SAND2019-0073C

Fault testing a synthesizable embedded processor at gate level using FPGA emulation

Tom J. Mannos
Rad Hard/Trusted ASIC Products
Sandia National Labs
Albuquerque, NM 87185
Email: tjmanno@sandia.gov

Brian Dziki
Information Assurance Research
Department of Defense
Fort G. G. Meade, MD
Email: bjdziki@tycho.ncsc.mil

Moslema Sharif
Rad Hard/Trusted ASIC Products
Sandia National Labs
Albuquerque, NM 87185
Email: msharif@sandia.gov

***Abstract*— To characterize the fault behavior of a target CPU running application software, we developed an automated gate-level fault injection platform based on Ultrascale and Ultrascale+ development boards from Xilinx. Using a synthesizable saboteur circuit connected via scan chain and controlled with a state machine, we tested four different types of static faults on the LEON3 core running an AES 256 application, in under two hours, a 7200X speedup from gate-level simulation. Though neither the hardware nor software employed fault mitigation techniques, we were surprised to discover multiple static faults that resulted in total or partial key and plaintext leakage through the communications serial port. Overall, we identified 22 unique faulty behaviors, four of which involve some form of crypto or memory leakage. Of the four static fault types tested, delay faults were the most effective at stimulating the behaviors of concern.**

Keywords*— *Fault injection testing, FPGA emulation, ASIC prototyping, embedded processing, cryptography

I. Introduction

Safety and security critical systems rely on hardware and/or software mitigations to ensure critical components continue to operate in the presence of hardware faults [1]. Even unmitigated components within these systems call for a thorough understanding of their fault behavior and the conditions under which they fail safe (secure) or fail unsafe (unsecure) [2]. Many techniques exist to fault-test custom or TMR-mitigated circuits, such as fault simulation, formal fault analysis, and FPGA emulation. However, to our knowledge, these techniques have not been demonstrated for fault-testing unmitigated embedded processors running security critical software. In such systems, the concern is that a hardware fault, random or induced, will cause the software to fail in such a way that the security of the system is compromised, such as through key or plaintext leakage or weakened cryptography.

As a demonstration vehicle, we synthesized, placed and routed the LEON3 32-bit processor core (integer unit) [3] in an ASIC macro as our device-under-test (DUT). We programmed the DUT with AES256 test software configured to encrypt a short message and compare it against an expected ciphertext stored in memory. The software displays the encrypted message on the serial port in hexadecimal and indicates whether the comparison succeeded. It runs on bare metal and executes upon reset, so there is no external control required apart from resetting the processor between tests. Figure 1 shows the serial output of the self-test software when run on a fault-free processor.

```
 moving .text from 0x00001460 to 0x40000000
 moving .data from 0x00007d30 to 0x400068d0
txt:
00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF

key:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
---
enc:
8E A2 B7 CA 51 67 45 BF EA FC 49 90 4B 49 60 89

tst:
8E A2 B7 CA 51 67 45 BF EA FC 49 90 4B 49 60 89

Match
```

Figure 1. Expected serial output from running AES test software under fault-free condition

II. Prior Analysis

To establish a baseline, we performed gate-level simulation of the FPGA and ASIC netlists, systematically injecting stuck-at faults at each logic gate input within the integer unit and recording the serial output and the address of each external memory access. (We did not fault-test gate outputs in the interest of time.) The FPGA simulation was performed using Riviera Pro [4], and faults were inserted by modifying the netlist. The ASIC simulation was performed using Questa Core [5] with faults inserted using the parameterizable saboteur circuit of Figure 2. In each case, the LEON3 UART was replaced with a simulation module that printed the serial output to a file for evaluation. Each 35 ms simulation took an average time of 7 minutes to complete (10 minutes in the fault-free case), requiring over 30 days of total runtime with 4 simulations running in parallel.

```
module sabby #(
  parameter inject=0,
  parameter type=0
) (input in, output out);
  assign out = inject ? type : in;
  initial if (inject) $display("%m: injecting SA%0d", type);
endmodule
```

Figure 2. Parameterizable saboteur circuit used for gate-level simulation (behavioral Verilog)

To determine if formal analysis could help reduce the search space, we used a commercial fault propagation analysis tool [6] on the ASIC netlist. The tool marked only 2.4% of input faults

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

as "safe", and these were all inputs that had been tied to constant values. However, 12% of input faults were identified as not affecting the program counter, and these may be candidates for exclusion. We did not attempt to collapse the fault list by identifying equivalent faults, but this would be another way to reduce simulation time.

