Self-Assertion-Based Countermeasures Within a RISC-V Microprocessor for Coverage of Information Leakage Faults

Idris Somoye, *Member, IEEE*, Tom J. Mannos, *Member, IEEE*, Brian Dziki, and Jim Plusquellic, *Member, IEEE*

***Abstract*—The execution behavior of a microprocessor (μP) in the presence of a fault is difficult to predict because of the complex interactions across pipeline stages and between functional units within the architecture. In prior work, we have observed that fault effects do not introduce any type of anomaly in the input-output behavior for 10s of thousands to millions of clock cycles. These characteristics increase the difficulty of evaluating μP architectures for resilience to information leakage events, i.e., scenarios where a fault causes sensitive data, such as an encryption key, to be inadvertently diverted to a primary output channel. In this article, we use an accelerated fault emulation platform implemented on a Xilinx ZCU102 board to evaluate an ASIC implementation of the Potato RISC-V μP for information leakage events as faults from several different classes are introduced. The effectiveness and latency associated with a set of self-assertion-based countermeasures (SABCs), that perform simple consistency checks on instructions and datapath values, are investigated. The countermeasures are characterized as dynamic verification (or as a continuous symptom monitor) because detection occurs during program execution. The detection and latency results of the SABCs are compared against a periodic counter-based countermeasure proposed in previous work.**

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

I. Introduction

VALIDATING and maintaining the reliability of μP has been a main stream topic among researchers for more than three decades [1]. The challenges associated with providing highly reliable μP implementations stem from multiple sources, including the difficulty of validating complex system architectures, the heightened susceptibility of devices running at high frequency to soft errors introduced by particle strikes, power supply noise, adverse environmental conditions, wear-out and its impact on critical path delays and increasing levels of process variations.

A wide variety of countermeasures (CM) have been proposed to detect faults, categorized broadly into four main classes, including redundant execution, periodic built-in self-test, dynamic verification, and anomaly detection [2]. A dynamic verification method, also known as a continuous symptom monitor, is proposed in this article that targets faults which result in the leakage of confidential information on an output channel of the μP.

Fail-secure μP architectures are defined as those that detect and prevent leakage of sensitive information when internal faults occur. Countermeasures that detect and mitigate against information leakage events can be designed under more relaxed constraints when compared to those that attempt to minimize internal data corruption, where fast detection and response times are important. This is true because information leakage events typically have large latencies between the point in time when a fault becomes active and when information leakage occurs on a primary output channel. Moreover, as we determined in previous work, only a relatively small fraction of internal faults actually lead to information leakage. The reduced constraints associated with detecting and preventing information leakage make it possible to design lighter-weight countermeasures.

In this article, a set of self-assertion-based countermeasures (SABCs) are investigated that share similarities to those proposed in [3] but are designed for, and evaluated against, information leakage faults. The goal of the SABC is to detect inconsistencies in a set of monitored signals, which occur because of the presence of a permanent fault, as instructions are executed and as data is manipulated through a pipelined μP architecture. Although all of the proposed SABC uniquely detect a nonzero fraction of the inserted faults, the range of their detection capabilities varies widely. More importantly, for the subset of faults classified as information leakage faults, only the SABC that monitor the register file are capable of detecting the vast majority of these faults.

FPGA emulation experiments are carried out in this work to determine the fault detection capabilities of the SABC. Once a fault is detected in an on-line system, some type of mitigation is typically performed. The simplest strategy adopted here is

Manuscript received 24 October 2022; revised 25 August 2023; accepted 10 December 2023. Date of publication 11 January 2024; date of current version 21 May 2024. This article was recommended by Associate Editor F. Koushanfar. *(Corresponding author: Idris Somoye.)*

Idris Somoye is with Sandia National Laboratories, Albuquerque, NM 87185 USA, and also with the Department of Electrical and Computer Engineering, The University of New Mexico, Albuquerque, NM 87131 USA (e-mail: isomoye@unm.edu).

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

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

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).

Digital Object Identifier 10.1109/TCAD.2024.3351592

to disable the microprocessor and return it to a fail-safe state. Our on-going work is investigating fail-safe monitors that shut down a microprocessor if an internal fault occurs in one of the redundant copies, such that the faulty microprocessor is not able to interfere with the continued operation of the fault-free copy. Unfortunately, permanent faults limit the options available for mitigation to redundancy-based techniques. For example, transient faults can be handled by restoring the state to an earlier recorded check-point. Such options are ineffective for permanent faults.

The specific contributions of this work include the following.

1) The evaluation of a set of self-assertion-based, continuous symptom monitors that can serve as countermeasures to information leakage faults.
2) A comparative analysis of the SABC with an off-line, periodic counter-based CM.
3) An analysis of both the fault detection capability and latency of the CM for a large set of faults, from which a set of information leakage faults are identified.
4) The identification of an architecture-independent component, namely, the logic within the branch comparator unit, that is the optimal location for a continuous symptom monitor and/or periodic, self-test-based CM.

The remainder of this article is organized as follows. Section II discusses additional related work. Section III describes the experimental design and attributes of the fault injection (FI) experiments. Section IV presents the details of the proposed SABC, while Section V presents SABC overhead, the testing process and fault classification. Section VI presents the fault coverage and latency results for the SABC and counter CM. Section VII presents our conclusions.

II. Related Work

An overview of the different strategies taken to detect faults through either continuous checkers (also called concurrent) or periodic testing is provided in [2], while an overview of different methods used in fault detection and mitigation are presented in [4]. Gizopoulos et al. [2] described four general approaches, including redundant execution, periodic built-in self-test, dynamic verification, and anomaly detection. The SABC described in this work fall into the dynamic verification category, while the counter CM is classified as a periodic built-in self-test method. The methods are uniquely applied here to the detection of leakage sensitive faults where the requirement for detecting faults with latencies on a per-instruction basis is relaxed and instead, the goal is to detect such faults before leakage occurs on the primary output channel(s) of the μP. The following summarizes the contributions of the most closely related techniques. The faults targeted by these previous techniques are not analyzed against information leakage events.

A μP dynamic implementation verification architecture (DIVA) for detecting transient and permanent faults is proposed in [3]. DIVA's checker recomputes the functional unit result using the instruction input operands and compares the results before allowing the instruction to commit. Although the checker design is simplified because it can leverage processor pipeline decisions, the checker pipeline overhead is still large, restricting its applicability to superscalar architectures. Moreover, it assumes that the register file and memory utilize ECC for error detection and correction as a mitigation against storage related faults.

A transient fault detection technique is proposed in [5], in which a program is duplicated to run concurrently as multiple threads on a μP. It leverages simultaneous multithreading to make use of μP resources that would otherwise remain idle waiting on data dependencies. The technique requires the insertion of a specialized delay buffer into the microarchitecture to enable comparisons between the execution result streams of the two threads. The authors presents simulation results that show the runtime penalty associated with executing two copies is between 10-30%.

Constantinides et al. [6] proposed a hardware-software fault detection and diagnosis technique that uses a set of special instructions to access state and control μP execution. The technique periodically suspends execution and runs a set of special tests designed to provide high-fault coverage ($\geq$ 99%) while minimizing performance (5.5%) and area (5.8%) overhead. The extended instruction set leverages the existing scan-chain intrastructure to access all microarchitectural state components while keeping hardware overhead low. The requirement to detect any type of fault increases the hardware area and performance overhead significantly.

A hardware-software high-level symptom-based fault detection technique is proposed in [7] that monitors software execution for anomalous behavior. Fault detection is performed at a high level by observing hardware traps and μP performance counters. Although the technique is able to detect 95% of the unmasked faults, the latency for detection can be high. While most are detected in less than 100 K instructions, others take longer, up to 10 million instructions.

A shadow register technique is proposed in [8], which integrates redundant registers and comparators into the existing register file as a means of detecting transient faults introduced by particle strikes. The technique as described cannot be used to detect permanent faults that occur in other components of the μP architecture.

A hardware-based permanent fault tolerance approach is proposed in [9]. While the work is performed on routers, the fault detection and mitigation as well as computing logic aspects are similar to μP. A reconfigurable hardware redundancy technique are used for handling permanent faults in important parts of the logic. A Built-in Self-Test (BIST) circuitry is integrated into input ports to detect the faults and configure the hardware redundancy circuit accordingly. Stuck-at faults are tested by performing cycle-to-cycle input versus output checks on specific registers, taking advantage of an inherit idle period to perform fault detection. Fault mitigation is achieved through the use of hardware redundancy in the form of spare registers, which are used when the BIST detects a fault.

