2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

Pre-Silicon Verification of Software Safety Mechanisms

A Hybrid Approach with SPI and NVDLA Case Studies

Moustafa Nabil, Siemens DISW, Cairo, Egypt (moustafa.nabil@siemens.com)
Ahmed Saied, Siemens DISW, Cairo, Egypt (ahmed.saied@siemens.com)
Ahmed Makram, Siemens DISW, Cairo, Egypt (ahmed.makram@siemens.com)
Mohamed Nasser, Siemens DISW, Cairo, Egypt (mohamed.nasser@siemens.com)
Sherif Khourshed, Siemens DISW, Cairo, Egypt (sherif.khourshed@siemens.com)

***Abstract*— Verifying software safety mechanisms for peripheral IPs presents a significant challenge during pre-silicon development, as these mechanisms require the entire System-on-Chip to execute. A software safety mechanism is a piece of software that detects hardware faults. This paper presents a novel hybrid verification approach that addresses this challenge by combining hardware emulation with software virtualization to enable comprehensive fault injection campaigns before silicon availability.**

Our architecture interfaces an RTL emulator running peripheral IP blocks with a host machine executing a virtualized CPU environment through a transaction-level interface. This hybrid approach provides signal-level accuracy while delivering practical execution times for fault campaigns. We demonstrate its effectiveness through two case studies: a memory-mapped SPI controller and the more complex NVIDIA Deep Learning Accelerator (NVDLA).

For NVDLA, we implemented a progressive multiple-point fault injection methodology, revealing that neural accelerators demonstrate moderate vulnerability to single-point fault injection (10% accuracy loss), and become increasingly susceptible with multiple simultaneous faults injected (95% accuracy loss with 32 faults). Experimental results show the platform achieves 10× speedup over fully-emulated environments and 1000× acceleration compared to RTL simulation approaches, while utilizing only 0.12% of available resources. For the SPI controller, combining safety mechanisms achieved up to 96.3% fault detection rate, which aligns with ISO 26262 estimation for safety mechanisms. The methodology enables early vulnerability detection and refinement of safety mechanisms, supporting the timely development of ASIL-compliant automotive systems.

Keywords—Software Safety Mechanisms, Fault Injection, Multiple-Point Faults, RTL, SoC, Emulator, QEMU, Hybrid Verification, ISO 26262, NVDLA, CNN accelerator, SPI.

I. INTRODUCTION

The increasing complexity of electronic and electrical systems in the automotive industry has necessitated stringent safety standards to ensure reliability and to manage the risk of random hardware failures. The ISO 26262 standard, "Road vehicles - Functional safety," provides a framework for managing these risks through systematic verification of safety mechanisms [1]. A critical aspect of this verification process for safety mechanisms is Fault Injection (FI), which assesses how electronic systems tolerate random hardware faults that could compromise safety-critical functions. Fault injection creates a gap in the verification workflow; hence, developers must find innovative ways to thoroughly verify software safety mechanisms for peripheral IPs before the entire System on Chip (SoC) is available. This paper addresses this gap by proposing a novel hybrid verification approach that combines hardware emulation with software virtualization. Our methodology enables comprehensive fault injection campaigns on peripheral IP blocks while the rest of the SoC is simulated in a virtual environment, providing both accuracy and performance advantages over existing approaches. Our research addresses three key questions: (1) How can fault injection campaigns be effectively performed on peripheral IPs during pre-silicon development? (2) Can a hybrid hardware-software approach provide sufficient speed and accuracy for ISO 26262 verification? (3) What performance improvements can be achieved compared to existing methods?

Key contributions of our study are:

1) A novel hybrid verification architecture combining hardware emulation for the Device Under Test (DUT) with software virtualization for the remaining SoC components.
2) Two case studies validating our approach for different IP types: a memory-mapped SPI controller and the NVIDIA Deep Learning Accelerator (NVDLA).
3) Novel insights on neural network accelerator resilience patterns, demonstrating inherent tolerance to isolated faults but increasing vulnerability with multiple concurrent faults.