III. FPGA EMULATION

For FPGA emulation, we replaced the behavioral saboteur module with the synthesizable circuit of Figure 3, allowing faults to be injected via a scan chain that runs throughout the DUT. A global fault_type signal selects between four fault types: stuck-at-0, stuck-at-1, delayed, and inverted. For stuck-at-0 and stuck-at-1 the saboteur output is tied to a constant value, like in the simulation. For a delay fault, we insert a flip-flop between the saboteur input and its output, delaying the signal by one clock cycle. An inversion fault inverts the saboteur input, giving it the opposite logic value.

```
module sabby ( clk, in, si, sck, ft, out, so );
  input clk, in, si, sck;
  input [1:0] ft;
  output out, so;
  reg out, so;
  reg di;

  always @(posedge sck)
    so <= si;  // scan chain propagation

  always @(posedge clk)
    di <= in;  // delayed input

  always @(so or ft or in or di)
    if (so)
      case (ft)
        0: out = 0;   // stuck 0
        1: out = 1;   // stuck 1
        2: out = di;  // delayed
        3: out = !in; // inverted
      endcase
    else
      out = in;  // no fault
endmodule
```

Figure 3. Synthesizable saboteur circuit used for FPGA emulation (Verilog RTL)

The saboteur circuit can be seen in the context of the fault injection platform in Figure 4, which consists of the DUT, the aforementioned scan chain, a control module, and a reporting module. The control module operates the DUT reset, scan chain, and reporting module to inject faults and perform each test in sequence. The reporting module takes control of the UART and displays summary statistics on the serial port at the end of each test: the number of faults tested, the type of fault currently being tested, the number of external memory accesses recorded during the test period, and the last 50 addresses accessed from external memory. Figure 5 shows a sample output from the reporting module.

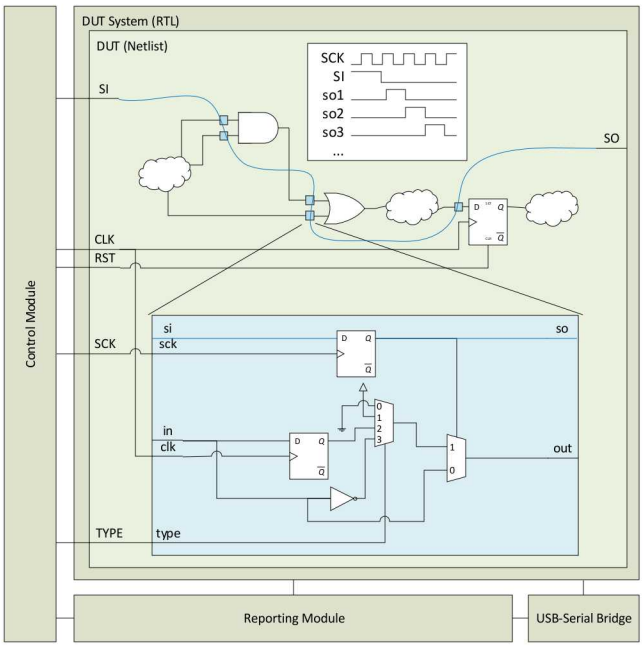

Figure 4. FPGA fault emulation platform

```
--- FN0000000000 FT0 ---
Addresses: 0000021538
00056C8
00056CC
...
0000808
000080C
0000810
0000814
0000818
000081C
---------------------
```

Figure 5. Reporting module output for the fault-free case

The control module state machine of Figure 6 releases the DUT from reset, waits a predefined amount of time for the user program to run, triggers the reporting module, and then scans in the next fault to test while the DUT is held in reset. The AES application takes 37 ms[1] to complete, but we run the CPU for 70 ms to allow unusual behaviors to manifest. The reporting module requires an additional 7 ms to report the memory access statistics, for a total runtime of 77 ms per test. Once all faults of a single type have been tested, the next fault type is selected, and the process repeats until all gate inputs have been tested with all four fault types. The full capture of the LEON3 running the AES application took 1 hour 19 minutes.