A hardware-software approach to protecting against FI attacks is proposed in [10]. The hardware-based fault detection unit (FDU) consists of sensors for detecting clock and voltage

glitches, and concurrent error detection and shadow latches for detecting faults injected into the datapath. The authors introduce shadow pipeline registers to detect changes in the state of the pipeline registers before and after the FI attack, where one copy is updated in an early pipeline stage and the second copy is updated in subsequent stages. The technique is not evaluated against permanent and/or information leakage faults.

A JTAG-based FI and detection method is described in [11], where researchers target the fault-injectable gate-level netlist of the targeted ASIC. The FI manager (FIM) contains the logic that drives control and dedicated signals, while the Observation Domain mainly houses the logic that implements the fault detection mechanism. An 8051 is implemented with the fault detection method built-in and is shown through emulation to be effective in detecting faults manifesting as partial signature mismatches. The method's resource utilization is given as 1445 LUTs and 1459 flip-flops for the non-ASIC parts of the system. As with other prior work in this field, the technique is not evaluated against information leakage, and requires an existing JTAG framework.

Kocher et al. [12] proposed a SRAM-based FPGA SEU assessment methodology that utilizes dynamic partial reconfiguration (DPR). DPR is used to emulate SEUs through the introduction of bit errors in the configuration memory bitstream of the FPGA, and the Xilinx Essential Bits technology is used to guide the selection of fault sites. The internal configuration access port is used to read and write frames with bit flip errors using an integrated FI infrastructure. The infrastructure is designed to be self-protecting and reliable, avoiding the insertion of bit flips in the infrastructure itself and protecting against the accumulation of errors from previous FI experiments.

Other recent work on subverting information security mechanisms within superscaler microprocessors [13], [14] and medium-scale microprocessors [15] are not directly addressed by the countermeasures proposed in this work. For example, the SABC are designed to establish the consistency of data and control signals within the micro-architecture of the processor, and therefore, are not capable of detecting malicious actions taken by correctly executing code. However, the counter-based countermeasure also described in this work may have extensions to detecting malicious code execution given that it records micro-architectural behavior of code execution in advance (under attack-free conditions), and assesses current execution behavior against this model. On-going work is investigating potential applications of this technique as mitigations against side-channel and covert-channel attacks.

III. System Overview

This section describes the RISC-V architecture used in the emulation experiments as well as the characteristics of the fault campaign, FIM and fault emulation (FE) engine. Also discussed are the CAD tools used in the synthesis and implementation, the testing process and details regarding the SABC.

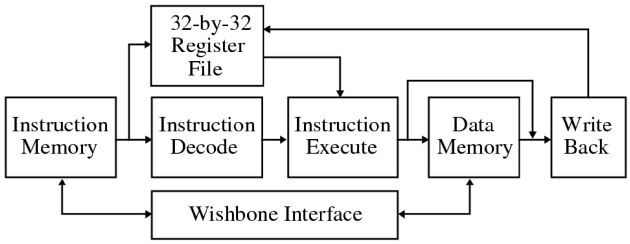

Fig. 1. Block diagram of potato's five stage pipeline [16].

A. RISC-V Architecture

The Potato μP [16] is used as the RISC architecture in this work, as an alternative to the Rocket design used in our previous work, because it is much smaller and possesses a well-structured VHDL description amenable to the insertion of the proposed SABC components. Potato is compliant with the RISC-V v2.0 standard and is implemented as a 32-bit, 5-stage pipeline (RV32I) (see Fig. 1) and possesses a complete set of integer instructions, with control and status register (CSR) and exception handling. All instructions except load and store execute in 1 cycle. The wishbone B4 standard is utilized as an internal bus.

B. Fault Campaign Characteristics

A fault campaign is a term used to describe the characteristics of the FI system [17]. Attributes of the FI include the computing and communication mechanism used by the FIM to communicate with and control the FE engine, the characteristics of the design-under-test and fault model, and the mechanism used to carry out the fault analysis. The fault campaign utilized in this research possesses the following characteristics.

1) The Xilinx UltraScale+ MPSoC FPGA on the ZCU102 development board is used as the emulation platform for the Potato μP.
2) A pair of 32-bit memory-mapped GPIO registers are used as an AXI-Lite-instantiated communication channel between the FIM, which executes under Linux on a Cortex A53 within the processing system (PS) of the FPGA, and the FE, implemented in the programmable logic (PL).
3) A set of 34 110 FI circuits are integrated into the Potato core during an ASIC-based synthesis and place-and-route (PNR) CAD tool flow. A set of three GPIO-connected scan chains are controlled by the FIM and are used to inject a fault and to read-out results for each FI experiment.
4) The FI circuits implement four fault types, including stuck-at-0 (SA0), stuck-at-1 (SA1), delay and inversion, and are configured using the GPIO-connected scan chains.
5) The FE is implemented as a set of state machines (SMs) that serve to collect serial and address bus data as Potato executes an advanced encryption standard (AES) algorithm. The SMs are configured via the FIM to constrain the number of run cycles, which, when used in

combination with a binary search routine implemented within the FIM C program, allow the latency of fault effects to be determined.

6) The fault detection capabilities of the SABC and counter CM, as well as their detection latencies, are determined off-line using the data collected from the scan chains.

Although alternative, highly flexible and modular RTL simulation techniques, e.g., those based on UVM [18], can be used as the FI platform, the requirement to analyze a large set of fault sites and fault types drove our decision to leverage MPSoC FPGAs as a means of accelerating the FI and data collection process. Moreover, the strategy used to introduce FI circuitry disabled logic optimization that would normally occur during synthesis and implementation, and allowed an assessment of the Potato microprocessor as it would be implemented using an ASIC methodology. The FI scan-based control infrastructure, when disabled, is completely transparent to the functional operation of Potato. An RTL simulation approach would have produced equivalent results but at the cost of longer execution times.

As indicated, the faults investigated in this work include SA0, SA1, delay and inversion. SA0 and SA1 are components of the stuck-at fault model, commonly used in the context of manufacturing test. Manufacturing defects can introduce permanent failure conditions where signals are not able to switch between 0 and 1, and vise versa, but instead remain fixed at 0 (SA0) or 1 (SA1). These faults are utilized in this work to represent some type of catastrophic event that occurs in a running system, e.g., a power surge or radiation strike. Similarly, permanent faults which introduce inversion of logic signals can occur in running systems by stuck-at defects on the outputs of upstream gates that drive downstream gates, for example, XOR gates, or by semi-permanent faults that flip bits in flip-flops that store configuration information. From a probabilistic point of view, delay faults represent the most significant concern. Trends in using increasingly aggressive timing constraints in combination with improving synthesis tool optimization techniques create a large number of paths that are classified as critical, increasing the likelihood of occurrence of delay faults. Normal wear-out, via hot-carrier injection (HCI) and negative-bias temperature instability (NBTI), or radiation exposure change threshold voltages over time, effectively adding small delta delays to the gates along critical paths. The cumulative increase in delay along the path over time can lead to critical paths failing to meet the setup-time requirement, especially when the device is exposed to adverse environmental conditions.

C. System Architecture

A block diagram of the system level architecture is shown in Fig. 2. The configuration of the PS and PL sides of the Zynq UltraScale+ MPSoC are shown, as well as some peripherals on the Xilinx ZCU102 development board. A Linux kernel is configured to run on the quad-core Cortex-A53 ARM μP [19]. The host computer communicates through an Ethernet connection to the Linux operating system. The FIM is implemented as a C program which communicates

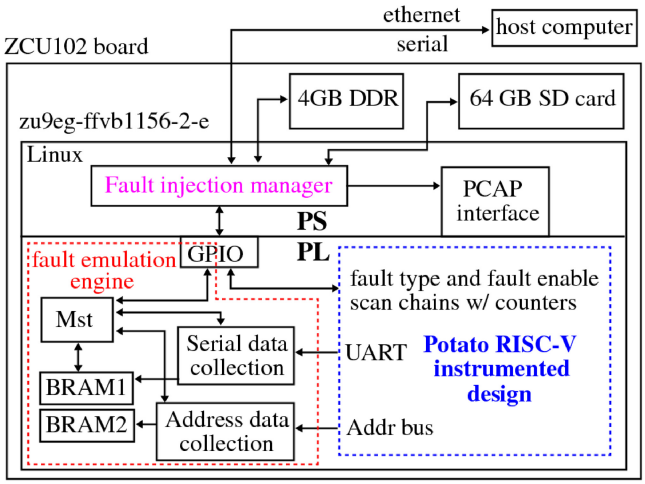

Fig. 2. Block diagram of the experimental setup with FI and counter circuit scan chains for accelerating data collection.

with the FE through GPIO registers. The FIM reprograms the PL before each FI experiment as a means of clearing the Potato processor state, in particular the block rams used to implement the caches, of any fault effects from the previous experiment. Although only a subset of the faults disrupt access patterns to the caches, and made program execution dependent on the state left by the previous fault, reprogramming clears the caches and guarantees that all FEs start from the same initial state. The processor configuration access port (PCAP) is used to minimize the time overhead associated with the reprogramming operation.

