Secured-by-design systems-on-chip: a MBSE Approach

Raphaële Milan, Loïc Lagadec, Théotime Bollengier, Lilian Bossuet, Ciprian Teodorov

Secured-by-design systems-on-chip : a MBSE approach

1st Raphaële Milan
Université Jean Monnet
Saint Etienne, France
0009-0006-3166-9235

2nd Loïc Lagadec
Lab-STICC, ENSTA-Bretagne
Brest, France
0000-0003-3778-3144

3rd Théotime Bollengier
Lab-STICC, ENSTA-Bretagne
Brest, France
0009-0006-7315-2736

4th Lilian Bossuet
Université Jean Monnet
Saint Etienne, France
0000-0001-7964-3137

5th Ciprian Teodorov
Lab-STICC, ENSTA-Bretagne
Brest, France
0000-0002-0722-5857

***Abstract*—Security by Design (SbD) has gained increasing interest over the past decade. While iterative processes and legacy preservation aim to reduce costs and mitigate risks through continuity, SbD encourages a break in the way we do things with a simple idea: dealing with new threats, leading to new risks, requires a complete rethink of our design processes.**

In embedded systems, security has been more or less left aside for a long time, with performance being the main objective. When security concerns emerged, the response was to adapt existing solutions with security patches. This is neither sustainable (to change from simple embedded systems to complex systems-on-chip) nor simply effective. It is necessary to change the mindset, which will lead to new practices. But the central question is: "How can we put security at the heart of the design process?" The aim of this paper is to contribute to this reflection by providing a rapid prototyping environment (modeling and simulation-based systems engineering) for the hardware mechanisms responsible for the deployment of rights management services.

***Index Terms*—Security, CAD Tool, Model-Based Engineering, Hardware, Systems-on-Chip**

I. Introduction

Security matters. But sometimes time-to-market constraints outweigh security, and reuse is the way to reduce design time. When writing software, reusing pieces of code opens security gaps. There are two main reasons for this. The first is the misuse or misunderstanding of the existing code. A zero-day attack may have been reported and corrected, without the solution being incorporated into the reused code. Context-related problems (e.g., overflows) are also likely to occur. The second problem is that reuse is sometimes based on unsourced code. In this case, incorporating code from a library is easy, but security control becomes impossible. Threats can obviously come from malicious suppliers, but also simply from poor-quality code (e.g. the C system() function calls binary files, which can be silently replaced by malicious files). A corrupted process that has access to shared storage or communication devices can compromise the entire system.

One solution is to use hypervisors. These hypervisors provide an interface layer between the software and the execution platform. Hypervisors check requests against legitimacy rules to grant or deny access to devices. Incorporating hypervisors, at first, promotes a secure by design approach.

While security as a whole has become a major issue, Systems-on-Chip (SoC) security in particular is a hot topic, because such systems are everywhere: computers, cell phones, IoT devices, and Supervisory Control and Data Acquisition (SCADA) systems, making them critical infrastructures.

When moving from software to system-on-a-chip, the analysis of security gaps remains. But with added complexity. On the positive side, there are some useful mechanisms. Vendors offer hardware solutions to ensure minimal rights management at a low level. For example, ARM's TrustZone [1] divides the system into two worlds based on the NR bit value. The rich world, dedicated to common execution, is separated from the secure world, which has additional rights. Processes in the rich world have no access to resources in the secure world. These architectures enforce strict security measures right from the powering on of the system to prevent misuse and compromise. However, on the other hand, such security measures have been found to be vulnerable where security design practices are not considered or are poorly implemented, particularly at software and hardware stack boundaries. Additionally, there is a wide range of devices, resulting in heterogeneity and therefore increased complexity. Peripherals are generally reused from previous designs or come from unreliable third parties. Particularly when considering reconfigurable implementation, for which the synthesis and optimization phase is costly, it is tempting to use off-the-shelf versions from hardware libraries.

This paper proposes an approach for Systems-on-Chip prototyping ensuring key qualities like trustworthiness and safety while maintaining performances. We show that it is possible to deploy in SoCs a hypervisor-based solution that would mediate between software and hardware but also between hardware devices, including synthesized tasks. This additional layer provides a secure by-design infrastructure. It conforms to a set of requirements expressed as rules. In such architectures, each architecture element (processor, FPGA, memory, etc.) has a wrapper that controls its security, based on a distributed

This work has been funded by French AID.

set of rights (authorization) dictionaries that specify who can perform what operation on what address range in what trust mode (secure, rich, etc.) on that element. Permissions can be specified statically at synthesis time, or dynamically, i.e., they can be modified between two runs of the platform. On the contrary, by exploiting these requirements through model-to-model (M2M) transformations, test cases can be generated, which challenge the infrastructure, in search of counterexamples for validation purposes. The M2M approach also leads to additional benefits, which are summarized in the perspective section.

