Information Leakage Analysis Using a Co-Design-Based Fault Injection Technique on a RISC-V Microprocessor

Jim Plusquellic, *Member, IEEE*, Donald E. Owen, Jr., *Member, IEEE*, Tom J. Mannos, *Member, IEEE*, and Brian Dziki

***Abstract*—The RISC-V instruction set architecture open licensing policy has spawned a hive of development activity, making a range of implementations publicly available. The environments in which RISC-V operates have expanded correspondingly, driving the need for a generalized approach to evaluating the reliability of RISC-V implementations under adverse operating conditions or after normal wear-out periods. Fault injection (FI) refers to the process of changing the state of registers or wires, either permanently or momentarily, and then observing execution behavior. The analysis provides insight into the development of countermeasures that protect against the leakage or corruption of sensitive information, which might occur because of unexpected execution behavior. In this article, we develop a hardware–software co-design architecture that enables fast, configurable fault emulation and utilize it for information leakage and data corruption analysis. Modern system-on-chip FPGAs enable building an evaluation platform, where control elements run on a processor(s) (PS) simultaneously with the target design running in the programmable logic (PL). Software components of the FI system introduce faults and report execution behavior. A pair of RISC-V FI-instrumented implementations are created and configured to execute the Advanced Encryption Standard and Twister algorithms. Key and plaintext information leakage and degraded pseudorandom sequences are both observed in the output for a subset of the emulated faults.**

***Index Terms*—Fault analysis, fault emulation (FE), FPGA, RISC-V.**

I. Introduction

MICROPROCESSOR system architectures that are resilient to internal faults are difficult to design because of timing and area constraints, complexity and the exponential number of possible faulty execution behaviors that are possible [1]. An alternative approach addresses reliability from a security perspective and focuses only on those faults that result in the leakage or corruption of sensitive information through I/O channels. This significantly reduces the number of faults that the system needs to detect and/or correct using countermeasures that, e.g., halt execution or take other reactive measures to protect sensitive information. The primary challenge to developing security-resilient architectures is deciding which faults or fault combinations lead to information leakage and/or corruption. Although formal and simulation-based approaches to fault analysis are possible, they are much slower and are, therefore, difficult to apply to complex systems such as those found in RISC-V system architectures, which possess a large number of fault sites [2]–[4].

We develop a hardware/software co-design-based fault injection (FI) system architecture that focuses on finding faults that leak or corrupt sensitive information. Hardware/software co-design refers to systems that partition the computational tasks associated with an algorithm into hardware and software components, where software components run on an embedded microprocessor while the hardware components run on dedicated hardware. Although application-specific integrated circuits (ASICs) can be designed and fabricated as dedicated hardware, it is more cost effective to use an FPGA system-on-chip (SoC) platform as the physical device to carry out the fault campaign. The arguments that support this use case relate to the ease at which both the software and hardware can be configured to, e.g.,

1) emulate any arbitrary architecture-under-test, e.g., a 64-bit version of a RISC-V processor;
2) dynamically reconfigure the architecture-under-test to change the set of active fault sites and/or implement specialized operations such as block RAM (BRAM) memory scrubbing;
3) dynamically tune the data collection process as a means of optimizing emulation speed;
4) accommodate and accelerate the testing and evaluation of countermeasures.

We built a system prototype with these capabilities and apply it to a logic-level netlist implementation of the Berkeley RISC-V (Rocket RV64IMA) microprocessor. FI is accomplished by instrumenting the netlist such that a special FI circuit is inserted on the inputs of gates and registers within the netlist. Faults are activated and execution behavior monitored using a combination of scan chain control, PL-side state

Manuscript received October 23, 2020; revised January 21, 2021; accepted February 28, 2021. Date of publication March 17, 2021; date of current version February 21, 2022. This article was recommended by Associate Editor S. Ghosh. *(Corresponding author: Jim Plusquellic.)*

Jim Plusquellic is with the Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM 87131 USA (e-mail: jimp@ece.unm.edu).

Donald E. Owen, Jr., is with Sandia National Labs, Albuquerque, NM 87185 USA, and also with the Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM 87131 USA (e-mail: deowen@sandia.gov).

Tom J. Mannos is with the Advanced CMOS Products/Design, Sandia National Labs, Albuquerque, NM 87185 USA (e-mail: tjmanno@sandia.gov).

Brian Dziki is with the Information Assurance Research, Department of Defense, Fort G. G. Meade, MD, USA (e-mail: bjdziki@tycho.ncsc.mil).

Digital Object Identifier 10.1109/TCAD.2021.3065915

machines, BRAMs, and a high-speed GPIO register interface to a C program running under the Linux operating system (OS) on the PS-side of a Xilinx FPGA SoC.

As indicated earlier, our fault emulation (FE) platform is designed to quickly find faults that result in the leakage or corruption of sensitive information. Cryptographic algorithms, such as the Advanced Encryption Standard (AES), maintain internal secrets that must not be revealed, fully or in part, after a fault(s) occurs. Therefore, such algorithms represent ideal candidates for information leakage analysis. Similarly, secure pseudorandom number generators, e.g., Twister, generate random sequences that must not be corrupted such that entropy is lost in the generated sequence. To this end, we create two RISC-V instantiations and configure them to execute compiled binary code that implement the AES and Twister algorithms. The data collected from these FI experiments are classified according to the severity of information leakage and entropy loss, respectively. Future work will investigate countermeasures to prevent the types of information leakage and corruption reported on in this article.

The techniques and results presented extend our previous work using the LEON3 microprocessor running the AES 256-bit cryptographic algorithm [5], [6]. The specific contributions of this work include.

1) A flexible hardware/software co-design architecture for accelerating the FI campaign applied to a RISC-V microprocessor implementation, which offloads complex decision making to a C program implementation of the FI manager (FIM) while utilizing simple and configurable programmable-logic state machines for providing real-time analytics to the FIM regarding the current state of the RISC-V.
2) A system architecture that leverages dynamic reconfiguration (DR) to implement BRAM scrubbing, which restores program memory and eliminates corruption created from previous FI tests.
3) A hardware/software technique, which monitors RISC-V output during program execution to enable early termination of FI experiments for faults that are benign.
4) An analysis of information leakage associated with the AES algorithm [6] and of the cryptographic strength of the bitstrings generated by the Twister pseudorandom number generator [7] under single and multiple fault conditions.
5) An extensive security analysis framework that classifies the level of information leakage and data corruption associated with the emulated faults and gives the frequency of occurrence of leakage and data corruption events.

The remainder of this article is organized as follows. Section II discusses related work. Section III presents the details regarding the FI campaign. Section IV presents our experimental results. Section V presents our conclusions and future plans.

II. Related Work

The use of FPGAs for emulation began in [14], where FPGAs were used to prototype ASIC systems, and extended by Wieler *et al.* [15] to FE using an instrumented approach, where faults could be dynamically introduced using an on-chip decoder which routed separate fault injector control signals to each gate. Burgun *et al.* [16] created partial bitstreams for each stuck-at (SA) fault and used reconfiguration to statically inject the faults. In [8], a scan-chain is introduced for FI, which enabled the insertion of SA-0 (SA0) and SA-1 (SA1) faults. Cheng *et al.* [17] extended the instrumented approach by allowing multiple independent and dependent faults to be dynamically injected simultaneously as a means of minimizing FPGA reconfiguration time.

An instrumented design using modern FPGA features is proposed in [9]. Their FI system is implemented as a software/hardware system, with a 6.67-MHz PCI bus interface between the FIM running on the host and the FE engine running on the PL side of a Xilinx Virtex FPGA SoC. Their system handles single and multiple SEUs in a gate-level representation of the design-under-test (DUT). The FIM selects faults, loads the faults into a mask scan chain, and asserts an inject signal to introduce a SEU into the target FFs at the time of FI. The mask scan chain can also be used to store the current state of the target system for scan out and analysis after the FI experiment is run. Their FI system is extended to SoC/microprocessors in [2].

Lopez-Ongil *et al.* [10] and [18] addressed the host-FPGA communication bottleneck using a fully instrumented design where the FI control functions, functions for generating the stimuli and fault list, as well as the fault classification are all implemented in the PL of the FPGA. The proposed system allows single fault SEUs to be injected at specific clock cycles. A time-multiplexed version is proposed in which the FFs are replaced with a circuit that is able to store the golden and faulty values, enabling fault detection in every clock cycle. A third FF enables the DUT's state to be restored, while a fourth scan chain FF is used to specify FI points.

Vanhauwaert *et al.* [11] proposed a hardware/software co-design method, whereby an embedded PowerPC on a Xilinx Virtex-II Pro, is used in addition to the host and programmable logic (PL) to carry out the FI campaign. The proposed system reduces the number of data transfers across the UART serial connection between host and embedded processor, addressing the bottleneck in many of the previously proposed FI architectures. In [19], they eliminate the time overhead associated with scan for FI by introducing a direct addressing scheme, where a binary code specifies the fault location instead of a scan-configured mask register. Kafka *et al.* [20] proposed a methodology for preserving the structural netlist characteristics of an ASIC DUT within the FE engine of the FPGA.

