Accelerators & Security: The Socket Approach

Luca Piccolboni, Davide Giri, *Graduate Student Member, IEEE*, and Luca P. Carloni, *Fellow, IEEE*

Abstract—We propose the concept of security socket to protect the integration of loosely-coupled accelerators in system-on-chip (SoC) architectures. The socket is placed between an accelerator and the rest of the SoC and provides security services that are completely decoupled from the accelerator implementation. Any accelerator can be plugged into the socket and transparently benefit from all the security services.

Index Terms—Hardware, heterogeneous architectures, security

1 Introduction

Heterogeneous system-on-chip (SoC) architectures combine general-purpose processors with many accelerators, which are application-specific computing engines [1], [2]. By having their hardware tailored to perform specific tasks, accelerators yield major speedups and energy savings compared to software executions. Some accelerators are tightly-coupled to the processors, and thus integrated as special functional units in the processors' pipelines. Others are instead loosely-coupled and they operate as autonomous computing engines [3], like for example NVDLA [4]. In this paper, we focus primarily on loosely-coupled accelerators.

Despite their benefits, the integration of accelerators in SoCs cannot be taken lightly as it affects security [5], [6]. As accelerators take a more central role in the system, there is a growing need for protection mechanisms that must give them the same consideration as general-purpose processors. However, being architecturally different from processors, accelerators demand ad-hoc protection mechanisms to meet the security requirements. The different characteristics of the accelerators require to analyze each accelerator design individually to look for vulnerabilities. While processors security has been extensively investigated, less effort has been put on proposing general protection mechanisms for accelerators. Most solutions only cover specific vulnerabilities and protecting a diverse set of accelerators is an open problem.

*Contributions.*We discuss the security of heterogeneous SoCs focusing on the accelerators and the applications that invoke them. We first define an *accelerator model* by drawing inspiration from the work of Olson *et al.* [5]. We then propose the novel concept of *security socket*. A socket is a hardware component placed between an accelerator and the rest of the SoC. The socket implements *security services*, i.e., protection mechanisms, such as memory access control [7], and primitives, such as encryption. While the socket instantiates many well-known services, we analyze their implementation considering their degree of decoupling from the implementation of the accelerators. We categorize the attacks that can be thwarted with fully-decoupled services and the attacks that might require changes to the accelerators. Finally, we discuss the integration of the security socket in ESP [2], an open-source platform for the design of heterogeneous SoC architectures.

- *The authors are with Computer Science, Columbia University, New York, NY 10027 USA. E-mail: {piccolboni, luca}@cs.columbia.edu, davide.giri@columbia.edu.*

Manuscript received 4 March 2022; accepted 31 March 2022. Date of publication 3 June 2022; date of current version 24 August 2022.
This work was supported in part by NSF under Grant 1764000.
(Corresponding author: Luca Piccolboni.)
Digital Object Identifier no. 10.1109/LCA.2022.3179947

2 Accelerator Model

We define a generic accelerator model (Fig. 1) as a reference. This is representative of many loosely-coupled accelerators [2], [4].

The model is organized in three parts: (i) a computing engine (*comp_engine*), (ii) a local memory (*local_mem*), and (iii) a set of registers $\{R_i\}$ to configure the accelerator. The computing engine contains the hardware that performs the specific tasks supported by the accelerator. For example, the engine of an image-processing accelerator includes the logic to read an image from memory, apply filters, and write back the result into memory. For a programmable accelerator that executes instructions, the engine consists of the stages that implement the instruction pipeline. The local memory holds data during the computation and it is only accessed by the accelerator. It usually contains a copy of a portion of the data in global memory (*global_mem*), i.e., the off-chip memory, also called main memory. The local memory can be implemented as a cache, which obeys a coherence protocol, or as a scratchpad, which is a multi-bank memory structure managed by the accelerator [8]. The registers are used to configure the execution of the accelerator. They are memory mapped, hence accessible by the software applications that run on the processors and invoke the accelerator. The registers define where the input and output data of the accelerator are located and other accelerator-specific parameters, e.g., the number of pixels of the image that needs to be processed by the accelerator.

