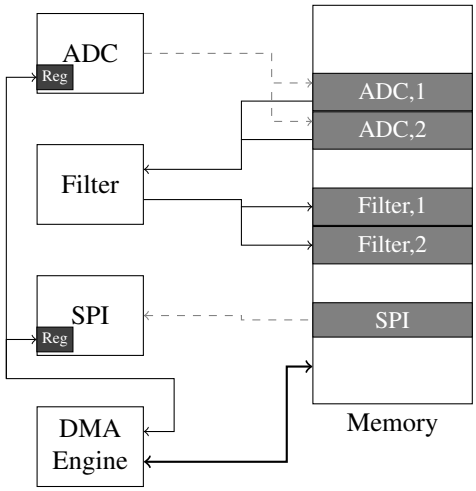

Fig. 3. The memory map of the SoC with double buffering. Dashed lines show the logical accesses performed by the DMA engine.

all rules for the current context are available at the LSEs. As the rules contain the context identifier, they are first activated when the actual context switch occurs.

VI. Case Study

In this paper, we utilize COSSEA to improve the security of two systems. Both are small embedded systems with a small footprint and only physical memory addressing.

Our system is based on the PULPissimo [9], configured with the Ibex RISC-V core. The peripherals with bus-master access to the system interconnect are inserted into the SoC as a HWPE, the existing mechanism in PULPissimo to add hardware acceleration units. To provide the HWPE bus access to the entire address range instead of just the memory, the HWPE is in our setup and connected to the interconnect in the same way as the RISC-V core.

A. Protection against Malicious Peripheral IP

The first scenario resembles the usage of a typical microcontroller. The CPU core is accompanied by several peripheral components, such as an ADC, an SPI master, and a DMA controller. The peripherals and the CPU can send requests to the DMA engine.

Our embedded system acquires analog data, filters it, and sends the processed measurements via SPI. Thus, the SoC contains an analog-to-digital converter (ADC), a hardware accelerator to perform the filter operation, an SPI master module, and a DMA engine. The DMA engine can perform memory operations between memory and peripherals and is shared between the peripherals. The ADC can send requests to the DMA engine, e.g., when sufficient samples have been captured.

Our application's execution flow is as follows: The ADC periodically instructs the DMA engine to store measurements alternately in two buffers in the memory (double buffering). The filter accelerator reads from the fully written buffer, performs the filter operation, and stores the data alternately in two other buffers. Further processing occurs on the CPU, which produces the final results of the application. The DMA engine is then requested to move the results to the SPI master peripheral. Finally, the results are available on the SPI bus. Figure 3 shows a sample memory map for such a system.

A malicious peripheral or software could now instruct the DMA engine to read from a different memory location, which may contain private security credentials, and store it in the buffer intended for the SPI transfers. This way, the attacker could exfiltrate secret data. Other attacks, such as data manipulation, are also possible.

Typical SoC protection mechanisms do not provide sufficient isolation in such a scenario. Physical memory protection (PMP / MPU) would block only requests from the CPU to the DMA controller, not the actual addresses of the DMA requests. In the more platform-wide concept of TrustZone, groups of peripherals can have different permission levels, but peripherals in the same group are not isolated. In order to access the same memory, the ADC and Filter peripherals would need to be in the same group, resulting in no isolation of the double-buffering. Even more, the ADC could directly write to the output buffer of the filter accelerator and inject manipulated data to the CPU.

With COSSEA, a protection scheme could be implemented with the contexts shown in Figure 4. Double buffering is divided into separate contexts to ensure that only one peripheral can access a buffer simultaneously. It would also be possible to merge context 1 with 2 and context 4 with 5. This would improve the parallelism, but it no longer enforces the sequential execution of the units, as it no longer guarantees that the CPU executes after the filter has finished its operation. As a possible side effect, parallel memory accesses of both CPU and filter accelerator could occur, resulting in memory congestion and longer execution times.

Due to the use of double buffering, reading the buffer before the previous component has written to it is not an issue here. The important part, however, is that the different permissions of the DMA engine are not combined in the same context. Otherwise, a malicious actor could instruct the DMA engine to leak data between the isolated memory regions.

B. RTOS Task Isolation

Our second scenario is a small embedded SoC running a real-time operating system (RTOS). Due to the SoC's embedded nature, it has no virtual addresses and only limited privilege levels. In the RISC-V world, this relates to a core with *machine* and *user* mode and a memory protection unit (MPU).

Each task of the RTOS runs at *user* mode, whereas task scheduling by the RTOS is implemented with a trap handler running in *machine* mode. With the MPU, the RTOS can restrict memory access and provide separate address ranges for each task. However, without an IOMMU, applying those restrictions to peripherals is impossible. As long as a task requires access to a peripheral, this peripheral has access to the entire memory region.