CrashTest is proposed in [12] where the PowerPC embedded processor on a Xilinx Virtex-II Pro SoC runs the FIM, which communicates to the DUT configured in the PL using an on-chip bus. The authors enable the emulation of SA, bridge, path delay, and SEU fault types by applying gate-level transformations. The time and duration of the FI are controlled by the FIM, which leverages a scan chain insertion technique. The authors describe the resources and performance of applying CrashTest to several designs, including the LEON3, using the

TABLE I
ARCHITECTURAL COMPARISON OF PROPOSED FI SYSTEM WITH SIMILAR TECHNIQUES

Author, et al.	Platform components	DUT	Fault model	Injection strategy	Static/DR/DPR	Fault analysis
Hwang [8]	Host + FPGA	ISCAS-89	stuck-at-0/1 multiple, permanent	Scan	Static	Fault sim. Off-line
Civera [9]	Codesign Host + FPGA	ITC'99 SPARC v8	SEU multiple, transient	Mask-scan	Static	Reliability On-line via host
Lopez-Ongil [10]	FPGA PL	ITC'99	SEU single, transient	Mask + state scan	Static	Reliability On-line via FPGA
Vanhauwaert [11]	Host + codesign SoC FPGA	multiplier ITC'99, i8051	SEU multiple, transient	Scan + direct address	Static	Reliability On-line via SoC FPGA
Pellegrini [12]	Host + codesign SoC FPGA	LEON3 OpenSPARC	stuck-at, bridge delay, SEU transient	Scan	Static	Reliability On-line via SoC FPGA
Nyberg [3]	Host + FPGA	8051	stuck-at, SEU multiple, transient	Direct address	Static	Safety On-line via FPGA
Fibich [4]	Host + FPGA	OpenSPARC USB-to-serial	stuck-at/open delay, SEU transient	Hardwired	Static	Safety On-line via FPGA
Oliveira [13]	Host + SoC FPGA	RISC-V	FPGA config. mem. SEU	Direct config. mem. insertion	Dynamic partial reconfiguration	Reliability On-line via SoC FPGA
Proposed	Host + codesign SoC FPGA	RISC-V	stuck-at-0/1, delay, invert multiple, permanent	Scan chain	Dynamic reconfiguration	Security Off-line

results obtained when a set of single and multibit faults are selected using a Monte Carlo sampling method. CrashTest is used to validate software anomaly treatment (SWAT) methods in subsequent FI experiments carried out on the OpenSPARC T1 core [21].

Nyberg *et al.* [3] described an FE engine that minimizes stalls between FI experiments while providing full single-bit and multibit FI capabilities. They utilized the direct addressing scheme proposed in [19] but added the capability to specify multibit faults using a subset mask. They also configured only the delta between successive FI configurations and allowed the fault model and fault duration to be specified. Their FIM allows simultaneous download of the next FI configuration while the current FI experiment is executing. They proposed a fault analysis technique that tolerates timing differences that can occur when FI is carried out on microprocessors.

Fibich *et al.* [4] argued that implementing FI in HDL changes the result of the synthesis and advocate instead for gate-level FI. Their tool, called FI instrumenter or FIJI, inserts saboteurs at locations specified by the user using a graphical user interface and implements a set of fault models similar to Pellegrini [12]. FIJI is designed to reduce the overhead associated with the FE engine while enabling precise timing and duration of fault insertions.

de Oliveira *et al.* [13] implemented the RISC-V Rocket processor on an FPGA and evaluate it in the presence of single event effects, with a focus on radiation-induced errors that occur in the configuration memory and BRAM. They introduced coarse grain TMR and memory scrubbing as countermeasures and measure time-out and silent data corruption (SDC) errors while running matrix multiply, AES, and Qsort algorithms.

A comparison of important architectural features associated with the previous work and the architecture proposed in this article is given in Table I. Note that we use the prefix "codesign" to refer to platform components that work together to implement the FI system. For example, "Codesign Host + FPGA" indicates that the Host is actively participating in the FI campaign, while "Host + codesign SoC FPGA" indicates that FI is implemented between the PS and PL sides of the FPGA, while the host provides only supplemental support. As a general rule, architectures that build larger fractions of the FI control elements into the FE engine achieve higher performance at the cost of increased complexity. Although several previously proposed methods are classified as "Host + codesign SoC FPGA," none of them are focused on a security evaluation while enabling runtime monitoring as proposed here.

III. FAULT INJECTION CAMPAIGN CHARACTERISTICS

The FI campaign refers to all aspects of the FI architecture, and is summarized as follows with each step explained in further detail in the following. Note that unlike the previous work, e.g., [3] and [10] among others, our goal is not to build the fastest FI architecture, but rather to architect a flexible and efficient engine that enables a thorough and statistically significant characterization of sensitive information leakage and data corruption. In other words, our security framework is focused on identifying nodes that under fault conditions, result in the leakage of private data, in contrast to a reliability analysis, which takes a more generalized view of identifying critical nodes that result in catastrophic failure when a fault occurs. Once identified, our long-term goal is to introduce countermeasures that prevent these security-sensitive nodes from revealing private information when they fail. Our framework, like a reliability analysis, cannot guarantee that we will find all security-sensitive fault conditions but instead exhaustively tests all single faults and applies statistical sampling techniques to obtain as much coverage as possible of multiple faults.

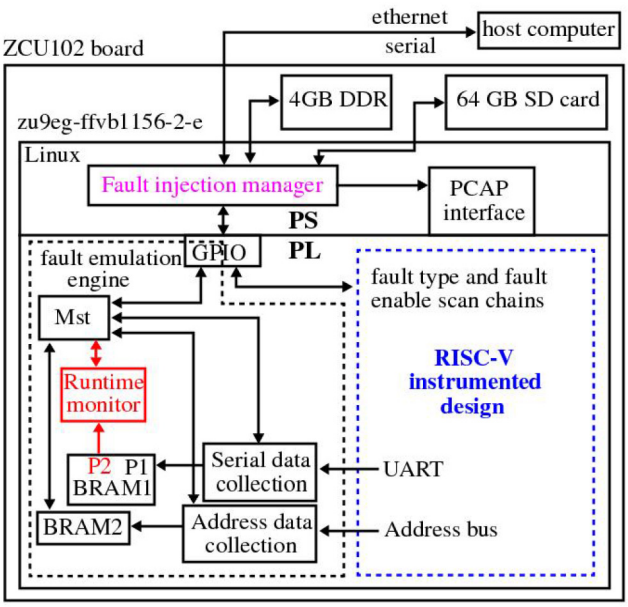


Fig. 1. Block diagram of the experimental setup with RM for accelerating data collection.

1) The DUT is the RISC-V River SoC (Rocket) with a 32-KB ROM for the application code and a 512-KB BRAM for scratch memory [22], [23]. An ASIC synthesis and place and route CAD tool flow is used to create a netlist, which is then processed into an instrumented design with faults inserted. A second FPGA CAD tool flow is used to process it into a bitstream.
2) The FIM runs as a C program on an embedded processor within a MPSoC FPGA, which communicates with the PL components through memory-mapped registers, similar to the approach taken in [24].
3) The set of emulated fault types include permanent SA-0/1, invert, and delay faults.
4) All single-bit faults and up to five randomly placed multibit faults are emulated in the experiments.
5) The FE engine consists of a set of state machines implemented in the PL along with the instrumented design. FI is carried out in the instrumented design using a set of scan chains, similar to the approach taken in [8] and [18]. The FE engine and scan chains are controlled by the FIM through the GPIO registers.
6) The bitstream is either programmed once (static) before the FI campaign or dynamically in between each FI experiment (dynamic).
7) The FE process implements a hybrid approach in which: a) a fault-free emulation is detected and terminated early by comparing a portion of the generated output with golden data and b) a faulty operation continues until a stop condition is met.
8) Sensitive data leakage and corruption analysis and classification are run offline using the faulty data sets.

A. Test Platform Design Characteristics

The proposed FI platform utilizes a Xilinx ZCU102 development board as shown in Fig. 1. The ZCU102 includes a Zynq UltraScale+ MPSoC that is partitioned into a processor side (PS) and PL side (PL) side. Petalinux is used to configure a custom Linux OS that runs on one of the 4 ARM Cortex-A53 processors embedded within the PS side [25]. The Linux OS provides both serial and Ethernet communication channels to a desktop server (host computer), as well as access to a 16-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 implementing the FIM can read and write these registers after obtaining virtual address information using the *mmap*() library function.

The FIM implements two distinct testing strategies, called static configuration and DR. The static configuration experiments program the FPGA exactly once at the beginning of the FI campaign while the DR experiments fully restore the fault-free state by reprogramming before each FI experiment. Both strategies use the processor configuration access port (PCAP) to perform reconfiguration of the PL side components. PCAP enables either the entire PL-side to be reconfigured (DR) or only portions of it (dynamic partial reconfiguration). The FI experiments described in this work use only DR.

The FIM coordinates operations with the FE master state machine (*Mst*) as shown in Fig. 1. The bit field connections associated with the GPIO control (GPIO Ins) and data (GPIO Outs) registers are shown along the top of Fig. 2. The scan chain control and data signals connect directly to the Rocket instantiation, providing the C program with complete control over the FI configuration.

The set of control flow operations carried out by the FE engine are given in the algorithmic state machine diagram (ASMD) in the bottom half of Fig. 2. At the beginning of each FI experiment, the cycle counter is reset and the state machines controlling the serial (UART) and parallel (address bus) data collection processes are started to clear the BRAMs. A configuration parameter that limits the number of cycles per test is then transferred from the FIM to the FE engine via the **get_params** and **wait_params** PS-PL handshake states.

