IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 4, pp. 610–632. DOI:10.46586/tches.v2024.i4.610-632

FaultDetective

Explainable to a Fault, from the Design Layout to the Software

Zhenyuan Liu, Dillibabu Shanmugam and Patrick Schaumont

Worcester Polytechnic Institute, Worcester, MA 01609, USA,
{zliu12,dshanmugam,pschaumont}@wpi.edu

Abstract. Hardware faults are a known source of security vulnerabilities. Fault injection in secure embedded systems leads to information leakage and privilege escalation, and countless fault attacks have been demonstrated both in simulation and in practice. However, there is a significant gap between simulated fault attacks and physical fault attacks. Simulations use idealized fault models such as single-bit flips with uniform distribution. These ideal fault models may not hold in practice. On the other hand, practical experiments lack the white-box visibility necessary to determine the true nature of the fault, leading to probabilistic vulnerability assessments and unexplained results. In embedded software, this problem is further exacerbated by the layered abstractions between the hardware (where the fault originates) and the application software (where the fault effect is observed). We present FaultDetective, a method to investigate the root-cause of fault injection from fault detection in software. Our main insight is that fault detection in software is only the *end-point* of a chain of events that starts with a fault manifestation in hardware and propagates through the micro-architecture and architecture before reaching the software level. To understand the fault effects at the hardware level, we use a scan chain, a low-level hardware test structure. We then use white-box simulation to propagate and observe hardware faults in the embedded software. We efficiently visualize the fault propagation across abstraction levels using a hash-tree representation of the scan chain. We implement this concept in a multi-core MSP430 micro-controller that redundantly executes an application in lock-step. With this setup, we observe the fault effects for several different stressors, including clock glitching and thermal laser stimulation, and explain the root-cause in each case.

Keywords: Fault Attacks, Fault Injection, ASIC, Microcontroller, Embedded Software

1 Introduction

A substantial body of literature addresses the adverse impacts of hardware faults on the execution of secure embedded systems, spanning over a quarter-century [BDL97]. These findings, derived from simulations or empirical studies, are used to guide research on fault countermeasures and techniques for fault detection or correction. Yet, the absence of a generally agreed-upon method in fault modeling leaves designers to rely heavily on assumptions regarding the statistical distribution, severity, and nature of faults. Research in fault injection and propagation on software falls into two broad categories: either the fault injection is simulated on a model of the design under test, or it is based on practical measurements on an artifact. Unfortunately, these two categories only show weak connections, raising uncertainties about how effectively the conclusions drawn from simulation-based research can be applied to the fault models identified through measurement-based research.

Simulation-based research in fault attacks assumes a fault attacker with arbitrary capabilities who induces a bit flip or a permanent fault with chosen precision and distribution.

Received: 2024-04-15 Accepted: 2024-06-15 Published: 2024-09-05

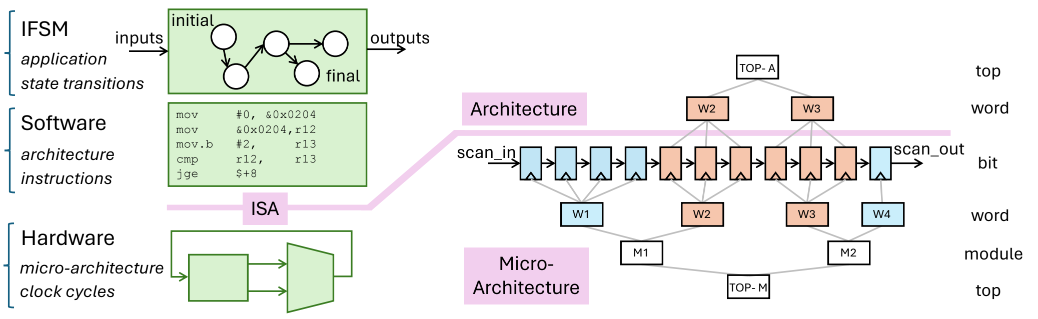
The impact of the fault is captured through random or constrained bit flipping techniques [YGS15, RSS$^+$21, AWMN20, NOV$^+$22]. At higher abstraction levels, simulated attacks revert to abstract fault models such as gate modification [RSG23], instruction-skip, or random-register corruption [TAC$^+$23]. These efforts have enhanced our understanding of the potential effects of chosen faults across both micro-architecture and architecture levels, thereby facilitating the development of tools for verifying countermeasures. However, underlying this research is an assumption on the nature and location of faults.

Empirical research in fault attacks, on the other hand, begins with the selection of a fault injection vector, which is then applied to a specific target such as a micro-controller or an FPGA. This approach provides insight into the real-world effects of a fault. Popular fault injection vectors include clock glitching, voltage glitching, electromagnetic pulses, and optical and thermal laser injection. However, due to the empirical nature of these experiments, visibility into the low-level hardware is often limited, and the precision of fault injection is limited. Thus, defining an effective fault model in this context poses a significant challenge and is sometimes the primary focus of research [VMDB20, KDD21]. Given the probabilistic nature of faults, the fault model itself also becomes probabilistic. For example, empirical results may describe a distribution of 1-bit and multi-bit faults [DBC$^+$18]. Additionally, it is a challenge to describe the fault model at higher abstraction levels, leading to various forms of instruction-skip, instruction-replication, or operand-substitution in micro-controller execution models.

Contributions. Our research investigates the gap between the fault models used in simulation and verification tools, and the probabilistic fault models encountered in practical microprocessor setups. To address this gap, we propose FaultDetective, a technique that precisely identifies fault locations and traces their propagation by integrating hardware redundancy with a scan chain. Hardware redundancy provides multiple perspectives on the effects of a fault injection on a microprocessor, ideally providing both correct and faulty versions of the same behavior. The scan chain facilitates access to the low-level system state of redundant copies. We compare these states to analyze the hardware origin of the fault, its propagation through micro-architecture, and its eventual manifestation in software and hardware. Our method utilizes layout-level design data to implement realistic fault modeling against a chosen fault vector. We demonstrate this approach using a six-fold redundant MSP430 micro-controller implemented in standard cells. Subsequently, we analyze a series of firmware applications to illustrate that our methodology can be applied to various targets, including PIN Verification and the ASCON SBOX, and to diagnose how both the software and hardware fail under fault injection across different firmware configurations.

Related Work. The impact of fault injection at design time can be anticipated by simulating the fault on a model of the implementation. An important use case is in the prediction of design reliability against random faults and single-event setup [MWE$^+$03, PG23b, PG23a]. In the security space, design-time evaluation is used to verify countermeasures in hardware [YGS15, RSS$^+$21, AWMN20, NOV$^+$22], software [HSP21], or mixed hardware/software systems [THN$^+$24, GS21, TAC$^+$22, TAC$^+$23]. Several of these efforts use formal verification techniques, which offer the advantage of exhaustive exploration of the fault response after fault injection [RSS$^+$21, THN$^+$24, GS21, TAC$^+$22, TAC$^+$23, NOV$^+$22]. However, in all of these cases, the fault model is chosen or constrained as an *input* of the simulation flow, rather than as an outcome of physical stress on the target design.