We implemented our fault emulation platform on the Zync UltraScale+ MPSoC ZCU102 Evaluation Kit [7] and the Virtex UltraScale FPGA VCU108 Evaluation Kit [8], both from Xilinx, Inc.

[1] This is slightly longer than the simulated time, owing to the use of the real LEON3 UART instead of a simulation model.

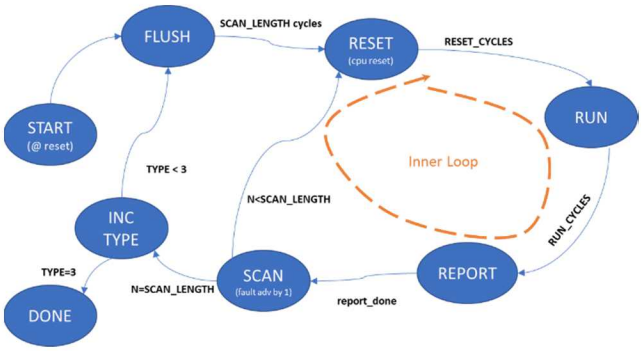

Figure 6. Fault emulation state machine

IV. Fault Analysis

We tested stuck-at, delay, and inverted signal faults on each of the 15,384 inputs of the DUT netlist using the FPGA emulation platform and examined the serial output resulting from running the AES user application with two different key/plaintext value pairs. The serial data outputs were logged and categorized, as were the address counts and last 50 addresses from the reporting module. The LEON3 UART and reporting module time-shared the USB-to-serial bridge running at 921600 baud, with a multiplexer granting control to the reporting module while it is active.

A. Faulty Outputs

Based on the serial output characteristics, we assigned each fault one of 22 behavioral categories and a severity score from 0 to 7, 7 being the most severe from an information protection perspective. To give a few examples from Table 1, the most common outcome of a fault is no discernible effect on the program output (perfect/match), and the second most common is catastrophic failure with no output on the serial port (empty). Several faults resulted in partially corrupted output, with or without a corresponding error message. There were also false positives, where valid output was reported as an error, and false negatives, where invalid output was reported as a match.

Of the faults that produced a long message--more than 500 bytes of non-repeated data--there were several that leaked the binary value of key or plaintext (leak/crypto) or other memory contents (leak/other) over the serial port. Finally, among the well-formed but non-matching outputs, we saw at least 4 bytes of the cryptographic key in the encrypted hexadecimal output (leak/key).

Table 1. Faulty behaviors from stuck-at, delayed, and inverted gate inputs (total counts)

Severity/Behavior	Stuck0	Stuck1	Delayed	Inverted
0: output as expected; correct ciphertext received and validated				
perfect/match	13,732	13,357	15,249	9,681
1: partially corrupt; correct ciphertext received and validated				
correct/match	416	377	514	315
repeat/match	10	6	12	8
2: wrong ciphertext received, but correctly reported as invalid				
agree/error	--	4	--	2
corrupt/error	18	10	16	4
disagree/error	615	400	776	275
3: no output, truncated or corrupt output				
agree/none	--	--	17	--
correct/error	77	79	73	115
correct/none	--	--	21	--
corrupt	212	232	182	198
disagree/none	8	8	12	2
empty	12,094	12,662	9,599	15,956
repeat/pattern	300	297	307	294
short	3,106	3,109	3,710	3,754
4: wrong ciphertext received, incorrectly reported as valid				
agree/match	60	56	48	34
corrupt/match	--	--	3	2
disagree/match	2	10	2	4
match/error	--	10	2	2
5: more output than expected, contents indiscernible				
long	2	8	24	10
6: output contains strings from memory				
leak/other	51	79	104	51
7: output contains strings from key and/or plaintext				
leak/crypto	13	11	43	9
leak/key	--	1	2	--

B. Faulty Control Flow

To determine the relationship between serial output and control flow behavior, we parsed the reporting module output and categorized the address behavior as follows. If the last 50 addresses match the expected end-of-program sequence from the fault free case, we mark the case as EOP. If the end-of-program sequence matches, but the total number of memory accesses is less than or greater than expected (21,538 accesses for the fault-free case), we mark it EOP early or EOP late, respectively. Otherwise, we mark the test case as one of the following: Loop, if the same address is seen more than once in the last 50 memory accesses; Sequential, if the last 50 addresses are in sequence (separated by 4, consistent with a 32-bit instruction fetch); or Nonsequential, if the last 50 addresses are not in sequence.