The remainder of this paper is organized as follows: Section II reviews existing fault injection methods and their limitations. Section III provides background on ISO 26262 safety requirements and metrics. Section IV presents our hybrid verification methodology. Sections V and VI describe our case studies and performance evaluation, respectively. Section VII discusses our findings, and the last section concludes with a summary of contributions and future work.

II. FAULT INJECTION

Fault injection deliberately introduces faults into systems to evaluate resilience and the effectiveness of safety mechanisms. [2] nicely described the history of fault injection techniques, which we categorize as follows:

A. Hardware-Implemented Fault Injection (HWIFI)

Hardware-Implemented Fault Injection applies faults to physical chips using instrumentation that emulates real-world fault sources like radiation or electromagnetic interference [3], [4]. While providing high fidelity, Hardware-Implemented Fault Injection requires manufactured silicon, making it unsuitable for pre-silicon verification, and incurring high costs and potential component damage.

B. Software-Implemented Fault Injection (SWIFI)

Software-Implemented Fault Injection mimics hardware faults by modifying the software during compilation [5] or runtime [6], primarily targeting CPU behavior through instruction or memory corruption. Though less costly than Hardware-Implemented Fault Injection, Software-Implemented Fault Injection requires functional hardware and inadequately models peripheral IP failures.

C. Simulation-Based Fault Injection (SFI)

Simulation-Based Fault Injection tries to create system models for fault injection before silicon availability. It can be divided into two categories:

1) Software Model-Based SFI

Uses behavioral models to emulate hardware, such as QEMU [7] for CPU emulation. Fault injection occurs through debugger interfaces [8] or API hooks [9]. ARMORY [10] provides an alternative model for ARM architecture that is not Qemu based. While offering fast execution, these approaches have key limitations: (1) Peripheral software models are simplified approximations that do not accurately represent hardware implementations, and (2) Fault propagation pathways may differ from actual hardware, leading to inaccurate safety assessments. These limitations make such models insufficient for ISO 26262 compliance verification.

2) HDL Model-Based SFI

Uses hardware description models at various abstraction levels. RTL models balance accuracy and performance but traditional simulation [11], [12], [13], [14] is prohibitively slow for comprehensive fault campaigns (It is the process of applying fault injection and measure how the RTL can tolerate these faults). FPGA-based acceleration [15, 16] improves speed but faces capacity constraints and requires specialized hardware modules.

Our hybrid approach addresses these limitations by combining hardware emulation for peripheral IPs with software virtualization for the system on chip (SoC). Key advantages of emulators like [17] over FPGA-based approaches include: (1) 100-1000× greater capacity for complex IP verification. (2) Non-invasive access to internal signals without design modifications. And (3) Fault injection through standard APIs without specialized hardware.

III. ISO 26262 BACKGROUND

This standard addresses safety throughout the lifecycle of critical electronic systems, from specification through development to decommissioning. During the electronic system development phase, the standard outlines a flow that consists of three steps: Analysis, Insertion and verification. But first we need to mention the following definitions:

1) Base Failure Rate (BFR): Quantifies random failures per billion operation hours. Standards like IEC 62380 [18] provide equations to estimate BFR using parameters such as transistor count.
2) Safety Mechanisms: Protect systems against random faults. They can be hardware safety mechanisms like parity logic, or software safety mechanisms like software test patterns. ISO 26262 part 5 provides examples of safety mechanisms and their diagnostic coverage estimates.
3) Diagnostic Coverage (DC): Measures safety mechanism effectiveness. For example, 90% DC indicates detection of 90% of possible random faults in the protected area. ISO 26262 uses DC with other metrics to evaluate the effectiveness of safety mechanisms.

A. Safety Flow

The safety flow runs parallel to SoC design through three main phases:

1) Safety Analysis

Standards [18, 19, 20] and tools like Questa One Safety Analyzer [21] estimate the base failure rate (BFR) before full RTL availability. Using Questa One Safety Analyzer [21], designers keep annotating RTL with planned safety mechanisms then recalculate safety metrics, while the RTL development is progressing. Safety mechanisms can be suggested on a small part like using parity bits to protect GPU registers, or can be a system level safety mechanism (e.g., a CRC module that helps the CPU to check the CRC of Ethernet packets). Safety analysis is an iterative process that continues until safety metrics meet requirements.

