cryptography

MDPI

Article

An Engineered Minimal-Set Stimulus for Periodic Information Leakage Fault Detection on a RISC-V Microprocessor

Idris O. Somoye [1,*], Jim Plusquellic [1], Tom J. Mannos [2] and Brian Dziki [3]

[1] Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM 87131, USA; jimp@ece.unm.edu
[2] Advanced CMOS Products/Design, Sandia National Laboratories, Albuquerque, NM 87131, USA; tjmanno@sandia.gov
[3] Information Assurance Research, Department of Defense, Fort G. G. Meade, Fort Meade, MD 24003, USA
* Correspondence: isomoye@unm.edu

Abstract: Recent evaluations of counter-based periodic testing strategies for fault detection in Microprocessor (μP) have shown that only a small set of counters is needed to provide complete coverage of severe faults. Severe faults are defined as faults that leak sensitive information, e.g., an encryption key on the output of a serial port. Alternatively, fault detection can be accomplished by executing instructions that periodically test the control and functional units of the μP. In this paper, we propose a fault detection method that utilizes an 'engineered' executable program combined with a small set of strategically placed counters in pursuit of a hardware Periodic Built-In-Self-Test (PBIST). We analyze two distinct methods for generating such a binary; the first uses an Automatic Test Generation Pattern (ATPG)-based methodology, and the second uses a process whereby existing counter-based node-monitoring infrastructure is utilized. We show that complete fault coverage of all leakage faults is possible using relatively small binaries with low latency to fault detection and by utilizing only a few strategically placed counters in the μP.

Keywords: fault emulation; RISC-V; FPGA; ATPG; fault analysis; fault detection; DFT; information leakage; cryptography

Citation: Somoye, I.O.; Plusquellic, J.; Mannos, T.J.; Dziki, B. An Engineered Minimal-Set Stimulus for Periodic Information Leakage Fault Detection on a RISC-V Microprocessor. *Cryptography* **2024**, *8*, 16. https://doi.org/10.3390/cryptography8020016

Academic Editor: Carlo Blundo

Received: 20 March 2024
Revised: 14 April 2024
Accepted: 18 April 2024
Published: 22 April 2024

1. Introduction

Information leakage in μPs, a security vulnerability that occurs when sensitive information is accessed or transmitted without proper authorization while executing applications such as cryptographic algorithms, has become a hotbed for research over the last couple of years [1,2]. The challenges associated with providing leakage-safe implementations are numerous and stem from the existence of countless vectors that lead to these situations. For example, Electromagnetic Induction (EMI) can cause a μP to enter an unexpected state, or a physical attack can damage the μP or disrupt its operation. Another common cause of stuck-at faults is hardware failure, such as a faulty transistor or a damaged electrical connection. Software bugs can also cause a system stuck-at fault. For example, a bug in the operating system or a malicious application can cause the μP to enter an infinite loop, preventing it from servicing other concurrent applications. When these faults occur, it is important to identify the root cause of the failure, which typically involves examining hardware components, analyzing software logs, and performing diagnostic tests.

In this paper, we propose a low-overhead method that utilizes already existing DFT scan chains and a handful of counters in conjunction with a specially designed binary to achieve low-latency hardware fault detection. We propose two methods of creating the specially designed binary. In the first method, we use a counter-based leakage detection method and a processor run-cycle analysis to determine the failure point on the μP caused by an injected fault and to recreate the processor state at that failure point. In the second method, a binary is constructed with the assistance of ATPG tools, which coerces ATPG

vectors into closely matched processor instructions and register file values. The goal is to create a relatively small binary program that provides high levels of fault coverage. Experimental results are provided for each method and compared to determine which process provides the highest coverage when considering binary generation complexity.

The specific contributions of this work include the following:

- The evaluation of fault propagation latency for leakage scenarios when executing a common cryptography algorithm on an RISC-V processor.
- An analysis and discussion of the process that converts exact processor states, i.e., register values, peripheral states, and instruction inputs, into specialized low-instruction-count binaries targeted at triggering hard-to-reach faults. Given that the binary will be stored in memory for periodic fault detection, minimizing size, e.g., less than 100 instructions, is an important design goal.
- An analysis of the specialized binary executables and an analysis that compares fault trigger latencies with the latencies obtained when executing common cryptography algorithms.

The remainder of this paper is organized as follows. Section 2 discusses additional related work. Section 3 describes the experimental design and attributes of the binary generation sequence. Section 4 presents the details of the proposed Periodic Built-In-Self-Test (PBIST). Section 5 presents the fault coverage and latency results for the generated binaries. Section 6 presents our conclusions.

2. Related Work

An overview of the different strategies that can be employed to detect faults through either continuous checkers (also called concurrent) or periodic testing is provided in [3]. The authors describe four general approaches, namely redundant execution, PBIST, dynamic verification, and anomaly detection. The periodical specialized binary run described in this work falls under the periodic built-in self-test category. The method is uniquely applied here to detect faults before information leakage occurs and is portable to a wide range of μP architectures and input–output peripherals. The methods described in previous work have higher overhead and do not address protection against information leakage.

Software-only fault detection methodologies are described in [4–6], which significantly improve reliability without requiring hardware modifications. This makes software redundancy techniques significantly cheaper and easier to deploy. For example, the authors of [4] used code transformation and specialized instructions to create fault-resistant binaries, which require a lengthy processor-specific, fault-agnostic run and do not provide true fault detection—only fault tolerance under certain circumstances.

The authors of [7] introduced an RISC-V framework for hardware–software codesign that can aid in the implementation of secure and safe SoCs based on RISC-V. The script-based framework provides cycle-true verification, ensuring accuracy in the simulation of hardware and software interactions. The framework's versatility makes it applicable in various scenarios, including designing systems resilient against Side-Channel Attacks (SCAs) and other vulnerability points. Additionally, the authors show that the framework enables the fast implementation, functional verification, and post-synthesis verification of projects such as the design of Post-Quantum Cryptography ISA extensions for RISC-V and cryptographic hardware accelerators for the Advanced Encryption Standard (AES). While this framework is effective in speeding up the evaluation of software-aware. hardware-dependent metrics such as performance, power consumption, and area utilization, it has not been shown to be capable of aiding in the detection of hardware-based information leakage faults.

The authors of [8] proposed a predominantly software-based fault detection scheme supplemented by hardware. They utilized a special instruction set, which they coined as Access-Control Extension (ACE), that interacts with a custom-instrumented, full-scan chain to test the μP. Unfortunately, the specialized instructions add complexity to the μP and create a side-channel attack vector. The implementation of their approach is complicated

because the ACE instructions are privileged to only the ACE firmware. Additionally, the tree architecture exposes no avenue to target information leakage sites. In contrast, our proposed periodic testing method introduces only a small set of counters and utilizes standard instructions, eliminating the need for custom instructions.