D. Fault Injection Circuit With Counter

The FI circuit with three scan chains is shown along the bottom of Fig. 3. The first scan chain, with input labeled *scan_in[0]*, is used to selectively enable one of the faults, while scan inputs [1] and [2] are used to select from one of four fault types. The scan chain consists of 34 110 elements, i.e., one instance of the FI is added to each of the gate input signals driving the logic gates within an instance of Potato's core ASIC design.

The term *FI with counter* or fault injection circuit with counter (FIC) refers to the entire circuit shown in Fig. 3, which includes both a counter and a FI circuit instance. The scan chains are extended into the counter circuit component to enable the count values to be scanned out after each FI experiment. The counters record the number of rising and falling transitions that occur on the node labeled *in* during execution of the program.

In this research, the counters serve two purposes. First, they are used to expand the list of Active (unmasked) faults, over the list derived from an analysis of the serial output and address bus behavior alone. Second, in recent work [20], we proposed counters as a periodic, built-in self-test CM for the detection of information leakage faults in the Rocket RISC-V design, showing that only a small subset of counters are needed to detect all such faults plus a large fraction of the Active faults. We carry out a similar analysis in this article and

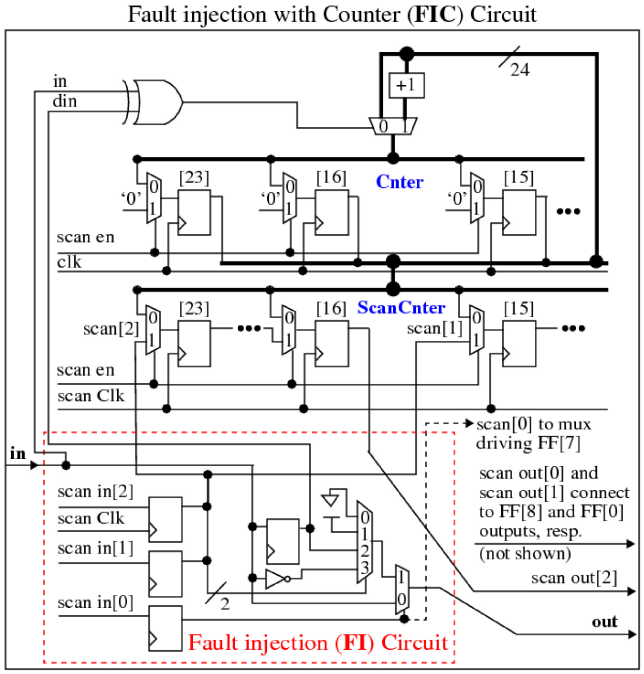

Fig. 3. Schematic of FI circuit (bottom) with counter (top). The Cnter and ScanCnter FFs[14:0] are not shown. The dotted line labeled scan[0] routes to the mux driving FF[7] of the ScanCnter, similar to the connection made by scan[1] to the mux driving FF[15]. The scan_out[1] and scan_out[0] signals connect to the outputs of FF[8] and FF[0] of the ScanCnter, respectively, similar to the signal labeled scan_out[2] on the output of FF[16].

compare the detection capability and latency of the counter CM with the results obtained for the SABC.

E. Potato Synthesis

The synopsys design compiler [21] is used to produce a gate-level netlist of Potato from a behavioral HDL description using an ASIC standard cell library [22], with integrated design-for-testability (DFT) scan chain. Cadence Encounter [23] is then used to perform PNR. A custom C program adds a set of three additional scan chains and FIC circuits shown in Fig. 3 into the netlist extracted from the layout to create a set of instrumented designs. Each design contains a set of 4000 FIC circuits, while the remaining 30 110 nodes are instrumented with only the FI circuit (no counter). This counter partitioning scheme is necessary because the size of the implemented design is too large to fit into the FPGA with all 34 110 nodes instrumented with the FIC circuit. A set of nine separate designs are created with the FIC circuit inserted on distinct subsets of nodes to enable counter values to be obtained for all nodes. This required the FE experiments to be repeated for each of the nine designs.

The instrumented designs are used as input to the Xilinx Vivado CAD tool to generate the bitstreams [24]. The structural constraints imposed by the FI circuits prevent Vivado from carrying out optimizations on the standard cell netlist, making the reported results relevant to a structurally equivalent ASIC implementation.

IV. Proposed Self-Assertion-Based Countermeasures

The SABC detect permanent faults by adding redundancy and assertions to specific computations carried out in the pipelined architecture of Potato. The signals monitored in Potato's pipeline are annotated with circled numbers in Fig. 4. In most cases, the labeled signals are routed to a countermeasure module which checks that their values are consistent with redundant calculations and/or state information. Each SABC has a corresponding error flag that is set when the assertion fails.

We note that certain types of permanent faults can disable a SABC, in particular, those that prevent its error flag signal from being asserted, and therefore, additional redundancy-based techniques are required to prevent information leakage for these cases. Assuming the probability of a permanent fault on a SABC error signal or within the counter is proportional to their area, the likelihood would only be a small fraction, less than 0.023, of a permanent fault occurring within Potato itself (this fraction is derived from the area overhead analysis in Section V-A and assuming error signal wires have negligible overhead.) Alternatively, it is possible to provide protection against their occurrence, using, e.g., a duplication-with-comparison technique (DWC). DWC is commonly used to enhance the reliability of critical system signals, while minimizing additional resources [25]. Given the redundant components associated with SABC are already present, only the error signals themselves would need to be duplicated. As we discuss in the following, this would involve duplicating a set of four signals, one for each of the four SABC. Similarly, the counter and corresponding error signal can be replicated with only a minor impact on area overhead. Doing so, would ensure that an instance of a permanent fault on a countermeasure would not disable it.

An overview of the four SABC is given as follows, with details provided in the following sections.

1) *Reconstitute Full Instruction (RFI):* The RFI SABC component is designed to check the uniformity of instruction-related control signals before and after decoding as a means of determining if the instruction fetched from memory is the same as the instruction that executed.
2) *Derive Intent of Operation (DIO):* The DIO SABC is similar but validates μP state information by comparing the expected state with the state defined by the opcode of the currently executing instruction.
3) *Cyclic-Redundancy-Check (CRC):* The CRC SABC is used to validate data against faults that corrupt memory-bound data, despite whether or not the instruction executed properly.
4) *Shadow Register Checker (SRC):* The SRC SABC is designed to detect faults that occur within the register file, utilizing a light-weight, CRC-based shadow copy of original register file contents.

The register file is central to μP operations, and therefore fault effects have a high likelihood of propagating to its inputs at some point during the execution of the program.