This paper also proposes a fast prototyping solution which conforms to modeling and simulation-based systems engineering (MBSE). A conceptual model describes the architecture (architecture elements, element wrappers, tasks, task allocation to elements, etc.). This model supports instance creation, size change, and reshaping. It offers a useful exploration feature for those who want to design an SoC, as it comes with a discrete element-based simulation framework to score metrics. This framework can be used to simulate various architecture and rights management scenarios at the model level before they are implemented at the lower level.

The remainder of this paper is organized as follows. Section II introduces the systems on the chip domain, and then the related security issues. Section III presents our two contributions: security through rights management and SoC modeling, respectively. Section IV discusses SoC prototyping using FPGA board, with an emphasis on M2M transformations to alleviate manual, repetitive, and error-prone process.

II. Background & Related Work

SoCs are complex electronic devices. They integrate heterogeneous resources (processors, memories, devices, hardware accelerators, etc.) interconnected by a shared bus (e.g. AXI for Advanced eXtensible Interface).

Designing and exploiting such devices require specific tooling, referred to as Computer-Aided Design (CAD) tools, that aid in the creation, modification, analysis, or optimization of a design. CAD processes and software have a long history and are of the highest quality. They form a solid basis for SoC implementation. However, they suffer from rigidity. This problem has been identified since the early 2000s and efforts have been made to remedy the situation. MBSE (Model-Based System Engineering) offers a range of possibilities: Madeo [2] provides a model along with a DSL to support exploration of the SoC domain space. The MARTE standard [3] (Modeling and Analysis of Real-Time and Embedded Systems) offers a methodology for SoC co-design [4], with GASPARD [5] as the SoC co-design environment to move from high-level MARTE specifications to an executable platform. Teodorov [6] has demonstrated regression-free inheritance refactoring while offering additional debugging facilities, These works bring new functionalities (exploration, debugging, automatic generation), while preserving legacy compatibility. This approach is suitable when targeting SoC security-by-design for two reasons. First, existing tools can be used to generate parts of the SoC that are secure (known IPs such as network-on-chip or custom IPs). Second, even if the complete SoC incorporates untrusted third-party IPs, new tools can address specific needs, including the addition of security mechanisms.

Although security requires a holistic approach, targeted contributions are based on certain assumptions. For example, Behani [7] demonstrated malicious tampering with AXI signals, and Gross [8] broke the TustZone mechanism, which had a devastating impact on the SoC as a whole. However, when it comes to focusing on incorporating untrusted third-party IPs into the SoC, not all of these IP cores have passed extensive integrity tests [9], so no security assumption can be made. Still, both mechanisms (Bus, TrustZone) should be considered correctly implemented.

Mal-Sarkar [10] and Jacob [11] demonstrated that third-party IPs can allow attacks such as hardware Trojans and malware that can be launched within any device using the compromised IP. A use case is to compromise a SoC running a secure update. Siddiqui [12] proposes an additional layer of hardware security when a vulnerability is found and exploited, which focuses on mitigating hardware trojans.

III. Contributions

The SoC must be immune to single failure points and must then contain an active response or mitigations for circumstances where a compromise occurs. The threat model consists of one or more IPs that are malicious or misprogrammed and attempt to perform transactions that violate the SoC access control policy. Our solution is based on wrappers as introduced by Siddiqui [12]. Unlike IPs, wrappers appear as relatively simple white boxes, hence are supposed to be immune to external non-physical attacks. Wrappers grant/deny access to remote address ranges depending on the requester, its world, and the requested operation (R/W). Compared to [12], wrappers are not bounded to filtering-out malicious IPs that alter the NS bit, but also cover broader use cases, as any device, including hard or soft IPs (implemented on FPGAs), is free to emit requests of his own. Such requests expose ID, trust mode, R/W mode, and target address; destination wrappers are responsible for reducing and ensuring system assets on the hardware, based on predefined rules.

Wrappers also have to initiate procedures if malicious activity is discovered, then in order to cope with new threats, they should be easily extendable to incorporate new functionalities. The deployment of new features (e.g. redundancy, reallocation, etc.) or policies (e.g. performing collaborative checks, preventing denial of service attacks) is beyond the scope of this article.