2) Safety Insertion

Teams implement the specified safety mechanisms; hardware safety mechanisms in RTL and software safety mechanisms in software (particularly in device drivers).

3) Safety Verification

Verification uses fault injection appropriate to the safety mechanism type. Hardware safety mechanisms can be verified with traditional module-level testbenches. However, software safety mechanisms (like CRC checking in Ethernet controller driver software) require the CPU and peripheral to function together, which typically only occurs near project completion when the entire SoC is integrated.

B. Hybrid Approach for Software Safety Mechanisms Verification

Software safety mechanisms verification presents unique challenges compared to hardware mechanisms, requiring connection to an entire SoC. Our proposed hybrid method combines virtual CPUs' flexibility with hardware emulation's speed, enabling accurate, high-speed fault injection campaigns by running RTL on an emulator while executing software on a virtual CPU.

IV. HYBRID VERIFICATION METHODOLOGY

Our methodology combines hardware emulation for peripherals with software virtualization for system components, creating an efficient verification environment for software safety mechanisms.

This hybrid approach provides signal-level accuracy with practical execution times for comprehensive fault campaigns.

A. System Architecture

As illustrated in the block diagram in Fig. 1, our architecture comprises two primary components connected through a transaction-level interface:

1) Emulator Environment

The emulator runs the RTL design with complete signal visibility and debugging capabilities. The RTL undergoes standard elaboration, synthesis, and placement processes. The emulator integrates an Extended RTL (XRTL) component that converts transaction objects from the Device Under Test (DUT) and connects to the AXI transactor.

2) Host Machine Environment

The host machine runs a virtualized Linux environment using QEMU, which emulates a virtual CPU and includes a complete software stack comprising device drivers that implement software safety mechanisms and the software applications for testing.

The host also runs a main control program (main.cpp) that interfaces with the RTL emulator via C++ APIs, enabling signal-level operations such as read, write, and force, as well as SSH connectivity to the virtual machine. The QEMU instance connects to the RTL emulator through the Innexis Developer Pro [22], which acts as an AXI transactor, bridging the software stack on the host with the hardware-level simulation in the emulator.

The Standard Co-Emulation Modeling Interface (SCEMI) [23] provides transaction-level communication between these environments, enhancing performance while maintaining fidelity. SCEMI is a standardized API developed by Accelera for connecting SystemVerilog or C/C++ testbenches to hardware emulators or FPGA prototypes. It enables transaction-level communication between host software and the design-under-test (DUT) running on hardware. The block diagram in Fig. 1 represents this entire system, showing the connections between the Emulator, Host Machine, and main program.

B. Fault Campaign Process

Our methodology follows a systematic four-step process for preparing and conducting fault verification campaigns, designed with reusability in mind to support diverse DUTs and software stacks with minimal integration effort.

1) Fault List Generation: Using Questa One Safety Analyzer [21], we analyze the RTL design to identify critical signals and registers for fault injection. This tool generates a comprehensive fault list optimized for maximum coverage with minimal simulation time.
2) DUT Integration: We integrate the peripheral IP with the AXI transactor on the emulator side, enabling communication with the virtual platform.
3) Software Stack Implementation: We develop device drivers implementing the software safety mechanisms and testbench applications on the virtual platform, creating a realistic software environment that exercises the DUT.
4) System Integration: After completing the previous steps, we pass the Fault List path and Software Testbench path to the main program, allowing for the execution of the entire system.