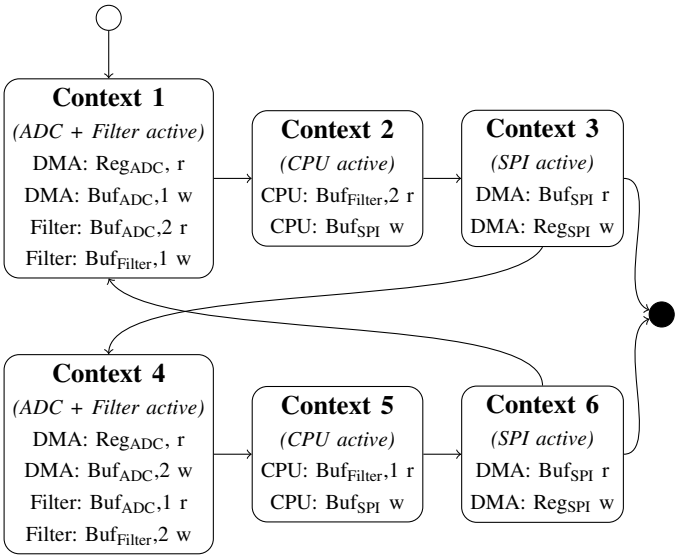

Fig. 4. FSM for the COSSEA contexts of the example embedded application. Dummy predecessor contexts that just handle the initialization of the double buffering are omitted to simplify the figure. The circles symbolize entry and exit points to the remaining parts of the application.

In the first configuration, each task has a COSSEA context associated with it. This way, a task can access peripherals and offload operations, e.g., to a DMA controller. COSSEA now ensures that even the peripherals can only access memory ranges that are allowed for this task. The RTOS trap handler triggers the context switches in the GCM and the configuration changes in the MPU.

In a more integrated configuration, a task can have several contexts with reduced permissions instead of just a single context. This way, more complex tasks can be further restricted for different segments of their execution. The transition to the next context must be initiated by the RTOS task itself so it does not protect against malicious tasks. However, it reduces the surface of attacks for code injection attacks and malicious IPs within the SoC. The task invokes a trap for the context transition so the RTOS can communicate with the GCM at the increased privilege level. To prevent the RTOS task from entering a context intended for another task, we do not allow a task to select a transition and only proceed sequentially in the FSM. In addition to the sequence of contexts intended for a task, each context in the FSM also has a second, *exit* transition to the following global context. When the RTOS scheduler runs, it can use this transition to enter a context outside the task securely. An example is shown in Figure 5.

VII. Evaluation

A. Hardware Overhead

We first evaluate the additional hardware overhead incurred by COSSEA with hardware synthesis for an FPGA and an ASIC target. For both technologies, the first case (*GCM-only*) implements COSSEA with a GCM and LSEs with 16 rules each. The second case (*GCM+CPD*) includes, in addition to the GCM, the CPD.

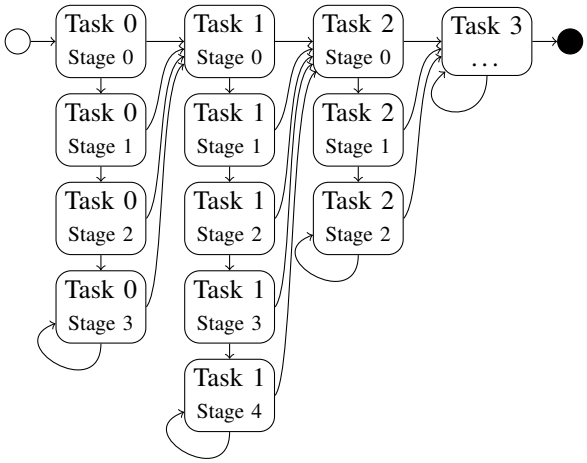

Fig. 5. FSM example for RTOS tasks divided into stages for fine-grain control within a task. Each context has an *exit* transition to proceed to the next task.

For both cases, we define different presets, which define the maximum number of GCM states and CPD rules. The *Small* preset allows 256 states and 256 rules, *Medium* allows 2048 states and 2048 rules, and *Large* allows 8192 states and 4096 rules.

The FPGA target is synthesized with Vivado 2020.2 for the ZCU102 board, which contains a Zynq UltraScale+ MPSoC. The results from Table I show the hardware overhead in terms of LUTs and Block RAM.

For evaluating COSSEA on an ASIC technology, we chose a GlobalFoundries 22 nm FDX process with a flow based on the Cadence reference flow. A Synopsys memory compiler is used to generate the required SRAM macros. Table I contains the required logic and routing area converted into kGE.

The resource overhead of the *GCM-only* setup on FPGA is below 6 %, which can be considered as insignificant. Only the *Large* preset on ASIC results in an overhead of over 10 %. In the ASIC synthesis, we can see that the main resource utilization is coming from the SRAM macros, and thus, the large size of the FSM memory is more visible in the results. On FPGA, even other parts of the SoC occupy Block RAM, so we cannot directly compare the overhead between RAMB36 and ASIC.

GCM+CPD incurs a much larger overhead. This is mainly due to an increase in the required memory. In contrast to *CPD-only*, where each LSE can only store 16 rules, we can store up to 4096 additional rules (in the *Large* preset) in the CPD. The memory in the CPD utilizes Block RAM (or SRAM macros on ASIC) and is thus more efficient than the register-based implementation in the LSE. For addressing the memory, the CPD requires additional address pointers in the FSM of the GCM, which approximately doubles the required size of the FSM storage. This is in addition to the actual storage required to store the rules in the CPD. Furthermore, the security interconnect is more complex. It now transmits the rules and not just the context identifiers.

