Information Leakage Analysis using Accelerated Fault Injection Emulation of a RISC-V Microprocessor

Tom J. Mannos
Advanced CMOS Products/Design
Sandia National Labs
Albuquerque, NM 87185
Email: tjmanno@sandia.gov

Jim Plusquelic
Electrical & Computer Engineering
University of New Mexico
Albuquerque, NM 87110
Email: jimp@ece.unm.edu

Brian Dziki
Laboratory for Advanced Cybersecurity
Research, Department of Defense
Fort G. G. Meade, MD
Email: bjdziki@tycho.ncsc.mil

***Abstract*—This paper investigates information security issues that result from the occurrence of faults within a RISC-V microprocessor. Faults are emulated using a specialized fault injection circuit which is inserted in series along paths within the synthesized netlist of the RISC-V microprocessor. The modified netlist is processed through the Xilinx Vivado implementation flow to produce a bitstream, which is used to program a Xilinx UltraScale+ MPSoC FPGA. Our previous work on using an FPGA to accelerate fault analysis is extended here using advanced FPGA capabilities including dynamic reconfiguration and high-speed GPIO between the processor and programmable logic (PL). These enhancements provide additional speedups and a more comprehensive investigation of fault types. The advanced encryption standard (AES) cryptographic engine is executed on this specialized RISC-V architecture as faults are introduced, one-at-a-time and in combinations. Ciphertext output is analyzed for each fault to determine the vulnerabilities inherent in a RISC-V ASIC implementation, across 2.7 million fault experiments.**

Keywords—Reliability analysis, RISC-V, FPGA emulation

I. Introduction

Tools and techniques to analyze and design reliable systems continues to be an active area of research. Complex, high performance microelectronic devices and systems can malfunction due to the occurrence of a hardware fault(s). Transient faults are particularly problematic because the system can, and often does, continue to operate despite the data corruption introduced by the fault. Corrupt system states represent vulnerabilities for many types of applications running on these systems, none more so then security applications. Cryptographic algorithms internally preserve and protect confidential information such as secret keys that may in fact be partially or fully revealed after a fault(s) occurs. An emerging area in the analysis of fault effects focuses on the design of test platforms and analysis techniques to identify system vulnerabilities that potentially lead to the leakage of confidential information. Once identified, countermeasures can be built into the design that minimize the likelihood that corrupt system states expose sensitive information on primary outputs and communication interfaces

The goal of this research is to pseudo-exhaustively investigate fault effects within a RISC-V architecture executing a cryptographic algorithm, to determine any effects that are detrimental to security, including leaked information associated with the plaintext or key in the corresponding ciphertext. The techniques and results presented extend our previous work using the LEON3 microprocessor running the Advanced Encryption Standard (AES) 256-bit cryptographic algorithm [1]. The following enhancements are made in this work over the previous testing architecture:

- The Berkeley RISC-V (Rocket RV64IMA) micro-processor [2] replaces the much smaller 32-bit LEON3.
- The use of advanced FPGA features, including dynamic reconfiguration and a high-speed GPIO communication interface between the processor (PS) and programmable logic (PL) components on the Zynq class SoC used as the emulation platform.

II. Fault Emulation using an MPSoC FPGA

A block diagram of the system level architecture is shown in Fig. 1. The Xilinx ZCU102 evaluation kit is used as the RISC-V emulation platform [3]. The board includes a Zynq UltraScale+ MPSoC that is partitioned into a processor side (PS) and programmable logic side (PL) side. Petalinux [4] was used to configure a custom Linux operating system (OS) that runs on one of the 4 ARM Cortex-A53 processors embedded within the PS side. The Linux OS provides both serial and ethernet communication channels to a desktop server, as well as access to a 64 GB SD card for storing data and programming bitstreams. C programs that run under Linux can access up to 4 GB of DDR4.

The Linux kernel is customized using a PL hardware configuration that includes an AXI general purpose I/O (GPIO) port. The port is composed of a 32-bit input and a 32-bit output register that is memory-mapped into the Linux kernel address space. A C program can read and write these registers after obtaining virtual address information using the *mmap()* library function (or by opening them using a Linux device driver).