C. Runtime Operation

During execution, the main program operates according to the algorithm in Listing 1, which orchestrates the fault injection process. After connections to both the emulator and virtual machine are established, the algorithm proceeds with fault injection as defined.

```
Algorithm FaultInjection
    INPUTS :
        1. FaultList path
        2. Software testbench program path on Qemu
    1. Wait for Linux booting
    2. GoldenRun(Run SW TB)
    3. while not at the end of Faultlist do:
        (a) Reset the design
        (b) Inject fault
        (c.1) Advance clock cycles
        (c.2) Run SW TB
        (d) Remove fault
        (e) Compare with Golden Run
    4. terminate Qemu and the Emulator
```

Listing 1: Fault Injection Algorithm

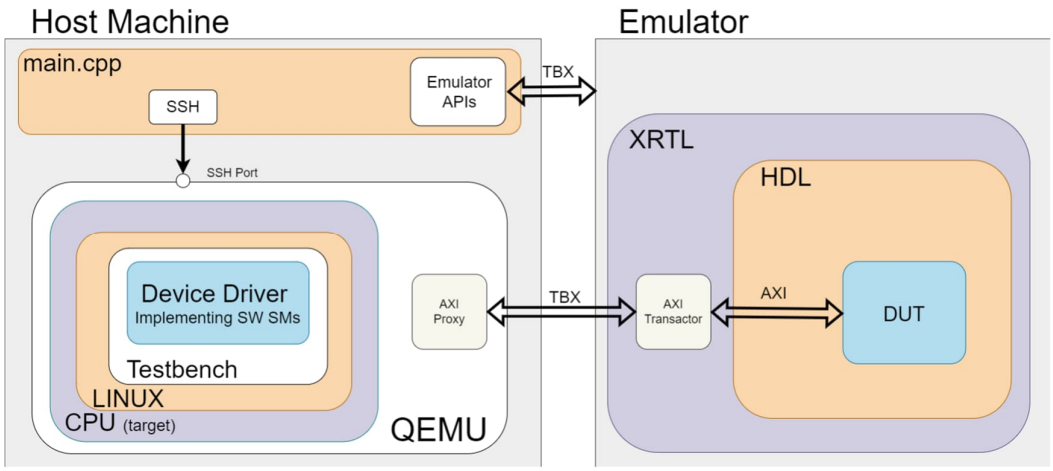

Figure 1: Hybrid Environment

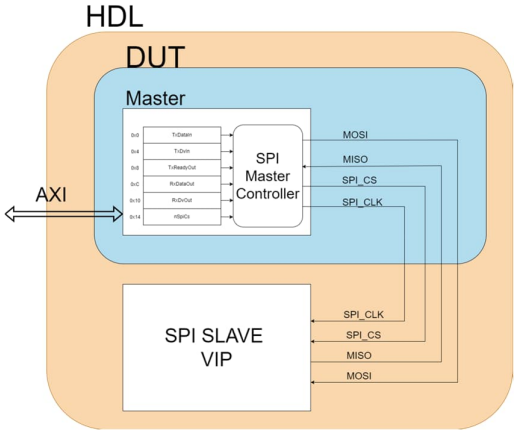

Figure 2: SPI Architecture Diagram

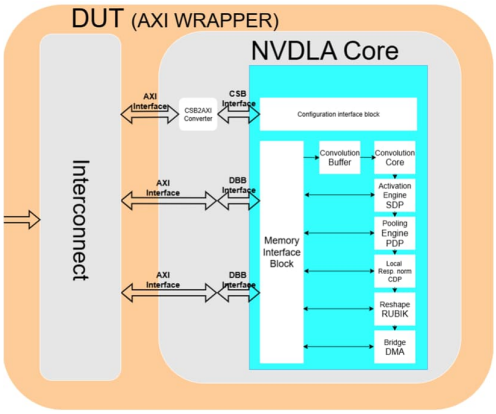

Figure 3: NVDLA Architecture Diagram

```
tbench.top.uIP.M_SPI_AXI.SPI_Master.RxByteOut[7]
tbench.top.uIP.M_SPI_AXI.SPI_Master.RxDvOut
tbench.top.uIP.M_SPI_AXI.M_SPI_REG.TxDVInReg
```