Table 2. Control flow fault behaviors observed, by severity (total counts)

Severity	EOP	EOP early	EOP late	Loop	Nonseq	Seq
0	46,683	430	1,029	88	3,779	10
1	1,076	170	160	18	232	2
2	4	1,529	216	9	362	--
3	261	771	46	31,020	34,148	178
4	108	67	12	--	48	--
5	--	--	2	--	6	36
6	--	--	--	19	14	252
7	--	2	--	26	17	34

It is interesting to see from Table 2 the range of control flow behaviors that still result in a perfectly matched serial output (severity 0). In 430 cases, between 1 and 7630 memory accesses were skipped, but the program still completed successfully and produced the correct output. 1029 cases resulted in up to 630,000 additional memory accesses without corrupting the serial output. Another 3877 cases terminated abnormally but still produced the correct output. These cases may nevertheless be susceptible to data dependencies and/or increased side-channel leakage. The two EOP faults that resulted in key leakage did so through the encrypted stream, each exposing 4 bytes of cryptographic data. The remaining severity 7 faults exposed all or most of the cryptographic data by dumping memory contents to the serial port.

4.3 Comparison to Simulation

Table 3 shows the testing results from gate-level simulation of the FPGA and ASIC implementations, compared to emulation of the ASIC netlist. Since the ASIC simulation did not cover delay or inversion faults, we limit our comparison to stuck-at faults. Though the FPGA and ASIC netlists were from vastly different implementations, the distribution of fault severities is roughly equivalent for the two. Simulation versus emulation, however, held some surprises: 182 faults were upgraded from lower to higher severity, and 694 were downgraded in severity from simulation to emulation.

We re-simulated these 876 cases using conditions nearer to those of the FPGA emulation: 1) SRAM was pre-loaded with all zeros, consistent with FPGA initialization, 2) the UART simulation model was replaced with the real UART used in FPGA emulation, and 3) the simulation time was increased from 35 to 70 ms. The result is the ASIC Simulation 2 column of Table 3, which is a closer match to our emulation results.

Table 3. Stuck-at fault behavior from emulation, compared to simulation

Sev-erity	FPGA Simulation	ASIC Simulation 1	ASIC Simulation 2	ASIC Emulation
0	13,096	13,165	13,189	13,557
1	242	308	309	418
2	416	393	490	499
3	12,504	16,734	16,627	16,090
4	55	69	70	69
5	11	16	1	5
6	8	8	25	67
7	37	23	5	11

V. Performance and Utilization

A. Speedup

Table 4 compares the netlist simulation and emulation times. Fault-free ASIC simulation took around 10 minutes, but because many of the faults resulted in reduced simulation activity, the average simulation time was 7 minutes. Each emulated test took just 77 ms of real time, a 5400X speedup, keeping in mind we also collected data for twice as long as in simulation. Accounting for this factor and the 40-minute FPGA compile time, the overall speedup was 7200X. From a practical perspective, this allowed us to test four fault types on all gate inputs of the DUT in just under two hours, whereas simulation of just two fault types took 37 days tying up four enterprise simulation licenses continuously. (The FPGA simulation tested three fault types in parallel, at an average of 10 minutes per simulation, for 91 days.)

Table 4. Runtimes for gate-level fault testing

Test	Average test time	Number of faults tested	Total test time
FPGA simulation	10 min	39,930	91 days[2]
ASIC simulation	7 min	30,768	37 days[3]
ASIC emulation	77 ms	61,536	79 min

B. 5.2 Overhead

Table 5 shows the relative utilization of each component of the emulation platform, including three different versions of the IU DUT: the original RTL, ASIC netlist with no saboteurs, and ASIC netlist with saboteurs inserted. It also shows the WNS (slack) for each version of the DUT when constrained with a 50 MHz clock. Though all variants had positive slack at the constrained frequency of 50 MHz, a WNS of 10.6 ns suggests the original design could be run at up to 106 MHz without additional synthesis effort. Inserting saboteur cells on the netlist slows this down to 57 MHz, about a 46% drop. The resource overheads were substantial: emulating the netlist alone (without inserting saboteurs) required 85% more LUTs and three times as many registers. Inserting saboteur cells called for 10 times the LUTs and 30 times the registers. The control and reporting modules added overhead as well, mostly from the reporting module FIFO, which could be minimized to save resources.