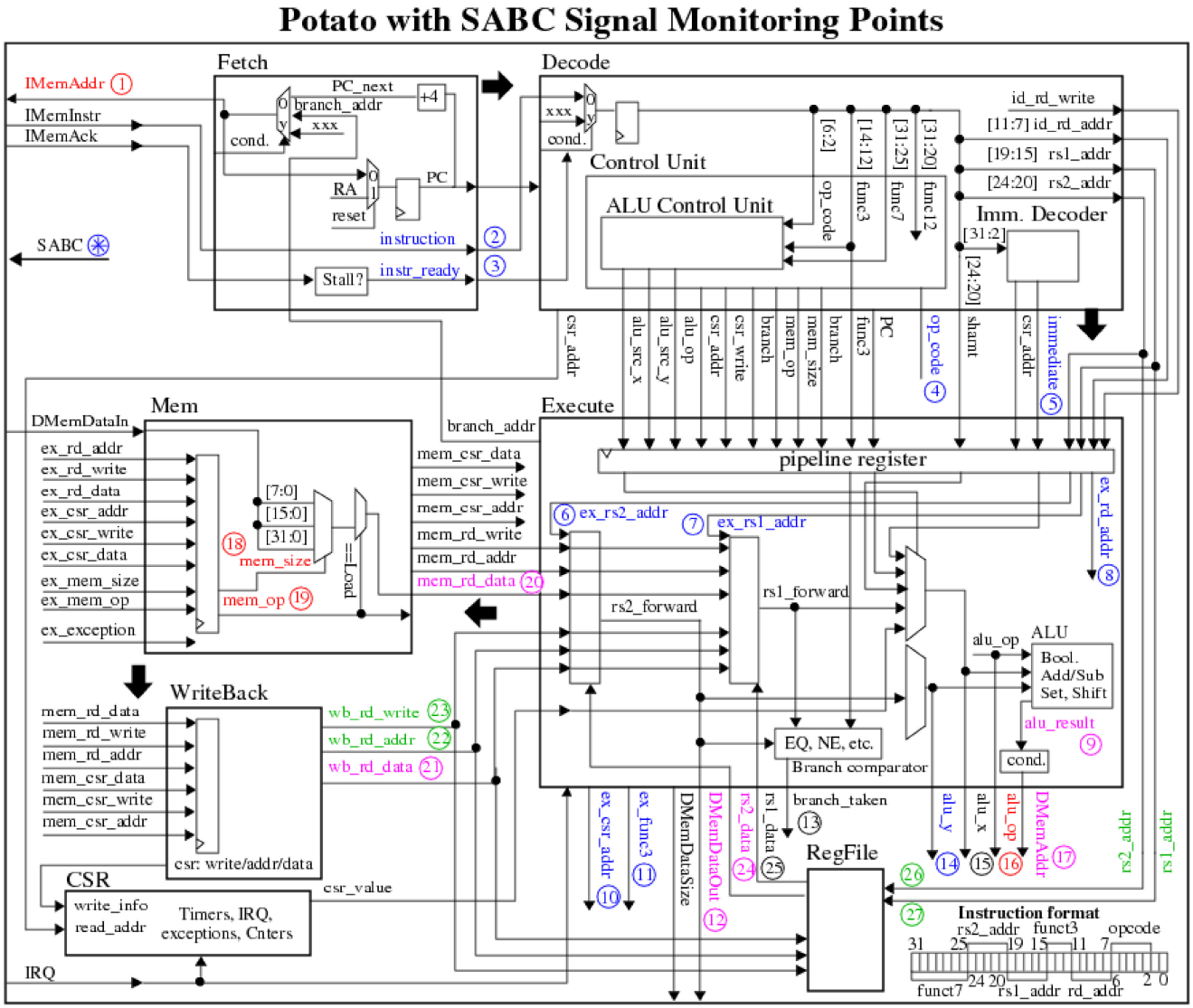

Fig. 4. Potato block diagram with numbered annotations indicating the signals monitored by the SABC. Each of the four SABC utilize distinct sources of information from the control and data path elements within Potato. For example, the RFI SABC creates assertions using data obtained from the wires labeled 2 and 3 in the fetch stage, from 4 and 5 in the decode stage and 6, 7, 8, 10, 11, and 14 in the execute stage. Color coding is used to show the signal components monitored in each SABC, e.g., blue is used for RFI SABC. Several of the labeled signals are used in more than one SABC, in which case the color from the first SABC described in Section IV is used.

An alternative XOR-based signature technique, referred to as the XOR Check (XRC) SABC, is also investigated, and the tradeoffs of the XRC and SRC SABC discussed.

A. Reconstitute Full Instruction SABC

The RFI SABC rebuilds the instruction using decoded signals from Potato's pipeline and compares the decoded instruction with the instruction fetched from memory. The instruction fetched from memory is conditionally obtained from the *instruction* (2) and *instr_ready* (3) signals in Fig. 4. All signals associated with the RFI SABC are color-coded in blue in the figure. The assertion is constructed by comparing a locally stored version of the instruction (called the left-hand-side or LHS of the assertion) with a version that is constructed from multiple datapath signals (called the right-hand-side or RHS of the assertion).

For example, the op_code signals (4) represent one of 43 instruction types defined by a decoder added to the Control Unit of the Decode pipeline stage (only the most commonly used instructions are checked by the RFI SABC). The remaining fields of the instruction, as shown within the *Instruction format* legend along the bottom right in Fig. 4, are retrieved from the signals labeled immediate (5), ex_rs2_addr (6), ex_rs1_addr (7), ex_rd_addr (8), ex_csr_addr (10), ex_func3 (11) and alu_y (14). Note that the prefix "ex" refers to the values obtained from the Execute pipeline register. Assertions constructed using signals from latter pipeline stages have the potential to detect larger numbers of faults given that fault effects propagate forward from earlier pipeline stages. A multiplexer in the reconstitution module assembles the RHS of the assertion using the op_code signals.

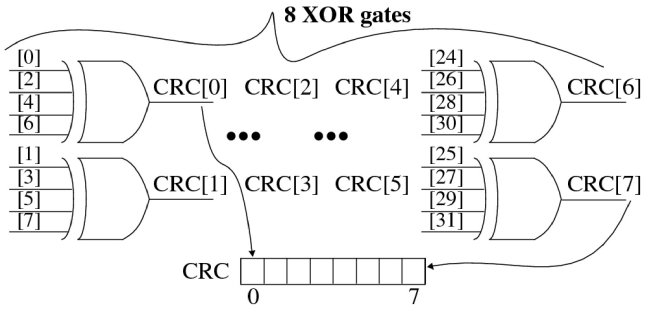

Fig. 5. CRC circuit used in the CRC SABC.

B. Derive Intent of Operation SABC

The DIO SABC targets faults that corrupt the operational state of several control signals that were discovered from simulation experiments to be sensitive to faults missed by the RFI SABC. The LHS of the assertion is obtained from signals labeled IMemAddr (1), alu_op (16), mem_size (18) and mem_op (19) in Fig. 4, while the RHS of the assertion is constructed using the op_code and the multiplexer defined for the RFI SABC in the countermeasure module (most of the DIO SABC signals are color-coded as red in the figure). The assignments to the fields in the RHS of the assertion are made using constants that define the expected operational state.

C. Cyclic-Redundancy-Check SABC

The CRC SABC targets faults that corrupt data and address bus values by comparing datapath signals that originate from distinct physical locations. A light-weight version of a CRC circuit is constructed assuming a single fault model, where the probability of fault masking is reduced. Experimental trials determined that the version shown in Fig. 5 is adequate to detect nearly all datapath faults. Note that a CRC needs to be computed for both the LHS and RHS of the assertion. Therefore, a 32-bit bus requires two copies of the CRC circuit for a total of 16 4-input XOR gates.

1) DataPath-Based CRC SABC: As indicated, datapath signals within Potato are checked at the source and destination when possible to maximize the sensitivity of the CRC SABC. For example, from Fig. 4, CRCs are computed and compared for databus signal pairs labeled rs2_data (24) and DMemDataOut (12), where the latter signal routes out to memory. Similarly, datapath pipeline signals mem_rd_data (20) and wb_rd_data (21) are also CRC-checked, as are the address bus signals labeled DMemAddr (17) and alu_result (9), which detects errors in the addresses computed for load instructions. These signals are color-coded as magenta in the figure.

2) ALU-Based CRC SABC: A set of ten additional CRC circuit instances are used to validate ALU operations using the op_code (4) and alu_result (9) signals. In these cases, a redundant, local instance of the functional unit is used to compute the result for add/sub, shift, jump and boolean instructions. The operands for the redundant functions are obtained from the alu_x (14) and alu_y (15), from the immediate (5) and from the register file (24 and 25) databus signals. Similarly, the branch_taken (13) is CRC-checked against locally computed branch conditions generated using the rs1_data and rs2_data databus signals. Three of these signals are color-coded as black in the figure.

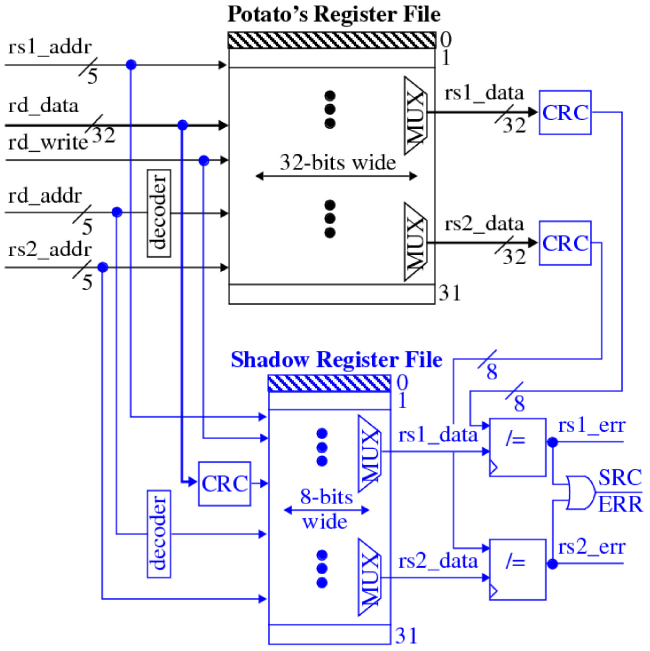

Fig. 6. Potato's register file (top) and shadow register file (SRC) SABC, shown in blue along bottom.

D. Shadow Register Checker SABC

The SRC SABC monitors the register file for faults in the decoders, MUXs and registers of Potato's register file by using a second, shadow register file, as shown in Fig. 6. In order to keep overhead low, the shadow register file stores only the CRC values of Potato's register file values, which reduces its width from 32-bits to 8-bits. The signals in Fig. 4 labeled (21) through (27) are used as inputs to the shadow register file. Four of these signal are color-coded as green in the figure.