Yet, fault models are challenging, and significant efforts have been invested in describing the impact of fault injection. Indeed, fault injection causes in essence an analog/electrical effect, and its transformation into a quantifiable digital effect in hardware or software is complex. For example, electromagnetic pulses have been described as local timing faults as



```
mov     #0, &0x0204
mov     &0x0204,r12
mov.b   #2,     r13
cmp     r12,    r13
jge     $+8
```

Figure 1: Software Execution as a Finite State Machine (Left) Intended FSM, Architecture Level FSM and Micro-architecture Level FSM (Right) The micro-architecture state of the processor covers all registers in a structural hierarchy of words and modules. The ISA hides some registers to the programmer, leading to a hierarchy of words in the architecture state.

well as sampling faults in hardware [OGM15, GYG+18], resulting in instruction-skip as well as instruction-repetition in software [PHB+19, BBG+19]. Laser pulses cause signal integrity effects that result in a distribution of single-bit and multi-bit flips [VMDB20, DBC+18]. The impact of laser pulses on software has been described as causing instruction-skip, instruction-replay, while sustained laser pulses can affect hundred instructions at a time [KDD21]. Similar complex software effects have been described for clock glitching, including partial update of variables [ACD+24] as well as complex micro-architecture effects that modify the instruction semantics [SMB+21]. Generally, these complex fault effects are in sharp contrast with the simple models used in simulation-based research.

We conclude that accurate fault modeling plays a crucial role in understanding the impact of faults on designs, and designing a comprehensive fault model for hardware remains a topic of ongoing research [RSG23]. For software, a comprehensive fault model is still elusive. Because of this reason, we adopt an approach to fault modeling that involves observing a fault effect as early as possible after fault injection, achieved by capturing the scan chain immediately following the fault injection. The fault injection itself can be performed either on a physical chip or through simulation using low-level (post-layout) design data. For example, simulation of laser fault injection requires the layout and the technology node of the design, and simulation of clock glitching requires the detailed post-layout netlist with timing distribution over the design. In this paper, we focus on simulating the fault injection process with realistic physical constraints. After fault injection, we simulate the impact of the physical fault on hardware across multiple abstraction levels, including micro-architecture, architecture and application-level software.

Outline. The paper is organized as follows: the next section presents the preliminaries that underpin the proposed FaultDetective. Section 3 details the FaultDetective methodology for root-cause analysis of fault injection, as well as the chip used as the analysis target. Section 4 presents a series of examples applying the fault root-cause methodology.

2 Preliminaries

This section describes foundational concepts for root-cause analysis in fault injection, including software and hardware modeling, attacker models, and fault propagation modeling across hardware and software.

Software execution as a Finite State Machine. We capture the execution of software as a hierarchical finite state machine (FSM), shown in Figure 1. The hierarchy is the result of the implicit layering between the application, the instruction-level architecture, and the micro-architecture. The instruction-level architecture hides certain registers within the micro-architecture, exposing only the programmer-visible state. In fault root-cause analysis, these abstractions play a crucial role since a fault may reside within the micro-architecture state and cause effects that cannot be anticipated from the programmable-visible state alone. The hierarchical layers are defined as follows.

1. At the application level, software developers construct the application, which we conceptualize as the Intended Finite State Machine (IFSM); this terminology is also used by other researchers who define a software exploit as a discrepancy between the IFSM and the actual implementation [Dul20, BGX$^+$21]. The IFSM operates on application-level state elements or variables. The number of states of the IFSM, and the number of variables it uses, is much smaller than the number of possible states that can be represented by the implementation. Therefore, the number of states and variables within the IFSM is considerably smaller than the total number of states that the implementation can potentially encompass.
2. The architecture level FSM is generated by compiling the application IFSM into instructions and programmer-visible state elements within a computer architecture. These programmer-visible state elements include processor registers, memory and other programmer-visible storage components. State transitions within the architecture-level FSM correspond to the execution of instructions.
3. The micro-architecture level FSM is characterized by the application executing on microprocessor hardware. The micro-architecture level encompasses all state elements utilized in the implementation, including storage that is both invisible and inaccessible to the programmer. Examples of such invisible storage elements include pipeline registers, storage buffers, and internal instruction controls. State transitions at the micro-architecture level are managed with cycle-accurate precision.

Software execution implies a hierarchical execution of these FSMs: each state transition in the IFSM corresponds to a sequence of transitions (instructions) at the architecture level, and each state transition at architecture level maps to a sequence of transitions (clock cycles) at micro-architecture level. Additionally, the state elements defined in the IFSM constitute a subset of the state elements of the architecture level FSM, which in turn comprise a subset of the state elements within the micro-architecture FSM. This hierarchy and the connection between levels through state elements and state transitions, is fundamental to understand the behavior of embedded software under fault injection. For example, a hardware fault injection such as a clock glitch affects the hardware and consequently alters the state of the micro-architecture level FSM. We refer to such a faulted micro-architecture state as a *weird state*. Weird states are distinct from sane states reached through normal software execution.

Attacker Model. In this paper, we consider an attacker who can maliciously change the micro-architecture state through non-invasive or semi-invasive fault injection, for example clock glitching or laser fault injection. We assume that faults are transient and disappear after processor reset. Our efforts focus on analyzing faults in the registers (flip-flops) of the micro-architecture that last more than one clock cycle, and we recognize that RAM storage can be protected through error correcting codes (ECC) [SRVL$^+$20]. We consider memory-corruption attacks [PV16], caused by feeding malformed inputs into buggy application software, to be out of scope. The objective of our work is not to detect or intercept a fault attack, but rather to pinpoint the root-cause of a simulated fault attack by simulating the fault injection on the design data, and analyzing exactly how the software fails the programmer's intent. Our analysis focuses on understanding how these faults manifest and propagate through hardware and software, providing deeper insights into their origin and impact from injection to detection and analyzing exactly how the software fails the programmer's intent.

Sane and Weird States. The concept of using state machines to capture security exploits, including fault attacks, is due to Dullien [Dul20]. Dullien defines the IFSM as the program executing on a processor according to the programmer's intent, and introduces the notion of a weird FSM to describe the behavior that occurs when the intended FSM enters an otherwise unreachable state. We found the concept of a weird FSM, coupled with the model of software execution as a hierarchical FSM, to be highly effective in investigating scenarios where, as Dullien describes, the IFSM 'goes of the rails'.

We define a system state as a concrete value of the micro-architecture state. In the broadest sense, this includes every flip-flop and memory bit in the system, including peripherals and the micro-processor itself. In the following, we restrict the discussion of system state to those state elements inside of the micro-processor. A *sane state* refers to any system state that is achieved when the implementation of the IFSM's implementation operates in accordance with the programmer's intent. Under normal conditions, in the absence of a fault attack, the processor's execution exclusively reaches these sane states. The *sane machine* is the collection of all possible sane states. The number of sane states within an IFSM can potentially be very large, particularly if the IFSM uses numerous variables with large domains. However, we don't have to visualize the entire sane machine. Instead, the purpose of the sane machine is to characterize the dynamic behavior of the application through a series of state transitions, and to identify instances where a transition occurs outside the boundaries of the sane state space. Implementing the IFSM as a program on a concrete micro-processor significantly increases the number of states in the implementation. This is due to the inclusion of a large number of state elements within the micro-processor, which collectively can accommodate many more states than those required by the IFSM. The result of a successful fault injection is the alteration of a state element against the programmer's intent. Depending on the method of the fault injection, any state element within the processor may be affected, causing the processor to transition from a sane state to a *weird state*. Occasionally, this weird state may coincide with another sane state, for example if specific fault-correction hardware on state elements is present [AHS09, NFM+19]. However, in general, a weird state does not occur within the boundaries of the sane machine.

