A Survey of Recent Developments in Testability, Safety and Security of RISC-V Processors

Conference Paper · May 2023

DOI: 10.1109/ETS56758.2023.10174099

Jens Anders
University of Stuttgart

Pablo Andreu
Polytechnic University of Valencia

Steffen Becker
University of Stuttgart

A Survey of Recent Developments in Testability, Safety and Security of RISC-V Processors

Jens Anders[**], Pablo Andreu[¶], Bernd Becker[*], Steffen Becker[||], Riccardo Cantoro[†], Nikolaos I. Deligiannis[†], Nourhan Elhamawy[‡], Tobias Faller[*], Carles Hernandez[¶], Nele Mentens[§], Mahnaz Namazi Rizi[§], Ilia Polian[‡], Abolfazl Sajadi[§], Mathias Sauer[††], Denis Schwachhofer[‡], Matteo Sonza Reorda[†], Todor Stefanov[§], Ilya Tuzov[¶], Stefan Wagner[||], Nuša Zidarič[§]

[*]*Dpt. of Computer Science, University of Freiburg, Freiburg, Germany*
[‡]*Institute of Computer Architecture and Computer Engineering, University of Stuttgart, Stuttgart, Germany*
[†]*Dpt. of Control and Computer Engineering, Politecnico di Torino, Turin, Italy*
[§]*LIACS, Leiden University, Leiden, Netherlands*
[¶]*Universitat Politècnica de València, Spain*
[||]*Institute of Software Engineering, University of Stuttgart, Stuttgart, Germany*
[**]*Institute of Smart Sensors, University of Stuttgart, Stuttgart, Germany*
[††]*Advantest Europe, Boeblingen, Germany*

***Abstract*—With the continued success of the open RISC-V architecture, practical deployment of RISC-V processors necessitates an in-depth consideration of their testability, safety and security aspects. This survey provides an overview of recent developments in this quickly-evolving field. We start with discussing the application of state-of-the-art functional and system-level test solutions to RISC-V processors. Then, we discuss the use of RISC-V processors for safety-related applications; to this end, we outline the essential techniques necessary to obtain safety both in the functional and in the timing domain and review recent processor designs with safety features. Finally, we survey the different aspects of security with respect to RISC-V implementations and discuss the relationship between cryptographic protocols and primitives on the one hand and the RISC-V processor architecture and hardware implementation on the other. We also comment on the role of a RISC-V processor for system security and its resilience against side-channel attacks.**

***Index Terms*—RISC-V, functional and system-level test, safety-related applications, cryptography, secure execution**

I. Introduction

RISC-V is an open, load-store instruction set architecture (ISA) based on well-known and established RISC principles that is provided under open source licenses. RISC-V was developed by the researchers of the Berkeley Architecture Research laboratory and began with a goal to make a practical ISA that is open-source, usable academically, and deployable in any hardware or software design without royalties. In the past decade, the RISC-V universe expanded and started conquering markets beyond personal computers, such as transportation and industrial [1]. One of the greatest advantages is the open ISA, which allowed a strong RISC-V community to emerge [2].

The ISA specification defines `XLEN = 32/64/128` address space variants and "mandates" a convenient, modular processor design, consisting of alternative base parts with added optional extensions. Although there are standard open and ratified ISA extensions, RISC-V allows custom extensions to be defined as well, e.g., PULP extension. The RISC-V ISA of an implementation, as shown in Table I, is abbreviated as: (i) the base (e.g., RV32I, RV32E etc.) and (ii) the extensions that are supported (each letter represents an extension [3]). For instance the PicoRV32 core implements the base integer ISA of 32-bits (RV32I) and the base integer ISA of 32-bits, using 16 registers (RV32E) and supports the integer multiplication and division (M) extensions and the compressed instructions (C), hence the ISA string RV32IEMC.

Table I: Selected RISC-V cores: academic and open source

Core	ISA	# st.	Comments	Interface	Tech.	Ref.
Rocket	RV64G,C	5	variable cache, opt. L2, MMU, b.p.	Tilelink Crossbar, RoCC	ASIC ✓	[4] [5]
BOOM (v1/v2/v3)	RV64G,C	6/9/11	variable cache, opt. L2, MMU, b.p., OoO	Tilelink Crossbar, RoCC, integrated RoCC	ASIC ✓ FPGA	[6] [7]
Ibex (Zero−RI5CY)	RV32 I/E, C,M,B	2 + 1	opt. b.p., opt. L0, LSU PULP	OpenHW eXtension, AXI4, APB, crossbar	FPGA ASIC ✓	[8] [9]
CV32E40P (RI5CY)	RV32IMC F	4	opt. L0, LSU, opt. L2, ‡ PULP	AXI4, APB, crossbar	ASIC ✓	[10]
CVA6 (Ariane)	RV64GC RV64IMC	6	variable cache, LSU, b.p., ⊕ PULP	AXI4, APB,	ASIC ✓	[11]
VexRiscv	RV32I MAFDC	2 + 3	opt. I/D cache, opt. LSU, OoO	AXI4, Avalon, wishbone	FPGA ASIC ✓	[12] [13]
PicoRV32	RV32I EMC	1	opt. I/D cache	AXI4, MM, PCPI	FPGA ASIC	[14] [15]
SCARV	RV32IMC	5	side-channel hardened, LSU,	mem. mapped	FPGA ASIC	[16]
SweRV (VeeR EL2)	RV32IMC	4	ultra-low-power core, I cache	AXI4, AHB-Lite, mem. mapped	FPGA ASIC	[17] [18]

# st. – number of pipeline stages + optional stages
b.p. – branch prediction
‡ – manycore possible or only manycore
L0 – prefetch buffer for I cache

opt. – optional features
OoO – Out-of-Order
⊕ – in-Order issue, commit, OoO write back
✓ – fabricated

In Table I we list selected RISC-V cores, with focus on academic and open source designs. In the first two columns we list the core and supported base ISA and possible extensions. Then we list number of pipeline stages and some of the implemented core features, e.g., branch prediction or out-of-order execution, and available interfaces; the latter is of importance for System-on-Chips (SoCs), ISA extensions and coprocessor designs. Finally we provide technology with ✓ denoting fabricated ASIC chips.

Recently, several surveys have emerged to cover the current state-of-the-art in RISC-V developments and features. For a comparative survey of selected open-source cores listing benchmark and maximum frequency, core area, and power consumption for both FPGA and ASIC technologies see [2]. The survey in [19] is focusing on open-source hardware integration of machine learning applications (ML). Authors of [1] classified the RISC-V software ecosystem into four categories: application fields (space systems, IIoT, AI-based heterogeneous systems), RISC-V implementations (soft-cores, System-on-Chip (SoC), emulators and simulators), software architecture (bare-metal, OS, development tools), and deployment features (security, reliability, low-power). The survey in [20] presents ISA extensions designed by the RISC-V community and the progress of official RISC-V ISA extension specifications. Abella *et al.* [21] provided an overview of why RISC-V is successful and some contributions on RISC-V and security against side-channel attacks, a dual core LockStep system with diverse redundancy, and System-Level Test.

In this paper, we survey the recent developments around RISC-V with focus on the following aspects: test, safety, and security. More specifically, in the area of functional test in Section II, we focus on test generation methods for Software-Based Self-Test, Burn-In and System-Level Test and the first steps in the development of a model of non-functional properties of hardware. Then, in Section III, we describe the potential and existing solutions of RISC-V for safety-related applications, with focus on functional correctness and timing verification. Finally, in Section IV, we provide an overview of RISC-V security from the cryptographic, hardware and ISA perspective, focusing on the granularity of added hardware and added custom ISA extensions in the context of hardware/software co-design, and conclude the section with RISC-V system security.