Table 5. Relative utilization on ZCU102

Design	WNS	LUTs	Registers
IU RTL	10.6 ns	1,652	838
IU ASIC netlist without saboteurs	11.7 ns	3,059	2,583
IU ASIC netlist with saboteurs inserted	2.57 ns (half speed)	17,178 (10X RTL)	25,082 (30X RTL)
Control/reporting module	--	3,236	8,631
LEON3 system (excluding IU)	--	17,032	3,486

[2] With three simulations performed in parallel

[3] With four simulations performed in parallel

Nevertheless, utilization on the ZCU102 and VCU108 platforms was less than 15% for all but block RAM, which could be reduced by moving the program code off chip.

Table 6. Total utilization on ZCU102 and VCU108

Utilization	ZCU102 (Zync MPSoC)	VCU108 (Virtex Ultra)
WNS	2.57 ns	1.18 ns
LUTs	37,684 (14%)	40,765 (7%)
Registers	37,199 (7%)	39,553 (4%)
LUT RAM	8,760 (6%)	9,059 (12%)
Block RAM	26 (3%)	363 (21%)

VI. Fault Locality

When performing fault analysis of an ASIC macro, we are able to back-annotate the results to the physical layout, shown in Fig. 1. Mapping fault numbers to gate pins, we can plot the location of each fault-tested pin and color it according to the fault behavior. This can help to identify potentially vulnerable areas of a given ASIC layout. In Fig. 2, for example, we see four gate pins near (280,230) µm that leak cryptographic data when faults are injected on them, yet are relatively isolated from pins that could cause suppressed or corrupted output from collateral damage.

VII. Conclusions

We have developed an automated gate-level fault injection platform based on Ultrascale and Ultrascale+ development boards from Xilinx and used this to characterize the behavior of the LEON3 integer unit running AES256 encryption software when exposed to four different types of faults (stuck-at-0, stuck-at-1, delayed input, and inverted input). The most severe of these behaviors involved verbatim leakage of program memory, key and/or plaintext over the serial communication channel; however, we also noted less severe behaviors that seem to depend on the value of the key and plaintext used. Of the four types of faults we tested, delay faults proved the most useful in uncovering the widest range of behaviors, at the expense of two registers per fault insertion point rather than one. The inverted logic fault proved to be the least useful, not covering many of the high-severity behavioral categories.

The FPGA emulation platform allowed for fault-testing the LEON3 in 1/7200th the time required for gate-level simulation, even accounting for FPGA implementation and bitstream programming time. The emulated behaviors matched their simulated counterparts, except in 701 cases (2% of faults tested), for which the differences could not be explained. Nevertheless, we saw the same behaviors between simulation and emulation, in nearly equal proportions.

The extreme acceleration will allow testing and characterization of more complex processors and user code, which would be prohibitive to do in gate-level simulation. Table 7 shows the projected scaling figures for three different flavors of RISC-V processor, based on preliminary ASIC synthesis results. Note that the estimated overhead of the 64-bit Rocket core exceeds the register count of either ZCU102 or VCU108, so this core would need to be fault tested in two parts. This minor inconvenience pales in comparison to the 4.5 years we would need to wait for simulation results.