To use an accelerator, an application prepares the input in global memory and then invokes its device driver. The driver configures the registers and starts the accelerator (signal *start*). The accelerator performs the tasks autonomously without interrupting the processor. The execution is divided in three phases (Fig. 2):

- *Configuration phase:* the memory-mapped registers of the accelerator are configured and the accelerator is ready to compute;
- *Computation phase:* the accelerator performs the assigned task; it communicates with the global memory through the channels *in_chan* and *out_chan* to load the input data and write back the output data, respectively;
- *Termination phase:* the accelerator frees up the resources that it has used (e.g., a portion of the local memory or a functional unit in the case that the accelerator supports the execution of multiple tasks); the accelerator raises an interrupt (signal *done*) after completing the execution of the task.

3 Attack Model

Accelerators potentially affect the *confidentiality*, *integrity*, and *availability* of the system in which they have been integrated [5]. Confidentiality guarantees that an attacker cannot access sensitive information. Integrity ensures that sensitive data can never be corrupted. Availability means that a resource can always be utilized by authorized users when necessary. Next, we describe our threat model by leveraging the taxonomy of Olson *et al.* [5].

Table 1 reports an attack classification based on our accelerator model, indicating the execution phase in which the attack takes place, the signals and/or channels involved in the attack, the attack description, and which of the three security properties may be affected. We consider attacks made possible due to improper uses of the accelerator by software applications (malicious or not), direct attacks to the accelerator, and attacks caused by design flaws in the accelerator. We consider only accelerators that do not include hardware Trojans (the accelerators do not deliberately attack other SoC components), which can be mitigated with techniques [9] that are orthogonal to the socket approach. Each of the attacks of Table 1 is possible depending on the attackers' capabilities, e.g., the attackers may need direct access to the

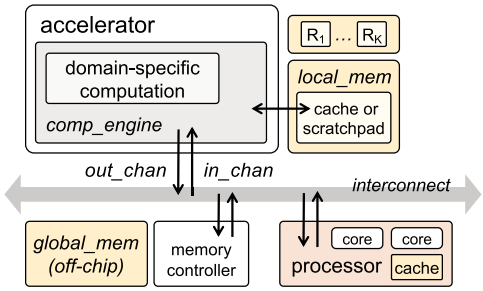

Fig. 1. Model of the target accelerators and integration in an SoC.

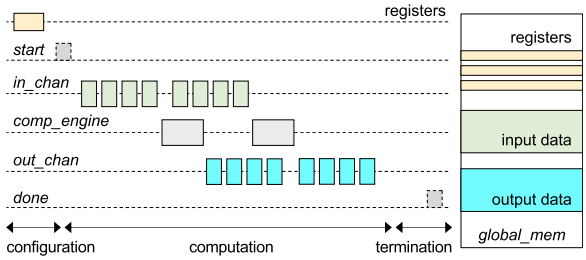

Fig. 2. Example of execution of the target accelerators.

SoC to implement a certain attack, or on the existence of a certain design flaw in the accelerator. We provide a classification of the most relevant attacks that are made possible by the presence of accelerators, without making specific assumptions. The socket is flexible and can instantiate several security services, which may be relevant or not depending on the properties of the SoC in which the socket is deployed. To analyze the potentiality of the socket approach in its entirety, we avoid making restrictive assumptions on trusted and untrusted components. We aim at protecting the accelerators and the applications that invoke them. Some attacks only affect the application that invokes the accelerator, but others have an impact on the entire SoC as discussed by Olson *et al.* [5].

Configuration Phase. During the configuration phase, there are two possible attack vectors: improper use of the signal *start* and improper configuration of the registers $\{R_i\}$. For example, if the signal *start* is utilized maliciously, the accelerator can be set to a "busy" state, therefore violating the property of availability [5], or the accelerator can be invoked by an application without permissions, thus affecting confidentiality. Also, the registers $\{R_i\}$ can be configured incorrectly on purpose. This leads to a variety of issues including unsafe accesses to the global memory [7], incorrect accelerator-specific parameters, and bad configurations of security metadata, such as memory tags [6].

Computation Phase. The computation phase is especially vulnerable because there are many attack surfaces. For example, the power consumption of *comp_engine* can be monitored to leak data, even remotely [10]. Another vulnerable component of the accelerator is *local_mem*, which can be exposed to side-channel attacks [11]. Also, incorrect configurations of the accelerator may cause unsafe read and write requests or an excessive number of requests to *local_mem*. This would affect the integrity of the tasks performed by the accelerator and the availability of the local memory, respectively. Finally, *in_chan* and *out_chan* must be protected as they interface the accelerator to *global_mem*.