II. RISC-V and Functional Test

In this section we present how RISC-V is able to support research on generating functional tests. In Section II-A we start with the challenges of generating Software-Based Self-Test (SBST) and Burn-In (BI) programs and how the RISC-V specification and ecosystem can help there.

Afterwards, we move from SBST to System-Level Test (SLT) in Section II-B, where we show how the open-source nature of RISC-V and the tools it spawned support research in automated test program generation. Here we explain why it is difficult to use traditional testing methods for SLT and why a greybox approach to generating tests becomes necessary. Then, we go into detail on how an open-source SoC generation framework supports the development of a high-level model of non-functional properties of hardware.

A. RISC-V and Functional Test Development

Software-Based Self-Test (SBST) [22] allows at-speed, native in-field testing of processors with respect to permanent faults by running software programs on the processor core, requiring no Design for Testability (DfT) infrastructure and averts over-testing of manufactured chips. It is well known that the manual development of SBST programs (also known as Self Test Libraries, or STLs) is an arduous task that requires in-depth knowledge of the underlying architecture and expensive project time - often months - to achieve a satisfying test coverage. In many instances, specific test details have to be derived from the hardware architecture implementation. Depending on the test, these details span from register initialization sequences up to reasoning about fault propagation paths.

RISC-V offers a modular and fully customizable ISA specification that features a variety of pre-defined extensions. Combining the base ISA with a set of extensions enables designs ranging from power-efficient embedded microcontrollers to high-performance chip designs. This modular structure of the RISC-V ISA leads to an incremental processor design. Consequently, the development of SBSTs is led alongside the incremental implementation of each ISA extension. As the ISA extensions form defined boundaries of SBST modules the reuse of those software modules for other RISC-V cores is encouraged.

For example, both RI5CY and Zero-RI5CY cores (see Table I) implement the RV32I base specification and the `M` extension. Hence, it is reasonable to assume that some parts and test structures of the same STLs can be re-used for both cores as shown in Figure 1. Though, when re-using the tests for a different implementation a loss in test coverage is to be expected. Nevertheless, with reduced effort a test engineer can extend and port the existing tests and merely develop additional tests for the remaining, not yet-covered `C` and `F` extensions supported by the `RI5CY` core. Although porting is possible, the reader should note that even though the two systems implement the same ISA extensions, hard-to-test faults are architecture-dependent [23].

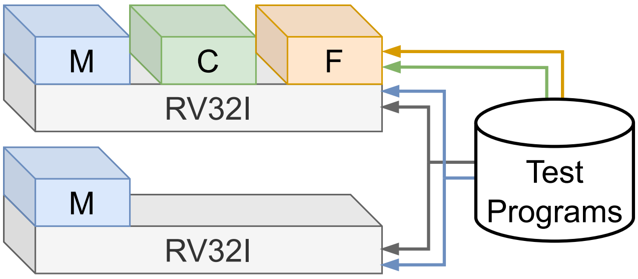

Figure 1: Reuse of SBSTs for designs with overlapping ISAs

The architecture dependency of hard-to-test faults implies a high complexity for test generation. For each new architecture, the development effort for testing hard-to-test faults has to be repeated. On top of that, the non-intuitive fault behavior and propagation make manual development unfeasible. Therefore, tool-aided, automatic SBST development is required for targeting these faults.

In the context of test generation, formal methods like bounded model checking (BMC) provide a facility to find test sequences for hard-to-test faults under functional constraints. The test engineer formalizes these functional constraints in a test specification. This test specification models the set of valid test behaviors that are allowed to be generated by the BMC solver. This test specification, among other things, includes the set of allowed instructions as specified by the ISA, allowed Control and Status Registers (CSRs) accesses and configurations, and the memory model.

Except for the most recently developed extensions, all parts of the official RISC-V ISA exist as formal specifications and build a ground truth for the RISC-V ecosystem of tools. According to the use case, these specifications are combined and inserted into various processes exploiting their reusability. Similarly, these formal specifications are reusable for test generation and allow for the automatic derivation of large parts of the test specification. Hence, RISC-V's open-source and openly available ISA specifications lay the groundwork for automated SBST generation per extension module.

It has been shown that SBST generation embedding parts of the formal RISC-V specification allows for rapid tool-aided SBST development targeting hard-to-test faults in scalar, single-issue pipelined processor core modules [24]. Depending on the supported ISA extensions of the processor core, a test specification is derived in a modular fashion. By transforming the required specification parts into a test specification and combining them with comparatively few additional constraints, a test procedure can be developed incrementally. The additional constraints are SoC architecture-specific and include parameters like memory regions and exception handling. Additionally, the test procedure type is specified depending on the environment the SBST is run. This environment can be a small-scale test after tape-in during development where the SoC is embedded into a custom test bench and the SoC's outputs are accessible; or an at-speed end-of-manufacturing test under harsh conditions to test for infant mortality of chips by artificially aging them (Burn-In test); or a timed self-test of chips in-field during operation to detect degradation due to electromigration and fatigue caused by external stress.

Burn-In test (BI) [25], is omnipresent in the safety-critical domain. More specifically, dynamic BI methodologies require functional stress-inducing stimuli. For the case where the devices under test (DUTs) are RISC-V processors [26], the stimuli generation process can be greatly aided and guided by the incremental ISA structure. For instance, such a process requires the identification of the appropriate vectors (i.e., assembly instructions) in order to compose a typically short and stress-effective program for the DUT. That is, a program that is able to maximize the number of logical switches in short periods of time. By having a clear reference of the implemented sub-set of instructions then we can rely on strategies (e.g., SAT-solvers [27], evolutionary algorithms [28]) to identify such sequences and an efficient stress program for the DUT.

Overall, RISC-V's formal specifications are a cornerstone for functional test constraint derivation e.g., the ISA itself, CSR behavior, and the memory model, directly providing the transition from a specification to functional test generation. RISC-V eases the complexity of STL development by providing an open, modular, well-organized ISA as well as an ecosystem of tools (e.g., compiler, instruction set simulator, virtual prototype) that can be used alongside electronic design automation (EDA) software to develop and validate RISC-V functional test programs.

B. How RISC-V supports research in System-Level Test

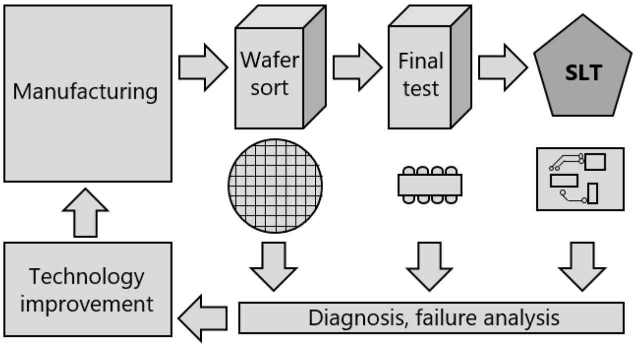

Figure 2: Test flow including System-Level Test (SLT) [29]

System-Level Test (SLT) has emerged as an additional test step in the last decade [30]. It is used to improve the quality assurance of complex SoCs, as traditional testing is not sufficient anymore. Figure 2 shows that SLT is executed as the very last step after Final Test.