We developed a custom C program that serves as the controller for the fault testing. The C program implements two distinct testing strategies; the first called *full-reconfiguration* is described in this section while the second, called *partial-reconfiguration*, is described in a subsequent section. Both

DISTRIBUTION STATEMENT A. Approved for public release: distribution is unlimited. SAND2020-0503 C. DoD #R2-PP006-20.

strategies communicate to state machines running on the PL side using the GPIO interface, and both use the processor configuration access port or PCAP to perform reconfiguration of the PL side components. PCAP enables either the entire PL side to be reconfigured or only portions of it.

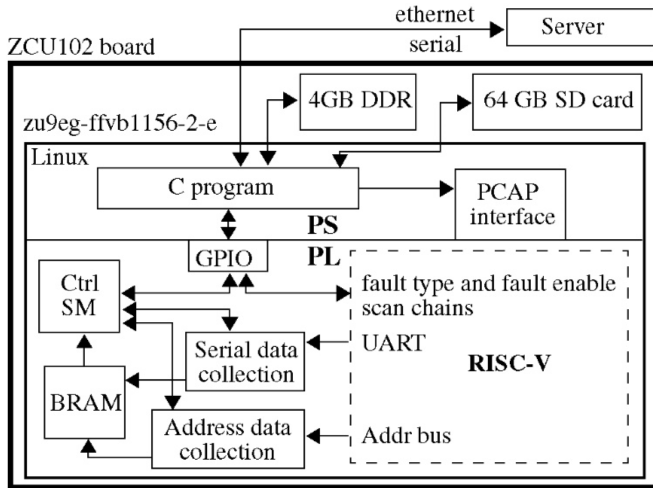

Fig. 1. System diagram using on-FPGA CPU to control fault injection.

The C program delivers commands to the *Ctrl* state machine (SM) in Fig. 1. The first command instructs the *Ctrl SM* to perform a reset on the RISC-V microprocessor. This causes the RISC-V to reboot and then execute the Advanced Encryption Standard (AES) algorithm. A ROM within the RISC-V is pre-configured with a RISC-V executable program that embeds a 256-bit key and a 128-bit plaintext. Once the emulated processor boots, it executes the binary code in the ROM which passes the key and plaintext to AES and then prints these components and the computed ciphertext in hexadecimal to its UART serial port. The RISC-V also compares the generated ciphertext with a pre-computed stored copy of the ciphertext and prints 'Match' or 'Fail'. Under fault-free conditions, it then enters an idle state.

As the processor executes the binary code, two state machines, labeled *Serial data collection* (SDC) and *Address data collection* (ADC) in Fig. 1, monitor the state of the RISC-V on a clock-by-clock cycle basis. The SDC collects and stores in block RAM (BRAM) 8-bit ASCII characters that appear on the RISC-V UART (up to 64 KB), while the ADC monitors and records activity on the RISC-V internal address bus. A separate 2048-word BRAM stores the 32-bit address bus values but only in cases when the address bus values change. The number of serial bytes transferred and the total number of memory accesses performed by the RISC-V are also recorded and stored. The serial data and address bus data are transferred to the C program after the test completes. The generated data files are transferred to the server for post-processing as described below. Example data from a fault-free test are shown in Fig. 4.

III. RISC-V SYNTHESIS

The goal of our analysis is to investigate fault behavior of a RISC-V processor core, in particular, the RISC-V Rocket core [2], implemented as a standard cell design within a CAD tool flow. Therefore, the RISC-V shown in Fig. 1 is not synthesized

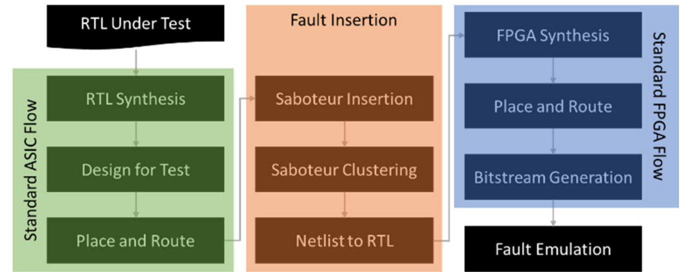

Fig. 2. Preparation of test circuit for fault emulation.