E. XOR-Check SABC

The XRC SABC also monitors the register file for faults but does so without creating a shadow copy. Instead, modifications are made directly to Potato's register file as shown in blue in Fig. 7. The fault detection mechanism requires all the register file bits within a column to be XOR'ed to create a 1-bit XOR signature for the column. XOR signatures are created for each of the 32 columns in this fashion to define the LHS of the assertion. The 32-bit bus labeled XOR_cols represents the 32-bit XOR signature for the entire register file.

An independent XOR signature (the RHS of the assertion) is created by the components shown along the bottom of Fig. 7. Updates to the XOR_reg occur when instructions write to the register file. The incoming rd_data register file value is XOR'ed with the existing XOR_reg, which effectively adds in the new value modulo 2. The old value of the destination register is simultaneously subtracted out modulo 2 from the signature stored in the XOR_reg. The comparator continuously monitors

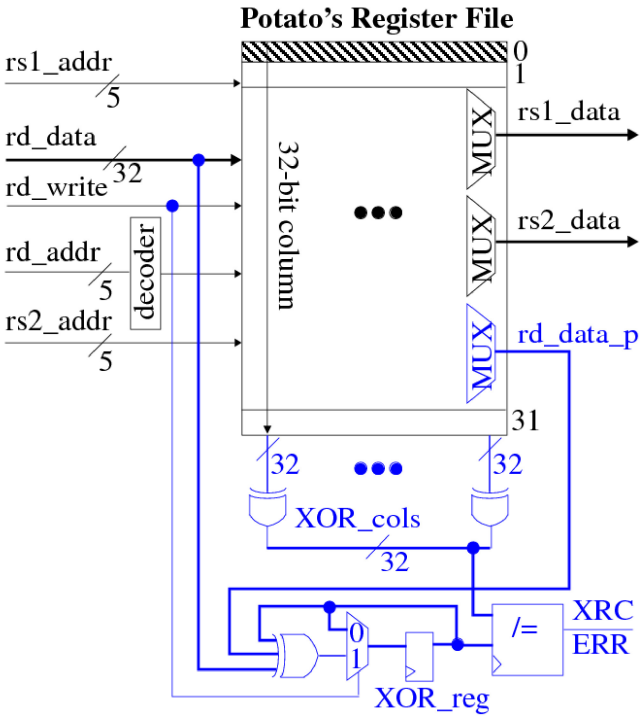

Fig. 7. Potato's register file outfitted with XOR-check (XRC) SABC. Elements added are shown in blue.

agreement between the XOR_cols and XOR_reg values and flags an error if they ever become different.

The XRC SABC is able to detect both permanent faults and any type of transient fault that upsets a register file bit, in contrast to the SRC SABC. Another difference between the SRC and XRC SABC is that the output MUXs for rs1_data and rs2_data are not monitored in the XRC SABC, which results in the latter SABC missing some severe faults. The fault detection capabilities of all SABC are presented in the following sections along with analysis of the overhead.

V. Overhead, Test Process, and Fault Classification

This section provides the performance and area overhead analysis of the SABC, as well as details regarding the testing, fault classification and data collection processes associated with the FE experiments.

A. SABC Area and Performance Overhead

The proposed SABC monitor activity on nodes within the μP architecture, and therefore, impact performance by only adding small capacitive loads to internal nodes. To determine the actual impact, we created two designs in Vivado, one using the original behavioral description of Potato and a second which includes the RFI, DIO, CRC and SRC SABC, and one copy of the counter-based CM. As we show in the following sections, the XRC SABC is excluded because it represents a less effective alternative to the SRC, and only one copy of the counter is required to detect all severe faults. The maximum operating frequency of the original design is 175 MHz, while the design with the SABC and counter is 165 MHz. Therefore, the impact on performance is $(175 - 165)/175 = 5.7\%$. A similar fractional change in performance is obtained using the ASIC synthesis tools.

TABLE I
Area Overhead Associated With the SABC

SABC	# Logic	# FF	Area (μm^2)	Area Fraction (%)
RDC	3423	192	5558	16.3
SRC	1581	225	3941	12.1
XRC	2071	33	3749	11.6
Cnter	354	48	682	2.3
Potato core	17,633	2283	28,510	100

For the area overhead analysis, behavioral descriptions for each of the SABC are used as input to the Synopsys synthesis tool to determine the gate count and area overhead associated with the layouts implemented using the ASAP standard cell library [22]. The results are given in Table I.

The fractional area overhead of the RDC includes the area of the RFI, DIO and CRC SABC, but excludes the area of the SRC and XRC SABC. Given the goals of the SRC and XRC overlap, only one of these SABC would be included in a RISC-V leakage-fault-protected design. As we show in the following, the SRC SABC is a more effective countermeasure, and can be used by itself to provide high levels of fault coverage.

B. Testing Process

A full execution of the AES algorithm takes 6 717 440 clock cycles to complete, which includes the clock cycles required to execute code for reading the key and plaintext before the encryption operation is started and the clock cycles needed to transfer the ciphertext through the serial port after the encryption operation completes.

Two sets of experiments are conducted. The first set, referred to as fault coverage experiments, is used for classifying faults as *Active* (unmasked) or *Benign* (masked), and for determining the detection capabilities of the SABC and counter CM. The analysis of the data from these experiments is also used to identify a subset of the Active faults that leak AES plaintext and/or key information on the serial port (hereafter referred to as *Severe* faults). The second set, referred to as the latency experiments, is used to determine the minimum number of clock cycles required for at least one CM to detect the fault.

The fault coverage experiments involve first carrying out a fault-free execution run to obtain the baseline count values. A sequence of FI experiments is then performed, each with a fault enabled at one of the 34 110 fault locations, and the counter values retrieved. The FI experiments are repeated for each of the fault locations and for each of the four fault types. The entire set of fault-free and FI experiments is then repeated nine times, once for each of the instrumented designs. The SABC fault assertion signals are also read out in the experiments which use the first instrumented bitstream (the results for the SABC are the same for all bitstreams).

The latency experiments are similar with two exceptions. First, only a subset of the 34 110 faults are processed, in particular, only those classified as Active or Severe from the fault coverage experiments. Second, for each Active or Severe

fault, the FI experiment is repeated using a binary search process to determine an upper and lower bound on the number of clock cycles required to detect the fault. The binary search process is terminated when a multiple of 1024 clock cycles is found in which the fault is not detected at the lower bound and is detected at the upper bound. Given the search is carried out over the region between 1024 and 6 717 440 clock cycles, in steps of 1024, the number of iterations required for each fault is 13.

Unlike the SABC, the binary search process for the counter CM requires fault-free count information to be available during the binary search process. This data is collected a priori in a clk-sweep experiment in which Potato is run under fault-free conditions through a sequence of experiments in which the number of clock cycles is set to an incrementally increasing multiple of 1024 clock cycles. The number of repetitions of the fault-free experiments per bitstream is given by $6\,717\,440 \div 1024 + 1 = 6561$, which includes an initial run of one clock cycle. Once this data is collected for all nine bitstreams, an appropriate subset of the full array of 223 795 710 fault-free counter values is utilized in the binary search process for each of the faults.

Each FI experiment involves the following sequence of operations, in reference to Fig. 3.

1) The PL is reprogrammed with one of the instrumented bitstreams.
2) The FIM transfers a parameter to the FE that specifies the number of clock cycles.
3) The FIM inserts a fault by driving the scan chain input signals.
4) The FIM asserts the *scan en* signal and pulses the system *clk* for one cycle to clear the counters.
5) The FIM deasserts *scan en* and starts the FE engine.
6) The FE engine signals the FIM after the test has completed.
7) The FIM scans out the contents of the scan chains to obtain the counter values.

The experiments were carried out on three copies of the ZCU102, and required approximately 1 month of runtime to collect the SABC and counter CM data for all of the experiments described herein.

C. Active and Benign Faults

The data collected from the fault coverage experiments is used to classify each of the faults into one of three fault classes.

1) *BenignFaults (Masked) Class:* Faults that corrupt only one or a small number of nodes within Potato's core, and do not affect AES program output.
2) *ActiveFaults Class:* Faults that introduce significant levels of internal corruption within Potato's core, and which may or may not affect output.
3) *SevereFaults Class:* Active faults that corrupt output and leak sensitive information.

In each FI experiment, we collect serial output data, the sequence of instruction address values associated with the last 50 changes to the address bus during the execution run, the counter values and the state of the SABC fault assertion signals. The faults are first classified according to whether they corrupt the serial output and/or change the address bus behavior. The analysis yields the following classification results.