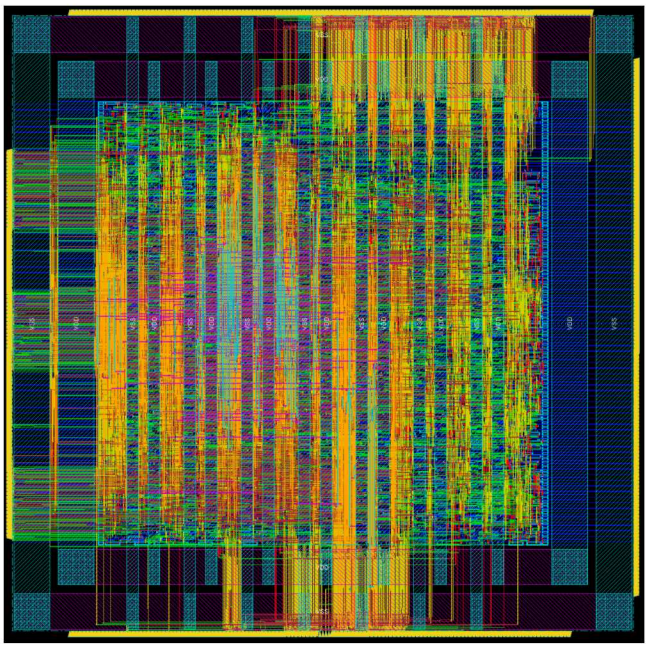

Fig. 1. DUT physical layout (ASIC macro)

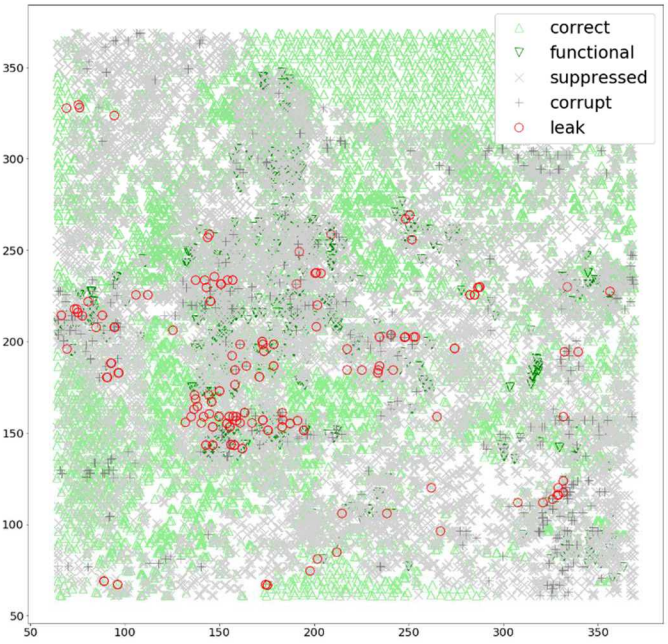

Fig. 2. Location of fault injection points within DUT layout

As neither the processor nor software included mitigations for fault tolerance, it was no surprise to see a variety of errant behaviors in response to injected faults. Nevertheless, understanding how the processor and user application respond to faults can help drive tailored mitigations that optimize the balance between performance and fault-tolerance in safety- and security-critical systems. This platform gives us the ability to

apply and rapidly test various hardware and software mitigations in sequence, driving better engineering decisions.

Table 7. Projected scaling to larger processors

Design	Fault points	Simulation time	Emulation time	Register overhead
LEON3 IU	15,384	74 days	2 hrs	32,875
RISC-V Potato	35,264	171 days	4 hrs	149,687
RISC-V Rocket 32-bit	67,385	328 days	6 hrs	278,171
RISC-V Rocket 64-bit	335,834	4.5 years	30 hrs	1,360,598

ACKNOWLEDGMENTS

We would like to thank Russell D. Miller for performing the ASIC synthesis and place-and-route used in this test and Sheng-Hao Huang for his early experiments on the RISC-V cores. We also thank Gerald Zuelsdorf for recommending security behaviors that should be investigated.

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] B. Yuce, P. Schaumont and M. Witteman, "Fault attacks on secure embedded software: threats, design, and evaluation," *Journal of Hardware and Systems Security,* vol. 2, no. 2, pp. 111-130, 2018.

[2] D. Shaw, D. Al-Khalili and C. Rozon, "Fault security analysis of CMOS VLSI circuits using defect-injectable VHDL models," *Integration, the VLSI journal,* vol. 32, no. 1-2, pp. 77-97, 2002.

[3] "LEON3 Processor," Cobham Gaisler AB, [Online]. Available: http://www.gaisler.com/index.php/products/processors/leon3.