Termination Phase. An attack can leak data through the signal *done* [12], in particular by measuring the execution time of the accelerator. In addition, similarly to the configuration phase, the signal *done* can be used maliciously to make the accelerator unavailable. Also, the resources used during the execution, e.g., $\{R_i\}$ and *local_mem*, should be cleared to avoid leaks [13]. For a similar reason, clearing *in_chan* and *out_chan* is necessary to avoid leaks of sensitive data through stale requests [5].

4 Security Socket

We propose the concept of security socket to protect the accelerator that it encloses, establishing a layer of protection between the accelerator and the applications that invoke it. The socket encloses the accelerator and manages its interactions with the rest of the SoC, including the local memory (Fig. 3). Differently from other approaches that focus on a specific vulnerability, the socket includes modular security services that protect the accelerator against a variety of attacks. The last column of Table 1 reports the protection mechanisms that can be integrated in the socket. While many of the services of Table 1 have been proposed in other papers, we analyze their implementation within the socket, while striving to respect three principles:

1. *Decoupling:* we do not require changes to the accelerator logic and we can freely add or remove services as needed;
2. *Automation:* we minimize the effort to deploy the socket;
3. *Efficiency:* we minimize the performance overheads.

We keep the services separated from the implementation of the accelerator. This makes it possible to (i) add and remove services depending on the particular requirements of the accelerator, and (ii) improve the reusability of the services. Table 1 shows the degree of decoupling of each service. We use ● to indicate if modifications to

TABLE 1
Threats to Our Accelerator Model and Protection Mechanisms. The Taxonomy is Derived From Olson *et al.* [5]

execution phase	attack target	attack description	affected security properties	protection mechanism
configuration	signal *start*	incorrect configuration	availability	○ configuration monitor
		incorrect permissions	confidentiality, integrity	○ configuration monitor
	register R_i	incorrect configuration	availability, confidentiality, integrity	◐ configuration monitor
		r/w critical information	confidentiality, integrity	○ encryption unit
computation	*comp_engine*	incorrect configuration	integrity	◐ configuration monitor
		power side-channel	confidentiality	● side-channel analysis
		r/w code and data	confidentiality, integrity	○ TEE support
	local_mem	too many r/w requests	availability	○ request monitor
		incorrect r/w requests	confidentiality, integrity	◐ request monitor
		r/w critical information	confidentiality, integrity	○ encryption unit
		timing side-channel	confidentiality	● side-channel analysis
		power side-channel	confidentiality	● side-channel analysis
	in_chan, *out_chan*	too many r/w requests	availability	○ request monitor
		incorrect r/w requests	confidentiality, integrity	◐ request monitor
		r/w critical information	confidentiality, integrity	○ encryption unit
		learnable r/w requests	confidentiality, integrity	◐ request monitor
		timing side-channel	confidentiality	● side-channel analysis
		power side-channel	confidentiality	● side-channel analysis
		missing metadata (DIFT)	confidentiality, integrity	◐ shadow DIFT logic
termination	signal *done*	bad configuration	availability	○ termination monitor
		timing side-channel	confidentiality	● side-channel analysis
	register R_i or *local_mem*	no resource release	availability	◐ termination monitor
		read old information	confidentiality	○ termination monitor
	in_chan, *out_chan*	r/w pending requests	confidentiality, integrity	○ termination monitor

○ *No modifications to the accelerator and no information required about the accelerator;* ◐ *no modifications to the accelerator and minimal information required about the accelerator;* ● *modifications to the accelerator are needed (future works can investigate decoupled solutions)*

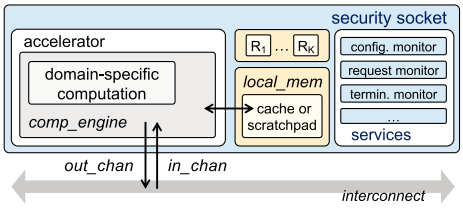

Fig. 3. The proposed security socket protecting an accelerator.

the accelerator are necessary depending on the current state-of-the-art techniques. We use ◐ if only high-level information is required, but no modifications are needed. When only high-level information is required, the security services can be automatically customized starting from a configurable template. We use ○ if no modifications and no information are required. Note that for most of the attacks in Table 1 no modifications are necessary, and we advocate for fully-decoupled services to simplify a safe accelerator integration. In these cases, the socket does not need to modify the computing engine of the accelerator nor its external interface. The accelerator does not need to know that it is protected by the socket: it executes normally, exactly as it would without the security socket.