The Meaning of the Weird Machine. Unlike the sane machine, which adheres to the semantics defined by the instruction-set, the behavior of the weird machine is largely unknown and unpredictable. Its execution semantics are defined by the low-level hardware implementation, and may express a much more varied response compared to the behavior observed in the sane machine. Following a fault injection that affects the micro-architecture, a software instruction may loose its intended meaning; an effect described by Dullien as the *emergent instruction set*. It is this uncertainty about the semantics of a processor under fault injection, that is responsible for the difficulty in understanding or predicting the fault response of embedded software. Dullien uses the weird machine model to create formal proofs of non-exploitability of the IFSM, under the assumption of an attacker who is able to corrupt the memory. He demonstrates a formal method to show that such an attacker cannot alter the outcome of the IFSM's execution. However, he does not attempt to visualize or explicitly uncover the concrete behavior of the weird machine at the micro-architecture or architecture levels. Instead, Dullien only concludes that it is difficult (but not impossible) to make statements about the non-existence of programs in a given machine language with only empirically accessible semantics [Dul20]. However, the weird machine *can* indeed be modeled when the weird state is fully specified, i.e. under

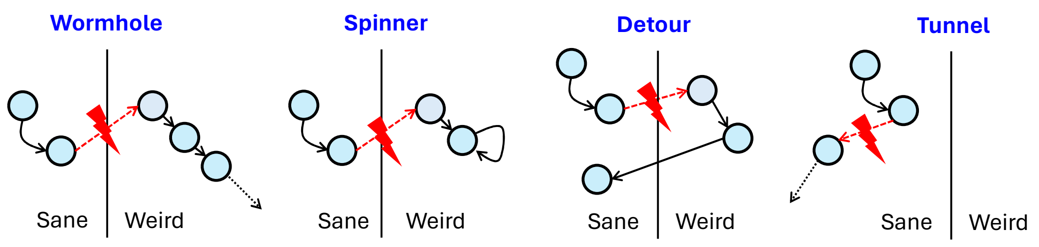


Figure 2: Four fault response patterns of an FSM after fault injection.

any concrete fault scenario. Also, the unknown state space of the weird machine does not imply that the processor operates non-deterministically. The underlying hardware remains identical whether operating under the weird machine or the sane machine. By fully modeling the hardware at the gate-level, it becomes feasible to analyze how the micro-architecture will respond to faults. A primary objective of FaultDetective is to support this analysis by dynamically tracking the state of the faulty processor.

Fault Patterns in the Weird Machine. To better understand the interaction between the sane machine and the weird machine, we have classified the fault response as a result of fault injection into four possible fault response patterns, illustrated in Figure 2. Each pattern is named to reflect its unique characteristics. A *wormhole* fault transitions from a sane state to a weird state and seemingly never returns from the weird state space, even though execution seems to continue. A *spinner* fault creates a weird state that loops back on itself, resembling the appearance of locking up the processor. A *detour* fault creates a number of weird state transitions but eventually returns to a sane state, giving the appearance of lost processor cycles. Finally, a *tunnel* fault jumps from a sane state to a different sane state, seemingly without an apparent valid execution path connecting both states. These classifications help clarify how fault injections manifest and affect processor behavior. The fault patterns in Figure 2 do not indicate whether a fault attack would result in a success or a failure. Determining the success of a fault attack requires verifying if a specific attacker-desirable state is reached, typically assessed through Boolean predicates applied to the state elements [THN$^+$24]. Instead, the focus of our work is on understanding the origin and propagation of hardware faults, and this fits well within the sane/weird machine model.

Scan Chain. To observe the state space of the processor, whether in a weird or sane state, we utilize a scan chain, which is a test structure that is commonly used for Integrated Circuit (IC) testing. The scan chain allows every register in a design to be configured as one element of a long shift register, enabling serial access to each flip-flop in the design. Scan chain synthesis is well supported in an Application Specific Integrated Circuit (ASIC) design flow as part of a design-for-test flow [WH10, Chapter 15]. Synthesis tools can automatically insert a scan chain by utilizing scan flip-flop cells available from standard cell libraries. The scan insertion algorithms also deal with special cases, such as handling multiple clock signals and disabling asynchronous reset during scan. The scan chain proves invaluable for fault root-cause analysis, providing comprehensive visibility into the low-level state of a design. During a scan operation, the processor halts its execution, temporarily suspending software operation. The intrusive nature of a scan chains turns it into a potential security liability, and additional test-access-controls may be required when a scan-enabled chip is deployed in the field [MBCB05]. However, the design of this access control is out-of-scope of the current work.

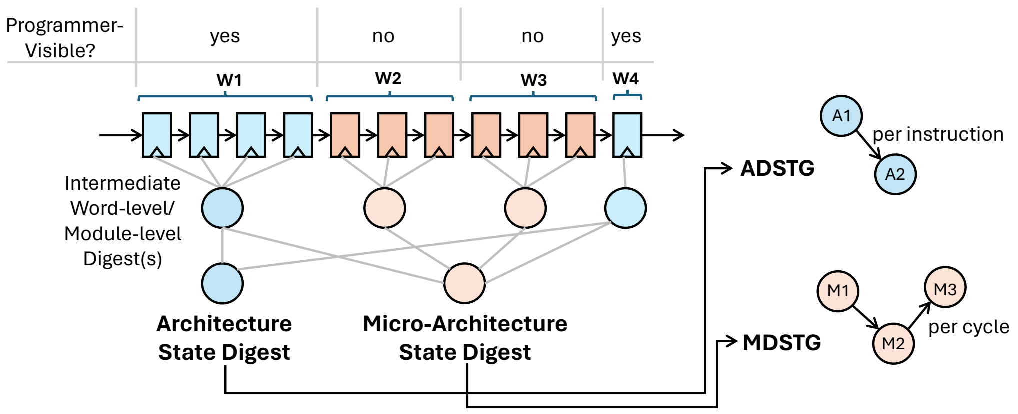

Programmer-Visible?
yes
no
no
yes
W1
W2
W3
W4
Intermediate Word-level/ Module-level Digest(s)
Architecture State Digest
Micro-Architecture State Digest
ADSTG
MDSTG
A1
A2
per instruction
M1
M2
M3
per cycle

Figure 3: Construction of the Architecture and Micro-architecture Dynamic State Transition Graph (ADSTG and MDSTG).

From Scan Chain to System State. We utilize the scan chain to visualize the state transitions within both sane and weird machine. Figure 3 illustrates the concept. The scan chain is transformed into a unique digest through a series of steps. We first compute a simple digest (`djb2`)[1] over the state bits per word and then per module, generating a tree structure of digests up to a processor-level digest. Subsequently, an architecture-level digest is computed using only architecture-visible scan bits, while a micro-architecture level digest incorporates all scan bits. The processor-level digest serves as a compact representation of the processor state, and it is sensitive to any single bit flip in the scan chain. Moreover, a tree data structure facilitates comparison among multiple scan chains (for example, resulting from redundant executions) to pinpoint which module or word or scan bit is responsible for differences in the digest, thereby pinpointing the origin or outcome of a fault. From a single scan chain state, we can compute additional system states through logic simulation, assuming that processor inputs are replayed. This process leads to the creation of a *Dynamic State Transition Graph*, which provides a compact representation of both sane and weird processor behaviors. The ADSTG is constructed using architecture-level bits, and records one transition per instruction. The MDSTG is created using micro-architecture level bits, and captures one transition per clock cycle.