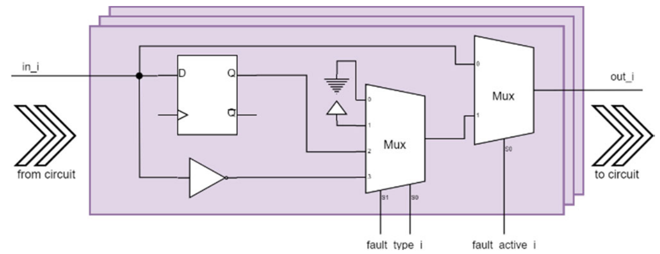

Fig. 3. Basic saboteur design used to inject faults on each gate input.

```
<------ test 2 -------->          (SERIAL OVERFLOW? 0)¶
Boot . . .OK^M¶
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¶
----¶
number of serial bytes: 288¶
number of address lines: 52147¶
1007FB80¶
1007FB90¶
1007FBA0¶
1007FBB0§
```

Fig. 4. Fault-free RISC-V serial output data.

by Xilinx Vivado directly from an RTL description, but rather is imported into Vivado as a RTL netlist. The CAD tool flow is shown in Fig. 2 where we first process the RISC-V RTL using Cadence Genus [5]. The ASAP7 7nm FinFET predictive PDK [6][7] is used during RTL synthesis as shown on the left in the figure. During synthesis, we also insert a scan chain and process the netlist through place and route to a layout using Cadence Innovus [8].

The netlist from the layout is extracted and post-processed as shown by the middle column in Fig. 2 to enable fault testing. In our previous work [1], we designed a scan-configurable saboteur cell that can introduce one of four fault types, namely a stuck-at-0, stuck-at-1, delay, or inversion fault. A schematic of the saboteur cell is shown in Fig. 3. A saboteur cell is configured using a *fault_active* scan chain, which either activates the fault or provides a fault-free by-pass path through the cell, and a 2-bit *fault type* scan chain, which selects from one of the four fault types. The stuck-faults hold the output signal at a constant value, while the delay and invert faults introduce a clock cycle delay or invert the input signal, respectively. The processing shown in the middle column of Fig. 2 inserts an instance of the saboteur cell on every input of every gate in the RISC-V netlist.

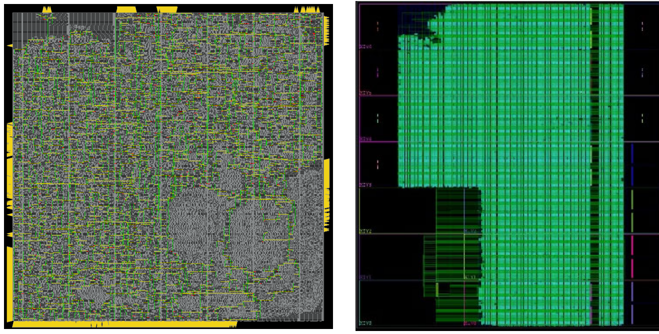

Fig. 5. RISC-V layout (left) and Zynq UltraScale+ FPGA version.

The right-most column of Fig. 2 shows the process flow using the Xilinx Vivado FPGA CAD tool [9]. The standard synthesis-implement-generate-bitstream flow is used for the *full-reconfiguration* test strategy described in this section (the flow used for the *partial-reconfiguration* test strategy requires additional steps described later). Fig. 5 shows the P&R layout of the RISC-V with the system clk highlighted on the left and the implementation view of the RISC-V on the Zynq UltraScale+ FPGA. The resulting block is 336 µm X 334 µm at 60% cell utilization. There are 34,196 logic gates, including 5,262 scannable flip-flops, and 2,414 hierarchical ports. Fault candidates include all combinational inputs and flip-flop data and scan-enable inputs, 85,714 fault insertion points in total.

A. *Testing Process*

The testing process is carried out by introducing a fault through the scan chain and then starting the RISC-V. Each of the 85,714 sites are tested individually by making the fault active and selecting the fault type using the scan chains, which are controlled by the C program via the GPIO interface. With four fault types, the serial and address bus data was collected from 342,856 fault injection tests. In addition, we repeated these experiments by inserting two faults of the same type in sequence, i.e., in adjacent saboteur cells in the scan chain. Last, in order to determine whether the observed fault behaviors are dependent on the selected key and plaintext, we repeated the single and dual fault experiments using a second key and plaintext, with and without memory scrubbing. Therefore, in total, we collected and analyzed data from 2,742,848 fault experiments.