Stuck-at-0: Active: 11487 Benign: 22623
Stuck-at-1: Active: 11678 Benign: 22432
Delay: Active: 9045 Benign: 25065
Invert: Active: 14443 Benign: 19667.

The faults from the ActiveFaults class are then inspected to determine if leakage of the plaintext or key occurs in the serial port output, and are added to the SevereFaults class if this occurs. The cardinalities of faults in the SevereFaults class are given as follows.

Severe Stuck-at-0: 65
Severe Stuck-at-1: 30
Severe Delay: 61
Severe Invert: 79.

The last component of our analysis further refines the ActiveFaults and BenignFaults class using the counter values. The counters enable a higher level of internal visibility of fault effects over what is available when analyzing the serial output and address bus behavior alone. The following conditions are checked to determine if a fault classified as Benign is to be reclassified as Active. Both conditions must be true for the fault to be reclassified.

1) A counter value in an FI experiment differs from the fault-free value by more than 1.
2) The number of such counters in the FI experiment exceeds a user-defined threshold of 100.

We consider faults that meet or exceed these conditions as introducing significant internal corruption to the state of Potato's core despite the absence of serial output corruption or changes to the instruction execution sequence. Using larger or smaller thresholds has only a minor impact on the number of faults that are reclassified except in cases where the threshold is set to a small value, e.g., less than 5. The final cardinality of each fault class is given as follows.

Stuck-at-0: Active: 13415 Benign: 20695
Stuck-at-1: Active: 14883 Benign: 19227
Delay: Active: 10105 Benign: 24005
Invert: Active: 17579 Benign: 16531.

VI. Experimental Results

The goal of our analysis is to select the minimum number of SABC needed to detect all faults in the SevereFaults class while maximizing the collateral coverage of faults in the ActiveFaults class, and to accomplish both of these goals at the smallest possible latencies. The SABC are analyzed in isolation and in groups as needed in the following sections in support of these objectives.

An analysis of fault coverage and latency is also carried out using a small set of counters, in particular, those that provide the highest-fault coverage. In previous work, we investigated the effectiveness of the counter CM and found that only a small

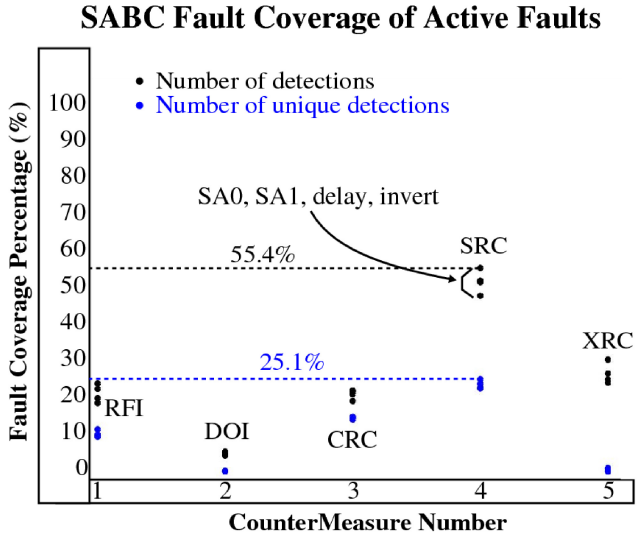

Fig. 8. SABC detection results expressed as a percentage of all faults in the ActiveFaults class.

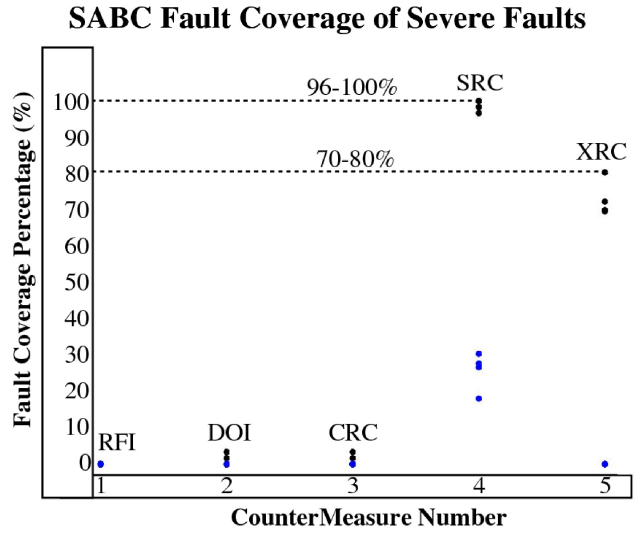

Fig. 9. SABC detection results expressed as a percentage of all faults in the SevereFaults class.

subset of 5 or fewer counters are needed to meet the goals stated above [20]. A similar result is attained here for Potato using only a single counter, referred to as the *Top Cnter*. For completeness, we also show the results attained using a set of the top three most effective counters but the improvement is marginal. The fault coverage results for the counter CM are presented concurrently with the latency results.

A. SABC Fault Coverage Analysis

The primary goal of the SABC fault coverage experiments is to determine the fault detection capabilities of the SABC and whether each SABC detects a unique subset of the faults not detected by other SABC. The analysis is carried out on the faults in the ActiveFaults class and SevereFaults class separately.

Figs. 8 and 9 plot the detection results for the SABC on faults in the ActiveFaults and SevereFaults classes, respectively, as percentages of the total number of faults for each fault type. The results for each SABC are given along the *x*-axis, labeled with the three-letter acronym defined in Section IV. The black points represent the fraction of the ActiveFaults and SevereFaults detected by each SABC while the blue points represent the fraction *uniquely* detected by each SABC. The points for each of the fault types are similar in value, indicating that the detection capability of the SABC is relatively insensitive to the fault type.

The SRC SABC provides the highest-fault coverage and highest-unique fault coverage for faults in the ActiveFaults class, with values of 55.4% and 25.1%, respectively. The fault coverages for the RFI, DIO, CRC and XRC SABC vary from 5% to more than 30%. The overall coverages for all faults of each fault type vary from 82% and 89% for the ActiveFaults class.

The results for the SevereFaults class show larger differences in coverage than those shown for the ActiveFaults class, with the SRC and XRC SABC providing the highest-overall coverages. The overall coverage using only the DIO, CRC and SRC SABC for each fault type is 100% for the SevereFaults class. The SRC SABC by itself detects all but two of the severe faults and represents the best SABC for detecting information leakage faults, as well as faults from the ActiveFaults class.

B. Latency Analysis

The objective of the latency analysis is to determine whether the SABC and a set of "top" counters are able to detect the presence of a fault before leakage occurs on the serial port. This goal is addressed for the SABC by evaluating latency using all SABC simultaneously, and then using the best-individual SABC. The term RDCS is used to represent the "all" group which includes the RFI, DIO, CRC and SRC SABC. The latency of the XRC SABC is also presented for completeness, although it is likely that only one of the SRC or XRC SABC would be used in an actual application.

For the counter CM, an analysis is carried out using a small set of top counters and then separately using the top counter. The counters are selected based on their fault coverage of Active and Severe faults. In particular, three counters are identified from the group of 34 110 counters that are able to detect all Severe faults while providing the highest-fault coverage of the Active faults. We refer to the counter analyses as *Top Cnter* and *Top 3 Cnters* in the following.

The latency results are presented as cumulative fault detection graphs, where the number of clock cycles that Potato is run for is plotted along the *x*-axis and the cumulative number of faults detected is plotted along the *y*-axis. It follows that curves which remain above and to the left of other curves in these plots represents a better result, implying the countermeasure detects a larger number of faults and does so in fewer clock cycles.

The latency results are presented for delay faults only and for all four fault types combined. As discussed further below, the results show that the countermeasures are insensitive to the fault type, and therefore, these subsets sufficiently represent our findings while minimizing redundancy. Figs. 10 and 11 show the latency results for delay faults from the ActiveFaults

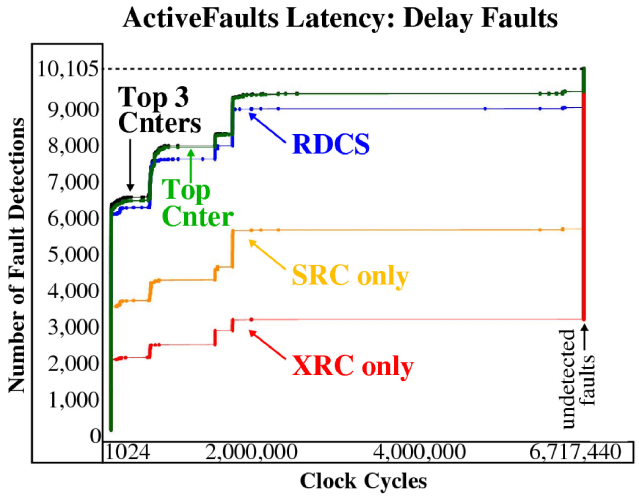

Fig. 10. Delay fault latency analysis of Active faults for the SABC and counter CM. The RDCS curve includes all SABC except for XRC.