SLT tries to approximate the end-user environment of the DUT as closely as possible. For example, a smartphone SoC might be placed into a board that resembles an actual smartphone, plus some monitors and sensors to observe the non-functional properties of the SoC. A test engineer or a system then boots an operating system, in the above example, it might be Android or iOS, and checks for unexpected behavior. If that is not the case, more applications that are typical in the DUT's mission mode are executed, e.g., a video is streamed onto the device or a browser is started. The DUT is considered defective if, at any point, unusual events such as a crash or unexpected errors occur. During SLT, the DUT is considered a blackbox. This is due to the complexity of these SoCs but also due to the fact that some Intellectual Properties are locked, and only I/O interfaces and behavioral descriptions are provided.

There are some issues with this approach. In particular, these test suites are currently manually composed by test engineers. Furthermore, the quality of those cannot be determined beforehand, as the only available metric is a simple pass/fail. So the defect level and diagnosis of returned ICs are used to determine the quality of the suite after the fact. Finally, typical testing times for SLT are ranging from 15 min to up to 2 h and SLT programs are also very large compared to structural tests or SBST.

With the above issues, it is unfeasible to use traditional methods such as fault simulation to gauge the quality of the test suite. Instead, different metrics and approaches for generating shorter SLT programs are developed [29]. One important factor in generating these tests is that we want to control non-functional properties, such as temperature or power consumption, of our DUT because some defects that are assumed to be detected exclusively by SLT are marginal defects [31]. The detection of this kind of defects depends on certain requirements, such as having a component at a specific temperature or a temperature gradient between components or a specific sequence of interactions that can only happen in the mission mode of the DUT.

Writing SLT programs that control non-functional properties by hand requires a lot of time and knowledge about the architecture of the DUT, which makes it infeasible in practical use. Instead, there is ongoing research on methods to automatically generate program snippets that are able to reliably control these properties. To develop these methods, it is necessary to analyze multiple different architectures. This is where an open-source SoC generator framework called *Chipyard* [5] can help.

Furthermore, in Section II-A, we already showed how we can easily derive constraints for methods such as evolutionary algorithms from the freely-available specification. This is useful for SLT program generation as well. For example, we can use greybox-based methods that are guided by feedback to generate program snippets that optimize a specific non-functional property of our DUT [32]. These methods include genetic programming [33] or mutation-based greybox fuzzing [34].

Chipyard has the distinct advantage that it allows us to freely configure SoCs based on either an in-order (Rocket [4], CVA6 [11] Ibex [8] and an educational core called Sodor) or out-of-order (BOOM [35]) core or even mix both. For example, we can add or remove caches and change their size or modify aspects of the cores themselves, such as the number of issues in the BOOM core, the size of the return order buffer, and much more. We can also freely enable or disable extensions and choose between a 32-bit and 64-bit architecture. The flexible configurability allows us to examine multiple configurations for real-world different scenarios, e.g., embedded processors or high-end smartphone SoCs, and identify common and architecture-specific features to derive a generic model of non-functional properties to use for SLT program generation.

Additionally, there is a wide range of well-tested periphery available that we can utilize as well to test interactions between the core and the periphery. And, due to the open-source nature of *Chipyard*, we are able to add our own periphery and make changes to existing ones, such as, for example, bus monitors, to observe activity on all the buses in the DUT. We can use this as feedback for a fuzzer to, for example, maximize the contention on each bus or cover all the possible transactions with all possible sources and destinations.

Chipyard already supports many of the designs listed in Table I. As several of these designs are also compatible with traditional ASIC design flows, it is possible to extract more information such as power simulations, map them to an FPGA to speed-up simulation time and get hardware measurements, and extract additional insight by adding output of bus monitors as feedback for the aforementioned greybox monitors.

Moreover, the RISC-V tools and cores are easy to set up with other commercial tools for further development and investigation. This allows us to work on new, more elaborate metrics to evaluate the performance of the different SLT snippets. Not just for non-functional properties but also for standard fault models (e.g., stuck-at or transition fault delays). Thus, we have some insights into possible causes for marginal defects and can figure out a way to imitate the conditions in which they manifest themselves. Causes of these marginal defects are still unknown, but by having such insights we might be able to find solutions to mitigate them and generalize the behavior of these.

On top of that, by having this transparency of the architecture, we can develop the SLT snippets to target specific peripherals or specific areas in the core that we believe to be problematic. Therefore, the SLT snippets can be developed as compact as possible, reducing the test times but maintaining the fault coverage or even achieving a higher fault coverage.

We also had a chance to compare a RISC-V design with and without DfT structures. We draw the conclusion that DfT structures are needed to facilitate the traditional structural test insertions as well as the SLT test insertion by speeding up the simulation process, having full controllability and observability of the DUT and resulting in much higher fault coverages than without DfT structures. This visibility is what helps to identify more faults and gives the insight on possible causes of a defect.

The freely available RISC-V specification and the rich ecosystem of open-source cores and tools allow researchers to work on functional program generation easily by, on the one hand, enabling them to easily derive constraints to automatically generate valid assembly snippets, and, on the other hand, allow to analyze and use architectures for different application scenarios to extract common and architecture-specific features to guide test program generation.

III. RISC-V for Safety-related Applications

In the context of safety-critical systems several different ISAs are widely used in different applications domains. Some illustrative examples are the case of the SPARC [36] for space, the Tricore [37] in the automotive domain, and the PowerPC in avionics and aerospace. Recently, ARM processors [38] have also increased their presence in safety-critical domain applications allowing safety-critical application developers enjoying from the wide ARM software ecosystem.

In this context, RISC-V arises as a great opportunity to break the existing safety-critical system fragmentation, and many industrial players in the safety-critical domain have shown their interest in RISC-V. The modularity, openness, and flexibility of RISC-V are the key features for its potential success in safety-critical systems. However, the adoption of RISC-V is also subject to having processor architectures that meet the stringent needs for certification of functional safety products. Currently, the RISC-V functional safety special interest group is working on a white-paper covering the best practices for the design of RISC-V processors safety-related applications. The following two subsections about functional correctness (Section III-A) and timing verification (Section III-B) elaborate on the impact of the functional safety in the design of RISC-V based processors. Finally, in the last Section III-C, we review some of the existing RISC-V processors targeting safety-critical applications.

A. Functional Correctness

The functional safety measures at hardware level commonly include fault tolerance, error detection and correction mechanisms. First of all, the processor internal memory structures, such as register files and L1/L2 caches, must be protected with error correction codes (ECC) [39]. Single-error correction and double-error detection (SECDED) Hamming codes are commonly considered in this context because of their moderate resource and performance overheads [40]. Likewise, the internal interconnect and external communication buses are subject to ECC protection.

The replication of entire on-chip components is another widely-used technique to detect and correct hardware faults. The basic fault detection capabilities are achieved by dual modular redundancy. In multicore processors it is typically implemented in the form of dual-core lockstep (DCLS) [41], a mechanism that executes the same program on two CPU cores in parallel, compares cores outputs at each clock cycle and signals any discrepancies. Some lockstep processors allow flexible usage of the redundant (slave) cores, providing a lock mode for reliable execution of critical applications, and split mode for high-performance execution of non-critical applications [42]. The triple-modular redundancy (TMR) or higher multiplicity replication schemes, for instance triple-core lockstep [43], enable real-time error correction.

The diverse execution of the replicated cores plays an important role in any replication scheme, whenever it is supposed to prevent common-mode failures. Diversity can be introduced at ISA and/or microarchitectural levels, resulting in heterogeneous fault-tolerant CPU architectures [44]. These solutions, however, are often considered impractical due to their increased cost and complexity of software running on top of heterogeneous cores. A more affordable (low-overhead) approach for diverse redundancy is time staggering mechanism, which introduces controlled inter-core delay to ensure that replicated (homogeneous) CPU cores execute different instructions at any given time instant.