A. System-on-Chip executable models

Capturing SoCs as executable models allows fast prototyping of both architectures and protection mechanisms thanks to rich expressiveness and simulation. Such models are dual: on the one hand, they offer an abstraction with pure concepts, such as those expressed in UML (Figure 1), on the other

hand, they provide an executable platform for efficient execution, verification, scenario elicitation, and change tracking for nonregression during refactoring. The automatic generation of code from the executable model and the deployment of this code directly as requirement conformance (within wrappers) and compatibility tests (test-driven development), make the process fast and reliable.

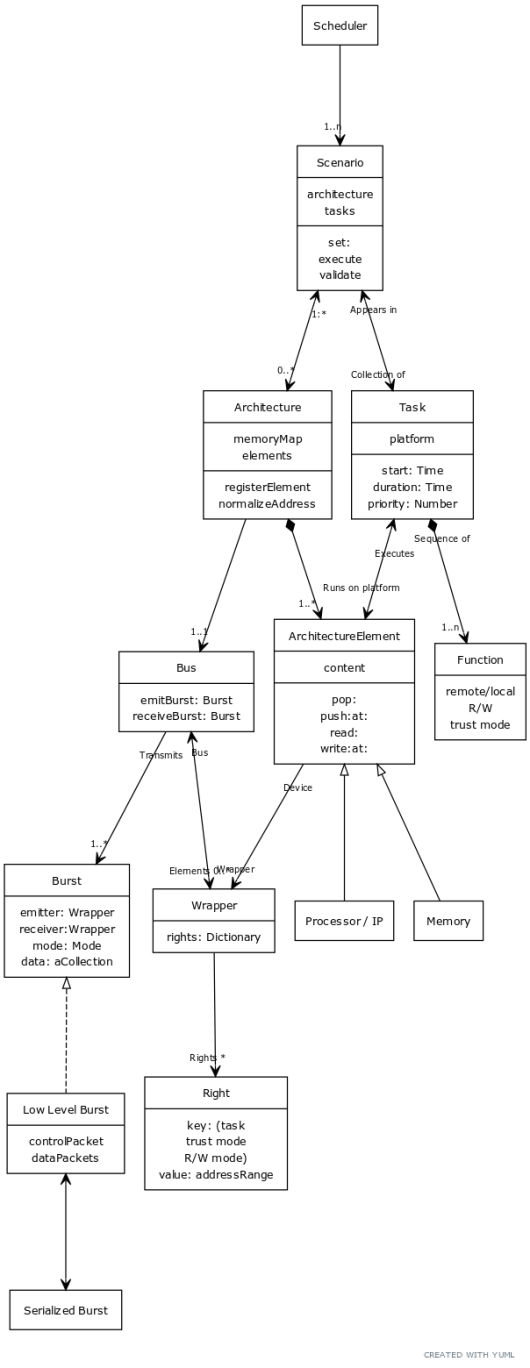

Fig. 1. Capturing the Security-oriented SoC domain as executable model

Embedded systems, including SoCs, exhibit both software and hardware parts. The model takes both into account. The right side of Figure 1 shows tasks composed of a sequence of functions. Tasks communicate with each other through local/remote read/write operations, with the potential for nondeterminism. The left side shows the hardware part, with an architecture owning a bus plus several architectural elements (Processors, IPs, Memories, ...). These elements and the bus are connected through wrappers. The wrappers know the rights associated with their monitored devices (i.e., who can perform what operation on what address range in what trust mode).

Supporting a gradation of abstraction levels arbitrates between simulation accuracy and computation time (e.g. the different burst types as shown in Figure 2). The simplification of the bus protocol (direct addressing of nodes, MMU-based memory shifting, etc.) supports debugging of the node implementation (mixed behavioral level validation). On the contrary, serialized bursts totally reflect the binary information that transits over the bus.

Finally, the scenario captures both dimensions of the model and operates by invoking a simulator on the tasks. The scenario also supports the exploration of the domain space using a parametric architecture and a pseudorandom distribution of tasks. One given architecture is specified as an instance of the model. Listing 1 illustrates a piece of an action language (Smalltalk Pharo dialect [13]) code, which defines a simple architecture, made up of a processor and a memory, interconnected via a bus. During the architecture setup phase, the elements connect to the bus. This results in creating a wrapper that will be further in charge of rights management. The wrapper is the only interface between the nodes and the bus that they both reference. A wrapper has no ID but delegates all its incoming requests to its node.