3 Root-cause Analysis for Faults

In this section, we first describe FaultDetective, a technique that combines a scan chain and redundant execution to determine the root-cause of a fault. We then demonstrate the application of FaultDetective on an ASIC, employing a design-layout aware simulation to perform root-cause analysis.

3.1 FaultDetective

FaultDetective relies on a redundant implementation of a micro-controller design that executes the cores in lock-step, as shown in Figure 4 (left). Each core operates as a completely independent unit with its own memories, processor, and peripherals. To detect faults in software, an on-chip network connects all cores, shown as blue connections in Figure 4 (left). The on-chip network allows the cores to exchange checksum values. The checksum computation is implemented in the software application running on the cores,

[1]Give an n-byte string `c = {c[1],..,c[n]}`, `djb2(c) = h[n]` computed iteratively as `h[i] = int(h[i-1]*33 + c[i])` and `h[0] = 5381`

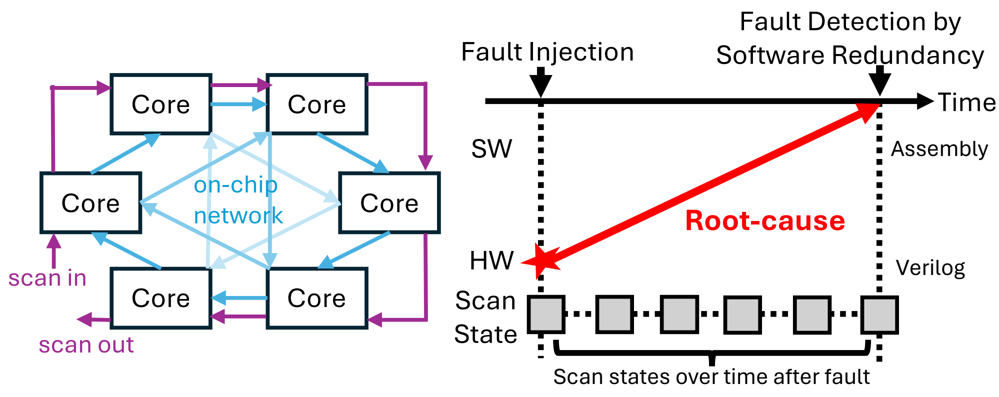


Figure 4: (Left) Scan chain in a multi-core system. (Right) Root-cause analysis for faults.

making it fully user-defined. Fault detection is fully distributed across all cores, and any core can trigger a redundantly implemented system-wide exception. We then scan the entire ASIC state through a scan chain, which enables a detailed investigation of a post-detection system state. This process allows us to compare the redundant scan chains and precisely pinpoint the location of errors. However, FaultDetective is not limited to post-detection fault analysis. By capturing the scan chain immediately after fault injection, as shown in Figure 4 (right), we can capture the early onset of faults in hardware before they are able to affect the software. Such post-injection fault analysis helps us to understand how the fault propagates from hardware to software, addressing the gap that currently exists between simulated fault models and empirical fault models. The fault root-cause is the earliest observable fault in the scan chain state after fault injection. The fault trajectory is the path from this initial fault in hardware to its eventual detection in software. Understanding this trajectory, in conjunction with knowledge of the hardware design (Verilog) and the software application, will ultimately help to explain how a fault propagates from the design layout to the application software.

A challenge with fault detection by means of redundancy is that there is no guarantee of a strict majority of correct executions among the redundant copies. If such a majority does not exist, the faulty bits in a scan chain cannot be identified. Therefore, FaultDetective uses one of two different fault detection principles: *a priori* fault detection, and *a posteriori* fault detection. A priori fault detection implies that the ground truth of correct behavior is known, while a posteriori fault detection relies solely on redundant execution and majority voting. In practice, we found both a priori fault detection and a posteriori fault detection useful. With highly localized fault injection vectors, such as lasers, it's easy to guarantee that there are sufficient redundant correct copies remaining to establish the correct scan chain state after fault injection. However, with global fault injection vectors, such as glitching (affecting a global clock network or power distribution network), it is harder to guarantee majority correctness. In such a case, we rely on a golden correct simulation and a priori fault detection, which involves comparing the scan chain state to the golden state.

3.2 Realization in ASIC

To facilitate a design-layout aware fault root-cause diagnosis, we developed an ASIC using 180nm standard cell technology to test the proposed idea of FaultDetective. The chip includes six identical MSP430 micro-controllers based on OpenMSP430 [Gir17], each with a 4KB of program memory, a 4KB of data memory, and peripherals including timer, UART, and GPIO. All cores are connected in a scan chain and each core contributes

Table 1: Scan Chain Composition of One Core in the Test Chip.

	Architecture	Micro-architecture	Total
openmsp430_0	**385**	**275**	**660**
watchdog_0	6	18	24
sfr_0	4	0	4
multiplier_0	66	4	70
mem_backbone_0	0	38	38
frontend_0	16	100	116
execution_unit_0	231	35	266
dbg_0	57	64	121
clock_module_0	5	16	21
peripherals	**712**	**330**	**1,042**
Total	**1,097**	**605**	**1,702**

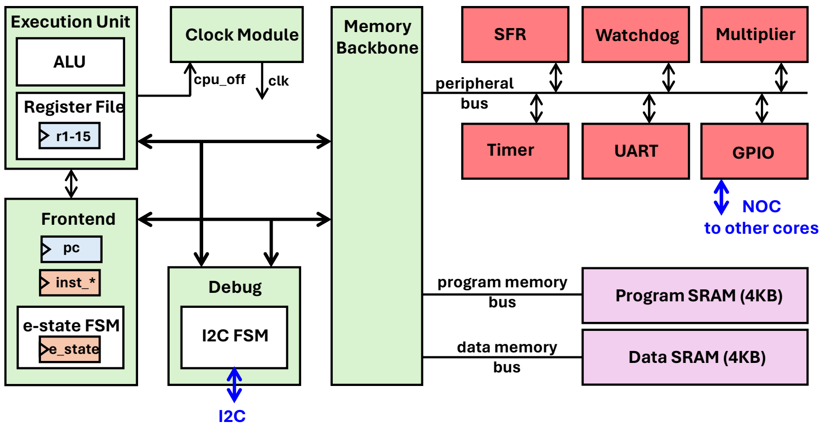


Figure 5: Micro-architecture of one core MSP-430 Micro-architecture.

1,702 bits of state as detailed in Table 1. In total, the scan chain across the entire chip consists of 10,212 bits. The ASIC is programmable through an I2C debug interface, and each core can be individually programmed and controlled. The I2C debug interface also supports global commands to control all cores jointly, enabling lock-step execution of identical programs. In this ASIC, lock-step redundant execution ensures that all cores compute the same instruction at the same clock cycle. Program- and Data-memory are replicated across each core so that the redundant fault detection can also cover the memory areas. Conducting experiments on a *six-core* ASIC chip had no specific reason beyond layout-symmetry considerations and budget constraints for the tape-out. Any multi-core setup that allows for *a posteriori* fault detection with majority voting would function similarly. The communication network on the ASIC is implemented by interconnecting a network of GPIO ports. Figure 4 (left) illustrates the network topology. The network is used to exchange chosen checksums at regular intervals during the execution of a test program. The system-wide exception, which is necessary to halt the system after fault detection, is redundantly implemented in hardware as a non-maskable interrupt to every core. Each MSP430 is a 16-bit micro-controller that executes instructions in one to six clock cycles, depending on the complexity of the instruction operands. Instructions range from one to three 16-bit words in length. Figure 5 demonstrates the micro-architecture of each core. Instructions are fetched and decoded by a frontend unit, which drives a datapath in the execution unit. Memory access is handled by the memory backbone unit, which multiplexes the MSP430 address space in separate regions for peripheral, program and data access.