Additional fault detection mechanisms, commonly provisioned in dependable CPU architectures, include watchdog timers, performance monitoring (profiling), trace and debug support units.

The RISC-V implementation with integrated fault tolerance and error detection mechanisms can be verified by means of fault injection (FI) experiments. FI is a well-known testing technique [45], used to verify the behaviour of critical system in presence of faults, evaluate the efficiency of integrated safety mechanisms, identify safety vulnerabilities and error propagation paths. In fact, the usage of FI testing during the system design is recommended by ISO 26262 standard [46]. At the early design stages, the simulation-based FI allows to verify safety mechanisms implemented in high-level HDL model of the system, e.g. ECC, lockstep execution, etc. Whereas at the late design stages the FPGA-based FI and physical FI are useful to evaluate safety features with all implementation details taken into account, including synthesis-place-route optimizations and impact of implementation technology.

B. Timing Verification

Functional safety certification processes impose several requirements to the software development and validation processes [47]. Several of these requirements are related to the timing behaviour of applications and are tightly coupled with the characteristics of the processor in which the software is executed. In general, and regardless of the functional safety application domain, the execution time of software needs to be bounded at the unit level and to enable a reliable integration of such elements the hardware/software architecture must ensure freedom from interference at the scheduling level. These software requirements are usually translated into the need of deriving the worst-case execution time (WCET) for each software unit and the need for temporal and spatial isolation between the different software items.

WCET derivation is challenged by the presence of hardware features that introduce execution time variability such as caches and shared resources. Thus, processor designs targeting safety-critical applications need to implement specific features targeted to control, reduce or completely remove such execution time variability and achieve a predictable behaviour. For instance, execution time predictability can be attained by replacing caches by scratchpads or by implementing restrictive policies for resource sharing such as time-division multiplexing [48]. However, in general, achieving constant execution time in specific resources comes at the expense of computing performance and/or flexibility which is not acceptable in many application domains. Thus, the current trend towards attaining predictability in complex hardware platform relies on the use of timing monitoring and quota enforcement policies [49].

Resource partitioning can be implemented with software only means and with both hardware and software support. However, fine-grain and performance efficient partitioning requires specific hardware support. In that respect, the RISC-V ISA has defined the `H` extension for virtualization. The `H` extension enables the execution of unmodified OS as hypervisor guests and thus, the co-existence of applications with heterogeneous needs in terms of performance and criticality. Unfortunately, the partitioning implemented at the H-extension level does not prevent the execution time interference originated at microarchitectural features like caches, queues and interconnects. For that, additional hardware support such as cache partitioning, time-predictable arbitration policies, and dedicated hardware for fine-grain isolation is needed [50].

C. Existing RISC-V Solutions for Safety-related Applications

Several RISC-V processors have been adapted or designed to meet functional safety requirements. The NOEL-V [51] from FrontGrade Gaisler is a processor for the space domain that implements the RV64GCH ISA configuration. However, the GPL version of the NOEL-V core does not implement fault-tolerant features. The NOEL-V has also been used to build the SELENE platform [52]. The SELENE platform is a multicore SoC that implements hardware-based diverse redundancy in the form of non-intrusive light-lockstep [53], the safeSU [54] hardware monitors for timing verification and enforcement, and provides virtualization support for partitioning hypervisors such as Jailhouse [55] and XNG, both already supporting RISC-V.

RISC-V based systems from the PULP project [56] have also been adapted to ease its adoption in safety-related applications. For instance, a RISC-V cluster based on the PULP project [57] implement hybrid modular redundancy to select between dual and triple redundancy with lockstep execution in a flexible manner.

Other RISC-V processors targeting safety-related applications that are commercially available (not open-source) are the SiFive P550 Application class processor with configurable dual-core lockstep, RAS capabilities, and ECC in interconnect and memories, and the Fraunhofer EMSA5-FS which has been certified as ASIL-D ready according to ISO 26262:2018 for functional safety in vehicles. A RISC-V processor tailored for high-energy physics applications (e.g. particles detectors at CERN) based on Fraunhofer AIRISC core [58] combines radiation-hardened fabrication technology with several architectural protection measures, including fine-grain TMR and extensive memory scrubbing.

IV. RISC-V for Security Applications

In this section, we discuss recent developments concerning RISC-V cores with respect to security. We begin with a conceptual overview of these developments from cryptographic, hardware, and ISA perspectives in Section IV-A, where we use secure communication applications running on RISC-V as an example. In Section IV-B, we further support our conceptual overview by presenting selected RISC-V cores with or without custom ISA extensions and cryptographic hardware, added as co-processors to the RISC-V core or integrated in the main datapath of the core. In Section IV-C, we highlight some recent approaches to secure RISC-V implementations that prevent or hinder Side-Channel Attacks. Finally, in Section IV-D, we briefly cover specific RISC-V security topics from a system point of view.

A. Conceptual Overview

In this section, we present RISC-V features, supporting secure communication, from three perspectives. We begin the discussion by categorizing the security features based on the type of cryptographic computation they support. This is presented as the *cryptographic perspective* (Figure 3 – right). The pyramid shows examples of basic building blocks, primitives, and protocols required to provide information security. They are placed in a pyramid to express the hierarchical dependency among them, i.e., the basic building blocks are used to construct (composed) primitives that are used to construct security related protocols. The *hardware perspective* (Figure 3 – middle) shows hardware design decisions distinguishing between dedicated hardware added to the main RISC-V core and a coprocessor interfaced with the main core. Finally, the *ISA perspective* distinguishes between Base ISA, Standard Extensions, and Custom Extensions (Figure 3 – left).

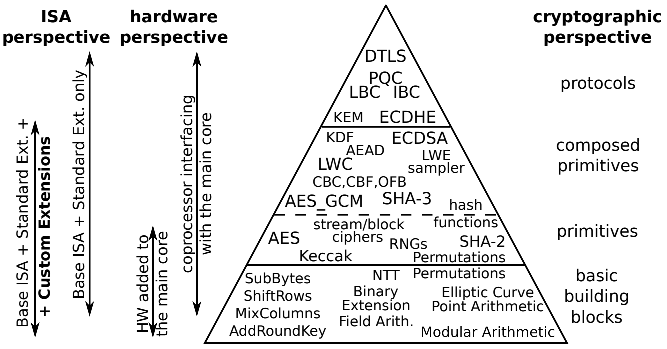

Figure 3: Supported security features in RISC-V based architectures, categorized from three different perspectives.

1) The cryptographic perspective: The pyramid in Figure 3 shows, in a top-down fashion, some main building components needed to assure secure communications. Cryptographic protocols providing secure communication channels (for example the DTLS protocol) are designed as challenge-response handshakes with appropriate steps to assure confidentiality, data integrity, authentication and non-repudiation, and furthermore to prevent known attacks such as replay, man-in-the-middle, etc. These protocols require several cryptographic primitives, a.o. digital signatures (like ECDSA) for authentication. Such cryptographic primitives are composed of basic building blocks. For example, the round function of the block cipher AES is composed of blocks such as SubBytes, ShiftRows, MixColumns, AddRoundKey, and relies on finite field arithmetic.