4.1 Security Services

Table 1 shows the services that can be added to the socket. Each service is "plug-and-play," thus it can be enabled or disabled.

Configuration Monitor. During the configuration phase, the configuration monitor checks that the accelerator is invoked safely by software applications. Specifically, it detects incorrect uses of the signal *start* and invalid values for the registers $\{R_i\}$. One of the goals of the configuration monitor is to ensure the availability of the accelerator. The monitor keeps the history of the invocation calls to the accelerator made by the applications and checks that the accelerator is shared fairly. In addition, the monitor allows only the applications with the right permissions (or that belong to a certain group) to use the accelerator. The configuration monitor can be customized with accelerator-specific checks to guarantee the correctness of the computation. For instance, for an accelerator that performs matrix multiplications, the monitor checks that the dimensions of the two input matrices are compatible.

Encryption Unit. The encryption unit encrypts and decrypts the data and registers of the accelerator, which have been selected at design time. The encryption tasks do not need to be performed necessarily in the socket: the encryption unit can offload these tasks to an external engine, which is shared by multiple sockets. The encryption unit is used in scenarios where the data are not safe in *global_mem* or if the accelerator works on encrypted data.

Request Monitor. The request monitor checks the safety of the memory requests to ensure the integrity and confidentiality of the stored data. For instance, the monitor checks that the requests only access data belonging to the task of the accelerator that made the requests (for *local_mem*) and the software application that invoked the accelerator (for *global_mem*) [7]. Unsafe memory requests happen for multiple reasons, from malicious configurations of $\{R_i\}$ to buggy implementations of the accelerator. The request monitor can guarantee the availability of *local_mem*and *global_mem* by monitoring the number of requests made by the accelerator. Also, if there is a risk of reverse engineering, the request monitor can make the requests to *global_mem* oblivious. The monitor can perform more complex checks, if needed. For example Olson *et al.* show how to check whether the requests to *global_mem* are consistent with the coherence protocol of the platform in which the accelerator is integrated [14].

Termination Monitor. The main goal of the termination monitor is to clean up the resources used by the accelerator, e.g., *local_mem* and $\{R_i\}$, so that sensitive data are not leaked. Many accelerators do not preserve their state across different invocations, but others must keep some data across invocations due to the computation they perform, e.g., a deep-learning accelerator that updates its model at each invocation to improve classification accuracy. In these cases, it is important to clean up all the used resources to prevent leakages. For instance, Di Pietro *et al.* show that the local memories of GPUs can be used to leak confidential data [13]. Besides cleaning up the used resources, the termination monitor must guarantee that there are no stale memory requests [5].

4.2 More Socket Services

In addition to the general services of Section 4.1, the socket can implement services that offer protection in specific scenarios. While here we describe how to support dynamic information flow tracking (DIFT) [6], [15], the socket is flexible enough to accommodate the implementation of other protection mechanisms, e.g., trusted execution environments (TEEs) [16], [17].

DIFT has been implemented mostly on processors [15], but it has been recently extended to accelerators. Piccolboni *et al.* showed how to support DIFT on loosely-coupled accelerators by proposing to handle tags at the interface, with low overhead and without modifying the implementation of the accelerators [6]. We can adopt this approach to support DIFT in the socket: the socket is extended with a module to propagate and check the tags (*shadow DIFT logic* in Table 1). This module intercepts the memory requests made by the accelerator and modifies the requests to load and store the tags in addition to the accelerator's data [6]. This module also implements the logic to propagate the tags, which can be customized for a given accelerator.

4.3 Side-Channel Attacks

Table 1 includes attack vectors caused by side-channels for which it might be hard (or not possible) to provide protection without modifying the accelerator. We can, however, leverage approaches that are orthogonal to the security socket.

Attackers exploit timing side-channels by finding a correlation between the execution time of the accelerator and the value of some sensitive signal. In most cases, preventing these attacks requires a thorough understanding of the internal implementation of the accelerator; this information is typically used to balance the execution time of the accelerator so that leakages are prevented. Jiang *et al.* [12] explain that it is possible to eliminate the timing vulnerabilities by enforcing constant execution time for input/output signals, e.g., the signal *done*. Their approach analyzes the accelerator and adds *enforcement finite state machines* (FSMs) to fix the vulnerabilities. The approach embraces some principles of the security socket, but it requires the enforcement FSMs to receive information from the internal implementation of the accelerator. Still, it is fully compatible with our socket approach, which can enclose an accelerator that has been modified to fix the side-channel vulnerabilities. In some specific scenarios, it may be possible to mitigate the risk of timing attacks by operating only at the level of the socket.

