DD-MPU: Dynamic and Distributed Memory Protection Unit for Embedded System-on-Chips

Carsten Heinz[0000−0001−5927−4426] and Andreas Koch[0000−0002−1164−3082]

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

Abstract. The integration of potentially untrustworthy intellectual property (IP) blocks into a System-on-Chip (SoC) poses significant risks, including data exfiltration and corruption due to unauthorized writes to memory or peripheral devices. Conventional countermeasures, such as memory protection or management units, tend to provide coarse protection granularity and impose substantial hardware overhead for embedded devices.
In this paper, we introduce DD-MPU, a custom memory protection unit specifically designed for individual third-party IPs. Our proposed solution features low area overhead and fine protection granularity while automatically adapting to dynamic system states by actively monitoring bus transfers and switching between different protection rules.
In our evaluation, we demonstrate the efficacy of the DD-MPU by integrating it into an SoC to isolate a potentially malicious accelerator block from the rest of the system. The area overhead of our approach for a single instance in a 22 nm technology ASIC node is a mere 0.3 %.

Keywords: Hardware Security · Embedded Systems · System-on-Chip · ASIC · IP Blocks

1 Introduction

As System-on-Chip (SoC) designs become increasingly complex, even small embedded SoCs necessitate the integration of intricate peripherals to improve aspects such as power consumption. This rising complexity requires significant development effort, which is frequently addressed by reusing external intellectual property (IP). However, incorporating third-party, *untrusted* IPs presents risks to hardware design [3].

In this paper, we introduce DD-MPU, a lightweight, dynamic, and distributed memory protection unit specifically designed for small embedded SoCs without virtual addressing. DD-MPU safeguards against malicious or malfunctioning IPs attempting to (a) manipulate the system by writing to unauthorized memory locations, and (b) exfiltrate information from memory.

Our proposed solution, DD-MPU, delivers protection against security threats posed by third-party IPs with a lightweight and easy-to-integrate hardware module. This module effectively segregates third-party IPs with master access to the memory bus from in-house designed SoC components. Each DD-MPU unit is configured during design time according to the associated third-party IP's specifications, thus permitting only memory accesses described in the IP's specifications. The DD-MPU can dynamically alter rules during runtime by monitoring data from a configuration interface and incorporating it into the rules.

This approach is suitable for peripherals such as a network controller, which necessitates reading and writing packets to DRAM memory. In order to configure this controller, the CPU writes pointers to memory locations into the controller's configuration registers. Considering that packet locations may be dispersed throughout the entire memory, a fixed protection rule would prove inadequate. Thus, the DD-MPU detects the data written to the network controller's control register and dynamically adapts the rules accordingly.

For multiple peripherals, each peripheral is encapsulated by an individual DD-MPU unit, reducing routing overhead, blocking malicious operations directly at the source, and minimizing the need for global address decoding.

In contrast to traditional security monitors that secure input interfaces, our approach analyzes outgoing traffic from IPs, eliminating the need to track traffic origin and often resulting in simpler transfer filtering rules. The granularity of memory protection is not restricted by page size and can be as small as one word. Dynamic monitoring is implemented in hardware, with no software modifications required.

The remainder of this paper is organized as follows: Section 2 provides a review of related work. Section 3 introduces our proposed architecture, which is applied to a hardware accelerator integrated into an SoC in Section 4 and evaluated in Section 5.

2 Related Work

Memory protection is typically managed by a *Memory Protection Unit* (MPU), which restricts memory accesses of tasks running on a core in non-virtual memory real-time operating systems, such as FreeRTOS [6]. However, since the MPU is integrated into the CPU, it does *not* offer protection for other bus masters on the SoC.

In more complex systems that employ virtual addressing, the MPU is replaced by a *Memory Management Unit* (MMU) to handle the lookup of physical addresses. While this approach can be extended to bus master devices in the form of an IO-MMU, virtual addressing is generally not utilized in low-power/cost, embedded SoCs. For instance, the ARM Cortex-M series, which lacks an MMU, is the industry-leading architecture in this domain. As we target the same domain in this paper, we will assume that an (IO)MMU is not present in the smaller systems under consideration.