The reset signal to Rocket is asserted in the **wait_clear** state and the run states of the serial, parallel, and runtime monitor state machines are consulted via the *xxx_done* signals to ensure the BRAM memory clear operations have completed. The Rocket reset signal is then released in the **start_rocket** state. This causes the microprocessor to boot and then execute a compiled C program from the preconfigured ROM. *MstCtrl* busy-waits counting clock cycles and then forces the Address (parallel) data collection state machine to halt after the designated number of clock cycles. A transition is then made to state **wait_serial**, which sets the flag *exit_enabled* and then busy waits for the Serial state machine to complete.

The ASMD of the serial data collection (SDC) state machine is given on the left side of Fig. 3. The "clear memory" operation is carried out if *Mst* starts this state machine with the *clear_mem* flag set (states are not shown

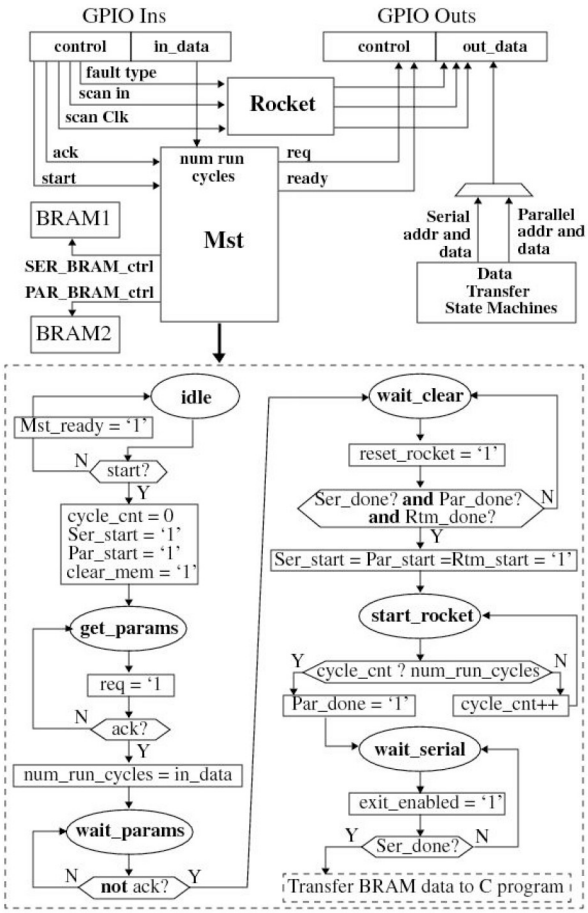


Fig. 2. Details of the PL-side components from Fig. 1 (top) and ASMD for master control state machine (bottom).

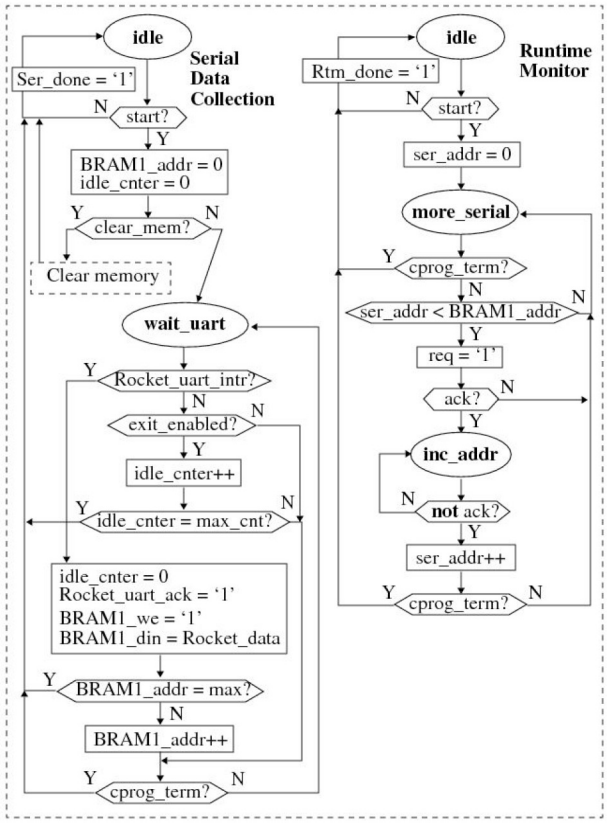


Fig. 3. (a) ASMD for SDC state machine and (b) ASMD for RM state machine from Fig. 1.

because they are straightforward). Otherwise, a transition is made to state **wait_uart**, which is responsible for collecting UART data as Rocket executes its program. Rocket asserts the *Rocket_uart_intr* interrupt signal when an ASCII character is available on its UART port. When this occurs, SDC acknowledges the interrupt while simultaneously transferring the character to the serial BRAM1. The *BRAM1_addr* is checked to determine if it has reached the maximum size of the available memory, and if so, the state machine returns to **idle**, otherwise, the address is incremented and a FIM (C program) flag, *cprog_term*, is checked. This flag enables the C program to terminate SDC at any instant in time. If the *Rocket_uart_intr* is not asserted, SDC checks the *exit_enabled* signal controlled by *Mst*, which as indicated earlier is not asserted until Rocket has run for at least 2^{22} cycles. If *exit_enabled* is not asserted or *idle_cnter* is less than a user-specified wait time (set to 2^{20} cycles in our experiments), then the SDC continues to monitor for additional UART activity, otherwise, it returns to **idle**. Note that *idle_cnter* is reset to 0 every time an interrupt occurs, allowing SDC to extend the data collection period significantly in special cases discussed further below.

The FE engine is modified for the Twister FI experiments as shown by the red annotations in Fig. 1. As discussed in detail later, the number of bytes transmitted through the UART interface between Rocket and the FE engine is no larger than 288 bytes in the AES experiments, but increases to more than 250 000 bytes in the Twister experiments, which dominates the runtime. However, in fault-free and some faulty test scenarios (to be discussed), the time overhead associated with the UART transfer can be reduced by monitoring the UART output as it is being generated. The runtime monitor (RM) shown in Fig. 1 is responsible for transmitting data back to the FIM in real time using a second BRAM port (P2). This enables the FIM to terminate the FI experiment early under certain conditions.

The ASMD for the RM is given on the right side of Fig. 3. Once started by *Mst*, it monitors the serial BRAM1 for additional characters written by the SDC state machine and transfers them to the FIM immediately. BRAM1 is a buffer between the SDC and the FIM, which ensures complete data transfers in cases, where the FIM is stalled by the Linux OS because of interrupt service processing.

The FI platform is designed to be adaptable for other types of algorithms and data collection requirements. Currently, Rocket needs to be resynthesized in order to execute other binaries, but this can be overcome by adding a port to enable the ROM to be loaded at runtime. Another feature that supports the adaptability of the FI platform is the RM, which can be used to accelerate the FI experiments (as we show here) by enabling control over execution based on the current state of Rocket. More importantly, it defers decisions, such as early termination, to the highly configurable FIM (software) component. The most difficult element of the FE engine to make adaptable relates to the type of state information to be

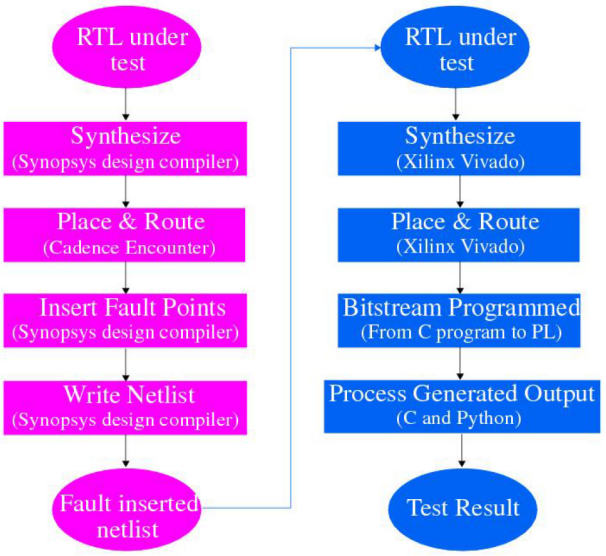


Fig. 4. ASIC synthesis, instrumentation, and FPGA implementation flow of Rocket design.

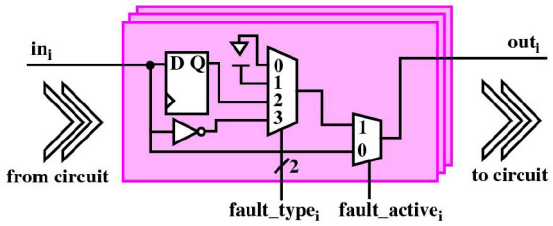


Fig. 5. FI circuit design used to inject faults on each gate input.

collected. The current version collects data from the serial port and address bus, which are highly sensitive to microprocessor misbehaviors with injected faults, but the runtime state of other components such as the program counter, ALU control signals, etc. may also need to be monitored. This can be accomplished by expanding the FE engine with small footprint state machines similar to those shown in Fig. 3.

B. RISC-V Synthesis

The objective of our analysis is to investigate the behavior of Rocket under faulty conditions in usage scenarios in which Rocket is implemented as an ASIC. ASIC synthesis is typically carried out using a standard cell library CAD tool flow. We use this approach to create a logic-level netlist that is then postprocessed into an instrumented design with faults. The CAD tool flow is shown in Fig. 4, where we first process the rocket behavioral description (RTL) using Synopsys design compiler [26]. The ASAP7 7-nm FinFET predictive PDK and standard cell library [27] is used during RTL synthesis as shown on the left side of the figure. The generated netlist is then processed through place and route to a layout using Cadence Encounter [28]. The netlist from the layout is extracted and a script inserts instances of a saboteur circuit in series with downstream gate inputs, along with a set of controlling scan chains.