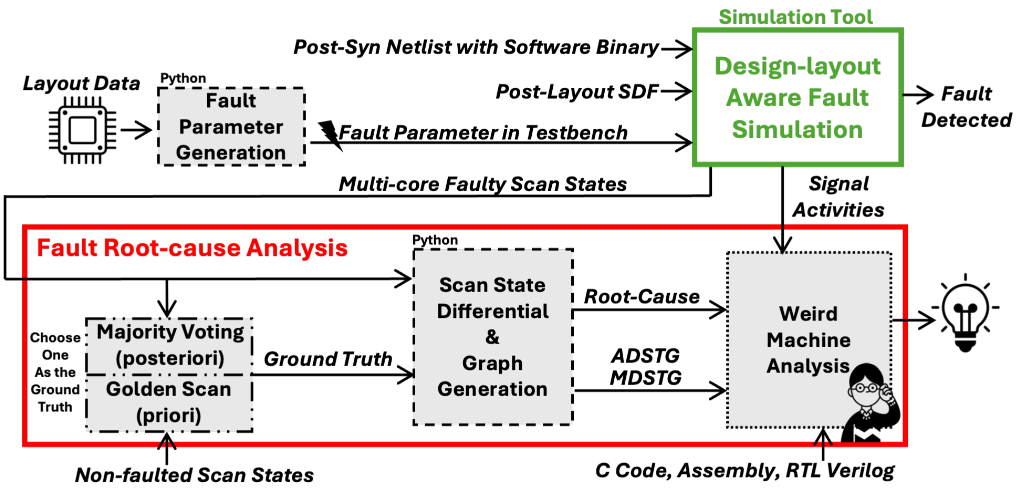

Simulation Tool
Design-layout Aware Fault Simulation
Post-Syn Netlist with Software Binary
Post-Layout SDF
Fault Parameter in Testbench
Layout Data
Python
Fault Parameter Generation
Fault Detected
Multi-core Faulty Scan States
Signal Activities
Fault Root-cause Analysis
Python
Scan State Differential & Graph Generation
Root-Cause
ADSTG MDSTG
Weird Machine Analysis
Choose One As the Ground Truth
Majority Voting (posteriori)
Golden Scan (priori)
Ground Truth
Non-faulted Scan States
C Code, Assembly, RTL Verilog

Figure 6: The design-layout aware simulation framework of FaultDetective.

3.3 A Design-layout Aware Simulation Framework

We present a design-layout aware simulation framework to demonstrate the root-cause technique in FaultDetective on the testing ASIC. The framework supports design-layout aware fault simulation and fault root-cause analysis, as outlined in Figure 6.

3.3.1 Step 1: Design-layout Aware Fault Simulation

Regardless of the type of fault injection, FaultDetective analyzes the root-cause of faults by examining scan chain differences between redundant copies. Figure 6 demonstrates an implementation of FaultDetective as a gate-level simulation. In this approach, a fault is injected directly into the gate-level netlist by manipulating the simulation testbench according to the desired fault-injection parameters. The application software (IFSM) is compiled and configured into the program/data memory, and then run in a gate-level simulation using the post-synthesis netlist with post-layout delay timing information (SDF). After injecting a fault, we capture the scan state of each core once per clock cycle for subsequent fault diagnosis. The testbench completes either when the application software (IFSM) detects a fault and halts the processor, or when the simulation continues to run until the application software finishes without detecting a fault, at which point it reaches a timeout. To expedite exploration of the fault space, we have automated the process of generating fault parameters in testbenches to create a large number of simulations across various unique fault vectors. Each simulation involves a single fault injection, residing in a single testbench, which can either be laser or clock injection as detailed below. Each simulation runs as a separate testbench with its own distinct fault injection. We now describe how we achieved implementation-faithful fault simulation for clock glitch injection and laser fault injection.

Fault Parameter Generation on Clock Glitch. Clock glitching is a common technique to inject timing faults into a circuit by temporarily shortening the clock period in a chosen clock cycle. The margin between the clock period and the minimum required time for a flip-flop to capture stable data is known as the *slack* of the signal path that ends at the flip-flop data input. Slack depends on the logic complexity and the interconnect between the output of one flip-flop and the input of another, the target flip-flop. When the slack turns negative, the target flip-flop may experience a timing fault, typically causing it to

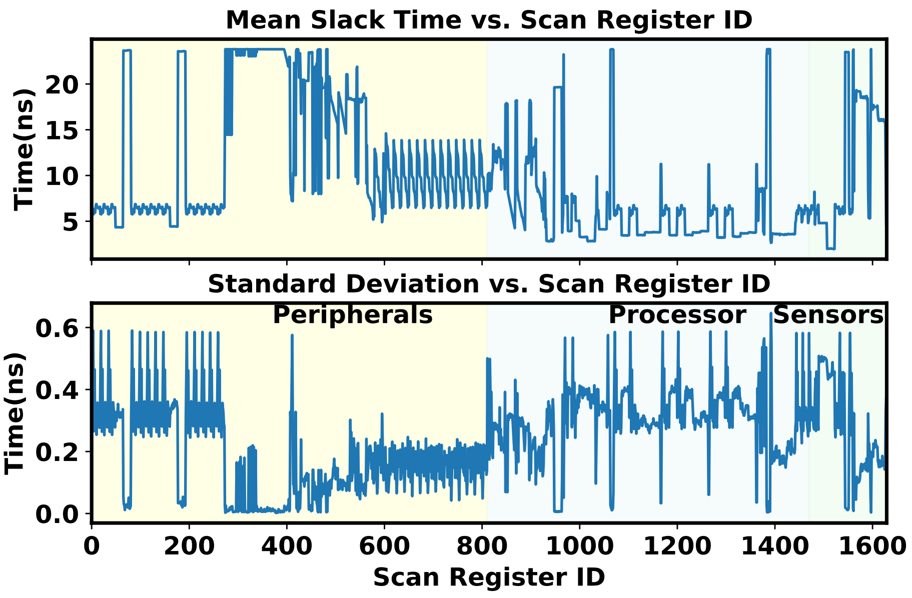


Figure 7: Distribution of slack time over the scan chain flip-flops, excluding flip-flops used as system inputs.

either retain the previous clock cycle's value or else to become metastable (resulting in unpredictable behavior with a new value). Slack time varies across the flip-flops of a large digital circuit. Figure 7 (top) illustrates the distribution of the slack among the flip-flops of an 'average' MSP430 in the ASIC, including registers from peripherals, processor, and sensor modules, labeling with different background color. By 'average', we refer to the mean slack of corresponding flip-flops across each MSP430 core. The figure demonstrates significant variation (over 20ns) across the entire register set. This variation implies that with a given glitch width, only a subset of flip-flops will be affected. However, we observe that this effect is not identical across all cores. The slack per register shows variations from core to core due to layout-level variations of standard-cell placement and routing. Figure 7 (bottom) shows the standard deviation of slack time. Especially in the processor region, we observe variations of 0.4ns or more, from one core to the next, across the set of registers. This implies that different cores may experience different timing faults even under identical glitching parameters. Clock glitch injection is simulated using timing simulation. Our simulation scripts generates a glitch sweep, applying a single clock glitch over every clock cycle of a software application. This sweep results in a large number of fault simulations, each involving a single clock glitch.