The C program automates the testing process. It first obtains the virtual addresses for the GPIO registers and then implements three loops. The outer loop sequences through the four fault types, the middle loop first tests single faults and then allows up to *n* sequential faults of the same type to be tested (we limited our analysis to only two), while the inner loop sequences through each of the 85,714 fault locations. Once the fault(s) is inserted, a soft reset is applied, which reboots the RISC-V. The *Ctrl SM* from Fig. 1 signals the C program when execution is complete, or in some cases, halts data collection when the fault causes the RISC-V to loop indefinitely. The data stored in BRAMs is then transferred and stored on the SD card.

As a validation experiment, we repeated the experiments using a special "scrubbing" process whereby we reprogrammed the PL side before running each test. First, the PL-side bitstream is transferred to Linux and stored in */lib/firmware*. The C program then makes a system call using the following argument before carrying out the fault insertion:

echo bitstream > /sys/class/fpga_manager/fpga0/firmware

This causes the *fpga manager* to read the stored bitstream and reprogram the entire PL side through the PCAP interface. The data collected from these experiments is referred to as the *reference* data. In experiments where this reprogramming is not performed, we found that the contents of the internal RISC-V SRAM impacts fault behavior. The reset operation does not clear the SRAM and the non-zero SRAM values from the previous encryption operation change the behavior of some faults. The reference experiments on the other hand are 100% reproducible and enabled the identification of fault behaviors that were SRAM dependent, as we discuss in the following results sections.

B. *Results*

We first report on the overhead associated with the fault injection circuitry and run times. The next sub-section provides details on the fault behavior, identifying cases in which the key and/or plaintext are 'leaked' through the UART.

1) *Run-times and Resource Overheads*

The time associated with the one-time processing of the RTL through fault insertion (left-most and middle columns of Fig. 2) is small relative to the Vivado synthesis and data collection times, and therefore are not reported here. The runtime associated with the standard-flow, i.e., synthesis, implementation and bitstream generation, is approximately 6 hours. Vivado synthesis was attempted at higher frequencies but we were ultimately forced to reduce the clock frequency to 24 MHz because of timing violations. For comparison, synthesis time and resource utilization are much smaller, i.e., approximately 2 hours and 20% of the LUTs and FFs, respectively, when faults are inserted directly into the RTL (Cadence place and route is not used). However, this approach does not represent a realistic usage scenario of the RISC-V.

Data collection run time is approximately 250 milliseconds (ms) per test in the fault-free case. The *Ctrl SM* allowed the RISC-V to run for 2^{22} clock cycles at a frequency of 24 MHz, which accounts for 174 ms. A portion of the remaining 75 ms is required for the scan and GPIO data transfer operations. Faults that caused the processor to reboot over-and-over again ran for several minutes and were eventually halted by the *Ctrl SM* once the 64 KB serial buffer filled up or no additional serial bytes were received after 2^{20} cycles. For the reference experiments, the PL side was reprogrammed before each test, which required an additional 250 milliseconds. On average, four tests could be completed per second without reprogramming, otherwise only two tests could be completed. Data collection time for one experiment which tests all 85,714 faults is approximately 6 hours and 12 hours, respectively. This represents a significant speed-up over simulation, which would take an equivalent of **10 years** to simulate each of the four fault types at 1 fault per hour.

2) *Faulty Output Results: Reference Experiment*

In previous work [1], we classified fault behavior into 22 categories based on the UART output, four of which represent some form of crypto or memory leakage through the UART port,

DISTRIBUTION STATEMENT A. Approved for public release: distribution is unlimited. SAND2020-0503 C. DoD #R2-PP006-20.

a condition typically referred to as "fail unsafe" [10-14]. A severity score is assigned from 0 to 7, with 7 representing the most severe from an information leakage perspective.

We extend the classification scheme here to 26 Severity/Behavior classes as shown by the left-most column in Table 1. The table gives the results obtained from the reference experiments where we reprogrammed the FPGA prior to running each test. We refer to this operation as **scrubbing** because the memory is cleared upon reprogramming, enabling the RISC-V to execute its code without being influenced by memory artifacts left behind from the previous execution. As discussed earlier, we repeated the reference experiments under four conditions, e.g., with 1 fault inserted at-a-time, and with 2 logically adjacent faults inserted simultaneously, and for two different key/plaintext pairs, identified here as C1 and C2. The results from these 4 experiments are given in separate columns in the table. The number of test instances shown on the rows for each class represent the sum across all 4 fault types, i.e., stuck-0, stuck-1, delay and invert. With 85,714 faults and 4 fault types, the sum of the values in any given column is 342,856.