Listing 2: Excerpt from the SPI fault list

```
NV_NVDLA_cmac.u_rt_out.mac2accu_data[307:0]
NV_NVDLA_sdp.u_NV_NVDLA_SDP_cmux.cmux_pd[511:0]
NV_NVDLA_pdp.pdp_dp2wdma_pd[63:0]
NV_NVDLA_cbuf.cdma2buf_wt_wr_data[511:0]
```

Listing 3: Excerpt from NVDLA Fault List

V. Case Study: Memory Mapped SPI Controller

To demonstrate the effectiveness of our hybrid approach, we implemented and verified a memory-mapped Serial Peripheral Interface (SPI) master controller as our first case study.

A. SPI Controller Architecture and Integration

As shown in Fig. 2, the SPI Master controller features dual interfaces: a standard SPI interface for external communication and an AXI interface for processor interaction—a common embedded system peripheral configuration ideal for validating our approach.

Following our methodology outlined in Section IV, we integrated the SPI controller into our verification environment through:

1) RTL Integration: The SPI master DUT connected to the AXI transactor enabling communication with the virtualized CPU, and simultaneously to an SPI Slave Verification IP for protocol validation Fig. 1.
2) Fault List Generation: Using the Questa One Safety Analyzer on our SPI controller, we identified 41 critical signals covering both control paths and data elements. We performed stuck-at-zero and stuck-at-one fault injections, creating 82 fault scenarios. Compared to real world designs, our fault list is relatively small. Because we implemented a small SPI controller for demonstration purposes. A representative excerpt from our fault list is shown in Listing 2.
3) Software Stack Development: We built a Linux environment with Buildroot providing UIO driver support for direct device memory access. The device tree exposed the SPI controller's address space, enabling direct hardware interaction from our software application.
4) Test Orchestration: A main control program orchestrated the fault injection process Listing 1, establishing connections to both emulator and virtual machine. The host machine communicates with the emulator's APIs using SSH to execute golden runs and inject faults while monitoring system behavior.

B. Software Safety Mechanisms Implementation

We implemented three software safety mechanisms in the SPI driver:

1) Cyclic Redundancy Check (CRC): A 1-byte CRC validates each 20-byte transaction, detecting data corruption but with limited control register fault coverage.
2) Read-After-Write (RAW) Verification: The master compares reflected data from the slave against transmitted data, identifying data path integrity issues.
3) Timeout Detection: Monitors transaction completion times, detecting blocking faults in control logic.

Following ISO 26262 recommendations for comprehensive coverage, we evaluated strategic combinations (CRC+Timeout, RAW+Timeout, CRC+RAW, and all three together), demonstrating how combined mechanisms significantly enhance diagnostic coverage across multiple fault domains—critical for automotive-grade safety systems.

VI. NVDLA Case Study

A. Background and Motivation

We selected NVDLA [24] for our second case study as it represents a significantly more complex peripheral than the SPI controller, with extensive computational blocks and internal state. Previous work by [16] demonstrated CNN fault injection using FPGAs but required hardware modifications to access internal signals. Unlike FPGA-based approaches, our hybrid platform enables comprehensive fault injection for NVDLA without design modifications, providing a realistic verification environment for automotive functional safety assessment with complete signal visibility.

B. Hardware-Software Co-Integration of NVDLA

We integrated the NVDLA into our framework following the methodology in Section IV, connecting its ports (2 masters and 1 slave) to an interconnect as shown in Fig. 3, while QEMU was connected to an additional master port in the same interconnect through the Innexis Developer Pro.

For the software part, we installed the device driver for the NVDLA and the runtime software provided by NVIDIA on the virtualized Linux environment.

Our platform executed inference workloads (ResNet18-CIFAR10) while injecting faults at points identified by the safety analysis tool [21] in the design. As shown in Table 1, the synthesis of NVDLA resulted in 726,308 in Lockup Tables (LUTs) which is only (0.12%) of StratoM [25] system capacity, achieving acceptable inference performance with negligible fault injection overhead.