A task requesting remote access delegates it to its wrapper. The wrapper generates a burst to be transmitted to the bus, then over the bus. The bus dispatches the burst to the proper destination wrapper. This wrapper is responsible for checking the rights (see Section III-B) before serving the request.

Figure 2 represents as an activity diagram the sequence of operations when receiving a burst. Three abstraction levels, ranging from high-level to serialized bursts, are considered. High-level bursts consist of a receiver, data and a relative address at the receiver, as well as a mode (R/W) and a trust mode (rich, secure, extended 1...n). In other words, these bursts contain the elements of the model (arbitrary complex data, real counterpart node, relative address instead of absolute address). This makes it possible to avoid numerous decoding operations, to easily visualize the content of the transmissions, and to be able to quickly verify the correct operation of the implemented mechanisms (rights, but also actual operations at the receiver). Deserializing a binary stream goes through slicing it into words based on the packet's size, and then using the first word to reconstruct a control packet. This packet carries all the structural information to rebuild a low-level burst. Then the following words are deserialized to rebuild the data fragments.

```
architectureExample: grants

| architecture proc ip1 ip2 bus memory
read processedData |
architecture := TSArchitecture new.

bus := TSAXIBus  name: 'AXI'
architecture: architecture.
proc := TSProcessor name:'Processor'
architecture: architecture addressSpace: 3000.
ip1 := TSIP name:'IP1'
architecture: architecture addressSpace: 256.
ip2 := TSIP name:'IP2'
architecture: architecture addressSpace: 256.
mem := TSMemory name:'RAM'
architecture: architecture addressSpace:  10000.

"Wrapper generation"
proc plugTo: bus mode:#master.
ip1 plugTo:bus mode:#master.
ip2 plugTo: bus.
memory plugTo:bus.

memory wrapper applyGrant: grants

^ architecture
```

Listing 1: Architecture representation as model instance creation

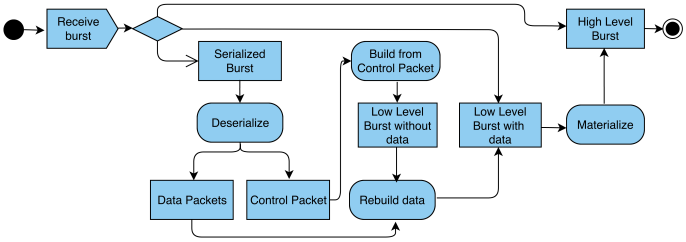

Fig. 2. Receiving high level, low level and serialized bursts

B. Security mechanism relying on rights management

When receiving the burst, the wrapper evaluates the security policy and delegates the operation to its node - be the operation legitimate. Figure 3 represents the sequence of operations when a write request is initiated: delegation of the platform on which the task is executed to the wrapper, invocation of the bus by the wrapper, which transmits the data to the appropriate destination wrapper. Then, two possibilities must be considered: either the operation is granted, in which case the target wrapper delegates to its device the execution of the operation, or it filters the request. In both cases, a return burst is emitted.

Rights are described as a dictionary whose associations link a tuple consisting of (requestor ID, mode, trust mode) to an address range. Authorization is granted if and only if the address range allows the task running in the mode (rich, secured, or an extended version of modes based on a set of security flags) to perform the operation (read/write). The absence of an entry in the dictionary means that there is no authorization. When it receives a request, a wrapper compares it with the registered authorizations and either serves the request or returns a missing-authorization warning.