In the open instruction set architecture RISC-V, an MPU, referred to as *physical memory protection (PMP)*, is defined as part of the privileged specification [2]. However, PMP is based on different execution privilege modes and does not provide protection against malicious peripherals. Recently, some vendors have begun addressing this issue: SiFive introduced security *gaskets* [5] and their improved successor, *WorldGuard* [11], while Andes extended the PMP concept to DMA masters with IOPMP [1]. Although these solutions offer enhanced protection against malicious or compromised peripherals, they lack the flexibility of our *dynamic* rule sets and are generally aimed at more complex, higher-performance applications, such as those employing virtualization.

Other current research already includes security monitors in one form or another, e.g., [7, 8]. Here, rules are stored in memory and require a bus master to write updates for changes. Compared to our proposed dynamic rules, this approach results in higher latency when modifying, for example, the base address in a rule. Moreover, the inclusion of these security monitors results in a substantial overhead. NoCF [7] reports a LUT increase of more than 23 % in their FPGA implementation. Additionally, [8] states a 9.18 % overhead in standard-cell area for their chiplet-based system. The evaluation of our proposed solution validates its lightweight nature, as indicated by a mere 0.3 % overhead in cell area.

3 Architecture

DD-MPU is a hardware module that connects to the interface ports of untrusted IPs. Serving as a firewall on the master interface, DD-MPU can interrupt transfers through its Protection Unit. On the slave interface, which typically functions as a configuration interface, DD-MPU monitors transfers using its Detection Logic. Figure 1 illustrates a simplified SoC containing a third-party IP with DD-MPU protection.

3.1 Detection Logic

The detection unit provides input to dynamic rules based on values extracted from transfers on the IP's configuration interface. Contrary to a conventional MPU, which necessitates rule configuration by a specific unit, the DD-MPU eliminates this requirement. Consequently, it can respond to configuration steps already implemented in systems without a DD-MPU. By monitoring an interface, the unit extracts information, such as address, length, and content, from data transfers. This extracted information is subsequently processed by a user-provided *trigger* module, which can range from simple matching to a given register address to more complex actions requiring state tracking. The processed data stream is then forwarded to the rules within a protection unit.

For example, an IP might have a register to configure an Ethernet frame's packet length. In this case, the trigger module detects write accesses to this register and utilizes the monitored value for the dynamic rule. Figure 2 depicts an example detection unit providing input for dynamic rules in protection units. In this example, two protection units (corresponding to two memory interfaces of an accelerator) are paired with two separate trigger modules. The extracted information from the monitored bus is forwarded to the trigger modules, which

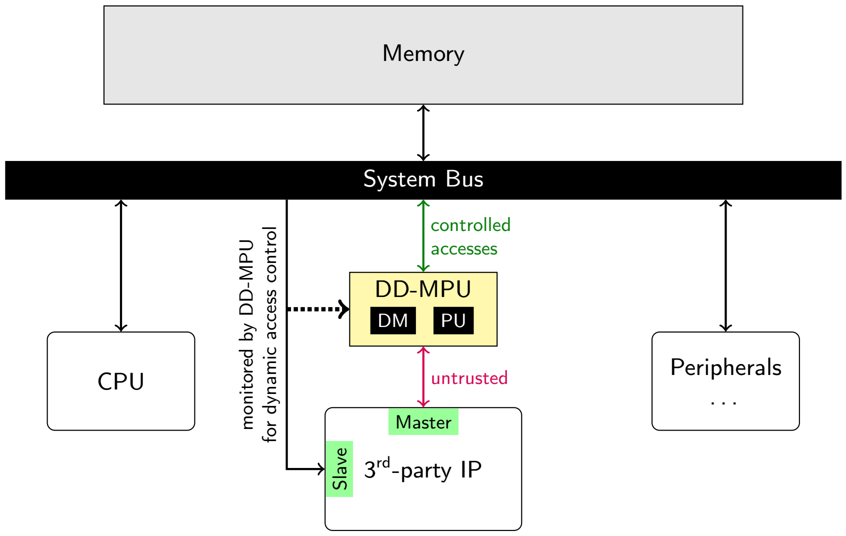

Fig. 1. DD-MPU with detection module (DM) and protection unit (PU) in an example SoC with a shared system bus.

check for specific addresses and use the transmitted data as `DATA` input for the protection units. This could be related to different memory pointers written into control registers at addresses `0xB0` and `0xC0`, allowing both protection units to access the specific memory location.