2) The hardware perspective: In the past decades, it became clear that security is a necessity for many embedded software applications. However, introducing security-related features in a system induces a communication and computational overhead. Such overhead can be reduced by adding dedicated hardware to a general-purpose embedded processor in order to accelerate the security-related computations in an application. In the middle of Figure 3, we show the hardware perspective of security-related features existing in RISC-V based cores and distinguish between dedicated co-processors tightly/loosely coupled with the main core and dedicated hardware additions to the main core. For example, attaching a cryprographic co-processor, such as the DTLS Cryptographic Engine [59], is a common solution. Alternatively, dedicated hardware, such as the AES functional unit [63], can be added to the datapath of the main core. The work presented in [73] analyzes the building blocks of the Lightweight Cryptography (LWC) finalists to find the most suitable acceleration granularity, e.g., their smallest hardware addition is for the Xoodyak parity plane manipulation (xorrol) with only a 1.155× increase in the area of the main core.

3) The ISA perspective: The RISC-V ISA is designed to be modular including a relatively small stable base ISA and multiple standard extensions while it also allows the design of custom instruction extensions. The distinction based on the presence of custom extensions is shown on the left of Figure 3. For example, the aforementioned DTLS engine interacts with the main core via a memory-mapped interface

Table II: Cryptographic applications and granularity of added hardware and added custom ISA extensions in the context of hardware/software co-design

RISC-V ISA	**RISC-V Main core**	**HW perspective**			**Application and granularity** (Figure 3), **Cryptographic perspective**	**Implementation comments**	**Technology CAD tools**	**Ref.**
		add to core	co-proc.	**control details**				
RV32 I	BlueSpec RISC-V		✓	memory mapped interrupts (`wfi`) FSM in coproc.	entire **DTLS** as coprocessor: ECDHE & ECDSA AES-128-GCM, SHA2-256	SPA-hardened configurable prime field ECC accelerator, stand-alone crypto possible	65nm LP CMOS, fabricated	[59]
RV32 IM	BlueSpec RISC-V		✓	memory mapped, interrupts (`wfi`), Sapphire insn. decode+control	entire **LWE-PQC** as **Sapphire** coprocessor: Frodo, NewHope qTESLA, CRYSTALS-Kyber CRYSTALS-Dilithium	modular ALU with config. prime, SRAM-based NTT butterfly unit, Keccak core distribution sampler	40nm LP CMOS, fabricated SPA on fabricated chip	[60]
RV32 I	Rocket		✓	RoCC	entire **crypto** coprocessor: AES, ECC, SHA-256	4-stage pipelined coprocessor, finite field multiplier	28nm CMOS, post-layout, Synopsys VCS and PrimeTime PX	[61]
RV32 IMC + Custom	Ibex	✓		AES IP connected to CSRs	**AES** coprocessor with one custom instruction	AES IP in decode stage mode as insn. parameter (ECB, CBC, CFB, OFB)	Nexys Artix-7, Xilinx Vivado 2018.1 post-P&R simulation	[62]
RV32/64 IMC + Custom	SCARV 32-bit	✓		main core decoder	**AES-GCM** with custom insns for: SubBytes, MixColumns, and SubBytes + MixColumns	AES-FUs for 4 32-bit extension sets and 1 64-bit set, carry-less mult. for GCM	Yosys post-synthesis RISC-V Cryptography Extension K proposals	[63]
	Rocket		✓	RoCC				
RV32 IMCF	CV32E40P (RI5CY)	✓		Accelerator connected to FPR	**finite field accelerator**, used for IBC (PQC): SIKE	Montgomery Muliplier, Modular Adder and Subtractor	Xilinx ZedBoard, Xilinx Vivado 2017.4	[64]
RV32 I	VexRiscv		✓	APB decoder connected to the main core	**prime field coprocessor**, used for IBC (PQC): SIKE	Modular arithmetic: adder, dual Montgomery multiplier for all SIKE primes	Xilinx Virtex-7, post-P&R	[65]
RV32 IM	CV32E40P (RI5CY)		✓	32-bit AHB Intecrconnect	**NTT accelerator, hash, accelerator**, used for LBC (PQC): NewHope	DMR hardened NTT and NTT-LUT core, hash core with Keccak	Xilinx ZedBoard Zynq-7000, FI simulation	[66]
RV32 IM+ Custom	VexRiscv	✓		Flexible pipeline: plugins	**prime field accelerator**, w/ 4 custom insn, used for LBC (PQC): Kyber, NewHope	2 variants: synthesized with a fixed prime, and flexible prime as parameter	Artix-35T, Xilinx Vivado 2019.1, iCE40 UltraPlus, Icestorm	[67]
RV32 IM+ Custom	CV32E40P (RI5CY)	✓		main core decoder	**finite field accelerator**, w/ 4 custom insns. used for LBC (PQC): LAC	PQ-ALU, ternary multiplier, Chien module, modular reduction , SHA256	Xilinx ZCU102 Zynq UltraScale+	[68]
RV32 IMC+ Custom	SCR1		✓	32-bit AHB-Lite Interconnect	**vector coprocessor** w/ 12 custom insns. used for LBC (PQC):Kyber, NewHope, LAC	NTT core, Keccak core, binomial sampler, Random Number Generator	TSMC 28nm HPC, fabricated	[69]
RV32 IMC+ Custom	Ibex	✓		Vector Instruction Interface	**vector coprocessor**, w/ 16 custom insns. used for LBC (PQC): CRYSTALS-Kyber	NTT, CWM, modular arithmetic for each exLane, register pooling, automatic index generation	Xilinx Alveo U250 Vivado 2019.2	[70]
RV32 IMC+ Custom	SweRV-EL2	✓		main core decoder	**finite filed accelerator**,w/ 4 custom insns. used for: AES, Reed- Solomon codes	carry-less multiplication, polynomial reduction, parametrized degree, irred. poly.	Nexys Artix-7, Verilator v4.032	[71]
RV32 IMCF+ Custom	CV32E40P (RI5CY)	✓		main core decoder	various **accelerators**, w/ 29 custum insn., used for LBC (PQC): NewHope, Kyber, Lightsaber , Firesaber, Saber	NTT, modular arithmetic, Keccak in decode stage binomial sampling, Karatsuba, Toom-Cook multipliers in execute stage	Xilinx Zynq-7000, UMC 65nm, Cadence Incisive Enterprise Simulator, Joules	[72]
RV32 GC + Custom	Rocket	✓		main core decoder	**LWC finalists:** separate extensions for each candidate by benchmarking, e.g., Ascon sigma.lo, sigma.hi insns, e.g., XOODYAK xorrol insn	main core, Zbkb/x FU, ans ISE-specific LWC FU for each candidate, also implementing AES-GCM for reference	SASEBO-GIII board Kintex-7 target, Xilinx Vivado 2019.1	[73]
RV32 IMC+ Custom	CV32E40P (RI5CY)	✓		main core decoder	**Ascon-p** with a custom insn: permutation with number of rounds as parameter	Ascon-p in Decode stage, Internal state in GPR Ascon, Ascon-hash, ISAP	65nm LL CMOS Cadence Encounter RTL Compiler, NanoRoute	[74]

PQC = Post-Quantum Crypto, IBC = Isogeny Based Crypto., LBC = Lattice-Based Crypto, LWE = Learning With Errors, ECC = Elliptic Curve Crypto.
LWC = Light-Weight Crypto., SPA = Simple Power Analysis, FI = Fault Injection, DMR = Dual Modular Redundancy
CSR = Control and Status Registers, FPR = Floating Point Registers, GPR = General Purpose Registers

and does not require any (custom) additions to the ISA [59]. In contrast, the work in [62], [63] presents a design of custom AES instructions for RISC-V. In general, with custom ISA extensions, ISA and hardware designers can choose a suitable level of granularity of the instructions and dedicated hardware added to the RISC-V based architecture. For example, the authors of [62] implemented the entire AES primitive with different modes, including ECB, as a custom instruction, while [63] presents separate RISC-V ISA instructions for utilizing the SubBytes and MixColumns basic blocks as well as a single instruction for utilizing both SubBytes + MixColumns.