Fault Parameter Generation on Laser Injection. Laser pulse injection is a highly localized fault injection mechanism, which can either set or reset a flip-flop. The diameter of the laser spot and the number of affected transistors are correlated [DBC+18]. Typically, the laser spot diameter in a laser injection setup ranges from 1 micron to 20 micron (μm), depending on the optical path configured. We assume a 15 μm spot size which can easily cover several flip-flops. In Figure 8 (left), a small section of the test layout shows the scan flip-flop cells marked using black outlines. The simulation of a laser pulse involves flipping the content of the flip-flop struck by the laser spot.

To simulate a laser fault injection campaign, we utilize the layout distribution of the registers in the testing ASIC, as shown in Figure 8 (right). In this figure, each register is represented as a grey dot. Due to the large amount of flip-flops, each core appears as

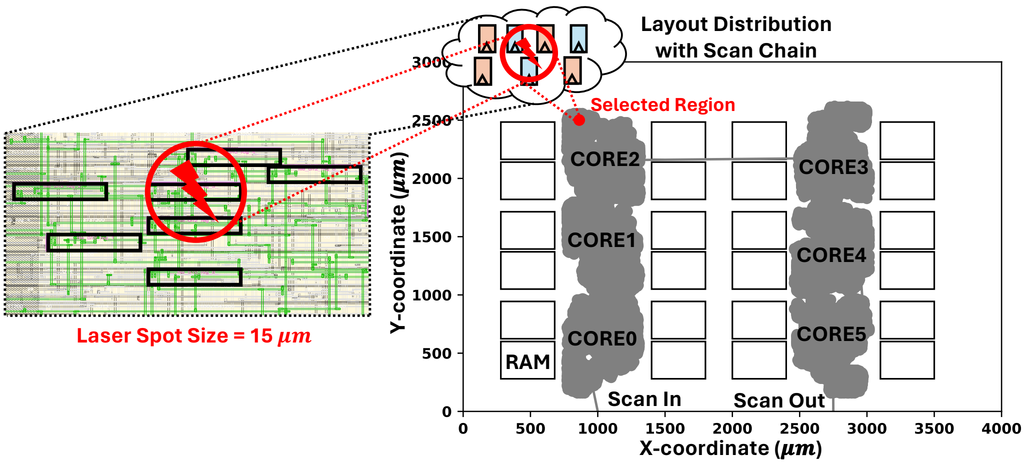


Figure 8: Fault parameter generation of laser injection. (Left) A 15 μm diameter laser pulse covers multiple 180nm standard cell flip-flops. (Right) Chip topology with flip-flop cell locations marked grey.

a grey blob. The laser fault simulation now proceeds as follows. Initially, we define a circular region (colored in red on the layout), and set its diameter as a configurable input. We then sweep this region over the entire layout, generating a new testbench for each location. In the testing ASIC, we identified 2606 different locations that would impact from one to three registers with a 15 μm laser spot. In total, we have 2060 testbenches for laser fault simulation, each representing a unique laser spot selection within its own simulation testbench. A single laser fault injection testbench proceeds as follows: we randomly select a clock cycle during the execution of the software application, and flip the registers at the selected location and the chosen clock cycle. By selecting a random clock cycle to inject laser faults, we aim to explain the root-cause for any faults we observe using FaultDetective instead of exhaustive simulation of all possible faults. In comparison with the glitch fault injection, the laser fault injection simulation is thus highly targeted.

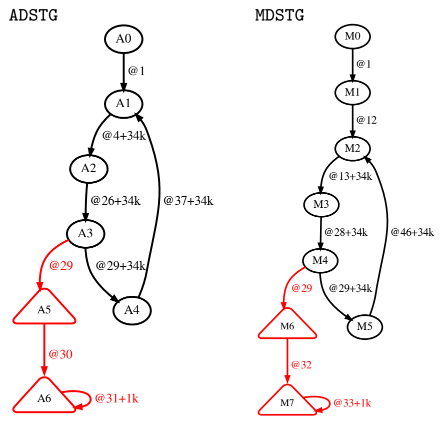
3.3.2 Step 2: Fault Root-cause Analysis

Using the scan chain state, we aim to explain the effects of faults as the fault propagates from hardware through the micro-architecture states to the software architecture states. To support this process, we have developed additional tools, as illustrated in the bottom half of Figure 6. Initially, by choosing either a priori or a posteriori fault detection, we compute the scan chain differential across redundant copies to pinpoint fault locations per clock cycle. Subsequently, for each core, we generate the Architecture and Micro-architecture Dynamic State Transition Graph (ADSTG and MDSTG). The subsequent weird machine analysis involves using the ADSTG/MDSTG, fault root-cause identification, as well as inspection of the C code, Assembly, RTL verilog, and signal activities (VCD). This is not an automated process and requires manual inspection of the data by an application designer. Automated analysis of the fault root-cause would likely require advanced pattern-matching, and is the scope of future work. While the ADSTG and MDSTG are processor-specific, they are created from scan chain data and therefore applicable to other processor architectures as long as the complete processor state can be recorded.

The following example illustrates the fault diagnosis process and the notation used throughout the paper. Figure 9 (left) shows an application running on the six cores in lock-step. The cores are configured in a ring network using GPIO ports, such that core ($i \mod 6$)'s `P2OUT` port is connected to core ($(i+1) \mod 6$)'s `P2IN` port. Each core

```
IMPLEMENTATION
while (1) {
  P1OUT = 0x80;
  switch (state) {
  case 0:
    P2OUT = 0x0;
    if (P2IN == 0x1) state = 1;
    break;
  case 1:
    P2OUT = 0x1;
    if (P2IN == 0x0) state = 0;
    break;
  }
  P1OUT = 0x00;

  if (P2IN != P2OUT) {  -> A5
      P1OUT = 0x05;
  }
}
```



```
FAULT ROOT CAUSE
CORE2:
openmsp430_0
-execution_unit_0
  register_file_0
   r2                13
```

Figure 9: Root-cause analysis of a *spinner* fault. (Left) Implementation. (Middle) The ADSTG and MDSTG. (Right) Fault Root-cause captured on the scan chain.

starts with a local `state` variable, initially at 0. A core switches its state only when the input changes, and it will detect a fault (`P1OUT=0x5`) when the input is different from the expected output. The middle section of Figure 9 shows the ADSTG and MDSTG. The black nodes and edges represent the non-faulted execution path. Each node corresponds to a unique architecture or micro-architecture digest (state). Cycle counts associated with these transitions are labeled on the edges. In cases where edges are traversed multiple times, the differences in cycle counts between transitions are also indicated. To simplify the graph notation, we apply a clustering technique where a linear sequence of nodes and edges is condensed into a start node, an end node, and an edge labeled with the transition from the start node to the end node. The ADSTG and MDSTG in Figure 9 demonstrate that the regular operation consists of a repetitive sequence of instructions (`A1, A2, A3, A4`) with a round-trip time of 34 cycles.