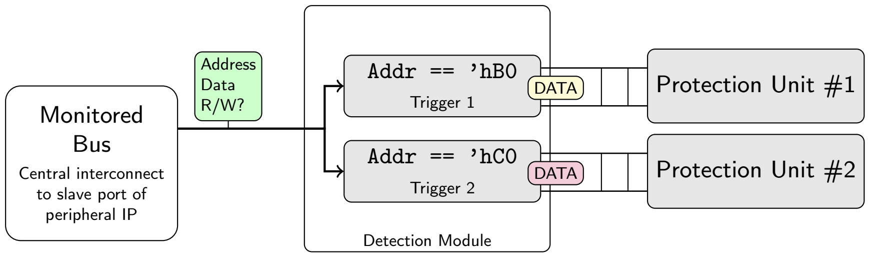

Fig. 2. The detection module extracts information from the control bus, which connects the central interconnect to the slave port of an peripheral IP, and forwards it to the dynamic rules of the two protection units.

3.2 Protection Unit

The decision to permit or deny a memory transfer is made within the protection unit, safeguarding memory from unauthorized access. This decision-making process involves analyzing the memory transfer's base address, transfer length, and whether it is a read or write transfer.

This information is combined with the current state of the detection logic to evaluate a set of rules. If the transfer matches at least one rule, it is allowed to proceed and forwarded to the memory. If not, the transfer is redirected to a dummy sink, which responds with protocol-compliant messages without revealing any real information (refer to Figure 4).

Common bus protocols require handshakes for data exchange, necessitating a response from the other party. In the absence of a response, the IP stalls during a transfer and typically halts until the entire chip is reset. The dummy sink allows transfers to complete without side effects, enabling subsequent IP executions with updated firewall rules to succeed.

A protection unit is configured with a list of rules during design time. Each rule specifies a start address and length for the allowed memory region and can enforce write-only or read-only transfers. In addition to fixed addresses, one can choose to dynamically update start address or length using values from the detection module. Additionally, a rule can be enabled or disabled by the detection module. Figure 3 presents the grammar for specifying rules.

```
<dd-mpu-rule>   ::= <start_addr> <length> <configuration> <direction>
                    <is_dynamic> <outstanding>

<start_addr>    ::= <ADDRESS> | dynamic
<length>        ::= <LENGTH> | dynamic
<configuration> ::= DEFAULT_DISABLED | DEFAULT_ENABLED | ALWAYS_ENABLED
<direction>     ::= READ_WRITE | WRITE_ONLY | READ_ONLY
<is_dynamic>    ::= NONE | DYN_ADDRESS | DYN_LENGTH | DYN_ADDRESS_LENGTH
                    | DYN_ENABLE
<outstanding>   ::= <NUMBER>
```

Fig. 3. Grammar for defining rules for DD-MPU

Apart from dynamic rules, DD-MPU properties can only be modified during design time, reducing the attack surface but making it challenging to accommodate fundamental changes in an IP's functionality (e.g., due to new firmware). As a compromise between the security of hardened rules and the flexibility of full configurability, DD-MPU offers the option to *dynamically* enable/disable *statically* defined rules: It is possible to define additional rules that are initially marked as `DEFAULT_DISABLED`. The secure configuration interface, distinct from the main interconnect, can then be utilized to enable these additional rules, and disable the now obsolete rules to stop the from interfering with the new firmware. This configuration could be facilitated by a central instance, such as a *Secure Element*. In contrast, enabling a more flexible runtime configuration would require the utilization of registers, rather than hard-wired signals. However, this approach introduces additional hardware overhead and timing path complexities, which directly contradicts our objective of achieving a *lightweight* solution. Thus, we impose restrictions on the available configuration options to ensure the desired levels of both efficiency and security.

Information from the detection unit is used to update rules during runtime, such as updating an address range. After updating, the previous content of the rule becomes invalid, and transfers are approved based on the updated rule.