Finally, there exists an ongoing effort to propose and standardize RISC-V cryptography ISA extensions containing two orthogonal sets of specific instructions: first, scalar & entropy source instructions [75], and second, vector cryptographic instructions [76]. The scalar & entropy source instructions are cryptographic instruction proposals for smaller cores that do not implement vector extensions. They include a subset of "constant-time" instructions with data-independent execution time to prevent timing side-channels. In contrast, vector cryptographic instructions are proposed to be used in large and high performance cores that have large vector registers that are reused. Some examples are AES and SHA-2, with instructions on different granularity, for example single-round (building block) and all-round (primitive). It is also possible to vectorize SHA-3 using a number of general-purpose instructions, allowing for simultaneous processing of multiple pieces of data to increase the performance of SHA-3 [76].

B. Selected RISC-V Cores with Security-related Features

In this section, we present selected state-of-the-art proposals for RISC-V implementations with security support. These proposals are shown in Table II where the order of the rows roughly follows the top-down cryptographic perspective illustrated with the pyramid in Figure 3. The columns in Table II are organized starting with the ISA perspective and the RISC-V main core, in the first two columns, followed by three columns summarizing the hardware perspective, with checkmarks for dedicated hardware added to the main core and co-processor designs. When custom extensions are present, indicated in the ISA column, the instruction decoder in the main core has to be modified but we do not consider this as an addition to the main core in Column 3. Column 5 lists some implementation details to show the relationship between the dedicated hardware and the main core. For example, the Sapphire co-processor in Row 3 has its own instructions decoder and controller, and could be reused with any main core.

Column 6 provides some application details from the cryptographic perspective, illustrated with the pyramid in Figure 3, together with the granularity of the ISA extensions and corresponding hardware additions. Column 7 provides very brief implementation comments, such as expensive cryptographic computations, i.e. sub-modules illustrated with the pyramid in Figure 3, and architectural decisions. Finally, in Column 8, we provide some information on the technology and CAD tools used to design the RISC-V cores, focusing mostly on fabricated ASIC and configured FPGA designs.

C. Towards Side-Channel Attack Resilience for RISC-V

During cryptographic operations, computations are performed using sensitive data such as cryptographic keys. It has been shown that such sensitive data can be revealed by exploiting side-channel leakage, such as power consumption or electromagnetic emanations. Thus, in the past two decades, we have seen the rise of side-channel analysis (SCA) for cryptographic circuits and systems. More recently, such SCA trend could be witnessed for RISC-V based systems featuring support for security-related applications. We briefly discuss some side-channel hardened RISC-V solutions, aiming to prevent, or at least to hinder, the extraction of sensitive data from RISC-V based systems.

As mentioned earlier, the DTLS Cryptographic Engine [59] is often attached as a co-processor to a RISC-V core. This engine implements countermeasures to prevent simple power analysis (SPA) from distinguishing point addition and point doubling. Similarly, the Sapphire co-processor implements constant-time SPA-hardened sub-modules [60]. The designers of Sapphire also conducted simple power analysis on the fabricated chip to verify the constant-time execution. The Sapphire co-processor has no differential power analysis (DPA) countermeasures, but its designers have discussed how to include such countermeasures using the Sapphire's programmability.

The authors of [77] propose a secure RISC-V ISA and architecture by including an additional secure pipeline which is completely separated from the unprotected (original) pipeline. The instruction decode stage disables the unprotected pipeline during execution of protected instructions to prevent leakage. The secure pipeline executes a set of instructions protected by Domain-Oriented Masking (DOM) with multi-cycle non-linear operations, e.g., 4-cycle 1-bit ADD instruction with two fresh random numbers per share. The idea of custom ISA instructions to mitigate SCA has been extended in [78] to include DOM and other masking approaches. The authors have presented 22 custom instructions such as conversion of operands (Boolean to/from Arithmetic masking), Boolean masking operations, arithmetic masking operations, and masked finite field arithmetic for $GF(2^8)$. They place a masked ALU containing dedicated hardware, including random bit generators, in parallel to the unprotected ALU in the SCARV core, with a 1.45× area increase for the ASIC implementation. Additional randomness and re-masking are used to compensate for not duplicating the datapath. The work in [78] also provides preliminary SCA results on FPGA implementations of the SCARV core by running and comparing unmasked AES using only the base ISA and masked AES using the aforementioned custom extensions.

Finally, the authors of [79] have used formal verification methods to design a secure Ibex core with additional features, preventing leakage, such as secure register file, AND-gated computation unit, and clear for the hidden LSU buffer. The security features increase the total Ibex core area by only 9.9%.

D. RISC-V System Security

This section briefly discusses selected RISC-V security topics, such as Root-of-Trust (RoT), trusted boot, memory protection and isolation, and trusted execution environment (TEE), from the perspective of added (cryptographic) hardware. Some of these topics are also covered in [1], [20], [80].

A secure bootloader is considered to be the smallest RoT, on top of which the chain of trust and trusted code base are built. For example, Sanctum [81] uses trusted on-chip ROM to hold the bootloader. In addition, the system includes a hardware TRNG and PUF, and software SHA-3, ECDSA and AES implementations [82] to provide strong isolation using enclaves and to protect against software attacks, such as cache timing attacks and passive address translation attacks. Sanctum has been designed for Rocket RISC-V with minimal hardware extensions and implemented on Xilinx Zynq 7000 FPGA.

Keystone [83] is an open-source framework for building customized TEEs with only basic primitives implemented in hardware. The framework is comprised of a device-specific secret key available only to the trusted boot process, a hardware source of randomness, and a trusted boot process. The Keystone RoT can be either a hardware crypto engine or tamper-proof software. It uses physical memory protection (PMP) and a security monitor running in machine mode to enforce memory isolation. For example, a SoC with two Rocket cores using Keystone implements two hardware accelerators, SHA-3 and ECDSA, connected via Tilelink [84]. Another example is the ITUS RISC-V based secure SoC [85] which includes two Rocket cores and integrates Keystone-enclaves for TEE. The SoC, implemented on Xilinx Kintex 705, also includes a hardware key management unit with one-time programmable memory, TRNG and PUF, a hardware signature verification unit with boot sequencer, SHA3, the XMSS and ECDSA digital signature schemes, and a memory protection unit with AES-GCM. In addition, [85] also considers countermeasures for side-channel attacks. Keystone has been also modified to run on the CVA6 core and on a GPU-scale accelerator with 4096 cores, divided into clusters of 8 [86].

V. Conclusion

The RISC-V ISA and processors implementing it are receiving an ever-increasing attention from both the scientific community and the industry. The test, safety and security aspects discussed in this survey are key prerequisites for RISC-V processor being useful for actual products. While many basic concepts and approaches previously developed for different architectures can be reused with minor modifications, the open nature of RISC-V ISA calls for new and more universal solutions for some of the problems mentioned. Therefore, this survey can only provide a snapshot of recently obtained results, and the scientific work will continue in the foreseeable future. In order to obtain the best solutions, it is important to establish and maintain connection between RISC-V architects and specialists in test methods, safety and hardware-oriented security.