Austin [9] proposed a μP with a unique architecture called Dynamic Implementation Verification Architecture (DIVA), designed to detect both transient and permanent faults. In DIVA, a checker validates the functional unit result by recomputing it using the instruction's input operands and compares this result before permitting the instruction to commit. Despite the advantage of a simplified checker design due to the leveraging of processor pipeline decisions, there is considerable overhead in the checker pipeline. This limitation restricts its practical use to super-scalar architectures. Additionally, the effectiveness of DIVA relies on the assumption that the register file and memory employ Error-Correcting Code (ECC) for error detection and correction, serving as a mitigation strategy against faults related to storage.

In [10], a high-level, symptom-based fault detection technique combining hardware and software was introduced. This method monitors software execution to identify anomalous behavior. The fault detection process occurs at a high level by observing hardware traps and utilizing μP performance counters. While the technique demonstrates an ability to detect 95% of unmasked faults, it comes with a potential drawback of high latency. Most faults are identified within the first 100,000 instructions, but some may take longer, extending up to 10 million instructions.

In recent work [11], a high-speed fault emulation platform was developed on an FPGA to assess the Potato RISC-V μP [12]. A dynamic verification or continuous symptom-monitoring approach was proposed to evaluate information leakage events introduced by faults from various classes. The study delved into the effectiveness and latency associated with a set of countermeasures based on self-assertions called Self-Assertion-Based Countermeasures (SABC)s. Self-Assertion-Based Countermeasures (SABC)s perform consistency checks on instruction and datapath values during program execution. The fault detection results and the associated latency are compared to those provided by a periodic counter-based countermeasure proposed in [13]. The evaluation of the SABCs includes assessing the number of severe faults they can detect, the latency associated with these detections, and the extent of collateral coverage for active faults. The results demonstrated that SABCs are nearly as effective as the node counter-based CMs in detecting all active faults and are nearly equivalent in effectiveness for detection of severe faults. Notably, all severe faults are successfully detected by SABCs, highlighting the effectiveness of the proposed countermeasures in preventing information leakage during program execution. The SABCs, however, are expected to scale somewhat poorly to more complex microprocessor architectures, including super-scalar architectures. Integrating them will demand adjustments and additional resources to navigate the heightened intricacies of the pipeline. Specifically, this involves synchronizing assertions with out-of-order executed instructions and managing the complexities associated with branch prediction and execution.

In other recent work [13,14], a counter-based node-monitoring technique and a fault injection technique were proposed. In this paper, we will expand on previous work to explore μP information leakage by analyzing internal node fault effects and discuss a low-overhead fault detection methodology that enables periodic fault detection without the need for special instructions or to take the μP offline.

3. System Overview

This section describes the RISC-V architecture used in the emulation experiments, including a special add-on feature referred to as Emulation ROM Side Loading (ERSL), which enables binary executable loading to be accomplished at run time, as well as the characteristics of the fault campaign, Fault Injection Manager (FIM) and Fault Emulation Engine (FE). Also discussed are the CAD tools used in the synthesis and implementation, the testing process, and details regarding counter-based periodic testing.

3.1. RISC-V Architecture

The architecture of the Potato μP utilized in this research is shown in Figure 1. Potato is compliant with the RISC-V v2.0 standard [12] and is classified as a 32-bit RISC-V ISA CPU core (RV32I). It possesses a complete set of integer instructions with Control and Status Register (CSR) and exception handling while supporting RISC-V integer (I), multiplication and division (M), and CSR instruction (Z) extensions (RV32IMZicsr). All instructions except load and store execute in one clock cycle. Potato utilizes the wishbone B4 standard [15] as an internal bus.

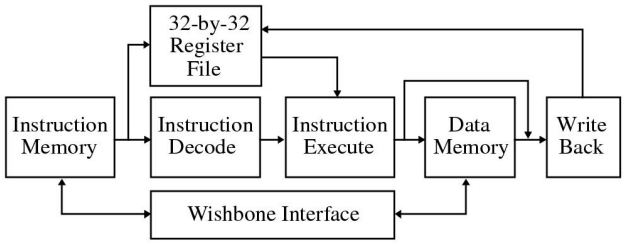

Figure 1. Block diagram of Potato's five-stage pipeline [12].

3.2. Fault Campaign Characteristics

A fault campaign refers to the characteristics of the Fault Injection (FI) system [16]. The architecture employed in this research is shown in Figure 2 and has the following features:

- The Potato μP [12] serves as the processor under test, configured with a 32 KB ROM for application code and a 132 KB BRAM for scratch memory. The netlist for Potato is generated using an ASIC synthesis and place-and-route computer-aided design (CAD) tool flow in which 34,110 fault injection circuits are integrated. The netlist is instrumented with scan chains, which provide access to fault injection circuits and counters. The instrumented netlist is used as input to an FPGA CAD tool flow to produce the programming bitstream for the FPGA.
- The Xilinx UltraScale+ Multiprocessor System-On-Chip (MPSoC) FPGA on the ZCU102 development board serves as the emulation platform for the Potato μP.
- The Fault Injection Manager (FIM) is implemented as a C program that runs on an embedded processor within the FPGA. Similar to the FI architecture proposed in [11], we leverage two 32-bit high-speed, memory-mapped General-Purpose Input/Output (GPIO) registers to facilitate fault injection, control, and counter data retrieval between the Processing System (PS) and Programmable Logic (PL) components.
- The fault injection circuits implement four fault types, namely stuck-at-0 (SA0), stuck-at-1 (SA1), delay, and inversion, and are configured using the GPIO-connected scan chains.
- The FE is realized as a set of State-Machines (SM)s designed to collect serial and address bus data as Potato executes the Advanced Encryption Standard (AES) algorithm [17]. Configured by the FIM, the SMs limit the number of run cycles. When combined with a binary search routine implemented within the FIM C program, this setup enables the latency of fault effects to be determined.
- A wishbone-based independent ROM binary side-load architecture is integrated into the design, which significantly accelerates the testing process.
- The C program running in the PS of the FPGA is used for communication with and control of the Fault Emulation Engine (FE) and the ROM wide-load module, which are both implemented in the PL.
- The fault detection capabilities and detection latencies of the countermeasures (CMs) are assessed offline using data collected from the scan chains.