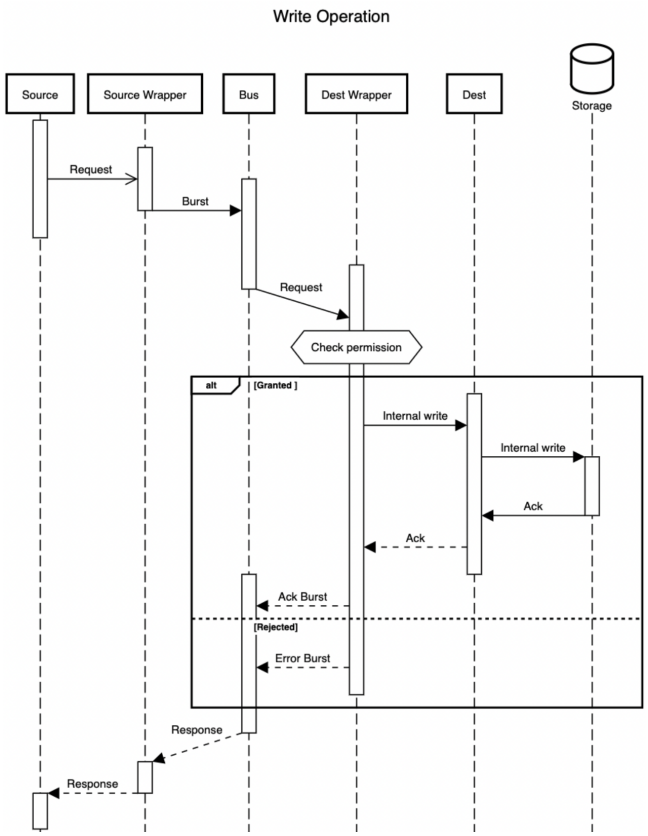

Fig. 3. Write operation sequence

Algorithm 1: Rights management

$$\forall(task, mode, trustMode, \overrightarrow{@})$$

$$\exists!(task, mode, trustmode\overrightarrow{@}) \implies REJECT$$

$$\cup\{\overrightarrow{@_{\text{right}\geq\text{requested}}}\}\cap\overrightarrow{@_{\text{requested}}} = \overrightarrow{@_{\text{requested}}} \implies OK$$

The algorithm 1 summarizes how the decision is made. Address ranges, denoted as $\overrightarrow{@}$ support set operators $\cup$, $\cap$, and $\subset$.

Algorithm 2: Expressing the SR_MEMORY2 rule: Secure and non-secure application memory pages are physically isolated from each other

$$\forall(task, mode, trustmode), (task, mode, trustmode')$$

$$\left\{ \begin{array}{l} \exists(task, mode, trustmode, \overrightarrow{@_{\text{A}}}) \\ \exists(task, mode, trustmode', \overrightarrow{@_{\text{B}}}) \end{array} \right\} \implies \overrightarrow{@_{\text{A}}}\cap\overrightarrow{@_{\text{B}}} = \emptyset$$

Conforming to the security rules goes through deploying authorizations accordingly, as illustrated in Algorithm 2 for memory segregation. These rules are formulated as OCL [14] code, the default language for expressing all types of (meta) model query, manipulation, and specification requirements. Listing 2 encodes the property of Algorithm 2.

```
Context Right Inv  SR_Memory2
Select (i | i key.task = self.key.task and
r.key.trustMode <> self.key.trustMode and
r.value = self.value ) -> isEmpty
```

Listing 2: Encoding the SR_MEMORY2 rule as OCL code

C. Simulation

Executable models can be easily simulated using a discrete-event engine, which represents the operation of a system as a list of events ordered by date [15]. Each event occurs at a given time t and marks a state change in the system. Between two consecutive events, no change occurs in the system; therefore, the simulation time can jump directly to the timestamp of the next event and then resume.

The tasks and respective duration δ are known; a task is triggered as soon as all its predecessor tasks have been completed. In the context of this work, events are model activation: task start/stop, plus internal events (burst, dispatch, permission checks, etc.). Starting a task generates its end as a future event at $t + \delta$, and triggers its associated action, modeled as a closure that takes the platform on which the task is running, as a parameter. Figure 3 provides an illustration in which the values δ are reflected by rectangles of respective height. The simulation loop ends when there are no more pending tasks. The task graph can be reshaped at will, on the fly, to add/remove dependencies (for example, adding "virtual" dependencies can be used to force sequential execution), modify task duration (including interactive change), define triggers to stop execution on specific events such as rights violation, and the simulation supports past snapshot restoration, to test multiple execution paths.

Rules implement rights policies in wrappers, but also serve as a basis for automatic test generation [16] via scenarios, as illustrated by Listing 3. Tests are used to validate compliance with the policy. As an example, Listing 4 illustrates the compliance check of the OCL code of Listing 2, with $isConsistent$ an atomic test ensuring non-overlapping memory addresses. Compliance-oriented scenarios are pseudo-generated and aim to find a counterexample that would identify the violation of a rule. Scenario parameters are generated randomly: topology, tasks, tasks' implementation (IP, process), events distribution (read/write), etc.Once generated, the scenario is saved to be replayed again.