Acknowledgment

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0132 and by the Italian ICSC National Research Centre for High Performance Computing, Big Data and Quantum Computing within the NextGenerationEU program. UPV researchers have received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 877056, Agencia Estatal de Investigación from Spain under grant agreement no. PCI2020-112092 within the NextGenerationEU program. Furthermore, this research was supported by Advantest as part of the Graduate School "Intelligent Methods for Test and Reliability" (GS-IMTR) at the University of Stuttgart. For this work, Leiden University received funding from the Dutch Research Council (NWO) through the PROACT project (NWA.1215.18.014).

References

[1] B. W. Mezger *et al.*, “A Survey of the RISC-V Architecture Software Support,” *IEEE Access*, vol. 10, pp. 51 394–51 411, 2022.

[2] A. Dörflinger *et al.*, “A comparative survey of open-source application-class RISC-V processor implementations,” in *CF*, 2021, pp. 12–20.

[3] *RISC-V ISA, Unprivileged Specification*, https://riscv.org/technical/specifications/.

[4] K. Asanović *et al.*, “The Rocket Chip Generator,” EECS Department, University of California, Berkeley, USA, Tech. Rep. UCB/EECS-2016-17, Apr. 2016.

[5] A. Amid *et al.*, “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs,” *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[6] C. Celio *et al.*, “BROOM: An Open-Source Out-of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS,” *IEEE Micro*, vol. 39, no. 2, pp. 52–60, 2019.

[7] J. Zhao *et al.*, “SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine,” 2020.

[8] *Ibex documentation*, https://readthedocs.org/projects/ibex-core/downloads/pdf/latest/, May 2023.

[9] P. Davide Schiavone *et al.*, “Slow and Steady Wins the Race? a Comparison of Ultra-Low-Power RISC-V Cores for Internet-of-Things Applications,” in *PATMOS*, 2017, pp. 1–8.

[10] M. Gautschi *et al.*, “Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices,” *IEEE Trans. VLSI Syst.*, vol. 25, no. 10, pp. 2700–2713, 2017.

[11] F. Zaruba *et al.*, “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-bit RISC-V Core in 22-nm FDSOI Technology,” *IEEE Trans. VLSI Syst.*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019.

[12] *GitHub - SpinalHDL/VexRiscv: A FPGA friendly 32 bit RISC-V CPU implementation*, https://github.com/SpinalHDL/VexRiscv, 2018.

[13] T.-T. Hoang *et al.*, “Low-power high-performance 32-bit RISC-V microcontroller on 65-nm silicon-on-thin-BOX (SOTB),” *IEICE Electronics Express*, vol. 17, no. 20, pp. 1–6, 2020.

[14] *GitHub - YosysHQ/picorv32: PicoRV32 - A Size-Optimized RISC-V CPU*, https://github.com/YosysHQ/picorv32.

[15] K. P. Ghosh *et al.*, “Technology mediated tutorial on RISC-V CPU core implementation and sign-off using revolutionary EDA management system (EMS) - VSDFLOW,” in *CSTIC*, 2018, pp. 1–3.

[16] B. Marshall *et al.*, “Implementing the Draft RISC-V Scalar Cryptography Extensions,” in *HASP*, 2020, pp. 1–8.

[17] W. Digital, *EL2 SweRV RISC-V CoreTM 1.4*, http://www.azhothot.com/risc-v.html, Jan. 2020.

[18] T. Kruiper, “Area-Optimized RISC-V-Based Control System for 22nm FDSOI Analog and Mixed-Signal Test Chips,” M.S. thesis, University of Twente, Jan. 2023.

[19] S. Kalapothas *et al.*, “A Survey on RISC-V-Based Machine Learning Ecosystem,” *Information*, vol. 14, no. 2, p. 64, 2023.

[20] E. Cui *et al.*, “RISC-V Instruction Set Architecture Extensions: A Survey,” *IEEE Access*, vol. 11, pp. 24 696–24 711, 2023.

[21] J. Abella *et al.*, “Security, Reliability and Test Aspects of the RISC-V Ecosystem,” in *ETS*, 2021, pp. 1–10.

[22] M. Psarakis *et al.*, “Microprocessor Software-Based Self-Testing,” *IEEE Des. Test. Comput*, vol. 27, no. 3, pp. 4–19, 2010.

[23] M. Prabhu *et al.*, “Functional test generation for hard to detect stuck-at faults using RTL model checking,” in *ETS*, 2012.

[24] T. Faller *et al.*, “Constraint-Based Automatic SBST Generation for RISC-V Processor Families,” in *ETS*, 2023.

[25] R. Vollertsen, “Burn-In,” in *IIRW*, 1993.

[26] M. Pietzsch, “RISC-V Processor for Network Platforms According to ISO 26262,” *ATZelectronics worldwide*, vol. 16, no. 11, pp. 8–13, 2021.

[27] N. I. Deligiannis *et al.*, “Automating the Generation of Programs Maximizing the Repeatable Constant Switching Activity in Microprocessor Units via MaxSAT,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 2023.

[28] N. I. Deligiannis *et al.*, “Automating the Generation of Programs Maximizing the Sustained Switching Activity in Microprocessor units via Evolutionary Techniques,” *Microprocessors and Microsystems*, vol. 98, 2023.

[29] I. Polian *et al.*, “Exploring the Mysteries of System-Level Test,” in *ATS*, 2020, pp. 1–6.

[30] S. Biswas *et al.*, “An Industrial Study of System-Level Test,” *IEEE Des. Test. Comput*, vol. 29, no. 1, pp. 19–27, 2012.

[31] H. H. Chen, “Beyond structural test, the rising need for system-level test,” in *VLSI – DAT*, 2018, pp. 1–4.

[32] D. Schwachhofer *et al.*, “Automating Greybox System-Level Test Generation,” in *ETS*, 2023.

[33] F. Corno *et al.*, “Automatic Test Program Generation for Pipelined Processors,” in *SAC*, 2003, pp. 736–740.

[34] A. Zeller *et al.*, *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021.

[35] P.-F. Chiu *et al.*, “An Out-of-Order RISC-V Processor with Resilient Low-Voltage Operation in 28nm CMOS,” in *Proc. IEEE Symp. VLSI Circuits*, 2018, pp. 61–62.

[36] C. Gaisler, *Quad core LEON4 SPARC V8 processor - LEON4-NGMP-DRAFT - data sheet and users manual*, 2011.

[37] Infineon, *AURIX Multicore 32-bit Microcontroller Family to Meet Safety and Powertrain Requirements of Upcoming Vehicle Generations*.

[38] X. Iturbe *et al.*, “Addressing Functional Safety Challenges in Autonomous Vehicles with the Arm Triple Core Lock-Step (TCLS) Architecture,” *IEEE Design and Test*, vol. PP, no. 99, pp. 1–1, 2018.

[39] A. Dörflinger *et al.*, “ECC Memory for Fault Tolerant RISC-V Processors,” in *ARCS*, 2020, pp. 44–55.

[40] E. B. Annink *et al.*, “Preventing Soft Errors and Hardware Trojans in RISC-V Cores,” in *DFT*, 2022, pp. 1–6.

[41] M. Peña-Fernández *et al.*, “Dual-Core Lockstep enhanced with redundant multithread support and control-flow error detection,” *Microelectronics Rel.*, vol. 100, p. 113 447, 2019.

[42] F. Kempf *et al.*, “An Adaptive Lockstep Architecture for Mixed-Criticality Systems,” in *ISVLSI*, 2021, pp. 7–12.