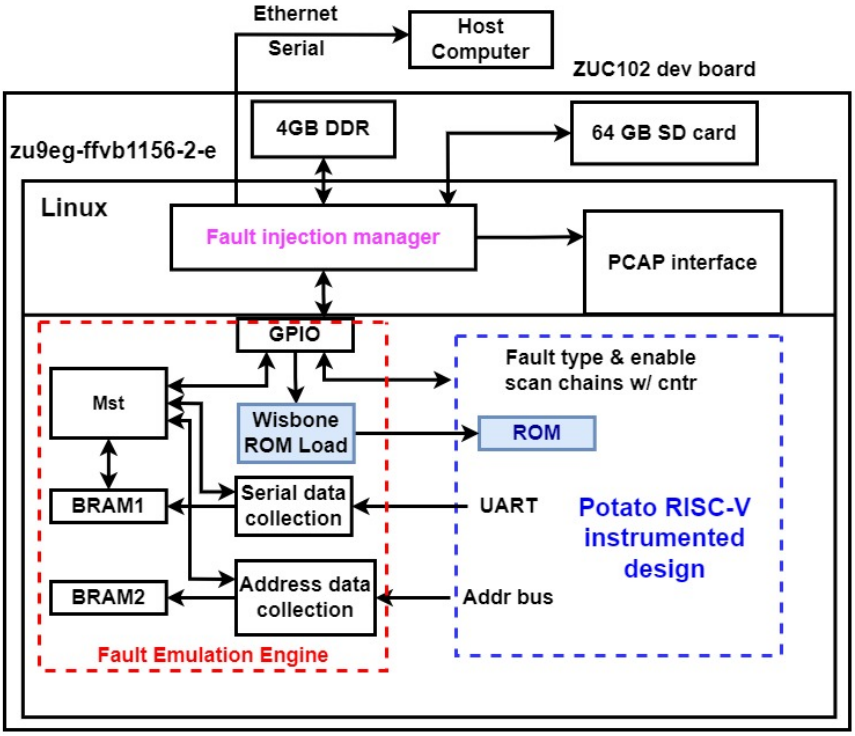

Figure 2. Block diagram of the experimental setup with ROM side load for rapid binary loading and testing. Adapted from [13].

3.3. System Architecture

The emulation hardware platform uses a Xilinx ZCU102 development board [18], in which both the PS and PL are utilized in a codesign-based system architecture.

3.4. Fault Injection Circuit with Counter (FIC)

The Fault Injection (FI) circuit structure is implemented using three scan chains, namely *scan_in[0]*, *scan_in[1]*, and *scan_in[2]*. The first scan chain, with *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 fault injection with counter, or FIC, refers to the encompassing circuit, which includes both a counter and an FI circuit instance. The scan chains are extended into the counter circuit, as shown in Figure 3, 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 during the program's execution.

In prior work [11], we showed that a substantial proportion of the active faults within various fault types, including Stuck At 0 (SA0), Stuck At 1 (SA1), delay, and invert fault classes, can be identified by a relatively small number of counters. In particular, a set of five counters has been shown to identify a large fraction of all faults. Notably, this identical set of counters also proves adept at detecting all severe faults, underscoring their efficacy across all active fault scenarios. However, several severe faults have been shown to have high latency, with as many as 6 million cycles during program execution.

In this work, we demonstrate that a small number of μ*P*s with carefully crafted instructions designed to exercise specific nodes and the small subset of counters identified in previous work, referred to as TopCounters, can be used to detect all faults that lead to information leakage with very low latency. Therefore, an effective countermeasure can be constructed with the node-monitoring counters in Figure 3 (without the fault injection portions) to serve as a part of a Periodic Built-In-Self-Test (PBIST) Counter-Measure (CM) for the detection of information leakage faults in the Potato RISC-V design.

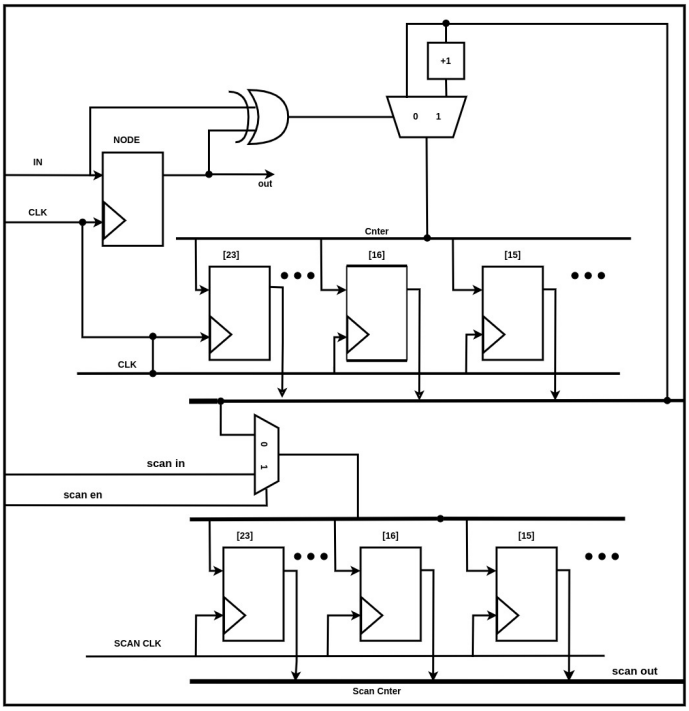

Figure 3. Schematic of the counter circuit without fault injection signals.

3.5. Testing Process

To initialize each test, the C program uses the The Processor Configuration Access Port (PCAP) interface to configure the FPGA fabric with the instrumented Potato bitstream. This bitstream incorporates the default AES binary executable stored in the BRAM-based emulated ROM of the μP. The ERSL shown in Figure 4 is used to override the boot memory locations with the executable binary being tested. The fault-free counter values associated with each designed binary are then computed by executing a fault-free run of Potato at 1024 clock cycle increments for the entirety of the AES algorithm execution, which spans 6,717,440 clock cycles. In subsequent steps of the testing process, faults are injected in a set of faulty runs of the selected binary to determine the latency to fault detection.

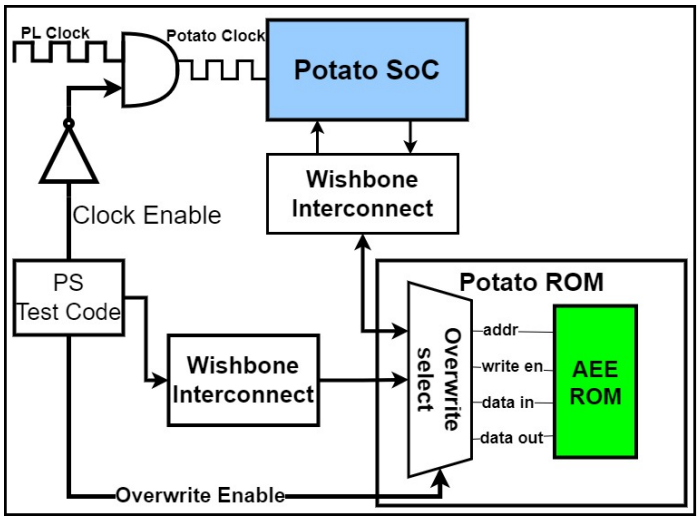

Figure 4. Block diagram of the Wishbone side-load architecture.