The right-most column of Fig. 4 shows the standard synthesis-implement-generate-bitstream process flow

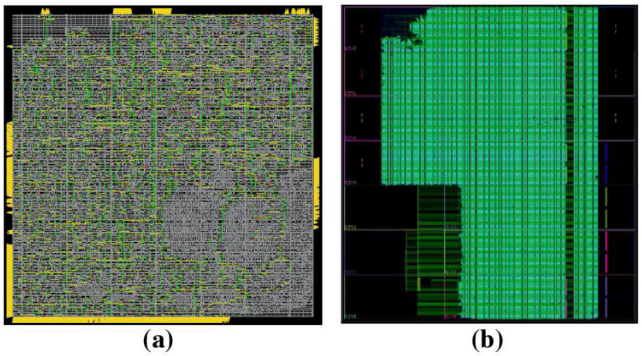


Fig. 6. (a) RISC-V layout with clock highlighted and (b) Zynq UltraScale+ FPGA version.

implemented within the Xilinx Vivado FPGA CAD tool [29]. Note that the scan chains in the instrumented design prevent Vivado from collapsing multiple standard-cell gates into a single LUT, and therefore, the netlist structure is preserved throughout the Vivado flow. Maintaining the ASIC netlist structure is an important objective of our CAD tool flow to ensure the results reported from our experiments are directly transferable to a structurally equivalent ASIC implementation.

C. Instrumented Design Flow

The instrumentation design flow adds three scan chains for controlling a set of 85 713 FI circuits, one on every input of every gate in the Rocket netlist. A schematic of the FI circuit is shown in Fig. 5. Two scan chains are used to select from one of four fault types while the third scan chain controls the *fault_active* signal. 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 *fault_active* signal either activates the fault or provides a fault-free by-pass path through the cell.

Fig. 6(a) shows the ASIC layout of Rocket with the system clock highlighted while Fig. 6(b) shows the Vivado implementation view of Rocket on the Zynq UltraScale+ FPGA. The ASIC layout is 336 mm X 334 mm with 60% cell utilization. There are 34 196 logic gates, including 5262 scannable flip-flops and 2414 hierarchical ports. Fault insertions include all combinational inputs and flip-flop data and scan-enable inputs, for a total of 85 713 fault insertion points.

The time associated with the one-time processing of the RTL through fault insertion (from Fig. 4) is small relative to the Vivado synthesis and data collection times. In contrast, the runtime associated with the Vivado standard CAD tool flow is approximately 6 h. We limited the operating frequency of the FPGA to 24 MHz because anything higher resulted in timing violations during the Vivado implementation phase. For comparison, synthesis time and resource utilization are much smaller, i.e., approximately 2 h with only 20% utilization of the LUTs and FFs, when faults are inserted directly into the behavioral RTL. However, as noted, this approach does not represent a realistic fault model of the ASIC implementation.

D. Testing Process

The FIM 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 controls the number of simultaneous faults, while the inner loop carries out the fault insertion by clocking in configuration data using the three scan chain inputs. Once the fault(s) is inserted, the FIM starts *Mst* to execute the test, collect test data from the BRAMs and, to receive signals regarding the status of the Rocket execution. The collected test data are stored onto the SD card and later transferred to the host computer through Ethernet.

Each FI experiment consists of optionally reprogramming the FPGA, introducing one or more faults through the scan chain and then resetting Rocket, which causes it to boot and then execute either the AES [6] or Twister [7] programs. The FIM automatically carries out a set of 342 852 FI tests (four fault types * 85 713 FI sites) for each of the AES and Twister instrumented designs. We repeated these experiments by inserting up to five faults of the same type in sequence, i.e., in adjacent FI cells in the scan chain. We again repeated these experiments with two different key-plaintext pairs in the AES experiments as a means of identifying data dependent fault behaviors. In total, 3 428 520 and 1 714 260 FI experiments were carried out using the AES and Twister algorithms, respectively.

The soft reset operation clears the registers and resets the execution state of the Rocket microprocessor but it does not reset the internal BRAM memory utilized by Rocket. Faults that disrupt Rocket's internal BRAM memory will accumulate throughout the FI campaign. To provide a baseline for determining how uncorrected accumulated faults impact the leakage and data corruption characteristics of the algorithms, we carried out a separate set of experiments where we reprogram the PL side before running each test. Reprogramming is easily accomplished using the PCAP interface by: 1) storing the bitstream in */lib/firmware/* within the Linux filesystem and 2) making the following system call before carrying out the scan-based fault insertion operation:

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

In experiments, where DR is not performed, Rocket exhibits distinctive behavior in some cases as a result of the accumulation of residual errors in Rocket's internal BRAM resources. In contrast, the DR experiments reset the BRAMs to its initial state, eliminating memory corruption from previous faults. The total number of experiments is, therefore, twice that given earlier at 6 857 040 and 3 428 520 for each of the AES and Twister FI campaigns, respectively.

E. Security-Oriented Algorithms

1) AES Test Case Characteristics: The AES algorithm that was compiled and incorporated in the ROM uses a 256-bit key and is freely available at [30]. For each FI experiment, we configured AES to fetch the key and plaintext from the ROM and then print them along with the computed ciphertext in hexadecimal to Rocket's serial port. The program also compares the generated ciphertext with a precomputed stored copy of the ciphertext and prints "Correct" or "Error." Under fault-free conditions, it then enters an idle state. An example snippet of the data collected by the FIM for a fault-free test is shown in Fig. 7.

```
<------ test 2 -------->
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
Correct
----
number of serial bytes: 288
number of address line changes: 52147
1007FB80        ⎫
1007FB90        ⎪
1007FBA0        ⎬ 50 addresses
1007FBB0        ⎪
...             ⎭
```

Fig. 7. Fault-free AES serial and address bus change data snippet.

As the processor executes the binary code, the SDC state machine collects and stores the sequence of ASCII characters generated on the Rocket UART as described earlier. A second address data collection (ADC) state machine runs in parallel (see Fig. 1) and records activity on Rocket's internal address bus. A separate 2048-word BRAM2 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 address bus changes are also recorded and stored (see Fig. 7). The serial data and the last 50 address bus changes are transferred to the FIM after the test completes. We determined that the last 50 addresses, in combination with the serial data, were sufficient to differentiate between distinct failure behaviors.

Several stop conditions are programmed into the SDC to deal with faults that prevented the AES program from completing execution. The number of cycles required for the fault-free AES encryption program execution is upper bounded by 2^{22}, therefore, every FI experiment is allowed to run for this number of clock cycles unabated. After this initial clock cycle interval, the SDC module then begins monitoring for additional byte transfers from the Rocket UART interface. If no additional characters appear after 2^{20} additional cycles, the FE engine signals the FIM to terminate the run. This occurs in the fault-free case and in cases where the fault causes the processor to hang and generate no output.

If, on the other hand, Rocket continues to transmit characters through the UART, the SDC module collects them until the 64-KB BRAM buffer fills up. In this case, the emulated fault causes the processor to malfunction and extends the runtimes in these rare cases into the minute range. We choose to collect this data because the additional output occasionally reveals internal secrets related to the AES encryption key.

From Table I, the best case runtime per FI experiment for the static configuration is approx. 220 ms, which occurs for most of the tests. At a frequency of 24 MHz, the $(2^{22} + 2^{20})$ clock cycle runtime takes approx. 218 ms with the scan and GPIO data transfer operations taking the remaining 2 ms. The total scan time is negligible because only one scan clock operation is required to move the fault to the next position in the scan chain and each scan clock operation takes 488 ns (we connected the scan clock from the GPIO register in Fig. 2 to a clock tree in the PL to make it fast). The GPIO transfer time is also small because at most 288 serial bytes and 50 address bus transfers occur (see Fig. 7).

TABLE II
FE ENGINE RUNTIMES

Experiment	Ave. DR time (ms)	Ave. scan time (ms)	Ave. GPIO transfer time (ms)	Best/ave/worst time per test
AES, static	0	0.000488	1.7	220 ms/NA/5 mins
AES, dynamic	215	21	1.7	456 ms/NA/5 mins
Twister, static	0	0.000488	0.61	211 ms/1.64 secs/16.7secs
Twister, dynamic	215	21	0.61	447 ms/1.88 secs/16.9 secs

```
<------ test 709 -------->
Number of 'Boot' strings 1
Number of non-ASCII chars 0
Force correct 1
Num matching 35000
Num chars fetched/checked 35000
Time (secs): 1.641464
f3a8d24e
57286251
ed0bb215
c54297c6
1372d3d1
9aebb2fd
4074858d
...
----
number of serial bytes: 35000
number of address lines: 19550
1007E830
1007E820
1007E830
80001000
...
```

Fig. 8. Fault-free Twister serial and address bus data snippet.

```
<------ test 736 -------->
Number of 'Boot' strings 1
Number of non-ASCII chars 0
Force correct 0
Num matching 14
Num chars fetched/checked 281264
Time (secs): 16.761667
Boot . . .OK^M
495c689c
a8ffa91a
92bd95d3
fd69d332
f8bde9da
64f0c8f0
8ce56963
...
----
number of serial bytes: 281264
number of address lines: 19550
1007FBD0
1007E830
1007E820
1007E830
...
```

Fig. 9. Faulty Twister serial and address bus data snippet.