C. *Fault Injection Campaign for NVDLA*

For safety verification, we targeted critical signals in the NVDLA RTL responsible for convolution, pooling, and activation functions as shown in Listing 3.

We implemented a software test pattern safety mechanism comparing inference results against golden references.

D. *Multiple-Point Fault Injection*

Our initial experiments with NVDLA revealed a baseline vulnerability to single-point faults, with approximately 10% accuracy degradation observed when faults were injected into critical signals. To characterize fault susceptibility under more realistic scenarios, we implemented a progressive multiple-point approach, systematically increasing fault density by simultaneously injecting 4, 8, 16, and 32 faults during inference runs.

VII. Experimental Results

We evaluated our hybrid verification methodology through two case studies: a memory-mapped SPI controller and NVIDIA's Deep Learning Accelerator, demonstrating our approach's effectiveness for different classes of peripheral IPs. We used the StratoM [25] emulator setup, and for simulation results we used Questasim [14].

A. *Fault Coverage Analysis*

Table 2 summarizes results for the SPI controller, showing detection rates for each safety mechanism. While individual mechanisms provided moderate coverage (CRC: 48.7%, RAW: 65.0%, Timeout: 44.0%), combining mechanisms significantly improved results. The Timeout+CRC and Timeout+RAW combinations achieved the highest detection rates (91.4% and 96.3% respectively), demonstrating the effectiveness of complementary safety mechanisms as recommended by ISO 26262.

For the NVDLA, we executed a comprehensive fault injection campaign. Our approach included both single-point and multiple-point scenarios with concurrent faults in 4, 8, 16, and 32 signals simultaneously. Table 3 shows the detection rates under multiple-point fault injection.

As illustrated in Fig. 4, NVDLA shows moderate resilience to isolated faults with a 10% detection rate for single-point failures, while demonstrating more significant inference accuracy degradation as fault density increases.

B. *Performance Evaluation*

Fig. 5 demonstrates the significant performance advantages of our hybrid verification approach. We conducted comparative benchmarking by executing identical inference workloads across three environments: our hybrid platform, a fully emulated SoC implementation, and a fully simulated SoC. Performance measurements reveal that our hybrid methodology achieves a 10× acceleration compared to full emulation and a 1000× speedup versus traditional simulation approaches. These substantial performance gains enable comprehensive fault injection campaigns to be completed within practical development timeframes, effectively addressing the execution time constraints that typically limit pre-silicon safety verification efforts.

Table 1 compares platform characteristics and inference performance. While higher clock speeds on the Jetson Xavier and Tesla T4 deliver faster inference times, our hybrid platform excels in pre-silicon verification capabilities. The FPGA implementation reaches 35.39% utilization with just the NVDLA small variant, limiting scalability, while our emulator-based approach utilizes only (0.12% without fault injection, 0.68% with fault injection) of available resources for the same design.

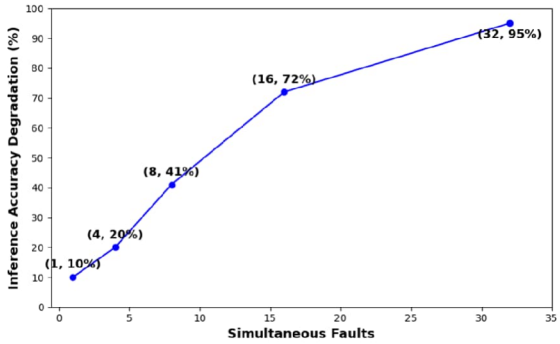

Figure 4: Accuracy Degradation vs Fault Injection Density

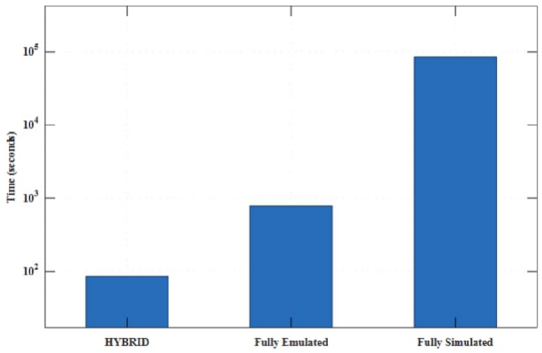

Figure 5: Runtime Performance Across Verification Platforms

The emulator transparently synthesizes additional Lockup Tables (LUTs) for fault injection capabilities without requiring manual instrumentation, preserving design integrity and eliminating the risk of inadvertently modifying circuit behavior.