4. Fault Trigger Binary Executable (FTBE)

The Fault Trigger Binary Executable (FTBE) is a minimized set of instructions designed to recreate a processor state that leads to information leakage in the presence of a fault

with minimal latency. In this section, we describe two methods that can be used to generate an FTBE and discuss the tradeoffs between the two. The first, which we call the Fault-Run-Cycle-Based FTBE (RCBE), is created by utilizing the counters already present in the FE, where we identify the run cycle in which faults are observable while running the cryptography algorithm encode/decode sequence. The second method, which we call the ATPG-Based FTBE (ATPGB), is created by using an ATPG flow to create high-coverage test vectors, which are coerced into μP instructions and represent a sequence of concise stimuli.

4.1. Run Cycle-Based Binary

The process of generating the Fault-Run-Cycle-Based FTBE (RCBE) is visually presented in Figure 5 and is categorized into three segments. The first segment is called Incremental Search Fault-Detected Cycle (IFTC) and is color-coded in blue. It involves determining the specific run cycles during which faults become observable on the node counters while the AES algorithm is executing. The second set of steps handles the determination of Microprocessor (μP) state at IFTC, color-coded in purple, is executed once the IFTC is found and used to determine the state of the Microprocessor (μP) at the identified fault-observable run cycles. The last segment, color-coded in green, is called binary creation for fault-triggered state replication. In this step, using the knowledge of the IFTC and μP state, a binary is crafted to replicate the state of the μP where faults are triggered. Each step is discussed in greater detail in the following subsections.

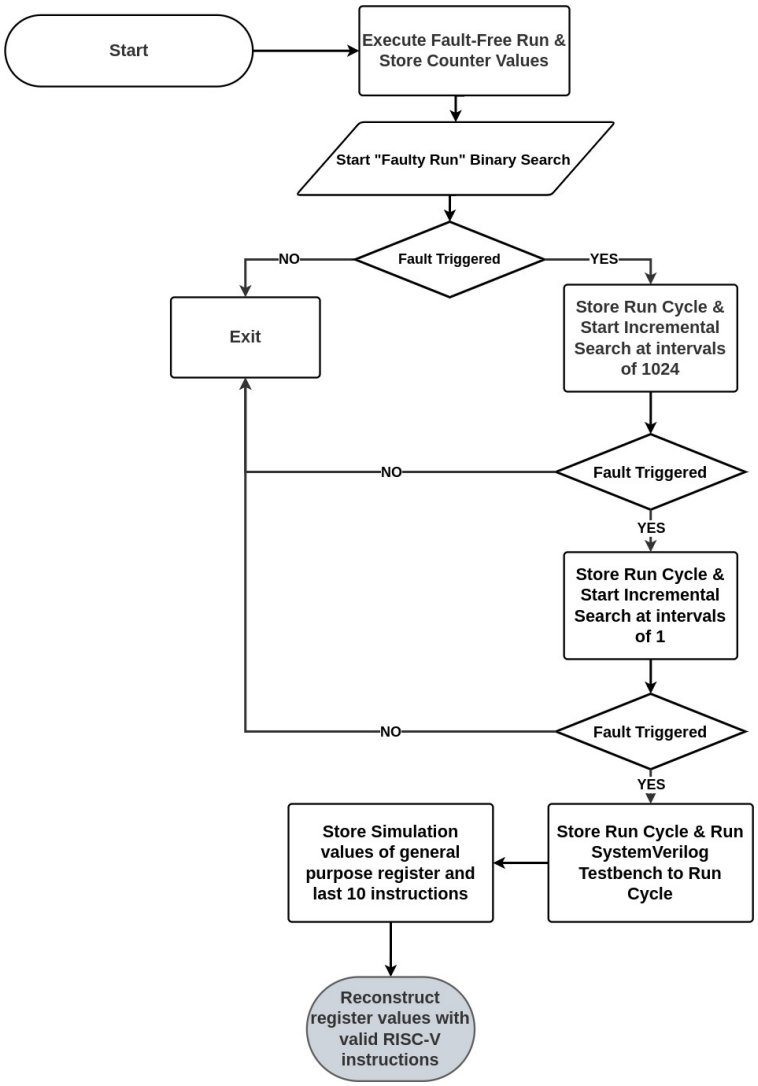

Figure 5. Flowchart for the generation of the RCBE.

4.1.1. Identification of Fault-Observable Run Cycles (IFTCs)

Identifying the IFTC requires generating and searching through node counter values for over 6 million clock cycles. Carrying out this search in a single-cycle incremental search would be impractical; therefore, the task is broken down into stages that combine a binary search and an incremental search to define a solution that has much lower memory and run-time overhead.

The process for identifying the IFTC is listed as follows:

1. Fault-free counter values for each node while running the AES binary are generated and stored in increments of 1024 cycles from cycle 0 to cycle 6,717,440.
2. A binary search with an exponentially increasing multiple of 1024 clock cycles is then performed.
   - The binary search concludes when a multiple of 1024 clock cycles is identified where the fault remains undetected at the lower bound but is detected at the upper bound.
   - Since the search spans from 1024 to 6,717,440 clock cycles in increments of 1024, each fault necessitates 13 iterations to complete the process.
3. When a fault is detected through the binary search method, the run cycle is stored as a Binary Search Fault-Detected Cycle (BFTC).
   Table 1 shows the BFTCs (severe-fault detect cycles) at 1024 cycle increments for Potato when running the AES binary. Each run cycle is passed onto the next step in the process, incremental sweep search, as a starting value for the single-cycle increment analyses. The highest latency fault triggered is at cycle 6,078,464 and correlates with delay faults. Detecting these delay faults at much lower latencies is crucial for this work.

Table 1. AES BFTCs on Potato.

Fault Type	Run Cycle Triggered
Stuck-At-0	1024; 552,960; 557,056; 1,728,512
Stuck-At-1	1024; 557,056; 552,960; 1,728,512; 1,478,656
Delay	1024; 552,960; 557,056; 573,440; 1,478,656; 1,728,512; 1,732,608; 6,078,464
Invert	1024; 557,056; 552,960; 1,728,512

4. Perform an incremental sweep strategy.
   - The incremental sweep begins at run cycle $BFTC - 1024$ due to the binary search being implemented at 1024 clock cycle increments and iterates at single-clock period increments. Both the fault-free and faulty tests are executed in this 1024-cycle incremental run.
   - The clock cycle in which the counter values differ is referred to as the Incremental Search Fault-Detected Cycle (IFTC) and is the exact cycle in which the fault is first observable by analyzing the node counters.
   - A script that takes multiple BFTCs as input arguments and performs the incremental search while iterating through the BFTCs is utilized to automate this process.

Table 2 shows the IFTCs, (severe-fault detect cycles)at single-cycle increments for Potato when running the AES binary. Each run cycle is passed onto the next step as a stop cycle for the simulation run.

Table 2. AES IFTCs on Potato.