Determinism-oriented scenarios are based on a new randomization step. They are derived from scenarios that do not violate any property. A random mix of abstraction levels (thanks to the M2M transformation) is used to run the same

```
scenarioExample: param
| arch proc ip1 ip2 bus mem data read pData |
"param is (@W1, @W2, @W3, grants)"
arch := TSArchitecture architectureExample:
            param last.

data := (Array new:2000 withAll:1).

"arbitrary addressing"
ip1 write:(1 to:250)  at: (param at:1).
ip1 write:(1000 to:1250)  at: (param at:2).

read := proc read:
(TSAddressRange from: param at:1) +50 to:
                      (param at:2) + 200).

"Simple example here"
pData := (read collect:[:a| a *2]).
"arbitrary addressing"
proc write: pData  at: (param at:3).
^arch
```

Listing 3: Parametric scenario definition

scenario repeatedly. Examples of distinct abstraction levels are high-/low-level burst transactions, mock computation (XOR vs. AES), etc. In addition, data are moved into new address ranges, and all relative addresses are modified accordingly.

Faulty behavior means getting different results for variants of the scenario. Such behaviors are detected through known result assertion violation, as illustrated in Listing 5.

```
behavioralCompliance: param
| arch |
arch := TSArchitecture architectureExample:
            param last.
self assert: (arch allDevices
  reject:[:a| a wrapper rights
              isConsistent not]) isEmpty.
```

Listing 4: Scenario compliance check

```
behavioralCompliance: grantedParameters
| arch |
arch := TSArchitecture scenarioExample:
                grantedParameters.
self assert: (mem contents from: (param at:3)
    to: (param at:3)+150=(2 to: 300 by:2).
self assert: (mem contents from: (param at:3)
    +51...
```

Listing 5: Scenario behavioral check

IV. Hardware design

A. Wrappers

Wrappers serve as a firewall, filtering requests to IPs based on a permissions table, as shown in Figure 4. The experiments were conducted on a ZYBO (ZYnq BOard) platform [17], an entry-level digital circuit and embedded software development platform built around Xilinx Z-7010 [18]. Vivado Design

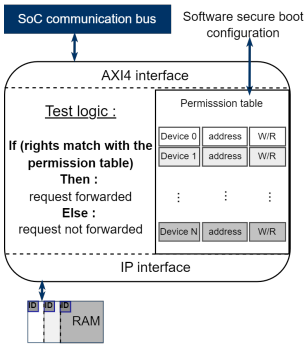

Fig. 4. Permissions table within a wrapper

Suite [19], from Xilinx, addresses productivity bottlenecks in system-level integration and implementation. It provides synthesis and analysis facilities of hardware description language (HDL) designs, with features for system-on-a-chip development and high-level synthesis.

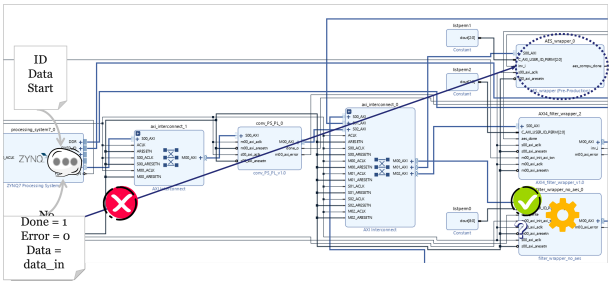

Fig. 5. Low level of architecture (3 IPs + Bus)

Considering a commercial platform for prototyping purposes brings additional complexityas some practical concerns must be addressed. As an example, Figure 5 illustrates that the AXI bus connecting the various IPs is separate from the one associated with the processor. In the ZYBO platform the processor is hardwired and connected to an AXI-3 bus. IPs are synthesized on the FPGA, as is their bus, with an AXI-4 version. Therefore, an AXI-3 to AXI-4 converter must be inserted into the design. Adding this module has no meaning from a conceptual point of view, but relates to low-level platform-specific constraints.

B. Design

A first proof-of-concept has been implemented. Wrappers have been added to the SoC nodes at design time, as shown in Figure 5. Experiments have shown an average increase of 0.34% in terms of LUTs, and 0.1% in terms of registers, without impact on frequency, over a set of five representative IPs. The design has been specified using the Vivado graphical user interface. In addition to being nonintuitive for nonspecialists (e.g. computer scientists), a significant part of our target audience, such interfaces degrade productivity, still reflect users' traditional way of doing things. Experiments demonstrated that the wrappers adequately mitigated unauthorized access.