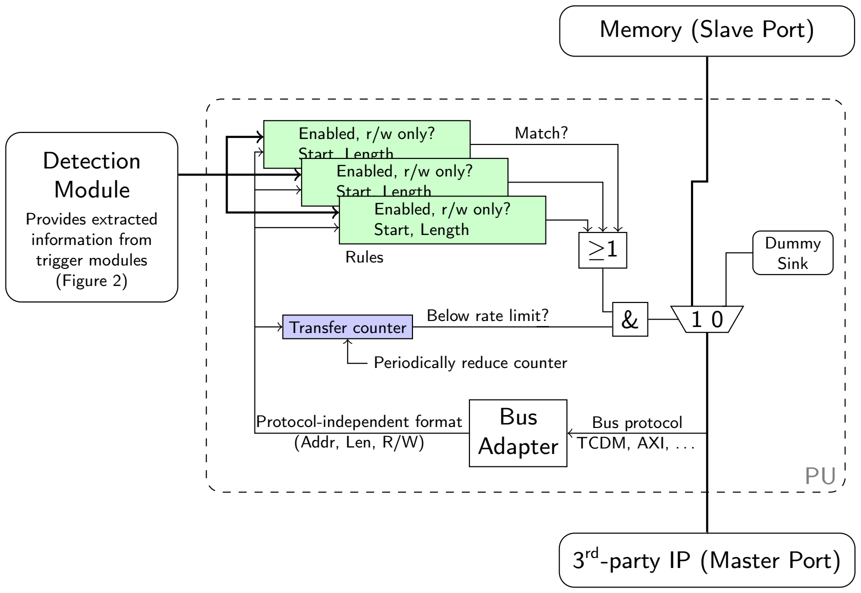

Fig. 4. Protection Unit of DD-MPU with rules and rate limiting.

To address the issue of *outstanding* transfers in IPs, DD-MPU optionally supports outstanding rules. Here, rules are replicated by a given number and are initially disabled. Dynamic updates are distributed across the rules in round-robin order, enabling a rule after the first dynamic update. This approach allows DD-MPU to store information about outstanding transfers and avoid refusing an IP's outstanding transfers.

An additional optional feature of the protection unit is rate-limiting. Typically, a memory bus shared among different modules is negatively affected by a high number of transfers. A malicious (or erroneous) IP could exploit this for a Denial-of-Service attack by issuing numerous transfers in a short time. DD-MPU can be configured to restrict memory bus utilization, blocking additional transfers when the IP attempts to perform more transfers than allowed within a given time. This rate-limiting is implemented by counting all transfers and decrementing the counter according to the allowed utilization at specific points in time. Transfers are denied once the counter reaches the limit and resumed when the counter falls below the limit.

3.3 Customization for IPs

For ease-of-use, it could be expected that we would define a standalone Domain-Specific Language (DSL) which would express the DD-MPU rules and protocol parsers at a high level, and which could then be automatically compiled, e.g., to Verilog, for actual hardware synthesis. This was actually our initial plan when starting this project.

However, it turned out that we can achieve the same goal with far less development effort by instead *embedding* our DSL into Bluespec SystemVerilog (BSV), which provides a much higher level of abstraction (e.g., in terms of type systems and model of computation) than existing HDLs, or the the more modern Hardware Construction Languages (HCLs) such as Chisel. By proceeding in this manner, we can then leverage the very robust BSV RTL generation capabilities to create efficient synthesizable hardware descriptions.

A custom DD-MPU designed to protect against (wrap) a new IP block requires only two components: (1) a list of rules, and (2) trigger functions. Rules define allowed memory accesses and specify which parts are provided dynamically. For simple blocks where existing static MPU-like functionality is sufficient, trigger functions can be omitted. These two components allow the instantiation and connection of firewall and trigger modules to bus protocol-specific adapters.

To enhance portability across a wide range of IP blocks, DD-MPU internally uses an intermediate bus description, while protocol-specific adapters provide the actual bus interface. This approach simplifies the addition of new protocols. The current implementation supports AXI4, AXI4 lite, APB, and the Tightly-Coupled-Data-Memory (TCDM) bus from PULP [9].

Figure 5 demonstrates an example involving `mkAddressTrigger` (the trigger module) and `mkMyDDMPU` (the custom DD-MPU module), which contains bus abstractions, custom trigger modules, detection modules with a list of trigger modules and communication type to the protection unit, and protection units with a list of rules.

4 Case Study

We present a demonstration of DD-MPU using PULPissimo [10], a low-power, embedded System-on-Chip (SoC) built around a RISC-V core with various peripherals. PULPissimo does not incorporate virtual addressing and lacks memory protection. The SoC enables the addition of hardware processing engines (HWPEs) with direct memory access for hardware acceleration. We examine an HWPE, the Hardware MAC Engine, to showcase the application of DD-MPU.