Fault-Type	Run Cycle Triggered
Stuck-At-0	184; 641; 553,064; 550,800; 1,728,272
Stuck-At-1	184; 641; 550,800; 553,064; 553,320; 1,728,272; 1,477,576
Delay	184; 641; 553,064; 553,320; 550,800; 569,576; 1,477,576; 1,728,272; 1,728,696; 1,729,024; 1,728,512; 6,076,072
Invert	184; 641; 550,800; 553,064; 1,728,272

4.1.2. Determination of Microprocessor (μP) State at the IFTC

The μP state at the IFTC is extracted with a SystemVerilog test bench, which instantiates a clean, non-instrumented version of Potato loaded with the same AES executable as the instrumented Potato. This ensures that, in a fault-free run, both the simulated and emulated Potato would be in the same state, including register values, peripheral states, and instruction inputs. The test bench starts at run cycle 0 and simulates Potato to the IFTC; it then stores the values of each general-purpose register, as well as the previous 10 instructions before that point. The test bench also stores the values read from and written-to-execution memory and the Wishbone interconnect bus for that time frame.

4.1.3. Binary Creation for Fault-Triggered State Replication

With the μP state values at the IFTC extracted, the FTBE is created with the following steps:

1. First, register states are recreated. This is accomplished by utilizing Load upper immediate (LUI) and ADD Immediate (ADDI) instructions (Note that Potato does not implement the Load Word Immediate (LWI) instruction and that LWI should be used for processors that do implement the instruction).
2. Second, memory locations that are accessed by the binary executable are reconstructed by storing the values read by the μP in the simulation in the addresses from which the binary executable will later be read.

Combining these into a binary program, with the memory and peripheral state reconstruction instructions executed first, followed by the register load instructions, creates an executable that mimics the processor state at the IFTC. Figure 6 shows a system-level view for generating the RCBE from node counter values and which device is utilized. The counter values are extracted from the Potato core by the PL, the fault trigger cycles are analyzed by the PS, the processor state values are determined by a simulation running on the host, and the binary is then constructed. Finally, the binary is loaded into the Potato core for testing using the ERSL.

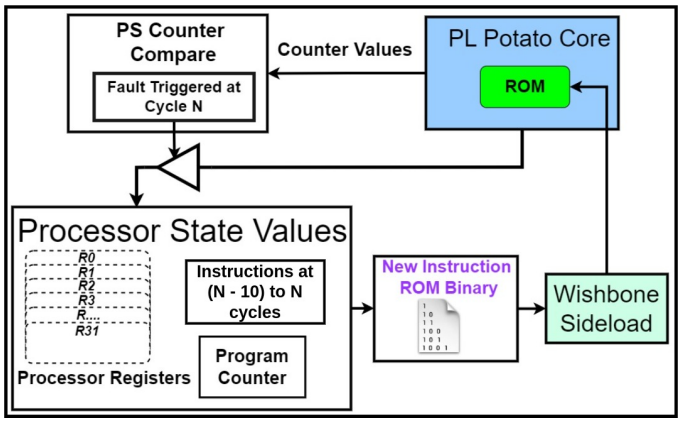

Figure 6. Block diagram of the Fault-Run-Cycle-Based FTBE (RCBE).

4.2. ATPG-Based Binary

Creating an ATPG-Based FTBE (ATPGB) requires the generation of ATPG test vectors and conversion of those normally serial test vectors into binaries that correlate with high fault coverage of the ATPG test vectors. Figure 7 shows a flowchart for the ATPGB generation process. The first step is to modify the Potato RTL so that the values of the instruction memory and general-purpose registers are observable on output ports at the top level. This is done to make extracting the μP state values easier following synthesis and to ensure minimal impact on the generated test patterns. Next, an open-source Design For Testing (DFT) solution, *AUCOHL-Fault* (Fault) [19], is utilized to automatically generate test patterns. The test patterns are then converted into an FTBE.

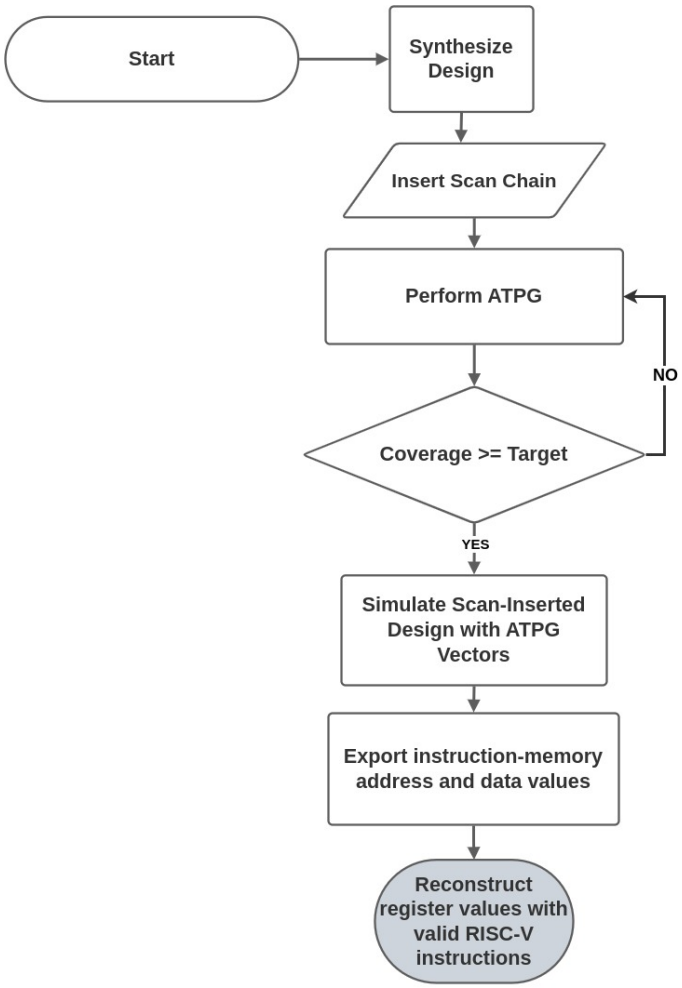

Figure 7. Flowchart for generating the ATPG FTBE.

To create the ATPG test vectors, the Potato module is synthesized and mapped to the osu035_stdcells library [20] by calling the Fault synth command. This initiates a Yosys [21]-based synthesis script and generates a flattened netlist. The netlist is then *cut* using the *fault cut* command, which eliminates the flip-flops in the netlist, converting it into a pure combinational netlist, which is utilized in conjunction with the original netlist to generate the test vectors. Lastly, test vectors are generated with the *AUCOHL-Faults* built-in PODEM [22] test pattern algorithm. Patterns are generated with default values of 100 test vectors and an expected minimum of 95% coverage. The generated test vectors for Potato offer 82% fault coverage of the design.