The best case runtime for the DR experiments is more than twice that of the static experiments because of the DR and scan operations. As noted, the PL side is reprogrammed before each test, which requires an additional 215 ms, and the scan operation must start over because the fault is cleared by the reprogramming. The 21 ms for scan represents the average scan time given the scan chain length is 85 713 elements long and half of the elements need to be scanned on average. The total data collection time for each fault type experiment (which tests all 85 713 faults) is approximately 6 h for the static configuration experiment and 12 h for the DR experiment. In either case, our FE setup represents a significant speed-up over simulation, which would take nearly ten years per experiment at 1 fault per hour.

2) Twister Test Case Characteristics: For each FI experiment, Twister is started with a fixed random seed and is configured to generate one million bits to satisfy the NIST statistical test suite requirements discussed later. The bits are packed four at a time into hexadecimal ASCII characters by Twister before being transmitted over the UART to the SDC state machine. Therefore, the number of UART characters in the fault-free case is 281 264 bytes, which accounts for the hex-encoded pseudorandom numbers, the boot message, etc., as shown by the fault-free data snippet in Fig. 8.

The fault-free pseudorandom sequence does not need to be stored, only the faulty sequences. Therefore, we configure the FIM and RM to parse the UART output and decide if the sequence is fault free. We determined from full runs of the FI experiments that no latent fault effects show up after 35 000 hex digit characters of the pseudorandom sequence are generated. Therefore, the FIM terminates the run early for fault-free test scenarios and replaces the pseudorandom sequence in the data files with the string "CORRECT."

The output snippet in Fig. 8 also displays information about other characteristics of the UART output stream. For example, the "Number of "Boot" strings 1" indicates that only one boot operation occurred. We configure the FIM to terminate the execution if more than 10 of these boot strings occur to avoid the long wait times described earlier in the AES experiments. Similarly, the FIM will terminate Twister execution if the number of non-ASCII characters is >10. Although this type of output represents a security issue, it can be easily flagged as abnormal by countermeasures and the program terminated.

Of greater concern are faults that produce well-formed but incorrect (WFBI) output sequences, i.e., the fault causes Twister to deviate from its expected execution behavior to produce a possibly nonrandom sequence. The sample output snippet in Fig. 9 shows one such example. The FIM is configured to identify these WFBI sequences and, in cases where they occur, Rocket is allowed to execute until all 1 million bits are generated. The focus of our security analysis in the following sections is on these output sequences.

The runtimes given in the right-most column of Table II reflect the different termination conditions just discussed. For example, for faults that cause processor hangs, continuous reboots, or non-ASCII character output, the FI experiment is terminated quickly, resulting in runtimes of 211 and 447 ms, for the static and DR experiments, respectively. Fault-free FI experiments are also terminated early at 1.64 and 1.88 s. FI experiments producing WFBI sequences take nearly 17 s. Note that the RM enables the serial data transfer to be carried out simultaneously with execution of Twister, effectively eliminating the serial data transfer time component. The 0.61-ms time interval given in the table represents the transfer time associated with the 50 address bus values. The total data collection time for each fault type experiment (which tests all

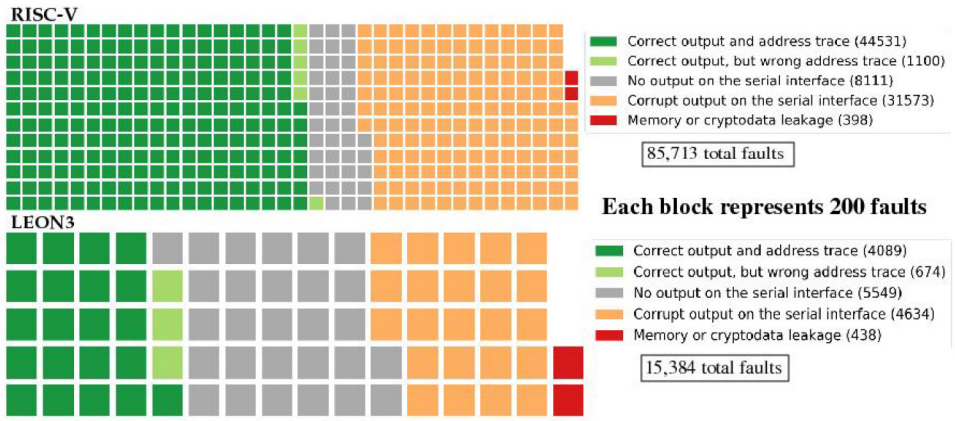


Fig. 10. Comparison of Leon3 and RISC-V results. Graph shows proportion of fault behaviors collapsed to five major classes.

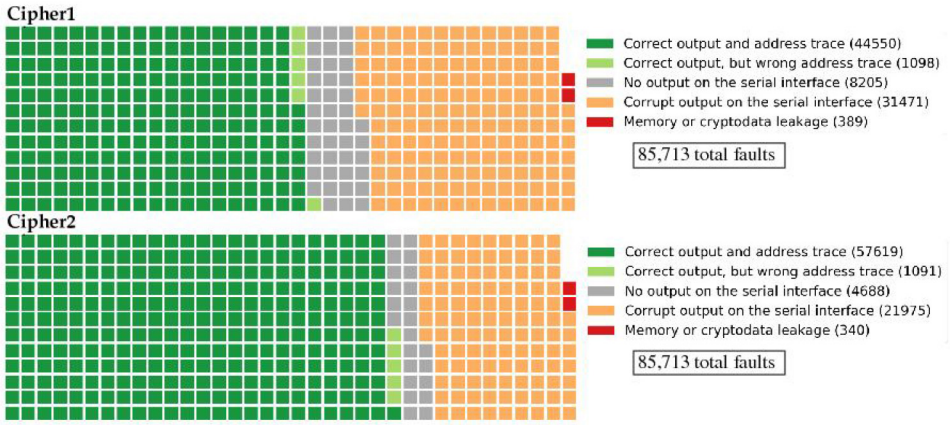


Fig. 11. Impact of data dependency on fault behavior using worst-case severity classification among all four fault types in the static configuration experiments with one simultaneous fault inserted.

85 713 faults) varies from 68 to 70 h for both the static and DR experiments because the WFBI output sequences dominate the runtime.

IV. Experimental Results

The results from our security analysis are presented in this section, with an emphasis on faults that cause at least some portion of the key and/or plaintext to be leaked through the UART for the AES experiments and, for Twister, on faults that produce WFBI sequences, a condition referred to as "insecure failure" in [31]. We are particularly interested in WFBI sequences that appear to look random but fail the NIST statistical test suite [32].

A. AES Analysis

The UART output and address bus traces vary widely across the FI experiments. However, despite this diversity, we were able to partition the set of fault behaviors into five severity levels with the last level representing the insecure failure class of interest. The five severity levels are characterized as follows.

1) *Correct Output and Address Trace:* Fault is benign, with UART and address bus output matching fault-free output.
2) *Correct Output But Wrong Address Trace:* The UART output matches perfectly, but the address trace is incorrect (the number of memory accesses is different from expected and/or the last 50 addresses accessed do not match the fault-free case).
3) *No Output on the Serial Interface:* Processor is hung.
4) *Corrupt Output on Serial Interface:* UART output is corrupted in some fashion. We observed a lot of diversity in this severity level, e.g., sometimes the "enc" and "tst" components in Fig. 7 are correct, other times they are not, sometimes the output is non-ASCII, other times it is a combination of ASCII and non-ASCII, or is ASCII but has missing or extra characters.
5) *Memory or Cryptodata Leakage:* The UART output displays the key or plaintext as part of the encrypted message, or otherwise leaks cryptographic data from memory.

The block graphs shown in Fig. 10 compare the current results (top) with those published previously for the LEON3 processor (bottom) [5]. The data from the SA0 static configuration experiment with one simultaneous fault are used here for both experiments. Each block represents 200 faults from one of the five color-coded severity levels. Here, we see Rocket has approximately the same number of leakage susceptible faults as LEON3 (398 versus 438), but has a lower percentage than LEON3 (0.46% versus 2.8%). Moreover, approximately half of Rocket's gate-input pins are single-fault safe.

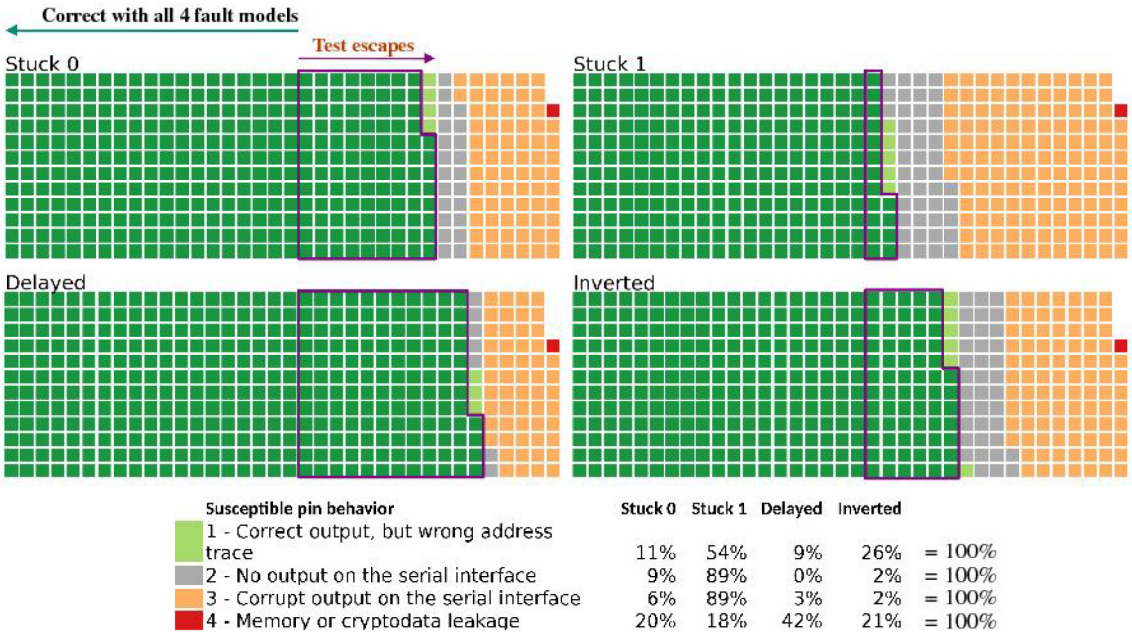


Fig. 12. Fault type analysis, showing breakdown according to severity class.