TABLE I. REFERENCE EXPERIMENT COMPARATIVE ANALYSIS WITH ALL 4 FAULT TYPES COMBINED.

Severity/ Behavior	1 Fault C1	1 Fault C2	2 Faults C1	2 Faults C2
0: perfect match				
perfect/match	267317	267220	243289	243198
1: correct match, with some extra output				
correct/match	113	119	120	126
repeat/match	21	21	19	19
2: incorrect output reported as error				
agree/error	2	2	1	1
corrupt/error	144	137	156	152
disagree/error	7559	7626	7230	7263
3: benign corrupt output				
agree/none	5	5	10	9
correct/error	143	130	101	86
correct/none	314	321	309	320
corrupt	3706	3700	4057	4056
disagree/none	300	305	246	251
empty	17814	17814	31091	31091
repeat/boot	234	234	302	302
repeat/error	2	2	1	1
repeat/pattern	845	841	1106	1090
repeat/txt	46	38	52	41
short	43504	43575	53971	54064
4: incorrect output reported as match				
agree/match	154	157	209	210
corrupt/match	28	25	20	18
disagree/match	53	52	54	52
match/error	13	13	9	9
5: corrupt long output indicating possible but unconfirmed memory leakage				
long	195	140	230	185
6: corrupt output with confirmed memory leakage w/ key and/or plaintext				
leak/other	0	0	0	0
leak/crypto	0	16	3	15
7: key and/or plaintext presented as part of the encrypted message				
leak/key	312	331	245	273
leak/txt	28	28	21	20

Similar to our previous results, the largest number of test instances is associated with class 0. Approximately 78% of the test executions were not affected by the inserted fault for the 1 Fault case. This drops to approximately 71% for the 2 Fault testing scenario. Although not shown, this trend continues for increasingly larger numbers of simultaneously inserted faults. This is not surprising as intuitively, the probability of incorrect execution should increase as more and more nodes become faulty.

In contrast to our previous results [1], the second largest class is **short**, which indicates that the output was corrupt and had length shorter than the correct output. The **empty** class comes in 3rd in these results. Both of these faults are considered benign because no sensitive information is leaked. The number of instances of false negatives (incorrect output reported as a match) from severity class 4 are relatively small, as are the false positives associated with class 3, **correct/error**.

The most significant distinction between these results and those reported in [1] is the increase in the number of severity level 7 test cases. These cases represent real information leakage where the key and/or plaintext, in whole or in part, are transmitted through the UART. Interestingly, the number of test results in this class drops from approximately 0.1% in the single fault case to 0.07% when 2 faults exist simultaneously.

Also unique to this analysis is the characterization of data dependent fault behavior. Data dependencies are reflected in the differences between the C1 and C2 column pairings. The closeness of the corresponding pairs of values suggests that data dependencies play only a small role on the RISC-V faulty execution behavior. However, significant differences occur for several higher severity classes, namely, class 5, **long** and class 7, **leak/key**, indicating the level of leakage can be controlled by key and/or plaintext selection.

3) Faulty Control Results

As discussed earlier, we additionally monitored activity on the address bus of the RISC-V in order to better understand its execution behavior under faulty conditions. Here, we analyze the address bus behavior and correlate them with the behavior of the UART output. We expect faulty UART output would always be accompanied with abnormal address bus behavior, but this is not always the case. In addition, we found the opposite

DISTRIBUTION STATEMENT A. Approved for public release: distribution is unlimited. SAND2020-0503 C. DoD #R2-PP006-20.

condition does not hold as well, i.e., correct UART output was not always accompanied by correct execution behavior.