The Hardware MAC Engine, derived from the XNOR Neural Engine infrastructure [4], performs simpler computations. It features a *control* interface connected to the RISC-V core via an APB bus and four *data* interfaces linked

```
// Formulate the detection properties ("protocol parsing")
module mkAddressTrigger(TriggerModule);
    method ActionValue#(DetectionType) evaluate(Trace t);
        if (t.addr == 'hB0) begin // CPU writes to peripheral register 0xB0
            return tagged Address pack(t.data);  // extract the data
        end else begin
            return tagged Invalid;  // do nothing for other accesses
        end
    endmethod
endmodule

// Create customized DD-MPU module
module mkMyDDMPU(MyDDMPU);
    let apb_mon <- mkAPB_Monitor; // Monitoring logic for APB bus
    let my_trigger <- mkAddressTrigger; // Trigger module

    // Instantiate Detection Module to connect Trigger to Protection Units
    let dm <- mkDetectionModule(List::cons(
        Channel {f: my_trigger, impl: FIFO}, Nil
    ));
    // Connect APB monitor to Detection Module
    mkConnection(apb_mon.trace, dm.trace[0]);

    // Instantiate protection unit (PU)
    PUServer pu <- mkPU(defaultConfig, List::cons(
        Rule { start_addr: 0, length: 'h60,
               configuration: DEFAULT_DISABLED, direction: WRITE_ONLY,
               is_dynamic: DYN_ADDRESS, outstanding: 2
        }, Nil
    ));
    let tcdm_adapter <- mkTCDMAdapter; // peripheral master port is TCDM
    mkConnection(tcdm_adapter.pu, pu.check); // connect adapter to PU
    mkConnection(dm.pu[0], pu.trigger_in); // connect DM to PU

    /* connect interfaces to external world */
    interface APB_Monitor_Fab apb_slave_ifc = apb_mon.monitor;
    interface tcdm_master_ifc = tcdm_adapter.master;
    interface tcdm_slave_ifc = tcdm_adapter.slave;
endmodule
```

Fig. 5. Simplified code snippet showing a sample DD-MPU using the Bluespec SystemVerilog library

to PULPissimo's memory subsystem. The data interfaces are employed for loading values and writing computed results back to memory. This example can be applied to HWPEs with *proprietary* RTL.

We attach the DD-MPU monitor to the control interface, where it observes protocol transactions to extract base addresses and sizes of input and output data. The HWPE's master interfaces connect to the DD-MPU's protection units, allowing only permitted traffic to access memory. Our approach supports the development of more sophisticated transaction parsers to extract information from complex custom protocols.

In the HWPE scenario, the protection units' rules only require address and length of the allowed memory regions, provided by the detection logic after straightforward extraction. Additionally, transfer direction enforcement ensures that the first three units allow read transfers only, while the last unit permits write transfers exclusively.

5 Evaluation

In the evaluation, we implemented a PULPissimo SoC featuring the Hardware MAC Engine and its associated DD-MPU (as discussed in the previous section) using a GlobalFoundries 22 nm FDX process with Synopsys standard cells. The synthesis and implementation flow employed Cadence Genus and Innovus. The design was constrained for a frequency of 200 MHz at multiple temperature points and an operating voltage of 0.72 V. For the L2 memory component of the PULPissimo, we utilized Synopsys SRAM macros.

Upon analyzing the results from the firewall described in the previous section, we observed 1595 cells for the firewall module and a total area of 1562 μm^2, which includes the routing area in addition to the cells. In comparison to the unsecured Hardware MAC Engine, this constitutes an area overhead of 13 %. It is important to note that the MAC engine is a relatively small module, primarily consisting of a 32-bit multiplication and basic logic for data fetch and store operations. In contrast, compared to the entire PULPissimo, the overhead of DD-MPU accounts for just 0.28 % of the total area. In Table 1, we also analyzed the overhead when securing only a subset of the data ports. As anticipated, the scaling is proportional to the port count. When examining the area of the entire SoC, no distinct trend is observable, likely due to variations within the place-and-route algorithms, which appear to be greater than the minor overhead of DD-MPU. The same applies to power measurements in Table 1, where no clear trend can be discerned. Furthermore, the table contains results when protecting a single data port only with static rules.

The DD-MPU approach does have an impact on the timing of an SoC. The trigger module connects to the control interface and increases fan-out. This can be mitigated using pipeline registers, with a minor latency cost for rule updates. For many applications, this penalty will be tolerable without any adverse impact: With a post-pipelining latency of three cycles, meaning that the value becomes active in a rule in the third clock cycle after the handshake occurred on the control interface, we no longer observe any negative impact on timing. When considered within a broader context, this apparent latency increase becomes negligible: In a typical hardware accelerator, both the control registers and memory bus are already driven by registers, equivalent to a pipelining depth of two. Moreover, address generation or other computations that take place before the first memory request soccur, thus increasing the base latency to three or more. In such settings, the rule update latency of the DD-MPU will be entirely concealed by the regular operation latency.