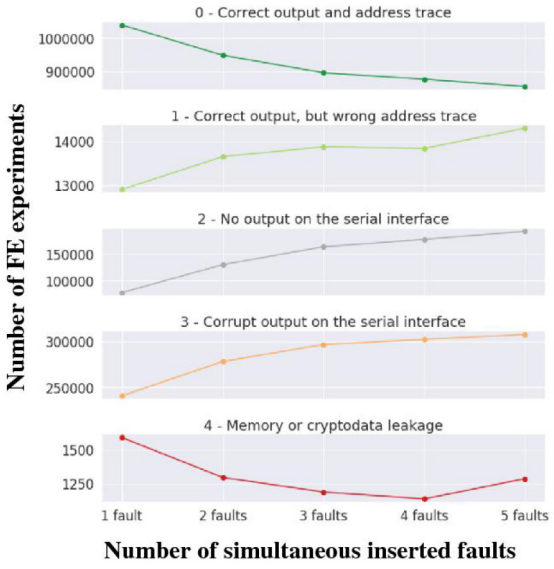


Fig. 13. Trends in the fault severity levels as additional faults are injected.

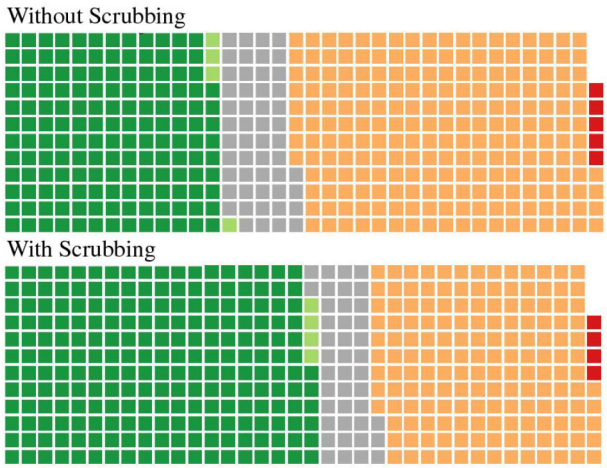


Fig. 14. Static (without BRAM scrubbing) and dynamic (with BRAM scrubbing) experimental results showing breakdown according to the severity class.

The impact of data dependencies on fault behavior is depicted in the block graphs of Fig. 11, which uses data from all four fault type experiments with 1 simultaneous fault inserted. For each fault site, the severity class for each of the four fault types is parsed and the fault type classified as worst-case (largest) is used as the final classification for that fault site. The top block graph shows the fraction of faults for each severity class when the encryption key/plaintext pair from Fig. 7 is used (cipher1) while the bottom graph shows the results when a second, different key/plaintext pair is used (cipher2). The cipher1 FI experiments detected considerably more null and corrupt output conditions and slightly more leakage conditions than the cipher2 experiments. This is not unexpected given that execution behavior, i.e., whether an *if* or *else* branch is taken, depends on the data being processed even under fault-free conditions. The key take-away of this analysis is that fault countermeasures must be evaluated using multiple key/plaintext pairs to be inclusive.

The severity-level block diagrams shown in Fig. 12 help determine which of the fault types is best at detecting susceptible pins in the layout. Here, the color-coded blocks count the number of fault sites associated with each severity class separately in each fault type experiment. The regions labeled "Test escapes" show the percentage of fault sites that are classified as "correct" in the associated fault type experiment but are classified by at least one other fault type experiment as "incorrect". Although each fault type identified susceptible pins not detected by any other fault type, SA1 identified the largest number of new susceptible pins across levels 1 through 3.

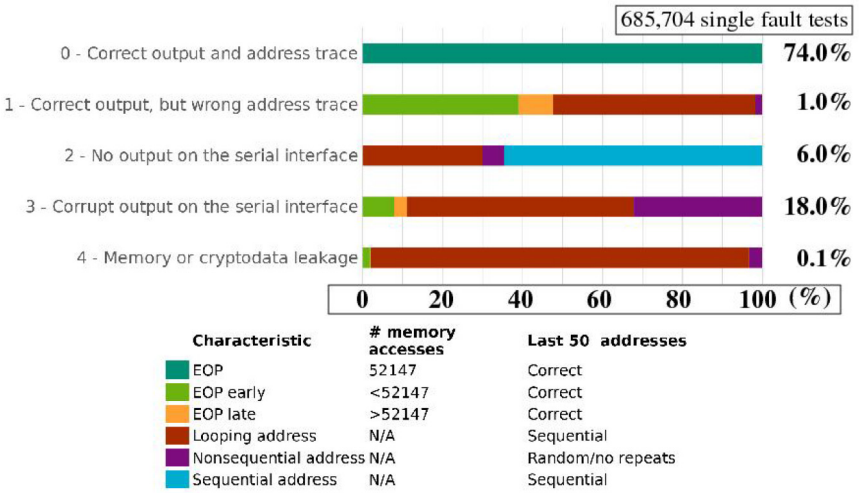


Fig. 15. Correlation of fault-free and faulty UART output with Address bus behavior.

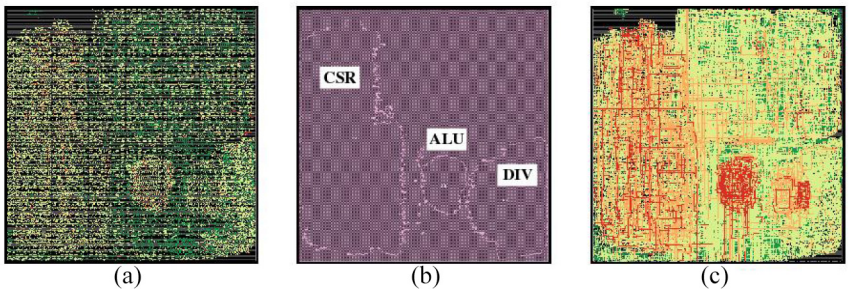


Fig. 16. (a) Standard cell placement, (b) hierarchical cell grouping, and (c) locations of nets and pins resulting in sensitive information leakage.

Moreover, the delay fault type found twice as many new susceptible pins in level 4 (leakage) than any other fault type. This observation is reinforced by the table shown along the bottom of Fig. 12. Here, the percentages reflect the number of unique faults in each severity class > 1 that each fault type experiment detected normalized to 100% across the fault type experiments. Clearly, SA1 and delay dominate the fractions across the severity classes.

The graphs in Fig. 13 portray the trends in fault behavior as the number of simultaneous faults is increased. As expected, fault-free behavior decreases and faulty behavior increases as more simultaneous faults are added with the notable exception of severity class 4. Here, leakage trends unexpectedly downward but then upticks with five simultaneous faults.

The static and DR experiments are differentiated only by BRAM scrubbing. The block graphs in Fig. 14 show the severity-level distributions without BRAM scrubbing (top) and with BRAM scrubbing (bottom). The larger fault-free operation region associated with scrubbing indicates that scrubbing as a countermeasure is somewhat effective in reducing null output, corrupt output, and leakage severity levels. However, additional mitigations are necessary to ensure a secure, robust system.

1) Correlating Address Traces With Program Behavior: As discussed earlier, we additionally monitored activity on Rocket's address bus in order to better understand its execution behavior under faulty conditions. Here, we analyze the address bus behavior and correlate it 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 condition does not hold as well, i.e., correct UART output was not always accompanied by correct execution behavior.

The legend in Fig. 15 identifies six address bus change classes, with EOP indicating end-of-program. Classes EOP early and EOP late identify cases, where the number of address bus changes is less than or greater than the expected, respectively (see snippet in Fig. 7). Looping address identifies cases where Rocket loops indefinitely. The nonsequential address class indicates address changes are random with no repeats while the sequential address class identifies cases, where the address simply increments. The five horizontal bars partition the results into the five UART severity levels discussed earlier, with the fractional number of cases for each of the address bus change classes color-coded according to the legend.