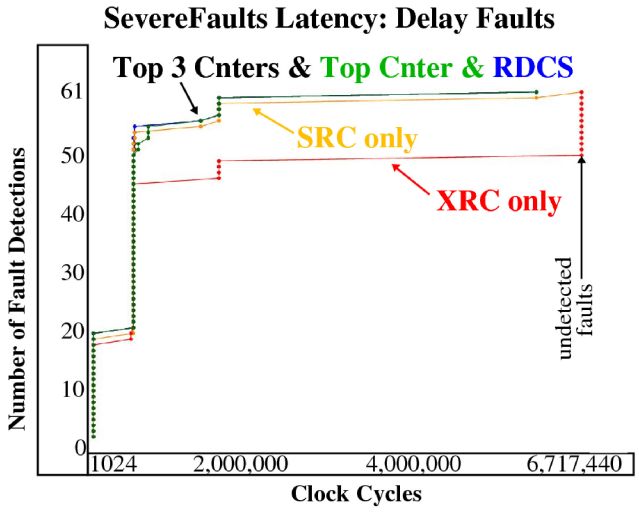

Fig. 11. Delay fault latency analysis of severe faults for the SABC and counter CM.

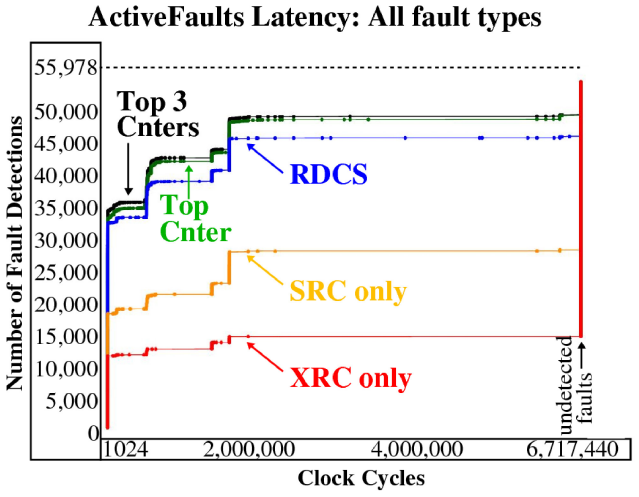

Fig. 12. SABC and counter CM latency analysis of all fault types in the ActiveFaults class.

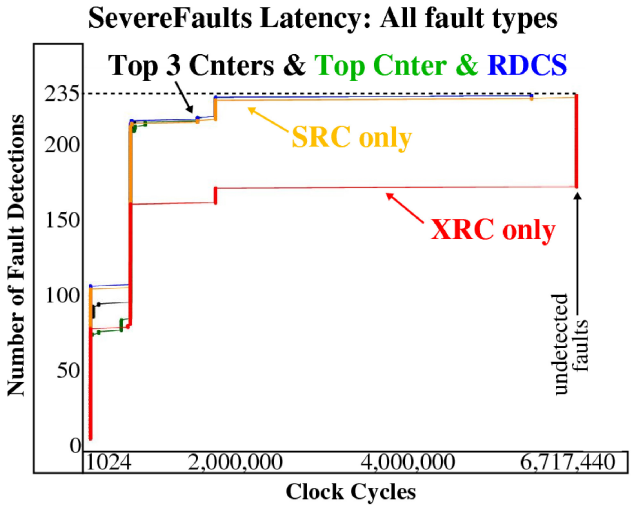

Fig. 13. Companion graph to Fig. 12 for the SevereFaults class.

and SevereFaults classes, with cardinalities of 10 105 and 61, respectively. Curves that extend out to the maximum number of clock cycles (6 717 440) indicate that some fraction of the faults from that class are not detected.

The latency curves for the SABC and counter CM are superimposed to illustrate which countermeasures provides the best results. As indicated earlier, the curves labeled RDCS include the RFI, DOI, CRC, and SRC SABC. For the ActiveFaults class shown in Fig. 10, the *Top 3 Cnters* CM, the *Top Cnter* CM and the RDCS curves dominate the "SRC only" and "XRC only" curves. Although the counter CM curves provide slightly better results both in terms of larger numbers of fault detections and smaller latencies, the RDCS curve is a close second. Moreover, the RDCS SABC is a continuous symptom monitor designed to be independent of the executed program. The curves for the SRC only and XRC only CMs are clearly less effective, failing to detect large fractions of the faults.

The results shown in Fig. 11 for the SevereFaults class portray a different result with regard to the SRC and XRC SABC. Here, the SRC SABC shows only a slight degradation with regard to the number of faults detected (all but two as indicated earlier) and provides nearly identical latencies when compared to the RDCS and counter CM curves. The latter result is expected given the results for fault coverage presented earlier, but the near-equivalence to the counter CM curves suggests that the SRC SABC is detecting errors on wires also monitored by the top counters.

Figs. 12 and 13 show the latency results for the ActiveFaults and SevereFault classes, respectively, when all four fault types are combined. Notwithstanding the larger number of faults considered, the shapes of the curves in both graphs are nearly identical to those shown for the delay faults only, suggesting that both countermeasures are insensitive to the fault type.

1) Serial Port Latency Analysis: The RDCS and *Top Cnter* CM are able to detect all of the Severe faults before leakage occurs on the serial port. As an illustration, the curves in Fig. 14 plot the serial port latencies in black and the RDCS latencies in blue for each of the faults in the SevereFaults classes plotted along the x-axis. The curves in Fig. 15 show

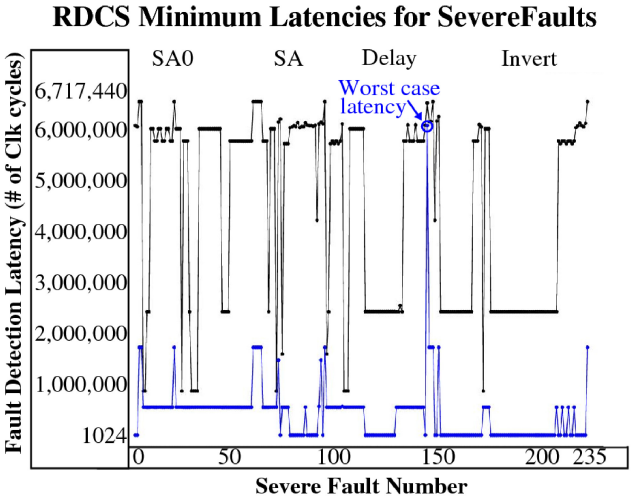

Fig. 14. SABC and actual serial port latencies for the severe faults.

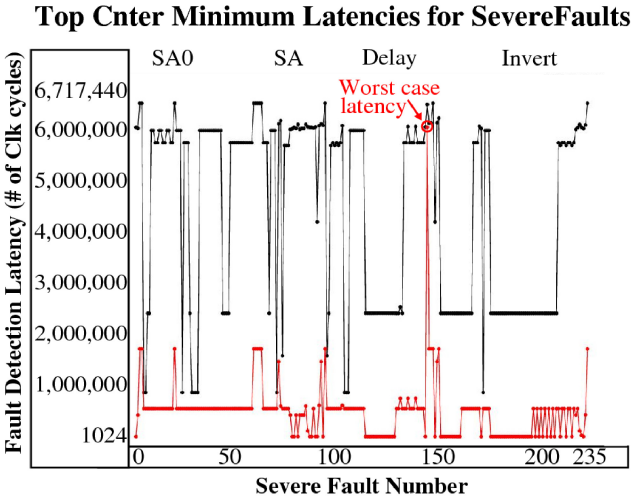

Fig. 15. *Top Cnter* and actual serial port latencies for the severe faults.

the results in the same format for the *Top Cnter*. In nearly all cases in both graphs, the black points are significantly larger than the corresponding countermeasure points, indicating the fault is detected well in advance of any leakage. Only in one case is the fault detection latency greater than 1.8 million clock cycles (highlighted in both graphs). This worst-case detection latency occurs at nearly 6 million clock cycles for both the RDCS and the *Top Cnter* CM.

C. Architectural Analysis

In this last section, we investigate the architectural significance regarding the locations of the counters identified earlier as the *Top 3 Cnters*. Although these locations were identified in previous work as significant [26], in this work, we analyze this region further to determine the mapping of the nodes from the netlist to specific behavioral VHDL statements in the Potato design. The nodes monitored by these counters are located within the *Branch comparator* submodule of the *Execute* pipeline stage shown in Fig. 4. The *Branch comparator* module takes three inputs; a 3-bit *funct3* field from the instruction and two 32-bit databus signals labeled *rs1_forward* and *rs2_forward*, and produces a 1-bit result called *branch_condition* internally (not shown). The branch_condition signal is then used to define the *branch_taken* output of the Execute stage (labeled 13 in the figure) but is used only if the instruction decoded is a branch instruction.