Figure 8 shows a diagram of the conversion process from ATPG test vectors to the FTBE, which is described as follows:

1. The DFT scan-inserted Potato μP is simulated in test_mode, i.e., scan_en asserted, with the ATPG vectors as a stimulus in the scan_in port.
2. During the simulation, instruction memory values, as well as the general-purpose registers, are captured from the signal exported to the top-level ports and stored for each test vector.
3. Processor state values are parsed to extract only those that contain valid instruction memory inputs and within valid address ranges. The processor state values extracted from the ATPG vectors differ from the RCBE in that preceding instructions are not available due to the non-contiguous nature of ATPG vectors. As a result, the binary program conversion process is slightly different.
4. An RISCV assembler [23] is used to convert the instructions into the RV32I [24] set that Potato supports.
5. The set of instructions is used to heuristically construct a coherent binary program while avoiding endless loops and other unwanted processor states.

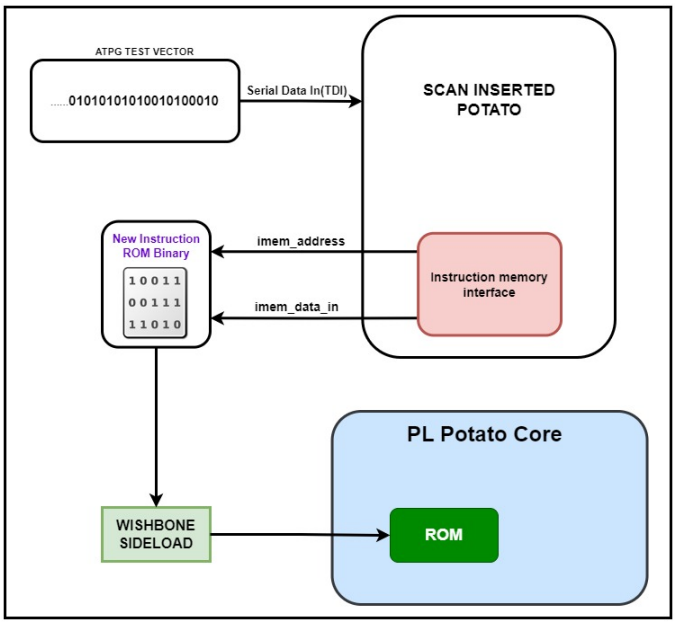

Figure 8. Block diagram of the ATPG-based FTBE.

5. Experimental Results

The primary goal of the FTBE fault coverage experiment is to determine the fault detection capabilities of each binary executable and to identify the minimum latency in which they trigger those faults. In this section, we focus on identifying the designed binary programs that provide the smallest possible severe fault trigger latency and compare the results with the latencies of other binary executables. The analysis in this section is carried out on the faults in the SevereFaults [14] class only and the five counters discussed earlier in Section 3.4 that are determined to provide the highest fault coverage.

For each test run, the ERSL is utilized to override the initial binary that Potato is synthesized with, saving hours of bitstream generation time by avoiding the need to re-synthesize Potato each time a new binary is tested. To avoid running Potato while the instruction memory is changing, the ERSL loads the binary under test to the emulated ROM module using a secondary clock while ensuring that the clock is de-asserted to the rest of Potato.

A fault-free run is then performed with each binary to obtain fault-free counter values at 1024 increments. Next, Potato is run with faults introduced for SA0, SA1, invert, and delay severe faults while examining the TopCounters values. Figure 9 plots the fault trigger results for each binary program on faults in the SevereFaultsclass as percentages of the total number of faults in the class. Each binary is assigned a unique color to differentiate it

from others. The size of each binary in kilobits is presented along the x-axis. The fraction of SevereFaults detected by each binary is presented along the y-axis. Each point represents the fraction of SevereFaults detected by each binary in comparison to the binary size. The results for each tested binary program are summarized below.

- For the Fault-Run-Cycle-Based FTBE methodology, the optimal number of binaries needed to satisfy requirements is determined to be three; B0, B1, and B2 are generated from IFTC *6,076,072*, *1,728,512*, and *550,800*, respectively. Each binary targets specific faults that are triggered at the IFTC from which it is derived.
  - *B0* triggers nine faults;
  - *B1* triggers one-hundred and thirteen faults;
  - *B3* triggers the remaining one-hundred and twenty-two faults.
- The ATPG binary requires fewer instructions in total, which directly correlates with the memory overhead needed to store the target binary; however, it also detects fewer SevereFaults than the previous binaries, at 188.
- The Coremarks algorithm requires the most instructions in total and triggers 191 faults in the SevereFaults class throughout a full program run.
- The hello world program, included in the analysis for surety, requires the smallest amount of instructions but only triggers one fault in the SevereFaults class.

We surmise from this analysis that while the ATPGB provides adequate coverage, it is likely limited by the test vector coverage achieved by the pattern generator, as well as the complexities of converting disjointed ATPG test vectors into effective coherent binary programs. A possible better utilization of ATPG principles in this research track could be the creation of a special five-node scan chain that is made up of the TopCounter nodes. This concept is discussed fully in Section 5.3.

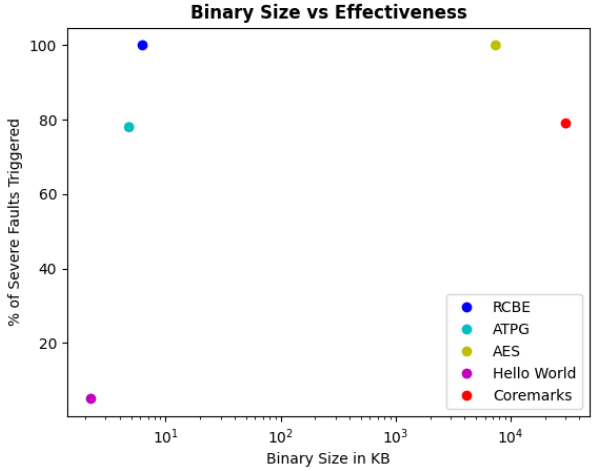

Figure 9. Binary size-to-effectiveness comparison for both FTBEs and several other general binaries.

Additionally, RCBEs are shown to be effective in detecting the specific faults observed at the IFTCs they are generated from but are not guaranteed to detect any other faults, even though they often do. Also, their modular nature means that multiple binaries or a larger contiguous binary would have to be stored to fully utilize this method. Combining multiple RCBEs in one is a relatively simple process, consisting of overwriting registers that differ between binary programs, then updating instructions. Overall, we believe the RCBE method provides a combination of binary size and low latency for the achievement of 100% information leakage fault coverage.