Table 1. Area overhead and power consumption of DD-MPU in the PULPissimo SoC. The base design is a PULPissimo without any DD-MPU, compared to designs with a DD-MPU configured to protect one to four master ports. This is contrasted with a DD-MPU configuration for a single master port, containing only static firewall rules without any dynamic updates or monitoring.

Configuration	Area (μm^2)			Power (mW)	
	DD-MPU	(rel. to base)	Total	Total	Dynamic
Base design	-	-	548,670	67.69	10.91
Dyn. 1 port	471.8	0.086 %	547,454	68.68	10.88
Dyn. 2 ports	779.1	0.14 %	550,527	68.25	10.55
Dyn. 3 ports	1149.2	0.21 %	547,112	68.32	10.87
Dyn. 4 ports	1562.0	0.28 %	556,677	68.42	10.59
1 static rule	59.37	0.01 %	551,121	68.11	10.77
8 static rules	63.94	0.01 %	545,224	68.90	10.88
16 static rules	69.15	0.01 %	547,456	68.64	10.96

After examining the DD-MPU control path for timing penalties, we also need to consider the data path of the protection unit, which is added to the client IP's (in this case, the MAC HWPE) memory access logic. It would again be possible to reduce that path delay by pipelining, incurring a latency penalty. However, we observed an increase of only 87 ps, which corresponds to two additional standard cells on the relevant path. Even with the lengthened memory bus, the path delay is *not* the SoC-level critical path, and thus, no additional registers are needed at all.

6 Conclusion

In this paper, we have introduced the DD-MPU approach as a method to restrict memory access by malicious IP cores. Our proposed solution offers protection without necessitating software modifications, as the DD-MPU autonomously responds to hardware transactions, and incurs no performance impact. The minimal area overhead of DD-MPU renders it suitable for implementation in even small SoCs.

Acknowledgements This research was funded by the German Federal Ministry for Education and Research (BMBF) in project 16ME0233.

References

1. Andes Technology: IOPMP Updates: The Protection of IOPMP. https://static.sched.com/hosted_files/riscvsummit2021/de/IOPMP%20Updates%20-%20Protection%20of%20IOPMP_Andes%20Technology.pdf
2. Andrew Waterman, Krste Asanović, J.H.: The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203 (December 2021)
3. Basak, A., Bhunia, S., Tkacik, T., Ray, S.: Security assurance for system-on-chip designs with untrusted ips. IEEE Transactions on Information Forensics and Security **12**(7), 1515–1528 (2017). https://doi.org/10.1109/TIFS.2017.2658544
4. Conti, F., Schiavone, P.D., Benini, L.: Xnor neural engine: A hardware accelerator ip for 21.6-fj/op binary neural network inference. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2018). https://doi.org/10.1109/TCAD.2018.2857019
5. Dabbelt, P., Graff, N.: SiFive's Trusted Execution Reference Platform. https://riscv.org/wp-content/uploads/2018/12/SiFives-Trusted-Execution-Reference-Platform-Palmer-Dabbelt-1-1.pdf
6. FreeRTOS: Memory Protection Unit (MPU) Support. https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html
7. LeMay, M., Gunter, C.A.: Network-on-chip firewall: Countering defective and malicious system-on-chip hardware. CoRR **abs/1404.3465** (2014), http://arxiv.org/abs/1404.3465
8. Nabeel, M., Ashraf, M., Patnaik, S., Soteriou, V., Sinanoglu, O., Knechtel, J.: 2.5D Root of Trust: Secure System-Level Integration of Untrusted Chiplets. IEEE Transactions on Computers **69**(11), 1611–1625 (2020)
9. Pullini, A., Rossi, D., Loi, I., Tagliavini, G., Benini, L.: Mr.wolf: An energy-precision scalable parallel ultra low power soc for iot edge processing. IEEE Journal of Solid-State Circuits **54**(7), 1970–1981 (2019). https://doi.org/10.1109/JSSC.2019.2912307
10. Schiavone, P.D., Rossi, D., Pullini, A., Di Mauro, A., Conti, F., Benini, L.: Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX. In: 2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S). pp. 1–3 (2018). https://doi.org/10.1109/S3S.2018.8640145
11. SiFive Technology: Securing The RISC-V Revolution. https://www.sifive.com/technology/shield-soc-security