The bar labeled "Correct output, but wrong address trace" refers to anomalies where the address bus behavior is wrong but the UART output is correct. In fact, we observed approximately 2500 of the tests that terminated early still produced the correct output. From the color-coding of the bottom-most bar, it is clear that most of the crypto-leakage behavior is associated with the Looping address class. Another notable feature depicted in the bar labeled "No output on the serial interface" is that most of the null output behavior is associated with a

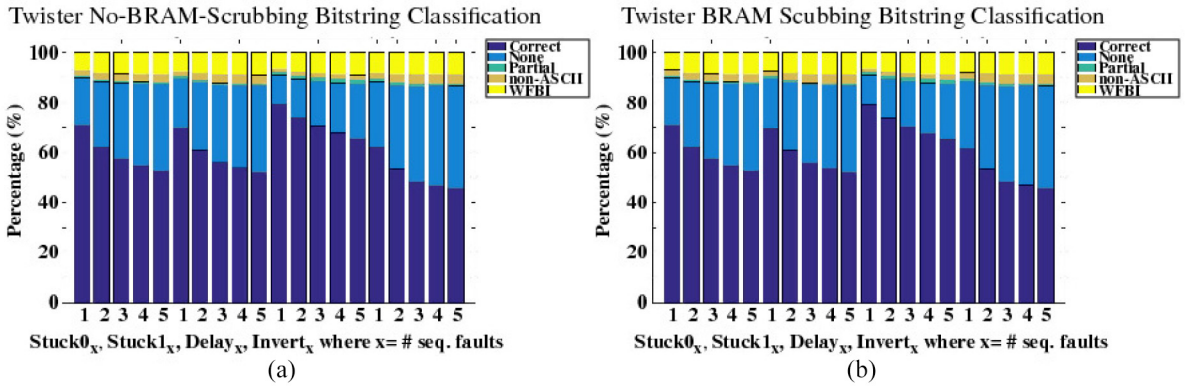


Fig. 17. Percentage breakdown of the 85 713 faults partitioned into five classes for 20 FI experiments given along the *x*-axis. The stacked bargraph in (a) gives the results without BRAM scubbing while (b) gives the results with BRAM scrubbing.

runaway sequential address. More importantly, this analysis reveals that the address classes are generally not predictive of program behavior.

2) Faulty Pin Locations: Using the UART output data and fault insertion location information, it is possible to back-trace to the layout those pins that result in cryptographic information leakage. The layout of Rocket is shown in Fig. 16(a) and (b), a specialized layout view showing the regions occupied by the components in the hierarchical design. The small circular region in the bottom center corresponds to the *ALU* component of Rocket. Fig. 16(c) uses red lines to identify the net connections between leaky pins. It is clear that the ALU component possess a large number of nets and pins that result in leakage when faults occur on them. The larger region to the left corresponds to the CSR, which also depicts a large number of leaky nets and pins, albeit more widely dispersed. Note that the ALU is composed solely of combinational logic gates while the CSR additionally includes registers, which are physically larger and, therefore, reduce the density of the faulty nets and pins. We are currently leveraging this back-annotation information for determining the best regions for inserting monitoring technology that is capable of detecting these leakage sensitive faults.

B. Twister Analysis

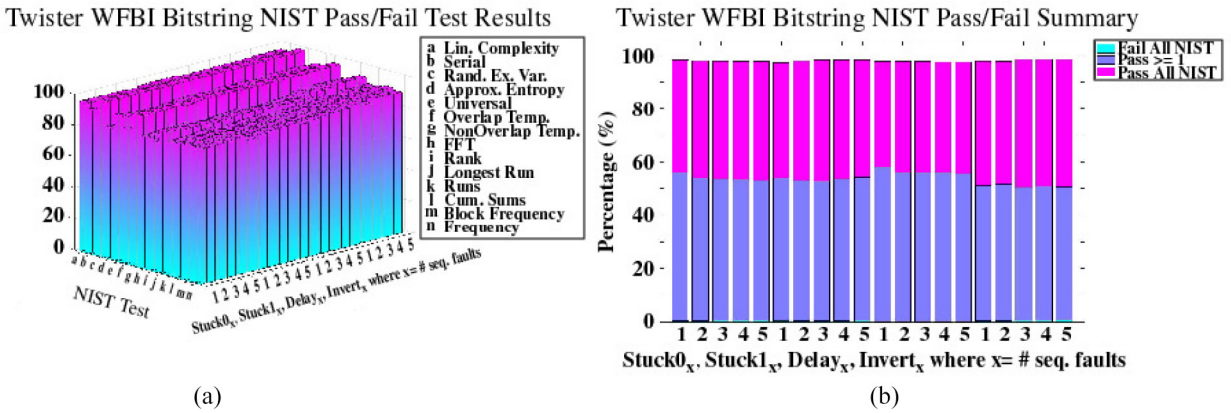
As indicated earlier, the FIM is configured to identify and fully collect Twister WFBI sequences that are produced in the correct format but do not match the fault-free sequence. Twister is programmed to produce 1 million bits under these conditions, which satisfies the input size requirements to run nearly all of the NIST statistical tool suite tests [32].

Note that the emphasis here is not on whether the ciphertext output is either correct or incorrect, or whether certain faults cause leakage of cryptographic key information as is true for the AES experiments, but rather on whether the strong cryptographic properties of Twister's pseudorandom sequence are weakened or eliminated. Unlike the key and ciphertext within AES, the random seed used in Twister can change frequently and therefore, it is difficult to design countermeasures here to determine on-the-fly whether the output sequence matches the expected sequence. The most straightforward countermeasure is to run simultaneous synchronized copies of Twister on two separate microprocessors and compare the outputs, but this may not be feasible and/or practical in some applications.

Similar to the AES experiments, the UART output observed in the Twister experiments varied widely. We classify the observed UART output into five categories, namely, Correct, None, Partial, non-ASCII, and WFBI. Correct indicates that the fault had no effect on Rocket's execution and the fault-free bitstring was produced. None indicates that no bitstring was produced, likely caused by Rocket entering some type of hung state. Partial refers to an ASCII bitstring with length less than the requested 1 million bits. The non-ASCII class refers to ROM dumps of the program code or other internal binary state information, and WFBI refers to well formed but incorrect full length ASCII bitstrings.

The stacked bar graphs in Fig. 17 give the fractional breakdown of the 85 713 faults within each of these five classes for each of the fault classes listed along the *x*-axis. The results are very similar for the experiments without BRAM scrubbing [Fig. 17(a)] and those with BRAM scrubbing [Fig. 17(b)]. For example, the WFBI class shown by the top bar segment varies between 6.72% for Delay_1 (only one delay fault) and 9.11% for Stuck1_5 (five simultaneous SA1 faults) without BRAM scrubbing and between 6.71% and 9.09% for the same FI experiments with BRAM scrubbing. Within each of the four fault classes, the fraction of WFBI cases increases by approximately 2% as the number of simultaneous faults increases from 1 to 3 but remains nearly constant for three, four, and five simultaneous faults. This trend contradicts the results shown earlier for AES where leakage decreased as the number of simultaneous faults increased except for the case of five simultaneous faults. Therefore, countermeasures must consider that as the number of simultaneous faults increases, the trend in the amount of leakage or data corruption will be application dependent.

Another notable trend in the graphs is the nonlinear decrease in the number of Correct bitstrings produced as the number of simultaneous faults increases from 1 to 5, and a corresponding fractional increase in the None class. Moreover, the number of Correct bitstrings is slightly larger, by less than 0.1% or approximately 100 bitstrings from a total of 85 713, in the DR experiments which indicates that scrubbing repaired corruption that carried forward from the previous fault in only



(a) (b)

Fig. 18. (a) NIST test (*y*-axis) pass percentage for 20 static FI experiments (*x*-axis) using the WFBI bitstrings. (b) Percentage breakdown of the WFBI bitstrings that fail all NIST tests, pass more than one test and pass all NIST tests in static FI experiments.

a small number of cases. The Partial and non-ASCII classes vary between 0.6% and 1.8%, and 1.2% and 3.5%, respectively, in both bar graph plots, with a slightly larger fraction of Partial bitstrings for the delay fault experiments.

A third notable trend is reflected in the nearly identical sequence of bars corresponding to the SA0 and SA1 FI experiments, while the Correct class for the delay fault experiments increases by nearly 10% over the SA fault percentages. Moreover, the Correct class for the invert experiments exhibit more than an 8% decrease in the fraction of Correct bitstrings. From these results, it is clear that permanent invert faults have the most significant negative impact on execution behavior, followed by SA and delay.

The bar graphs in Fig. 18 present the NIST statistical test results for the WFBI bitstrings. The goal of our analysis is to determine the cryptographic strength of the WFBI bitstrings that appear to be random. Therefore, in this analysis, we exclude WFBI bitstrings that are full length but are easily recognized as being invalid, e.g., those that posses a large fraction of zeros or have short repeated patterns. The number of these invalid WFBI bitstrings is small, i.e., between 2% and 3% of the total number of WFBI bitstrings.

The results shown in Fig. 18(a) give the fraction of the WFBI bitstrings from the static experiments that pass each of the fourteen NIST statistical tests identified along the *y*-axis (the NIST test RandomExcursions required more than 1 million bits and is excluded). The total number of WFBI bitstrings varies from approximately 5700 to 7800 across the 20 FI experiments. The pass percentages vary from approximately 77% to 100%. The NIST test that failed most often is the Rank test, which tests for linear dependence among fixed length substrings in the bitstrings. This may suggest that portions of the Twister algorithm are not executing, leaving artifacts that would normally be removed by Twister's full blown algorithm. On the other hand, the RandomExcusionsVariant test, which tests for deviations in the number of expected visits to various states in a cumulative sum random walk, passed more than 99% of the time. The number of WFBI bitstrings that passed all NIST tests is surprisingly high.