When a fault is injected, one core will exhibit scan states that are different from the other cores (or from the golden copy), and the faulty core enters a weird state marked in red. Edges that originate from or terminate in a weird state are also colored red. In this example, we injected a laser fault into an MSP430's status register `r2`. Figure 9 (right) shows the fault root-cause, which corresponds to the scan chain differential immediately after fault injection. The fault altered `r2` from `03` into `13`, affecting the `CPUOFF` bit of the status register. This bit disables further instruction-fetch, effectively halting the processor. The ADSTG and MDSTG illustrate that the fault is injected in cycle 29 (cycles are counted from the software instruction `P1OUT=0x80`), while the core was evaluating the value of `P2OUT`. After completing the current instruction, the core halts in state `A6` (ADSTG) and state `M7` (MDSTG). This results in the processor entering a *spinner* fault pattern.

4 Results and Analysis

This section illustrates the operation of FaultDetective through a collection of examples. Each example details the fault effects across multiple abstraction levels, following a similar approach as outlined in Section 3.3.2. Our objective is to explain the root-cause of faults and their propagation through both hardware and software across various scenarios.

4.1 Target Firmware

Table 2 shows the six test cases. We have developed two firmware targets: Simple Two States and Redundant Simple Two States, with an IFSM as illustrated in Figure

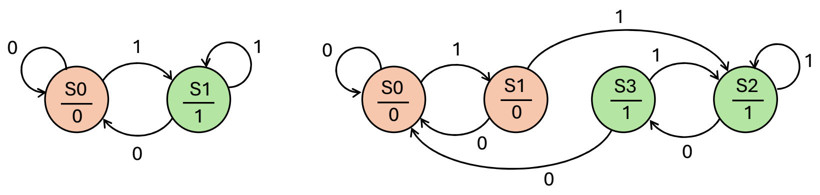


Figure 10: Application level FSM of (Left) Simple Two States. (Right) Redundant Simple Two States.

Table 2: Test Cases Discussed in Section 4.

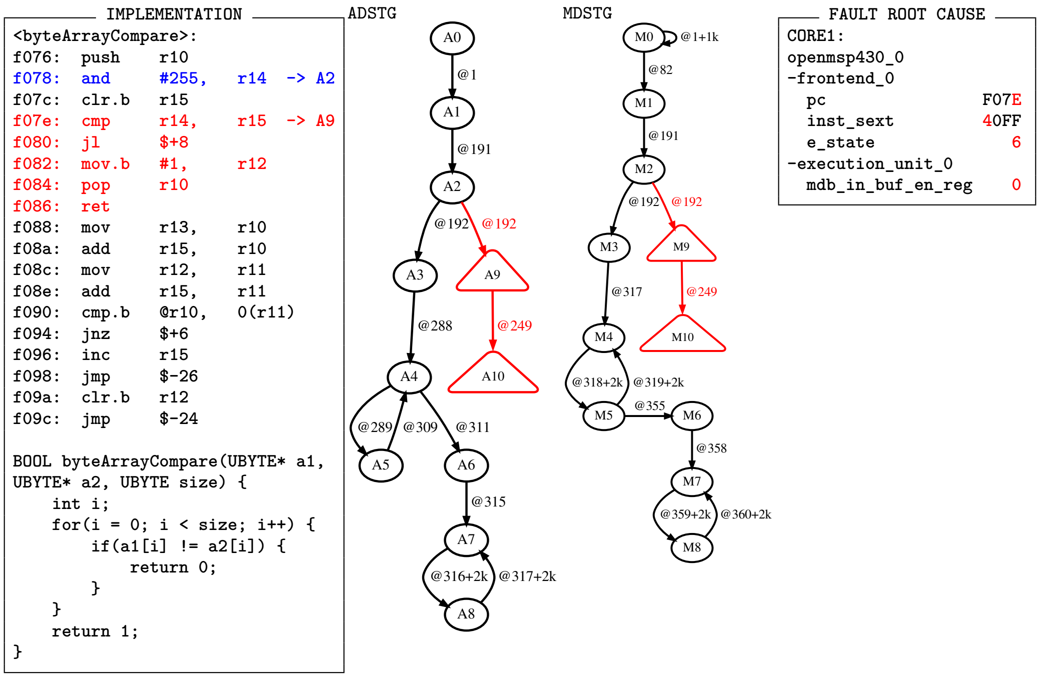
Test Case	Firmware	Source	Fault Type	Demonstration
1	verifyPIN0	[DPP+16]	clock	Wormhole pattern
2	Redundant Simple Two States	self	laser	Detour pattern
3	ASCON SBOX	[Sto16]	laser	Tunnel pattern
4	Redundant Simple Two States	self	clock	Countermeasure faults
5	verifyPIN5	[DPP+16]	clock	Countermeasure faults
6	ASCON SBOX	[Sto16]	clock	Faulty output

10. Simple Two States corresponds to the software demonstrated in Figure 9, while Redundant Simple Two States is a redundant version of this program that transitions to a `1`-output only after two consecutive `1`-inputs. We also made use of the FISSC toolbox [DPP+16], which offers a range of PIN verification routines with different countermeasures. Specifically, we selected `verifyPIN0`, an unprotected version, and `verifyPIN5`, a protected version featuring hardened Booleans, a fixed time loop, a step counter, and double-testing. Additionally, we employed a bit-sliced ASCON SBOX, optimized based on Stoffelen's SBOX [Sto16]. The first three cases illustrate specific fault patterns in the ADSTG and MDSTG. The next three cases demonstrate FaultDetective's application in the context of cryptographic engineering problems.

4.2 Illustrating Fault Response Patterns

In this section, we demonstrate fault response patterns that illustrate the transition from the sane state space to the weird state space, as shown in Figure 2.

A *Wormhole* Fault Pattern. Figure 11 demonstrates a fault injected during VerifyPIN0. A clock glitch occurs at cycle 192, right after the program enters `byteArrayCompare`, which compares the secret PIN with the trial PIN. The fault root-cause, identified by capturing the scan chain immediately after the fault injection, demonstrates that four different registers are affected. Among them, one is the program counter, which causes the program counter to increment both at the glitch as well as the subsequent clock cycle. Additionally, we can observe that the execution state (`e_state`) is faulted. This state variable contains the instruction sequencer, and the fault stops execution of `clr.b r15`. This prevents the loop counter in `r15` from being cleared, causing `r15` to retain its previous value, which happens to be larger than the loop bound stored in `r14`. Therefore, the comparison instruction at `f07e` concludes that the loop iteration bound has been exceeded, causing the `byteArrayCompare` procedure to return `1`, executing instructions from `f082` to `f086`. In the ADSTG and MDSTG, the program thus terminates in a weird state where the verifyPIN0 program is disrupted. The correct execution of the program ends in the fault-detection ADSTG state `A7 - A8`, after detecting that one core's `byteArrayCompare` returns true instead of false. The faulty copy ends in ADSTG state `A10`, successfully completing verifyPIN0 and exhibiting a *wormhole* pattern.