Hardware validation goes through three steps. First, we need to make sure that the unprotected hardware is working properly and that the processor is accessing the devices. This validation is essential, as communication must pass through the AXI-3 to AXI-4 converter in both directions. This check is carried out by executing a C code that writes and then reads back values (i.e., a simplified Listing 5-like mechanism). The second step then consists of reproducing the same approach, at a lower level of abstraction, by using a TCL code, as illustrated in Listing 6 which describes a test comparing the values written and read according to a burst. Last, if security-based filtering is added, the burst operation can be allowed or denied, as shown in Figure 5. For the sake of conciseness, the corresponding test code is not provided in this article.

The experiments showed a strict equivalence between the high-level executable model and the hardware implementation. However, adapting the hardware design demonstrated to be time-consuming, and hence preserves from design space exploration.

```
# Test: S00_AXI
# Create a burst write transaction at
# s00_axi_addr address
create_hw_axi_txn w_s00_axi_addr [get_hw_axis
$jtag_axi_master] -type write -address
$s00_axi_addr -len 4 -data $wdata_2
-burst INCR

# Create a burst read transaction at
# s00_axi_addr address
create_hw_axi_txn r_s00_axi_addr [get_hw_axis
$jtag_axi_master] -type read -address
$s00_axi_addr -len 4 -burst INCR
# Initiate transactions
run_hw_axi r_s00_axi_addr
run_hw_axi w_s00_axi_addr
run_hw_axi r_s00_axi_addr
set rdata_tmp [get_property
                DATA [get_hw_axi_txn
r_s00_axi_addr]]
# Compare read data
if { $rdata_tmp == $wdata_2 } {
puts "Data comparison test pass for - S00_AXI"
} else {
puts "Data comparison test fail for - S00_AXI,
expected-$wdata_2 actual-$rdata_tmp"
inc ec
}

# Check error flag
if { $ec == 0 } {
        puts "PTGEN_TEST: PASSED!"
} else {
        puts "PTGEN_TEST: FAILED!"
}
```

Listing 6: Tcl Code for burst test

V. Conclusion

Effective hardware design requires specific skills and often focuses on low-level implementation considerations. This can be mitigated by providing a broader view of the whole process.

Fortunately, model-based engineering (MBE) helps capture the expertise of hardware design while promoting a higher level of abstraction. Two key points strengthen this need: first, rapidly evolving security issues lead to short refactoring cycles, and second, the high modularity of chip systems requires point-to-point solutions combined with global composition. This work emphasizes the need to master several levels of abstraction. First, in modeling, we facilitate a detailed understanding of the interactions between the elements while preserving agility through transparent interoperability between these levels. MBE is part of the solution, and targeted development, debugging, and testing can take advantage of. When it comes to the application of rights policies, this leads to real productivity gains. Designing complex rights policies requires ease of specification, simulation facilities, and automatic property validation prior to moving to implementation.

In this work, the Vivado tool was used for demonstration purposes. It was used to design SoC nodes and wrappers.Then to incorporate parts of HDL code from libraries and interconnect them. Among these codes, we obviously find the wrapper code, which is secure by design. This hardware demonstrator has been proven to fully conform to the high-level model and to detect and prevent unauthorized access to devices. The execution of models allowed us to expand the testing by quickly exploring hardware topologies along with the distribution of variable rights. This demonstrated that the combined approach is valuable and viable.

This work opens up many perspectives. Modern commercial EDA tools usually provide end-users with a framework for application-specific customization. This aims at easy interfacing between general-purpose programming languages and underlying circuit object models. Vivado takes advantage of TCL, which offers an M2M transformation back-end process to accelerate the synthesis of parametric SoC and allow defining scenarios. This work perspective also includes the interface between LiteX from Enjoy-digital [20], a SoC builder/IP library, and utilities to create SoCs and complete FPGA designs. In addition to being open source and BSD-approved, its unique feature is that all IP components are described using Migen Python's internal DSL, which simplifies its design in depth. These two perspectives claim for code generation as an enabler to either prototype or implement the model as HDL. The application on the processor requires a C code, which [3] will help generate. The work presented in this paper allows to quickly prototype mitigation hardware mechanisms, but also will, through M2M, support Vivado TCL, LiteX, and Marte as backends to port global design to many technologies.

The agile methodology encourages consideringsimple cases, before tackling realistic and complex cases. Accordingly, we will validate the refinement of high-level abstractions against hardware description language code, as well as the accuracy of functional analyses. Once done, relevant measurements will be available and flexible processes will be used to examine increasingly complex and realistic cases. The objective of measurement is productivity, i.e. the time difference between designing a complex system by hand and modeling the same system in our environment. Also is the ability to interrupt a simulation, or even to go back in time and modify a parameter to measure its impact. This refers both to the hardware aspect (number of cores, heterogeneity, rights, reconfiguration, cryptography facilities, etc.) and to attack scenarios; taking these into account ensures that our protection mechanisms are effective enough to prevent known threats across the platform as a whole, as well as to prevent future threats, due to their malleability and scalability. The changes we make require very little hardware development time (e.g. rewriting wrappers).