The bar graphs in Fig. 18(b) provide a different perspective. The stacked bars show the percentage of WFBI bitstrings that fail all NIST tests (always less than 0.8%), pass at least 1 NIST test but not all NIST tests (varies between 50% and 58%) and pass all NIST tests (varies between 40% and 48%). Given these bitstrings are completely different from the fault-free bitstring, 3% to 4% of the 85 713 fault experiments, or between 2700 to 3,300 of the full length bitstrings generated, are deemed to be of high cryptographic strength by the NIST statistical tests. In contrast, the remaining full length bitstrings, between 3500 to 4000, fail at least one NIST statistical test.

V. Conclusion

This article investigated 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 microprocessor was created using an ASIC CAD tool flow and an instrumented design is created in which a fault-injection circuit is inserted in series with all gate inputs in the netlist.

A scan chain is used to enable emulation of four fault types at one or more fault locations. A set of FI experiments are carried out that systematically measure and characterize the RISC-V execution behavior as faults are enabled. Advanced FPGA features, including tightly coupled, on-chip, memory-mapped registers between the processor and PL, DR, and a runtime monitor, are used to accelerate the FI experiments.

The fault tests were classified according to the amount of information they leak in the AES experiments, and the cryptographic strength of the generated pseudo-random number sequence (PRNS) in the Twister experiments. The results indicated that the occurrence of key leakage and cryptographically weak PRNS, although rare, represent security holes and, therefore, need to be addressed. Follow-on research will leverage the flexibility and efficiency of the proposed FI platform to investigate other important security algorithms including those used in asymmetric and post-quantum cryptography, as well as cost-effective circuit-level alternatives to costly standard fault-tolerant techniques, such as triple modular redundancy.

Acknowledgment

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology

and 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 article describes objective technical results and analysis. Any subjective views or opinions that might be expressed in this article do not necessarily represent the views of the U.S. Department of Energy or the United States Government SAND2021-2696 J.

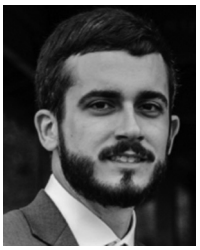
REFERENCES

[1] J. A. Clark and D. K. Pradhan, "Fault injection: A method for validating computer-system dependability," *Computer*, vol. 28, no. 6, pp. 47–56, Jun. 1995.
[2] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante, "New techniques for efficiently assessing reliability of SOCs," *Microelectron. J.*, vol. 34, no. 1, pp. 53–61, 2003.
[3] R. Nyberg, J. Heyszl, D. Rabe, and G. Sigl, "Closing the gap between speed and configurability of multi-bit fault emulation environments for security and safety–critical designs," *Microprocess. Microsyst.*, vol. 39, no. 8, pp. 1119–1129, 2015.
[4] C. Fibich, S. Tauner, P. Rössler, M. Horauer, M. Matschnig, and H. Taucher, "FIJI: Fault injection instrumenter," *EURASIP J. Embedded Syst.*, vol. 2019, no. 1, 2019, Art. no. 2.
[5] T. J. Mannos, B. Dziki, and M. Sharif, "Fault testing a synthesizable embedded processor at gate level using ultrascale FPGA emulation," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2019, p. 116.
[6] (2020). *Advanced Encryption Standard*. [Online]. Available: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
[7] (2020). *Mersenne Twister*. [Online]. Available: https://en.wikipedia.org/wiki/Mersenne_Twister
[8] S.-A. Hwang, J.-H. Hong, and C.-W. Wu, "Sequential circuit fault simulation using logic emulation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 8, pp. 724–736, Aug. 1998.
[9] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante, "An FPGA-based approach for speeding-up fault injection campaigns on safety-critical circuits," *J. Electron. Testing*, vol. 18, no. 3, pp. 261–271, 2002.
[10] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcia, and L. Entrena, "Autonomous fault emulation: A new FPGA-based acceleration system for hardness evaluation," *IEEE Trans. Nucl. Sci.*, vol. 54, no. 1, pp. 252–261, Feb. 2007.
[11] P. Vanhauwaert, R. Leveugle, and P. Roche, "A flexible SoPC-based fault injection environment," in *Proc. IEEE Design Diagnost. Electron. Circuits Syst.*, 2006, pp. 192–197.
[12] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. M. Austin, "CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework," in *Proc. IEEE Int. Conf. Comput. Design*, 2008, pp. 363–370.
[13] A. B. de Oliveira *et al.*, "Evaluating soft-core RISC-V processor in SRAM-based FPGA under radiation effects," *IEEE Trans. Nucl. Sci.*, vol. 67, no. 7, pp. 1503–1510, Jul. 2020.
[14] S. Walters, "Computer-aided prototyping for ASIC-based systems," *IEEE Design Test Comput.*, vol. 8, no. 2, pp. 4–10, Jun. 1991.
[15] R. W. Wieler, Z. Zhang, and R. D. McLeod, "Emulating static faults using a xilinx based emulator," in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.*, 1995, pp. 110–115.
[16] L. Burgun, F. Reblewski, G. Fenelon, J. Barbier, and O. Lepape, "Serial fault emulation," in *Proc. 33rd Design Autom. Conf.*, 1996, pp. 801–806.
[17] K.-T. Cheng, S.-Y. Huang, and W.-J. Dai, "Fault emulation: A new methodology for fault grading," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 10, pp. 1487–1495, Oct. 1999.
[18] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcfa, and L. Entrena-Arrontes, "Autonomous transient fault emulation on FPGAs for accelerating fault grading," in *Proc. 11th IEEE Int. On-Line Testing Symp.*, 2005, pp. 43–48.
[19] P. Vanhauwaert, R. Leveugle, and P. Roche, "Reduced instrumentation and optimized fault injection control for dependability analysis," in *Proc. IFIP Int. Conf. Very Large Scale Integr.*, 2006, pp. 391–396.
[20] L. Kafka, M. Danek, and O. Novak, "A novel emulation technique that preserves circuit structure and timing," in *Proc. Int. Symp. Syst. Chip*, 2007, pp. 1–4.
[21] A. Pellegrini *et al.*, "Crashtest'ing SWAT: Accurate, gate-level evaluation of symptom-based resiliency solutions," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, 2012, pp. 1106–1109.
[22] K. Asanović *et al.*, "The rocket chip generator," EECS Dept., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016–17, 2016.
[23] Sergeykhbr. (2020). *System-On-Chip Template Based on Synthesizable Processor Compliant WITH the Risc-V Architecture*. [Online]. Available: https://github.com/sergeykhbr/riscv_vhdl
[24] H. Cho, "Impact of microarchitectural differences of RISC-V processor cores on soft error effects," *IEEE Access*, vol. 6, pp. 41302–41313, 2018.
[25] (2020). *Petalinux Tools*. [Online]. Available: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html
[26] (2020). *Synopsys Design Compiler*. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html
[27] L. T. Clark *et al.*, "ASAP7: A 7-nm finFET predictive process design kit," *Microelectron. J.*, vol. 53, pp. 105–115, Jul. 2016.
[28] (2020). *Cadence Innovus Implementation System*. [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html
[29] (2020). *Vivado Design Suite*. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html
[30] (2020). *A Byte-Oriented AES-256 Implementation*. [Online]. Available: http://www.literatecode.com/aes256
[31] D. Shaw, D. Al-Khalili, and C. Rozon, "Fault security analysis of CMOS VLSI circuits using defect-injectable vhdl models," *Integration*, vol. 32, nos. 1–2, pp. 77–97, 2002.
[32] (2020). *Random Bit Generation*. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html

Jim Plusquellic (Member, IEEE) received the M.S. and Ph.D. degrees in computer science from the University of Pittsburgh, Pittsburgh, PA, USA, in 1995 and 1997, respectively.

He is a Professor of Electrical and Computer Engineering with the University of New Mexico, Albuquerque, NM, USA.

Prof. Plusquellic received an Outstanding Contribution Award from IEEE Computer Society in 2012 and 2017 for co-founding and for his contributions to the Symposium on Hardware-Oriented Security and Trust.



Donald E. Owen, Jr., (Member, IEEE) received the B.S. and M.S. degrees in electrical engineering from the University of Texas at Austin, Austin, TX, USA, in 2011 and 2013, respectively.

He is a Staff Member with the Digital Design and Verification Department, Sandia National Laboratories, Albuquerque, NM, USA, and a Graduate Student with the University of New Mexico, Albuquerque, NM, USA. His research interests include hardware security and its intersection with computer architecture and embedded system design.

Tom J. Mannos (Member, IEEE) received the B.S. degree in electrical engineering from the University of Utah, Salt Lake City, UT, USA, in 2002, and the master's degree in electrical engineering from the University of New Mexico, Albuquerque, NM, USA, in 2006.

He is an Integrated Circuit Designer with the Advanced CMOS Products/Design Department, Sandia National Laboratories, Albuquerque, NM, USA, conducting research in embedded FPGA security, security fault analysis, and superconducting electronics.

Brian Dziki received the B.S. degree in electrical engineering from Pennsylvania State University, Pennsylvania, PA, USA, in 1987, and the master's degree from John Hopkins University, Baltimore, MD, USA, in 1995.

He is currently working with the Department of Defense, Fort G. G. Meade, MD, USA, conducting research in microprocessor reliability, lightweight cryptography solutions, secure wireless networks, and is a member of Trusted Computing Group.