TABLE II. CONTROL FLOW BEHAVIOR

Severity	EOP	EOP early	EOP late	Loop	Nonseq	Seq
0	264006	771	81	2405	54	0
1	2	45	3	57	27	0
2	0	3569	471	3662	3	0
3	2	149	1042	34778	19450	11492
4	127	30	18	73	0	0
5	0	0	0	100	95	0
6	0	0	0	0	0	0
7	0	2	3	335	0	0

The address bus is monitored in two ways. First, the total number of address bus changes is recorded during each test, which is nominally equal to 52,147. Second, we also record the last 50 address bus changes and compare them with the expected fault-free sequence. Table 2 shows the results obtained from the reference experiment called **1 Fault**, **C1** in Table 1 (results for the other experiments are similar). The left-most column gives the severity level of the test as determined by the behavior on the UART output. The **EOP** column gives the number of tests which produce address sequences that match the expected sequence in number and value. The non-zero values associated with severity levels other than 0 indicate that UART output was corrupt despite the nominal behavior observed from the address bus. The columns labeled **EOP early** and **EOP late** identify test cases where the number of accesses was less than and greater than, respectively, the expected number. Again, the non-zero values in the row with severity 0 indicate that UART output was correct but execution behavior was abnormal. The column labeled **Loop** are tests that produced an address value that appeared more than once in the last 50. Columns **Seq** and **Nonseq** count test cases where the address values increment by 4 and are in sequence, and cases where this does not occur.

4) Scrubbed Vs. Not Scrubbed

The execution behavior of the RISC-V is dependent in some cases on the initial state of the SRAM memory. We confirmed this by repeating the reference experiments but without reprogramming the FPGA after each test. The results in Table 3 compare the UART output behavior observed for the **1 Fault**, **C1** and **C2** experiments with the corresponding reference experiment results from Table 1.

TABLE III. REFERENCE EXPERIMENT (SCRUBBED) VS. NOT SCRUBBED COMPARATIVE ANALYSIS WITH ALL 4 FAULT TYPES COMBINED.

Severity/ Behavior	Scrubbed 1 Fault, C1	Scrubbed 1 Fault, C2	Not Scrubbed 1 Fault, C1	Not Scrubbed 1 Fault, C2
0: perfect match				
perfect/match	267317	267220	267568	267479
1: correct match, with some extra output				
correct/match	113	119	165	167
repeat/match	21	21	59	51
2: incorrect output reported as error				
agree/error	2	2	2	2
corrupt/error	144	137	86	85
disagree/error	7559	7626	7556	7596
agree/none	5	5	4	4
correct/error	143	130	144	131
correct/none	314	321	238	250
corrupt	3706	3700	2811	2844
disagree/none	300	305	235	270
empty	17814	17814	16697	16698
repeat/boot	234	234	227	224
repeat/error	2	2	4	4
repeat/pattern	845	841	1181	1161
repeat/txt	46	38	90	90
short	43504	43575	40671	40639
4: incorrect output reported as match				
agree/match	154	157	183	183
corrupt/match	28	25	9	8
disagree/match	53	52	59	55
match/error	**13**	**13**	**1591**	**1594**
5: corrupt long output indicating possible but unconfirmed memory leakage				
long	**195**	**140**	**2082**	**2066**
6: corrupt output with confirmed memory leakage w/ key and/or plaintext				
leak/other	0	0	42	59
leak/crypto	**0**	**16**	**792**	**819**
7: key and/or plaintext presented as part of the encrypted message				
leak/key	312	331	348	365
leak/txt	28	28	28	28

The differences between columns 2 and 4 and columns 3 and 5 confirm that the starting state of the SRAM impacts the RISC-V execution. In some cases the differences are relatively small, e.g., the **perfect match** row pairs differ by less than 0.1%, while other show much larger differences, e.g., the bolded rows in severity levels 4, 5 and 6. The large differences in these higher severity classes exacerbate the concerns discussed earlier with respect to data dependencies, i.e., the amount of leakage is dependent on the operational state.

DISTRIBUTION STATEMENT A. Approved for public release: distribution is unlimited. SAND2020-0503 C. DoD #R2-PP006-20.

IV. On-Going Work: Partial Dynamic Reconfiguration

As discussed earlier, the reference design data collection process reconfigured the PL-side before each fault test using PCAP. Although PCAP provides fast reconfiguration, on order of 250 milliseconds, full reconfiguration does not address the size limits imposed by the fault circuitry when embedded within large designs. The 85,714 copies of the Saboteur circuit shown in Fig. 3 consume significant FPGA resources, reducing those available for the RISC-V implementation itself. An alternative strategy is to build a reconfigurable design where only one fault type at each fault site is included in the implementation, with the option of swapping different fault types in and out at run-time. The resource savings is substantial because the Saboteur cell is now much smaller because the other fault types, the 4-to-1 MUX and the fault type scan chain from Fig. 3 are eliminated.