[43] X. Iturbe *et al.*, “The Arm Triple Core Lock-Step (TCLS) Processor,” *TOCS*, vol. 36, no. 3, pp. 1–30, 2019.

[44] I. Marques *et al.*, “Lock-V: A heterogeneous fault tolerance architecture based on Arm and RISC-V,” *Microelectronics Rel.*, vol. 120, p. 114 120, 2021.

[45] A. Benso *et al.*, *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Springer Science & Business Media, 2003, vol. 23.

[46] L. Pintard *et al.*, “Fault Injection in the Automotive Standard ISO 26262: An Initial Approach,” in *EWDC*, 2013, pp. 126–133.

[47] J. Abella et al., “WCET Analysis Methods: Pitfalls and Challenges on their Trustworthiness,” in *SIES*, 2015, pp. 1–10.

[48] M. S. et al., “T-CREST: Time-predictable multi-core architecture for embedded systems,” *J. Syst. Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[49] H. Yun *et al.*, “MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms,” in *RTAS*, 2013, pp. 55–64.

[50] M. Chisholm *et al.*, “Cache Sharing and Isolation Tradeoffs in Multicore Mixed-Criticality Systems,” in *RTSS*, 2015, pp. 305–316.

[51] Cobham Gaisler, *NOEL-V Processor*, https://www.gaisler.com/index.php/products/processors/noel-v, 2020.

[52] C. Hernàndez *et al.*, “SELENE: Self-Monitored Dependable Platform for High-Performance Safety-Critical Systems,” in *DSD*, 2020, pp. 370–377.

[53] F. Bas *et al.*, “SafeDE: a flexible Diversity Enforcement hardware module for light-lockstepping,” in *IOLTS*, 2021, pp. 1–7.

[54] G. Cabo *et al.*, “SafeSU: an Extended Statistics Unit for Multicore Timing Interference,” in *ETS*, 2021, pp. 1–4.

[55] R. Ramsauer *et al.*, “Static Hardware Partitioning on RISC-V – Shortcomings, Limitations, and Prospects,” arXiv:2208.02703, 2022.

[56] *PULP Platform*, https://pulp-platform.org/.

[57] M. Rogenmoser *et al.*, “Hybrid Modular Redundancy: Exploring Modular Redundancy Approaches in RISC-V Multi-Core Computing Clusters for Reliable Processing in Space,” arXiv:2303.08706, 2023.

[58] A. Walsemann *et al.*, “STRV – a radiation hard RISC-V microprocessor for high-energy physics applications,” *J. of Instrum.*, vol. 18, no. 02, p. C02032, 2023.

[59] U. Banerjee *et al.*, “An energy-efficient reconfigurable DTLS cryptographic engine for securing Internet-of-Things applications,” *IEEE J. of Solid-State Circuits*, vol. 54, no. 8, pp. 2339–2352, 2019.

[60] U. Banerjee *et al.*, “Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols,” *TCHES*, vol. 2019, no. 4, pp. 17–61,

[61] W. Wang *et al.*, “An energy-efficient crypto-extension design for RISC-V,” *Microelectronics J.*, vol. 115, p. 105 165, 2021.

[62] A. Zgheib *et al.*, “Extending a RISC-V core with an AES hardware accelerator to meet IOT constraints,” in *SMACD / PRIME*, 2021, pp. 1–4.

[63] B. Marshall *et al.*, “The design of scalar AES Instruction Set Extensions for RISC-V,” *TCHES*, vol. 2021, no. 1, pp. 109–136,

[64] D. B. Roy *et al.*, “Efficient Hardware/Software Co-Design for Post-Quantum Crypto Algorithm SIKE on ARM and RISC-V Based Microcontrollers,” in *ICCAD*, 2020.

[65] R. Elkhatib *et al.*, “Accelerated RISC-V for Post-Quantum SIKE,” *IEEE Trans. on Circuits and Syst. I: Regular Papers*, vol. 69, 2022.

[66] T. Fritzmann *et al.*, “Towards Reliable and Secure Post-Quantum Co-Processors based on RISC-V,” in *DATE*, 2019, pp. 1148–1153.

[67] E. Alkim *et al.*, “ISA Extensions for Finite Field Arithmetic: Accelerating Kyber and NewHope on RISC-V,” *TCHES*, vol. 2020, no. 3, pp. 219–242, 2020.

[68] T. Fritzmann *et al.*, “Extending the RISC-V Instruction Set for Hardware Acceleration of the Post-Quantum Scheme LAC,” in *DATE*, 2020, pp. 1420–1425.

[69] G. Xin *et al.*, “VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture,” *IEEE Trans. on Circuits and Syst. I: Regular Papers*, vol. 67, no. 8, pp. 2672–2684, 2020.

[70] H. Li *et al.*, “A scalable SIMD RISC-V based processor with customized vector extensions for CRYSTALS-Kyber,” in *DAC*, 2022, pp. 733–738.

[71] Y.-M. Kuo *et al.*, “Versatile RISC-V ISA Galois Field arithmetic extension for cryptography and error-correction codes,” in *CARRV*, 2021.

[72] T. Fritzmann *et al.*, “RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography,” *TCHES*, vol. 2020, no. 4, pp. 239–280,

[73] H. Cheng *et al.*, “RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography,” *TCHES*, vol. 2023, no. 1, pp. 193–237,

[74] S. Steinegger *et al.*, “A Fast and Compact RISC-V Accelerator for Ascon and Friends,” in *CARDIS*, 2020, pp. 53–67.

[75] B. Marshall, Ed., *The RISC-V Cryptography Extensions, Volume I: Scalar & Entropy Source Instructions*. RISC-V Foundation, Feb. 2022.

[76] K. Dockser, Ed., *The RISC-V Cryptography Extensions, Volume II: Vector Instructions*. RISC-V Foundation, Mar. 2023.

[77] P. Kiaei *et al.*, *Domain-Oriented Masked Instruction Set Architecture for RISC-V*, Cryptology ePrint Archive, Paper 2020/465, 2020.

[78] S. Gao *et al.*, “An Instruction Set Extension to Support Software-Based Masking,” *TCHES*, vol. 2021, no. 4, pp. 283–325,

[79] B. Gigerl *et al.*, “COCO: Co-Design and Co-Verification of Masked Software Implementations on CPUs,” in *USENIX*, 2021, pp. 1469–1468.

[80] T. Lu, “A survey on RISC-V security: Hardware and Architecture,” arXiv:2107.04175, 2021.

[81] V. Costan *et al.*, “Sanctum: Minimal Hardware Extensions for Strong Software Isolation,” in *USENIX*, 2016, pp. 857–874.

[82] I. Lebedev *et al.*, “Invited Paper: Secure Boot and Remote Attestation in the Sanctum Processor,” in *CSF*, 2018, pp. 46–60.

[83] D. Lee *et al.*, “Keystone: An Open Framework for Architecting Trusted Execution Environments,” in *EuroSys*, 2020, pp. 1–16.

[84] T.-T. Hoang *et al.*, “Quick Boot of Trusted Execution Environment With Hardware Accelerators,” *IEEE Access*, vol. 8, pp. 74 015–74 023, 2020.

[85] V. B. Kumar *et al.*, “Towards Designing a Secure RISC-V System-on-Chip: ITUS,” *J. of Hardware and Syst. Secur.*, vol. 4, pp. 329–342, 2020.

[86] M. Schneider *et al.*, “Composite Enclaves: Towards Disaggregated Trusted Execution,” arXiv:2010.10416, 2020.