CONCLUSION

This paper presented a novel hybrid verification methodology combining hardware emulation of peripheral IPs with software virtualization of CPU subsystems for comprehensive verification of software safety mechanisms.

Our approach addresses a critical gap in pre-silicon safety verification, enabling thorough testing under realistic fault conditions before hardware availability.

Results from our case studies—a memory-mapped SPI controller and NVIDIA's Deep Learning Accelerator—show significant advantages over traditional approaches: 10× speedup compared to fully emulated environments and 1000× acceleration over simulation-based approaches. The platform provides complete signal visibility and non-invasive fault injection without requiring RTL modifications.

The emulation-based approach utilizes only 0.12% of available resources for implementing NVDLA compared to 35.39% on FPGA platforms, enabling verification of increasingly complex peripherals required in modern automotive systems. Our analysis confirmed ISO 26262 recommendations on complementary safety mechanisms, with combined approaches achieving significantly higher detection rates than individual mechanisms.

Future work will focus on enhancing our hybrid verification methodology, expanding coverage of ISO 26262 safety mechanisms, and conducting more in-depth fault injection analysis for neural network accelerators. As automotive systems grow in complexity, robust pre-silicon verification approaches become increasingly critical—addressing safety requirements earlier in development cycles reduces costly silicon iterations and accelerates time-to-market while ensuring functional safety compliance.

Table 1: Comparison of platform characteristics and inference performance

Platform	LUTs	Util. (%)	Clock (MHz)	Inference Time
Hybrid	726,308	0.12	1	1m 24.674s
Hybrid + FI	4,266,660	0.68	1	1m 24.674s
FPGA SoC[a]	77,373	35.39	100	36.53ms
Jetson Xavier	N/A	N/A	1,400	2.65ms
AMD Ryzen 7	N/A	N/A	3,800	11.57ms
Tesla T4	N/A	N/A	585	2.26ms

[a] FPGA SoC = ZYNQ 7000 [16]
All platforms run ResNet18 on CIFAR10 dataset

Table 2: Faults detected with each safety mechanism

SM	Faults Injected	Faults Detected	Detection (%)
CRC	82	40	48.7
RAW	82	53	65.0
Timeout	82	36	44.0
Timeout+CRC	82	75	91.4
Timeout+RAW	82	79	96.3

Table 3: NVDLA inference accuracy degradation under multiple-point fault injection

Simultaneous Faults	Faults Injected	Faults With Impact. (Observed Faults)	Inference Accuracy Degradation (%)
1	470	45	10
4	357	24	20
8	181	76	41
16	90	65	72
32	45	43	95

REFERENCES

[1] ISO, ISO 26262-1:2018 Road vehicles, Functional safety. Part 1: Vocabulary, Available: https://www.iso.org/standard/68383.html

[2] Y. B. Bekele, D. B. Limbrick and J. C. Kelly, A Survey of QEMU-Based Fault Injection Tools & Techniques for Emulating Physical Faults, IEEE Access, vol. 11, pp. 62662-62673, 2023, doi: 10.1109/ACCESS.2023.3287503

[3] U. Gunneflo, J. Karlsson and J. Torin, Evaluation of error detection schemes using fault injection by heavy-ion radiation, [1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers, Chicago, IL, USA, 1989, pp. 340-347, doi: 10.1109/FTCS.1989.105590

[4] G. Miremadi and J. Torin, Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection, IEEE Transactions on Reliability, vol. 44, no. 3, pp. 441-454, Sept. 1995, doi: 10.1109/24.406580