5.1. Latency Analysis

The objective of latency analysis is to determine whether the FTBEs can detect the presence of faults well before the AES algorithm, thereby justifying the area overhead that

their storage incurs. This goal is addressed for the FTBEs by evaluating latency for each designed binary, then comparing them to other generic binaries. For this analysis, only faults in the SevereFaults class are considered. First, fault-free runs are executed for each binary to acquire fault-free counter values. Then, fault-injected runs are performed for each binary and fault type at 4096 cycle increments from clock periods 0 to 1,024,000. The latency results are presented as a cumulative fault detection graph in Figure 10, where the number of clock cycles that Potato is run for is plotted along the x-axis and the cumulative number of detected faults is plotted along the y-axis. Each binary program is assigned a different color for differentiation, and a color guide is provided to the right. Both FTBEs demonstrate much better performance in terms of latency when compared to other standard binaries. The RCBE detects all SevereFaults by 4096 run clock cycles, while the ATPGB triggers 188 faults by the 4096 run clock cycle index.

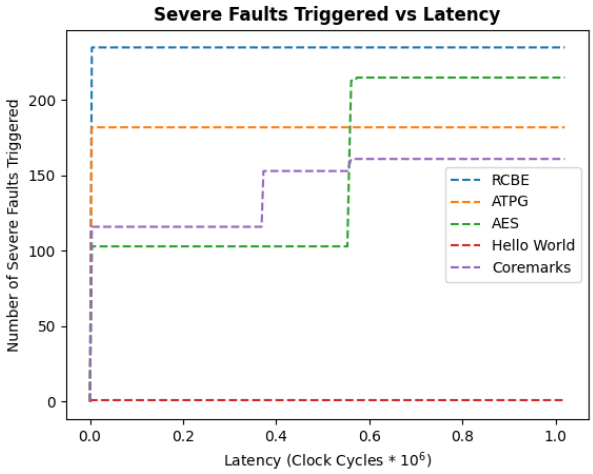

Figure 10. Subset of fault trigger latencies for select binaries.

5.2. Overhead Analysis

In this section, a comparative analysis is undertaken to evaluate the performance and area overhead of the proposed counter-based Counter-Measure (CM), along with the FTBE, in contrast to previous works. Because our primary focus is detecting leakage-sensitive faults, comparing the overhead of our technique with previous methodologies that aim for complete fault coverage is not a clear comparison, as they essentially address different problems. Leakage-sensitive faults have inherent latency and sequence dependencies, while generic faults might not; this discrepancy is likely to lead to differences in overhead costs of the detection techniques. Additionally, techniques such as continuous symptom monitoring (CSM) and PBIST also have unique complexities that must be taken into account when making comparisons.

The counter circuit, consisting of two 24-bit counters, is analyzed for area overhead using the Synopsys Design Compiler and the ASAP7 standard cell library. The synthesis report indicates an area of 339 µm² per counter, and deploying five counters results in an overhead of 1695 µm². In contrast, the Potato core has a larger area of 28,510 µm². The fractional area overhead is approximately 5.9%. The performance overhead is estimated using a checkpoint interval of 100 million instructions, akin to the ACE technique reported above. Notably, unlike the ACE methodology, the number of scan clock cycles is minimal (120 with 5 24-bit counters), and the vast majority of self-test time is attributed to program execution. Leveraging the full runtime required to reach maximum fault detection of the generated FTBEs' 1350 clock cycles for the RCBE and 1072 clock cycles for the ATPGB, the performance overhead is estimated at $(1350, 1072) \div 100$ million $\approx (0.0035\%, 0.00107\%)$ for the RCBE and the ATPGB, respectively. Thus, our proposed counter-based CM + FTBE methodology incurs minimal overhead compared to existing methods.

It is assumed that the FTBE will be stored in non-volatile memory, which is tightly coupled with the μP, perhaps on chip at some offset from the bootloader. This presents another overhead that must be considered and analyzed. Using an area of 0.52 µm² per memory bit, we estimate the total area of each FTBE. We calculate area overhead by taking into account the total number of bits per binary and the total area of Potato, namely $((6300, 4800) \times 0.52\ \mu m^2) \div 28{,}510) \approx (11.4\%, 8.7\%)$, respectively. Table 3 shows the area overhead comparison of our two generated binaries with the closest equivalent in the literature [8]. While the FTBEs require more overhead area than existing works, the Potato μP analyzed in this work is a much smaller core than the OpenSPARC T1 μP used in [8]. A comparison with the Rocket μP analyzed in [13], which has an overall area of 112,224 µm², yields better overhead results, at 2.9% and 2.2% for RCBE and ATPGB respectively. Our prior analysis of both Potato and Rocket [11,13] implies that the number of counters necessary to detect all severe faults is common in most μP architectures; therefore, we have reason to believe that the counter-based periodic testing methodology will scale well with larger processor systems.

Table 3. Memory allocation per Counter-Measure.

Method	Memory Allocation (KBits)	Area Overhead (%)
RCBE	6.3	11.4
ATPGB	4.8	8.7
ACE	32	18.7
ACE Hybrid	32	5.8

5.3. Next Steps

The output of the Fault Trigger Binary Executable (FTBE) is a counter value that could be stored in memory and later compared to the fault-free counter value also stored in memory. Additional efforts would be needed to create a test controller software binary that works in conjunction with hardware timers to periodically run the Fault Trigger Binary Executable (FTBE). This test controller would need to have special rights if running on an operating system, and analysis would be performed to determine what test frequencies would be ideal when PBIST-induced downtime is considered.

The ATPG-Based FTBE performs poorly compared to the Fault-Run-Cycle-Based FTBE. This is largely because ATPG vectors do not take into account sequential operations and transitions in the internal states of the circuit, which are typically associated with clock cycles or other triggering events. When bypassing state transitions, the objective is to directly set or reach a particular state without going through the intermediate states that would occur in normal circuit operation. This can be useful in certain testing scenarios where the primary goal is to reach a specific state quickly for fault detection or analysis. However, replicating these optimizations in a standalone binary and achieving the same levels of efficiency is difficult, as discussed in Section 5. A minimized scan chain consisting of only the TopCounter nodes could offer the precision of ATPG vectors without the need to serially scanin data to the over 34,000 nodes in Potato. Future work could investigate the process of mixing and chaining the TopCounter nodes into a "mini-scan-chain", as well as implementing a specialized BIST controller [25] that serially scans in the test vector and inspects the serial data output. These test vectors would likely be much shorter than those described in this work, while the BIST controller would likely consume minimal overhead due to only interacting with a handful of nodes.

In addition, the final goal for this track of research is to demonstrate the counter-based PBIST in a manufactured processor on a viable technology node. Future research could tape out the Potato μP with the counter-based countermeasures inserted at the five TopCounter nodes, with the FTBE and fault-free values stored in memory with a program set to run it periodically. This instrumented μP could then be inserted into a high-radiation environment

and tested for fault countermeasures. The insights gained from these experiments could be invaluable in accessing failure probability for both temporary and permanent faults and comparing theoretical countermeasure performance to actualized performance.