The internal *branch_condition* signal is driven upstream by a node connected to the *Top Cnter* CM. In particular, the *Top Cnter* CM monitors a node within the reduction logic network constructed by the synthesis tool to implement the comparison operators within the *Branch comparator* module. The first couple lines of the behavioral VHDL code for the *Branch comparator* module are given as follows.

```
case funct3 is
   when b"000" => -- EQ
      result <= to_std_logic(rs1 = rs2);
   when b"001" => -- NE
      result <= to_std_logic(rs1 /= rs2);
   when b"100" => -- LT
      result <= to_std_logic(signed(rs1) <
         signed(rs2));
...
```

The reduction logic operations that implement equality, inequality, etc, using the *rs1_forward* and *rs2_forward* databus signals as input, are always performed, independent of the instruction's opcode. Therefore any fault that propagates along any bit of these databus signals will, with high probability, manifest as a change within the reduction logic networks of this module. The second and third of the *Top 3 Cnters* CM also monitor nodes within the reduction logic network of the *branch_condition* signal. Moreover, several of the CRC-based SABC described earlier utilize signals derived from the *rs1_forward* and *rs2_forward* databus signals as input, which explains the close correspondence between the counter CM and the SABC results. Future work will investigate whether it is possible to design alternative SABC that perform assertions within this functional unit of the RISC-V architecture.

The proposed SABC can be applied to other more complex microprocessor architectures, including super-scalar architectures. However, their integration will require modifications and additional resources to deal with the more complex behavior of the pipeline, in particular, as a means of synchronizing the assertions with instructions executed out-of-order, and with handling complex branch prediction and execution.

VII. Conclusion

This article investigates a continuous symptom monitor CM called SABC and a periodic, built-in self-test counter-based CM for the detection of permanent faults in the Potato microprocessor. The SABC and counter CM are evaluated using an accelerated FPGA-based emulation platform, which emulates the execution of the AES algorithm on Potato. A behavioral HDL description of the microprocessor is instrumented with FI circuits, SABC and counter CM, and is synthesized using a standard cell ASIC CAD tool flow. A C program running on

a hardwired Cortex A53 μP within the FPGA inserts faults, controls execution runtimes and reads-out SABC and counter CM results. Four fault types are emulated, one-at-a-time, on each of Potato's 34 110 core circuit nodes. Fault coverage and latency experiments are carried out, and the effectiveness of the SABC and counter CM are reported and compared with the counter CM.

The SABC and counter CM are evaluated with respect to the number of Severe faults they can detect, the latency of the detections and the level of collateral coverage of the Active faults. The results show that the SABC are nearly as effective as the counter CM for Active faults and produce nearly identical results for the Severe faults. Moreover, all Severe faults are detected by the SABC indicating that the proposed countermeasures achieve the goal of preventing information leakage during program execution. An architectural inspection of the nodes monitored by the top counters reveals that nodes within the Branch comparator are highly sensitive to fault effects and therefore represent the best locations for the insertion of continuous monitors or periodic, built-in self-test countermeasures.

ACKNOWLEDGMENT

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 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.

REFERENCES

[1] S. Mittal and J. S. Vetter, "A survey of techniques for modeling and improving reliability of computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 4, pp. 1226–1238, Apr. 2016.

[2] D. Gizopoulos et al., "Architectures for online error detection and recovery in multicore processors," in *Proc. Des., Autom. Test Europe*, 2011, pp. 1–6.

[3] T. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Proc. 32nd Annu. ACM/IEEE Int. Symp. Microarchitecture (MICRO)*, 1999, pp. 196–207.

[4] P. Maistri, "Countermeasures against fault attacks: The good, the bad, and the ugly," in *Proc. IEEE 17th Int. On-Line Test. Symp.*, 2011, pp. 134–137.

[5] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," in *29th Annu. Int. Symp. Fault-Tolerant Comput. Tech. Dig. (Cat. No.99CB36352)*, 1999, pp. 84–91.

[6] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "Software-based online detection of hardware defects mechanisms, architectural support, and evaluation," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2007, pp. 97–108.

[7] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," *ACM Sigplan Not.*, vol. 43, no. 3, pp. 265–276, 2008.

[8] M. Kayaalp, F. Koc, and O. Ergin, "Improving the soft error resilience of the register files using SRAM bitcells with built-in comparators," in *Proc. 15th Euromicro Conf. Digit. Syst. Design*, 2012, pp. 140–143.

[9] H. J. Mohammed, W. N. Flayyih, and F. Z. Rokhani, "Tolerating permanent faults in the input port of the network on chip router," *J. Low Power Electron. Appl.*, vol. 9, no. 1, p. 11, 2019. [Online]. Available: https://www.mdpi.com/2079-9268/9/1/11,

[10] B. Yuce, N. F. Ghalaty, C. Deshpande, C. Patrick, L. Nazhandali, and P. Schaumont, "FAME: Fault-attack aware microprocessor extensions for hardware fault detection and software fault response," in *Proc. Hardw. Architect. Support Secur. Privacy*, 2016, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/2948618.2948626

[11] A. Floridia and E. Sanchez, "A JTAG-based fault emulation platform for dependability analyses of processor-based ASICs," in *Proc. IEEE 12th Latin Am. Symp. Circuits Syst. (LASCAS)*, 2021, pp. 1–4.

[12] S. Di Carlo, P. Prinetto, D. Rolfo, and P. Trotta, "A fault injection methodology and infrastructure for fast single event upsets emulation on Xilinx SRAM-based FPGAs," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, 2014, pp. 159–164.

[13] P. Kocher et al., "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Security Privacy (SP)*, 2019, pp. 1–19.

[14] M. Lipp et al., "Meltdown," 2018, *arXiv:1801.01207*.

[15] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz, "Processor hardware security vulnerabilities and their detection by unique program execution checking," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2019, pp. 994–999.

[16] K. K. Skordal. "A simple risc-v processor for use in FPGA designs." 2021. [Online]. Available: https://github.com/skordal/potato

[17] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin, "CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework," in *Proc. IEEE Int. Conf. Comput. Design*, 2008, pp. 363–370.

[18] M. Barbirotta et al., "Fault resilience analysis of a RISC-V microprocessor design through a dedicated UVM environment," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, 2020, pp. 1–6.

[19] (Xilinx Semicond. Manuf. Commer. Co., San Jose, CA, USA). *Petalinux Tools*. (2020). [Online]. Available: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html

[20] D. E. Owen, J. Joseph, J. Plusquellic, T. J. Mannos, and B. Dziki, "Node monitoring as a fault detection countermeasure against information leakage within a RISC-V microprocessor," *Cryptography*, vol. 6, no. 3, p. 38, 2022. [Online]. Available: https://www.mdpi.com/2410-387X/6/3/38

[21] (Synopsys Softw. Co., Sunnyvale, CA, USA). *Synopsys Design Compiler*. (2020). [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html

[22] L. T. Clark et al., "ASAP7: A 7-nm finFET predictive process design kit," *Microelectron. J.*, vol. 53, pp. 105–115, Jul. 2016.

[23] (Cadence Des. Syst. Comput. Softw. Co., San Jose, CA, USA). *Cadence Innovus Implementation System*. (2020). [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html

[24] (Xilinx Semicond. Manuf. Commer. Co., San Jose, CA, USA). *Vivado Design Suite*. (2020). [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[25] D. L. McMurtrey, "Using duplication with compare for on-line error detection in FPGA-based designs," M.S. thesis, Dept. Elect. Comput. Eng., Brigham Young Univ., Provo, UT, USA, 2006.

[26] J. Plusquellic, D. E. Owen, T. J. Mannos, and B. Dziki, "Information leakage analysis using a co-design-based fault injection technique on a RISC-V microprocessor," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 3, pp. 438–451, Mar. 2022.

Idris Somoye (Member, IEEE) received the B.S. degree from Morgan State University, Baltimore, MD, USA, and the M.S. degree from the Georgia Institute of Technology, Atlanta, GA, USA.

He is a Staff Member with Sandia National Laboratories, Albuquerque, NM, USA, and a graduate student with the University of New Mexico, Albuquerque. His research interests include hardware acceleration, and emulation-based hardware verification and security.

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

He is an Integrated Circuit Designer with Sandia National Laboratories, Albuquerque, conducting research in embedded FPGA security, security fault analysis, and superconducting electronics.

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

He is currently with the Department of Defense conducting research in μP reliability, lightweight cryptography solutions, secure wireless networks, and is a member of the Trusted Computing Group.

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

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.