[4] "Riviera-PRO Advanced Verification Platform," Aldec, Inc., [Online]. Available: https://www.aldec.com/en/products/functional_verification/riviera-pro. [Accessed 3 August 2018].

[5] "Questa Advanced Simulator," Mentor, a Siemens Business, [Online]. Available: https://www.mentor.com/products/fv/questa/. [Accessed 2 August 2018].

[6] "Fault Propagation Analysis," OneSpin Solutions GmbH, [Online]. Available: https://www.onespin.com/products/specialized-apps/fault-propagation-analysis/. [Accessed 2 August 2018].

[7] "Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit," Xilinc Inc., [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html. [Accessed 2 August 2018].

[8] "Xilinx Virtex UltraScale FPGA VCU108 Evaluation Kit," Xilinx Inc., [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-u1-vcu108-g.html. [Accessed 2 August 2018].

[9] Aeroflex Gaisler AB, "SPARC V8 32-bit Processor LEON3 / LEON3-FT," March 2010. [Online]. Available: https://www.actel.com/ipdocs/LEON3_DS.pdf.

[10] S. Lu, S. Huang, C. Wu and Y. Chen, "Speeding up emulation-based diagnosis techniques for logic cores," in *Design & Test of Computers, IEEE*, 2011.

[11] "Veloce Emulation Platform," Mentor, a Siemens Business, [Online]. Available: https://www.mentor.com/products/fv/emulation-systems/. [Accessed 1 August 2018].

[12] "Palladium Z1 Enterprise Emulation Platform," Cadence Design Systems, Inc., [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/acceleration-and-emulation/palladium-z1.html. [Accessed 1 August 2018].

[13] P. Yeung, D. Smith and A. Ayari, "Formal fault analysis for ISO 26262: Find faults before they find you," *Tech Design Forum,* 18 June 2018.

[14] J. S. Monson, M. Wirthlin and B. Hutchings, "A fault injection analysis of Linux operating on an FPGA-embedded platform," *Int'l Journal of Reconfigurable Computing,* vol. 7, 2012.

[15] F. Ghaffari, F. Sahraoi, M. Benkhelifa and B. Granado, "Fast SRAM-FPGA fault injection platform based on dynamic partial reconfiguration," in *Microelectronics (ICM), IEEE*, 2014.

[16] J. Gracia, J. C. Baraza, D. Gil and P. J. Gil, "Comparison and application of different VHDL-based fault injection techniques," *Proc. IEEE Int'l Symposium on Defect and Fault Tolerance in VLSI Systems,* pp. 233-241, 2001.

[17] E. Mojtaba, "A fast, flexible, and easy-to-develop FPGA-based fault injection technique," *Microelectronics Reliability,* vol. 54, no. 5, 2014.

[18] M. R. S. Reddy and R. S. Babu, "High speed fault injection tool (FITO) implemented with VHDL on FPGA for testing fault tolerant designs," *Int'l Journal of Modern Engineering Research,* vol. 3, no. 5, pp. 2894-2900, 2013.

[19] J. Grinschgl, A. Krieg, C. Steger and R. Weiss, "Automatic saboteur placement for emulation-based multi-bit fault injection," *6th Int'l Workshop on Reconfigurable Communication-centric Systems-on-Chip,* pp. 1-8, 2011.

[20] "Virtex UltraSCALE," Xilinc Inc., [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale.html. [Accessed 1 August 2018].

[21] G. S. Kumar, T. Malathi, P. Sankar and S. Saravanan, "Enhancement of Fault injection techniques using saboteurs and mutants for modification of VHDL code," *Int'l Journal of Innovative Science, Engineering & Technology,* vol. 1, no. 9, pp. 567-573, 2014.

[22] J. Stanis and M. Antony, "Reducing launch and capture power using saboteur and mutants method," *Int'l Journal for Scientific R&D,* vol. 3, no. 1, pp. 1511-1515, 2015.

[23] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: method, tools and experimental results," in *Embedded Software for SoC*, Boston, Springer, 2003, pp. 39-51.

[24] "Xception," CRITICAL Software, [Online]. Available: https://www.criticalsoftware.com/en/products/p/xception. [Accessed 2 August 2018].