6. Conclusions

This paper investigates the generation of specially designed executable binary programs for a counter-based periodic BIST intended for the detection of faults in the Potato RISC-V microprocessor using an FPGA emulation platform. The specially designed binary programs are generated using two different methods, with varying success. The detection and latency capabilities of the designed binaries using the counter-based approach are evaluated on a subset of the active faults referred to as severe faults, which are defined as faults that leak sensitive information, e.g., a portion of the plain text and/or encryption key, through the serial port output. The designed binary programs, when utilized in combination with a small set of strategically placed counters, are shown to achieve high fault coverage and low latency while adding little overhead when compared to competing approaches.

Author Contributions: Conceptualization, I.O.S., J.P. and B.D.; methodology, I.O.S.; software, I.O.S.; validation, I.O.S.; resources, I.O.S. and J.P.; data curation, I.O.S.; writing—original draft preparation, I.O.S.; writing—review and editing, I.O.S.; visualization, I.O.S.; supervision, I.O.S., J.P., T.J.M. and B.D.; project administration, I.O.S. and J.P.; funding acquisition, I.O.S., B.D. and T.J.M. All authors have read and agreed to the published version of the manuscript.

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

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Yuce, B.; Ghalaty, N.F.; Deshpande, C.; Patrick, C.; Nazhandali, L.; Schaumont, P. FAME: Fault-attack Aware Microprocessor Extensions for Hardware Fault Detection and Software Fault Response. In Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016, Seoul, Republic of Korea, 18 June 2016. [CrossRef]
2. Sere, A.A.; Iguchi-Cartigny, J.; Lanet, J.L. Automatic Detection of Fault Attack and Countermeasures. In Proceedings of the 4th Workshop on Embedded Systems Security, Grenoble, France, 15 October 2009; Association for Computing Machinery: New York, NY, USA, 2009. [CrossRef]
3. Gizopoulos, D.; Psarakis, M.; Adve, S.V.; Ramachandran, P.; Hari, S.K.S.; Sorin, D.; Meixner, A.; Biswas, A.; Vera, X. Architectures for online error detection and recovery in multicore processors. In Proceedings of the 2011 Design, Automation Test in Europe, Grenoble, France, 14–18 March 2011; pp. 1–6. [CrossRef]
4. Reis, G.A.; Chang, J.T.Y.; August, D.I. Configurable Transient Fault Detection via Dynamic Binary Translation Princeton University CS Publications 2006. Available online: https://liberty.cs.princeton.edu/Publications/war06_swiftpin.pdf (accessed on 1 June 2023).
5. Oh, N.; Shirvani, P.; McCluskey, E. Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. Reliab.* **2002**, *51*, 63–75. [CrossRef]
6. Reis, G.; Chang, J.; Vachharajani, N.; Rangan, R.; August, D. SWIFT: Software implemented fault tolerance. In Proceedings of the International Symposium on Code Generation and Optimization, San Jose, CA, USA, 20–23 March 2005; pp. 243–254. [CrossRef]
7. Zulberti, L.; Di Matteo, S.; Nannipieri, P.; Saponara, S.; Fanucci, L. A Script-Based Cycle-True Verification Framework to Speed-Up Hardware and Software Co-Design: Performance Evaluation on ECC Accelerator Use-Case. *Electronics* **2022**, *11*, 3704. [CrossRef]
8. Constantinides, K.; Mutlu, O.; Austin, T.; Bertacco, V. Software-Based Online Detection of Hardware Defects Mechanisms, Architectural Support, and Evaluation. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), Chicago, IL, USA, 1–5 December 2007; pp. 97–108. [CrossRef]
9. Austin, T. DIVA: A reliable substrate for deep submicron microarchitecture design. In Proceedings of the MICRO-32: 32nd Annual ACM/IEEE International Symposium on Microarchitecture, Haifa, Israel, 16–18 November 1999; pp. 196–207. [CrossRef]

10. Li, M.L.; Ramachandran, P.; Sahoo, S.K.; Adve, S.V.; Adve, V.S.; Zhou, Y. Understanding the propagation of hard errors to software and implications for resilient system design. *ACM Sigplan. Not.* **2008**, *43*, 265–276. [CrossRef]
11. Somoye, I.; Mannos, T.J.; Dziki, B.; Plusquellic, J. Self-Assertion-based Countermeasures within a RISC-V Microprocessor for Coverage of Information Leakage Faults. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2024**, 1. [CrossRef]
12. Skordal, K.K. A Simple RISC-V Processor for Use in FPGA Designs. 2021. Available online: https://github.com/skordal/potato (accessed on 15 May 2022).
13. Owen, D.E.; Joseph, J.; Plusquellic, J.; Mannos, T.J.; Dziki, B. Node Monitoring as a Fault Detection Countermeasure against Information Leakage within a RISC-V Microprocessor. *Cryptography* **2022**, *6*, 38. [CrossRef]
14. Plusquellic, J.; Owen, D.E.; Mannos, T.J.; Dziki, B. Information Leakage Analysis using a Co-design-Based Fault Injection Technique on a RISC-V Microprocessor. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2021**, *41*, 438–451. [CrossRef]
15. Herveille, R. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. Available online: https://cdn.opencores.org/downloads/wbspec_b3.pdf (accessed on 1 June 2023).
16. Pellegrini, A.; Constantinides, K.; Zhang, D.; Sudhakar, S.; Bertacco, V.; Austin, T. CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework. In Proceedings of the 2008 IEEE International Conference on Computer Design, Lake Tahoe, CA, USA, 12–15 October 2008; pp. 363–370.
17. Advanced Encryption Standard. 2020. Available online: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard (accessed on 12 February 2022).
18. AMD, Xilinx. Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. Available online: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html (accessed on 12 May 2023).
19. Abdelatty, M.; Gaber, M.; Shalan, M. Fault: Open-Source EDA's Missing DFT Toolchain. *IEEE Des. Test* **2021**, *38*, 45–52. [CrossRef]
20. Stine, J. System-on-Chip Designs for SCMOS MOSIS AMI 0.6 um, AMI 0.35 um, TSMC 0.25 um, TSMC 0.18 um, and FreePDK 45. Available online: https://vlsiarch.ecen.okstate.edu/flows/ (accessed on 12 February 2022).
21. Wolf, C. Yosys Open SYnthesis Suite. Available online: https://yosyshq.net/yosys/ (accessed on 12 February 2022).
22. Goel, P.; Rosales, B. PODEM-X: An Automatic Test Generation System for VLSI Logic Structures. In Proceedings of the 18th Design Automation Conference, Nashville, TN, USA, 29 June–1 July 1981; pp. 260–268. [CrossRef]
23. Teske, L. RISCV Online Assembler. Available online: https://riscvasm.lucasteske.dev/# (accessed on 12 February 2022).
24. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2. 2017. Available online: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf (accessed on 12 February 2022).
25. Patil, M.; Sharanabasaveshwar, H.B. Design and Implementation of BIST. In Proceedings of the 2018 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT), Msyuru, India, 14–15 December 2018; pp. 1142–1146. [CrossRef]