REFERENCES

[1] arm.com, "Arm security technology : Building a secure system using trustzone® technology." https://developer.arm.com/documentation/PRD29-GENC-009492/. PRD29-GENC-009492C.

[2] L. Lagadec and B. Pottier, "Object-oriented meta tools for reconfigurable architectures," in *Reconfigurable Technology: FPGAs for Computing and Applications II*, vol. 4212, pp. 69–79, 10 2000.

[3] omg.org, "About the uml profile for marte specification, version 1.2." https://www.omg.org/spec/MARTE/.

[4] I. Quadri, A. Gamatie, S. Meftali, j.-l. Dekeyser, H. Yu, and E. Rutten, "Targeting reconfigurable fpga based socs using the marte uml profile: from high abstraction levels to code generation," *International Journal of Embedded Systems*, vol. 4, 09 2010.

[5] R. Atitallah, P. Marquet, E. Piel, S. Meftali, S. Niar, A. Etien, j.-l. Dekeyser, and P. Boulet, "Gaspard2: from marte to systemc simulation," *Proceeedings of the DATE'08 workshop on Modeling and Analyzis of Real-Time and Embedded Systems with the MARTE UML profile*, 2008.

[6] C. Teodorov and L. Lagadec, "Model-driven physical-design automation for fpgas: fast prototyping and legacy reuse," *Software: Practice and Experience*, vol. 44, 04 2014.

[7] E. M. Benhani, L. Bossuet, and A. Aubert, "The security of arm trustzone in a fpga-based soc," *IEEE Transactions on computers*, vol. 68, no. 8, p. 1238–1248, 2019.

[8] M. Gross, N. Jacob, A. Zankl, and G. Sigl, "Breaking trustzone memory isolation and secure boot through malicious hardware on a modern fpga-soc," *Journal of Cryptographic Engineering*, vol. 12, pp. 1–16, 06 2022.

[9] M. Banga and M. S. Hsiao, "Trusted rtl: Trojan detection methodology in pre-silicon designs," in *2010 IEEE international symposium on hardware-oriented security and trust (HOST)*, pp. 56–59, IEEE, 2010.

[10] S. Mal-Sarkar, A. Krishna, A. Ghosh, and S. Bhunia, "Hardware trojan attacks in fpga devices: threat analysis and effective counter measures," in *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*, pp. 287–292, 2014.

[11] N. Jacob, C. Rolfes, A. Zankl, J. Heyszl, and G. Sigl, "Compromising fpga socs using malicious hardware blocks," in *Design, Automation & Test in Europe Conference & Exhibition*, pp. 1122–1127, IEEE, 2017.

[12] F. M. Siddiqui, M. Hagan, and S. Sezer, "Pro-active policing and policy enforcement architecture for securing mpsocs," *31st IEEE International System-on-Chip Conference (SOCC)*, pp. 140–145, 2018.

[13] pharo consortium, "Pharo: The immersive programming experience." https://pharo.org.

[14] J. Cabot and M. Gogolla, *Object Constraint Language (OCL): A Definitive Guide*, pp. 58–90. Springer Berlin Heidelberg, 2012.

[15] A. Goldberg and DavidRobson, *Smalltalk-80: The Language and its Implementation*. Addison Wesley, 1983.

[16] B. Verhaeghe, N. Anquetil, S. Ducasse, and V. Blondeau, "Usage of tests in an open-source community: A case study with pharo developers," *Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies*, 2017.

[17] Digilent, "Zybo Zynq-7000 ARM/FPGA SoC Trainer Board." https://digilent.com/reference/programmable-logic/zybo/start.

[18] Xilinx, "Zynq-7000 SoC Data Sheet: Overview." https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview.

[19] Xilinx, "Vivado." https://www.xilinx.com/support/university/vivado.html.

[20] F. Kermarrec, S. Bourdeauducq, H. Badier, and J.-C. Le Lann, "LiteX: an open-source SoC builder and library based on Migen Python DSL," in *OSDA 2019, colocated with Design Automation and Test in Europe*, (Florence, Italy), Mar. 2019.