4.4 Design Implications

The security socket is a general and flexible approach to protect accelerators. We demonstrate its effectiveness by showing that the socket (i) protects a relevant class of accelerators (Section 2), (ii) can support a broad variety of protection mechanisms, and (iii) can be extended at ease. The implementation of the socket is feasible since it does not require extensive modifications to the target SoC nor to the applications. The socket implements per-accelerator services. If the replication of a service is inefficient, a centralized implementation of the service can be made available to multiple accelerators. The socket increases design productivity, as it allows designers to focus on the engineering of the accelerators rather than on the security aspects.

5 Case Study

ESP [2] is an open-source research platform for heterogeneous SoC design (Fig. 4). ESP combines a scalable tile-based *architecture* and a flexible system-level design *methodology* [18].

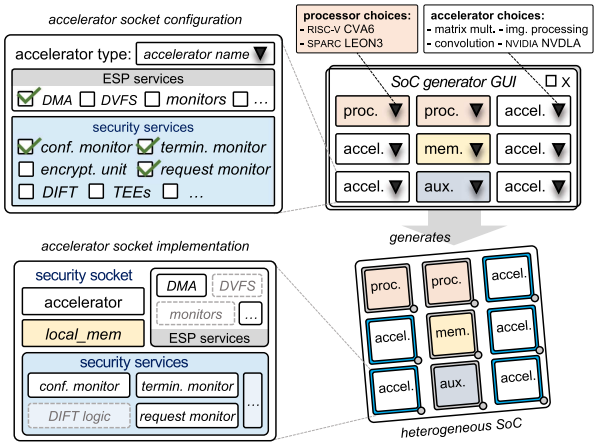

Fig. 4. ESP SoC design flow extended with the security socket.

Architecture. The architecture is structured as a grid [2] (Fig. 4) with four types of tiles: processor tiles, memory tiles for the communication with the global memory, accelerator tiles, and an auxiliary tile for peripherals like Ethernet. Each tile is encapsulated into a *socket* that interfaces it to a scalable network-on-chip and decouples its design from the one of the rest of the SoC [19]. The ESP accelerator socket is automatically generated and can host any accelerator compliant to the ESP load-store interface or to the AXI4 standard. The ESP accelerator socket provides services that are transparent to the accelerators and the applications (Fig. 4), e.g., direct memory access (DMA), dynamic voltage-frequency scaling (DVFS), performance monitors, etc.

Methodology. The methodology consists of two main categories of design flows. The *accelerator design flows* simplify the design of accelerators and their SoC integration. The *SoC design flow* supports SoC configuration, software build, and rapid FPGA prototyping. On the hardware side, ESP supports multiple accelerator design flows, including high-level synthesis flows and domain-specific flows, e.g., hls4ml [20]. All the generated accelerators become part of an intellectual property (IP) library, which includes third-party IPs, e.g., NVDLA. The automated SoC design flow (Fig. 4) allows the designer to select the number, mix, and placement of tiles for a target SoC and many other configuration parameters by using the ESP SoC generator. Once the floorplanning of the SoC is specified, the flow is push-button: ESP generates the RTL of the full system for the physical design of the chip and for FPGA prototyping. On the software side, ESP automatically generates the device drivers and provides an application programming interface (API) for invoking the accelerators from applications running on Linux.

5.1 Security Socket

Architectural Modifications. The ESP accelerator socket can be easily extended with the security services reported in Table 1. The services already present in the ESP socket simplify the integration of the accelerator; the new security services help to make this integration more secure. The ESP socket is automatically generated and the implementations of the security services can be generated through the same process. The implementation of the services that require customization (indicated with the symbol ◐ in Table 1) can be automatically obtained at design time by starting from a templated implementation.