TABLE I
HARDWARE OVERHEAD OF DIFFERENT COSSEA CONFIGURATIONS IN FPGA AND ASIC TECHNOLOGY. THE RELATIVE OVERHEAD IS COMPARED TO THE ENTIRE PULPISSMO SOC.

	Preset		LUTs	RAMB36	kGE
GCM-only	Small	FPGA	673 (1.54 %)	0 (0 %)	
		ASIC			21.4 (1.38 %)
	Medium	FPGA	690 (1.58 %)	1 (0.69 %)	
		ASIC			40.7 (2.63 %)
	Large	FPGA	721 (1.65 %)	8 (5.56 %)	
		ASIC			168.2 (10.86 %)
GCM+CPD	Small	FPGA	1851 (4.23 %)	3 (2.08 %)	
		ASIC			149.0 (9.62 %)
	Medium	FPGA	2099 (4.79 %)	10 (6.95 %)	
		ASIC			314.8 (20.32 %)
	Large	FPGA	2021 (4.62 %)	27 (18.75 %)	
		ASIC			659.6 (42.58 %)

Interestingly, the *Large* preset consumes fewer LUTs on the FPGA. This is probably a result of the non-deterministic behavior of the EDA tool. The design size for *Large* mainly increases in memory; the main difference in the logic is the increase of the context identifier by two bits to address the larger address space.

B. Latency Analysis

In the *GCM-only* scenario, we measure a latency of just four clock cycles for a context change. The latency includes the AXI request to the GCM, evaluation of the FSM, broadcasting the context, and finally activating the new rules within the LSEs.

The *GCM+CPD* scenario requires a more detailed analysis. When the rules have been prefetched, we measure a latency of six clock cycles for the context change. In contrast to the *GCM-only* setup, this is an increase of two cycles because we implement the context change in the CPD and incur additional communication and Block RAM latency. Contrary to typical prefetchers, which operate on the assumption of, e.g., a taken branch, the CPD in COSSEA prefetches both possible consecutive contexts. This implies that all rules are prefetched when there is sufficient time between context switches.

If a context switch occurs during prefetch, the remaining rules need to be transmitted before the context switch can happen. The highest latency occurs when the prefetch has not yet started, or no rules have been transmitted yet for the context of the *taken* transition. Here, the latency of six clock cycles is extended by one clock cycle for each rule contained in the context. Even for a large COSSEA system with many peripherals, we expect that a context does not require more than one hundred rules. The resulting latency of 106 clock cycles is above but still comparable to a function call on the CPU.

VIII. CONCLUSION

We have presented a security architecture that reduces the overall attack surface within an SoC. Our evaluation has shown that the resource overhead incurred for a small configuration is minimal, causing only a slight increase in latency in a limited number of cases.

For future work, we want to investigate a more automated tool flow for creating contexts and their rules. For example, a compiler could analyze the firmware, extract data buffers, and control flow information.

REFERENCES

[1] J. H. Andrew Waterman, Krste Asanović, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203," December 2021.

[2] J. H. Ng, C. H. Ang, and H. C. Law, "A Realization of IO Physical Memory Protection for RISC-V Systems," in *2022 IEEE 15th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2022, pp. 375–380.

[3] N. Wistoff, A. Kuster, M. Rogenmoser, R. Balas, M. Schneider, and L. Benini, "Protego: A Low-Overhead Open-Source I/O Physical Memory Protection Unit for RISC-V," in *Proceedings of the 1st Safety and Security in Heterogeneous Open System-on-Chip Platforms Workshop (SSH-SoC 2023)*. SSH-SoC, 2023.

[4] Arm Limited. TrustZone technology for Armv8-M Architecture Version 2.1. [Online]. Available: https://developer.arm.com/documentation/100690/0201/

[5] S. K. Saha, A. N. Butka, M. K. Ahmed, and C. Bobda, "OpenTitan based Multi-Level Security in FPGA System-on-Chips," in *2023 International Conference on Field Programmable Technology (ICFPT)*, 2023, pp. 302–303.

[6] C. Heinz and A. Koch, "Dd-mpu: Dynamic and distributed memory protection unit for embedded system-on-chips," in *Intl. Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS)*. Springer Nature Switzerland, 2023.

[7] J. Coburn, S. Ravi, A. Raghunathan, and S. Chakradhar, "Seca: security-enhanced communication architecture," in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 78–89.

[8] C. Spang, Y. Lavan, M. Hartmann, F. Meisel, and A. Koch, "Dexie - an iot-class hardware monitor for real-time fine-grained control-flow integrity," in *Journal of Signal Processing Systems*, vol. 94, no. 7, Jul. 2022, pp. 739–752. [Online]. Available: https://doi.org/10.1007/s11265-021-01732-5

[9] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini, "Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX," in *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, 2018, pp. 1–3.