Xilinx calls this methodology dynamic partial reconfiguration (DPR) and provides supports for it in their Vivado CAD tool using a bottom-up design flow methodology depicted in Fig. 6. The single project design flow is replaced with a set of projects labeled '1' along the bottom of the figure. The *static* design on the left includes all design components that are not reprogrammable. The modules shown on the bottom right include components that can be reprogrammed with other modules in the set. In our design, we create 181 DPR regions and create four modules for each of these regions, one for each fault type. The bottom-up synthesis flow allows the static design and all of the modules to be synthesized in parallel, and then saved as design checkpoints (DCP).

Once the DCPs are available, the implementation and bitstreams for the unified, full design can be created (labeled '2' in the figure). The static design components are 'locked down' as shown by the labels in Fig. 7, and a set of regions are created called *pblocks*. Vivado is used to place and route each of the modules, and their variants, within these regions. A static bitstream as well as a set of partial bitstreams are automatically generated in the final step of this process. The bitstreams are transferred to the Linux OS on the ZCU102, and PCAP is used to configure the regions with modules that implement each of the fault types.

We implemented the tool flow shown in Fig. 6 but have not yet been successful at creating a fully routed design. The big challenge associated with the RISC-V design is not the number of logic gates but rather the large fan-out networks that exist in the structural netlist. We hope to complete this design in the near future and demonstrate its capabilities. For example, any one of the individual regions can be reprogrammed in approximately 1 ms. Therefore, DPR time overhead is expected to be negligible.

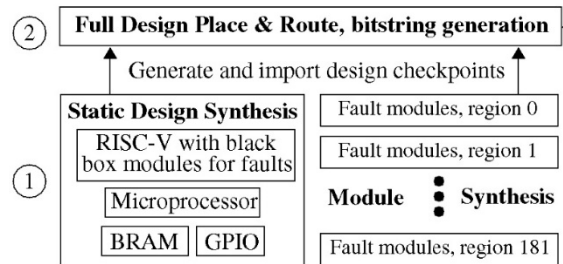

Fig. 6. Xilinx Vivado bottom-up dynamic partial reconfiguration tool flow.

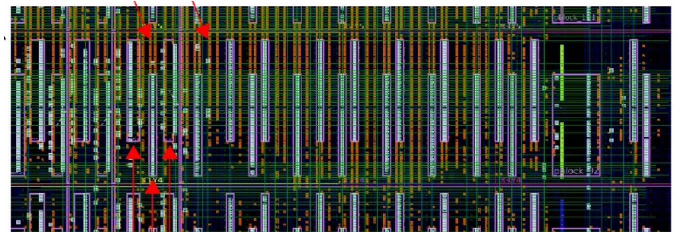

Fig. 7. Implementation view of locked full design.

V. Conclusion

This paper investigates the impact of static faults on the execution behavior of a RISC-V microprocessor, using an FPGA emulation platform. A standard cell implementation of the RISC-V is created using an ASIC CAD tool flow to obtain a realistic netlist of the RISC-V. A Saboteur circuit is inserted in series with all gate inputs in the netlist. A scan chain is used to activate one of four fault types at one or more fault locations. A set of experiments are carried out that systematically measure the RISC-V execution behavior. The fault tests are classified according to the amount of information they leak. Advanced FPGA features are used to accelerate the fault testing process.

References

[1] T. J. Mannos, B. Dziki, M. Sharif, "Fault Testing a Synthesizable Embedded Processor at Gate Level using UltraScale FPGA Emulation" *International Symposium on Field-Programmable Gate Arrays*, 2019.

[2] https://github.com/sergeykhbr/riscv_vhdl/blob/v2.0/rocket_soc/rocketlib

[3] https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html

[4] https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html

[5] https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html

[6] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthya, G. Yeric, "ASAP7: A 7-nm FinFET Predictive Process Design Kit," Microelectronics Journal, vol. 53, pp. 105-115, July 2016.

[7] http://asap.asu.edu/asap/

[8] https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html

[9] https://www.xilinx.com/products/design-tools/vivado.html

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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. 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. SAND2020-0503 C. DoD #R2-PP006-20.