Methodology Modifications. The accelerator design flows supported by ESP do not need to be changed, but some design flows could be extended to support those services that require modifications to accelerators, e.g., side-channel elimination [12]. The SoC integration flow should be adapted for the integration of the security socket. The ESP SoC generator should be extended to enable the selection of which security services are instantiated in each tile (Fig. 4). The device drivers of the accelerators need to be extended to configure the security services and to handle the exceptions that are raised by the socket. Some services may be implemented transparently to software (like the *encryption unit*) by hiding the implementation behind a software API. Other services (for example the *configuration monitor*) require software configuration as well as mechanisms to report unsafe uses.

6 Conclusion

We proposed the concept of security socket to make accelerator integration secure. We showed that the socket is an effective solution to thwart many of the attack vectors that may involve accelerators. We explained how the security socket can be implemented as part of ESP by enhancing the ESP accelerator sockets. We hope that the socket approach will be adopted to implement protection mechanisms for accelerators.

References

[1] W. J. Dally *et al.*, "Domain-specific hardware accelerators," *Commun. ACM*, vol. 63, no. 7, pp. 48–57, 2020.
[2] P. Mantovani *et al.*, "Agile SoC development with open ESP," in *Proc. ACM/IEEE Int. Conf. Comput. Aided Des.*, 2020, pp. 1–9.
[3] E. G. Cota *et al.*, "An analysis of accelerator coupling in heterogeneous architectures," in *Proc. 52nd ACM/EDAC/IEEE Des. Autom. Conf.*, 2015, pp. 1–6.
[4] NVDLA NVIDIA Deep Learning Accelerator. [Online]. Available: http://nvdla.org/
[5] L. E. Olson, S. Sethumadhavan, and M. D. Hill, "Security implications of third-party accelerators," *IEEE Comput. Architecture Lett.*, vol. 15, no. 1, pp. 50–53, Jan.-Jun. 2015.
[6] L. Piccolboni, G. Di Guglielmo, and L. P. Carloni, "PAGURUS: Low-overhead dynamic information flow tracking on loosely coupled accelerators," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2685–2696, Nov. 2018.
[7] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *Proc. IEEE/ACM 48th Annu. Int. Symp. Microarchitecture*, 2015, pp. 470–481.
[8] D. Giri, P. Mantovani, and L. P. Carloni, "Accelerators & coherence: An SoC perspective," *IEEE Micro*, vol. 38, no. 6, pp. 36–45, Jun. 2018.
[9] K. Xiao *et al.*, "Hardware trojans: Lessons learned after one decade of research," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 1, pp. 1–23, 2016.
[10] M. Zhao and G. Edward Suh, "FPGA-Based remote power side-channel attacks," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 229–244.
[11] L. Zhang *et al.*, "Memory-based high-level synthesis optimizations security exploration on the power side-channel," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2124–2137, Oct. 2020.
[12] Z. Jiang, S. Dai, G. E. Suh, and Z. Zhang, "High-level synthesis with timing-sensitive information flow enforcement," in *Proc. ACM/IEEE Int. Conf. Comput.-Aided Des.*, 2018, pp. 1–8.
[13] R. D. Pietro *et al.*, "CUDA leaks: A. detailed hack for CUDA and a (partial) fix," *ACM Trans. Embedded Comput. Syst.*, vol. 15, no. 1, pp. 1–25, 2016.
[14] L. E. Olson, D. H. Mark, and A. W. David, "Crossing guard: Mediating host-accelerator coherence interactions," *ACM SIGARCH Comput. Archit. News*, vol. 45, pp. 163–176, 2017.
[15] G. E. Suh *et al.*, "Secure program execution via dynamic information flow tracking," *ACM Sigplan Notices*, vol. 39, pp. 85–96, 2004.
[16] I. Anati *et al.*, "Innovative technology for CPU based attestation and sealing," in *Proc. 2nd Int. Workshop Hardware Architectural Support Secur. Privacy*, 2013, pp. 1–7.
[17] I. Jang, S. Gueron, S. Johnson, and V. Scarlata, "Heterogeneous isolated execution for commodity GPUs," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2019, pp. 455–468.
[18] L. P. Carloni, "The case for embedded scalable platforms," in *Proc. IEEE/ACM/EDAC 53nd Des. Autom. Conf.*, 2016, pp. 1–6.
[19] L. P. Carloni, "From latency-insensitive design to communication-based system-level design," *Proc. IEEE*, vol. 103, no. 11, pp. 2133–2151, Nov. 2015.
[20] F. Fahim *et al.*, "hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices," in *Proc. TinyML Res. Symp.*, 2021, pp. 1–10.