[5] J. Aidemark, J. Vinter, P. Folkesson and J. Karlsson, GOOFI: generic object-oriented fault injection tool, 2001 International Conference on Dependable Systems and Networks, Gothenburg, Sweden, 2001, pp. 83-88, doi: 10.1109/DSN.2001.941394

[6] G. A. Kanawati, N. A. Kanawati and J. A. Abraham, FERRARI: a tool for the validation of system dependability properties, [1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing, Boston, MA, USA, 1992, pp. 336-344, doi: 10.1109/FTCS.1992.243567

[7] QEMU, QEMU, Available: https://www.qemu.org/

[8] A Fault Injection Simulator for ARM, A Major Qualifying Project submitted to the Faculty of WORCESTER POLYTECHNIC INSTITUTE in partial fulfilment of the requirements for the degree of Bachelor of Science, Available: https://web.cs.wpi.edu/claypool/mqp/sv/2018/nvidia-sw/

[9] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann and O. Spinczyk, FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance, 2015 11th European Dependable Computing Conference (EDCC), Paris, France, 2015, pp. 245-255, doi: 10.1109/EDCC.2015.28

[10] M. Hoffmann, F. Schellenberg and C. Paar, ARMORY: Fully Automated and Exhaustive Fault Simulation on ARM-M Binaries, IEEE Transactions on Information Forensics and Security, vol. 16, pp. 1058-1073, 2021, doi: 10.1109/TIFS.2020.3027143

[11] Veripool, Verilator, Available: https://www.veripool.org/verilator/

[12] Synopsys, VCS, Available: https://www.synopsys.com/verification/simulation/vcs.html

[13] Cadence, Xcelium Simulator, Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html

[14] Siemens Digital Industries Software, Questa Advanced Verification, Available: https://eda.sw.siemens.com/en-US/ic/questa/ (accessed Nov 19, 2024)

[15] O. Ruano, F. García-Herrero, L. Aranda, A. Sanchez-Macian, L. Rodríguez and J. A. Maestro, Fault Injection Emulation for Systems in FPGAs: Tools, Techniques and Methodology, a Tutorial, Sensors, vol. 21, 1392, 2021, doi: 10.3390/s21041392

[16] F. Masar, V. Mrazek and L. Sekanina, Late Breaking Result: FPGA-Based Emulation and Fault Injection for CNN Inference Accelerators, 2025 Design, Automation & Test in Europe Conference (DATE), Lyon, France, 2025, pp. 1-2, doi: 10.23919/DATE64628.2025.10992992

[17] Siemens Digital Industries Software, Veloce CS system - IC emulation and prototyping, Available: https://eda.sw.siemens.com/en-US/ic/veloce/ (accessed Nov 19, 2024)

[18] IEC, IEC 62380 Standard, First edition 2004-08, Reliability data handbook - Universal model for reliability prediction of electronics equipment, PCBs and equipment

[19] Siemens, Siemens SN 29500-2 (2010-09), Failure rates of components Part 2: Expected values for integrated circuits

[20] IEC, IEC 61709 Standard, Edition 3.0 (2017-02). Electric components - Reliability - Reference conditions for failure rates and stress models for conversion

[21] Siemens, Questa One Safety Analyzer, Available: https://resources.sw.siemens.com/en-US/fact-sheet-questa-one-safety-analyzer/ (accessed Jul 2, 2025)

[22] Siemens Digital Industries Software, Innexis Developer Pro, Available: https://eda.sw.siemens.com/en-US/ic/hav/innexis/developer-pro/ (accessed Jul 3, 2025)

[23] Accellera Systems Initiative, Standard Co-Emulation Modeling Interface (SCE-MI), Version 2.4, 2023, Available: https://www.accellera.org/downloads/standards/sce-mi

[24] NVIDIA, NVDLA: NVIDIA Deep Learning Accelerator, Available: https://nvdla.org/

[25] ElectronicSpecifier, Veloce Strato Platform Scales Up to 15BG, Available: https://www.electronicspecifier.com/news/analysis/veloce-strato-platform-scales-up-to-15bg