IMPLEMENTATION

```
<byteArrayCompare>:
f076:  push    r10
f078:  and     #255,    r14  -> A2
f07c:  clr.b   r15
f07e:  cmp     r14,     r15  -> A9
f080:  jl      $+8
f082:  mov.b   #1,      r12
f084:  pop     r10
f086:  ret
f088:  mov     r13,     r10
f08a:  add     r15,     r10
f08c:  mov     r12,     r11
f08e:  add     r15,     r11
f090:  cmp.b   @r10,    0(r11)
f094:  jnz     $+6
f096:  inc     r15
f098:  jmp     $-26
f09a:  clr.b   r12
f09c:  jmp     $-24

BOOL byteArrayCompare(UBYTE* a1,
UBYTE* a2, UBYTE size) {
    int i;
    for(i = 0; i < size; i++) {
        if(a1[i] != a2[i]) {
            return 0;
        }
    }
    return 1;
}
```

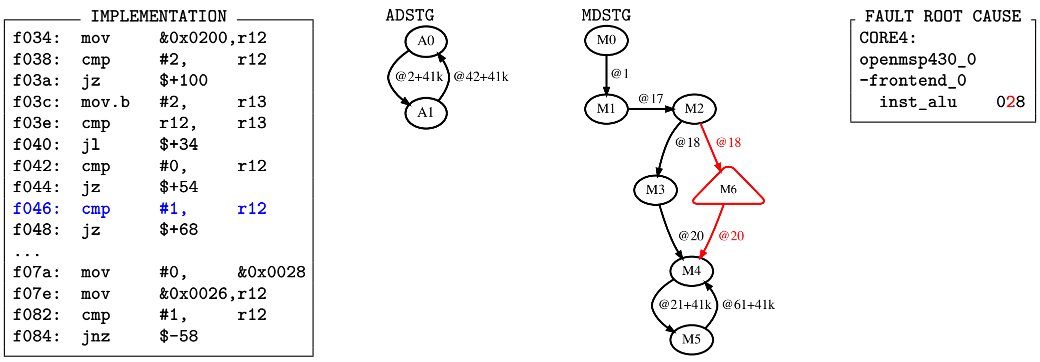
FAULT ROOT CAUSE

```
CORE1:
openmsp430_0
-frontend_0
  pc                  F07E
  inst_sext           40FF
  e_state                6
-execution_unit_0
  mdb_in_buf_en_reg      0
```

Figure 11: Root-cause of a *wormhole* fault on VerifyPIN0 breaking the PIN. (Left) Implementation. (Middle) The ADSTG and MDSTG. (Right) Fault Root-cause captured on the scan chain.

A *Detour* Fault Pattern. The second case uses laser fault injection on the Redundant Simple Two States program. We select a location in core 4, at XY location (2491, 1417) (ref Figure 8). Figure 12 illustrates the fault response of the program on core 4. The testbench injects a laser pulse 18 clock cycles after the start of the Redundant Simple Two States loop, corresponding to the execution of `jz $+54` and the decoding of `cmp #1, r12`. The laser fault affects a single micro-architecture register, `inst_alu`, causing a micro-architecture fault and a weird state `M6`. Since no architecture state was modified, the ADSTG shows no change. The fault in `inst_alu` impacts the logical operation of the ALU, and further evaluation of the Verilog code shows that this fault alters the ALU bitwise logic from AND to a combination of XOR and AND. However, during that specific clock cycle, the bitwise logic is not utilized. The instruction `jz $+54` computes a branch target using a separate adder that is independent of `inst_alu`. As a result, the branch is correctly taken, and the fault in `inst_alu` is overwritten by the subsequent instruction, resulting in a detour back to the sane state.

A *Tunnel* Fault Pattern. The tunnel fault pattern is more sophisticated than previous ones because it does not create faulty (weird) state. Instead, the injected fault is able to cause a time shift in the edges of the sane state machine. We illustrate on a laser fault injection on the ASCON SBOX application. We select a location in core 3, at XY location (2728,2431) (ref Figure 8). Figure 13 demonstrates the response of the ASCON SBOX computation to the laser fault injection. The testbench injects a single fault at 142 clock cycles after the start of the ASCON SBOX evaluation, during the execution of the instruction `mov r12, &0x0028` at `f0b4`. Both this instruction and the preceding `and #511, r12` are double-word instructions. The fault is injected when the program counter is at `fb06`, in the middle of the computation of `mov r12, &0x0028`. The effect of the fault, as shown in Figure 13 (right), alters the program counter from `fb06` to `fb02`,

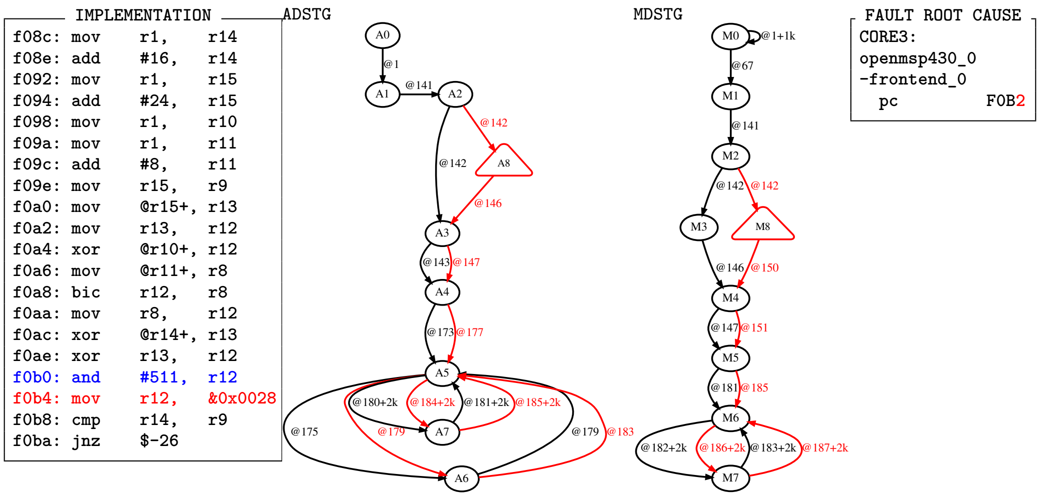

IMPLEMENTATION

```
f034:  mov      &0x0200,r12
f038:  cmp      #2,      r12
f03a:  jz       $+100
f03c:  mov.b    #2,      r13
f03e:  cmp      r12,     r13
f040:  jl       $+34
f042:  cmp      #0,      r12
f044:  jz       $+54
f046:  cmp      #1,      r12
f048:  jz       $+68
...
f07a:  mov      #0,      &0x0028
f07e:  mov      &0x0026,r12
f082:  cmp      #1,      r12
f084:  jnz      $-58
```

FAULT ROOT CAUSE

```
CORE4:
openmsp430_0
-frontend_0
  inst_alu     028
```

Figure 12: Root-cause of a *detour* fault on Redundant Simple Two States. (Left) Implementation. (Middle) The ADSTG and MDSTG. (Right) Fault Root-cause captured on the scan chain.


IMPLEMENTATION

```
f08c: mov     r1,     r14
f08e: add     #16,    r14
f092: mov     r1,     r15
f094: add     #24,    r15
f098: mov     r1,     r10
f09a: mov     r1,     r11
f09c: add     #8,     r11
f09e: mov     r15,    r9
f0a0: mov     @r15+,  r13
f0a2: mov     r13,    r12
f0a4: xor     @r10+,  r12
f0a6: mov     @r11+,  r8
f0a8: bic     r12,    r8
f0aa: mov     r8,     r12
f0ac: xor     @r14+,  r13
f0ae: xor     r13,    r12
f0b0: and     #511,   r12
f0b4: mov     r12,    &0x0028
f0b8: cmp     r14,    r9
f0ba: jnz     $-26
```

FAULT ROOT CAUSE

```
CORE3:
openmsp430_0
-frontend_0
  pc           F0B2
```

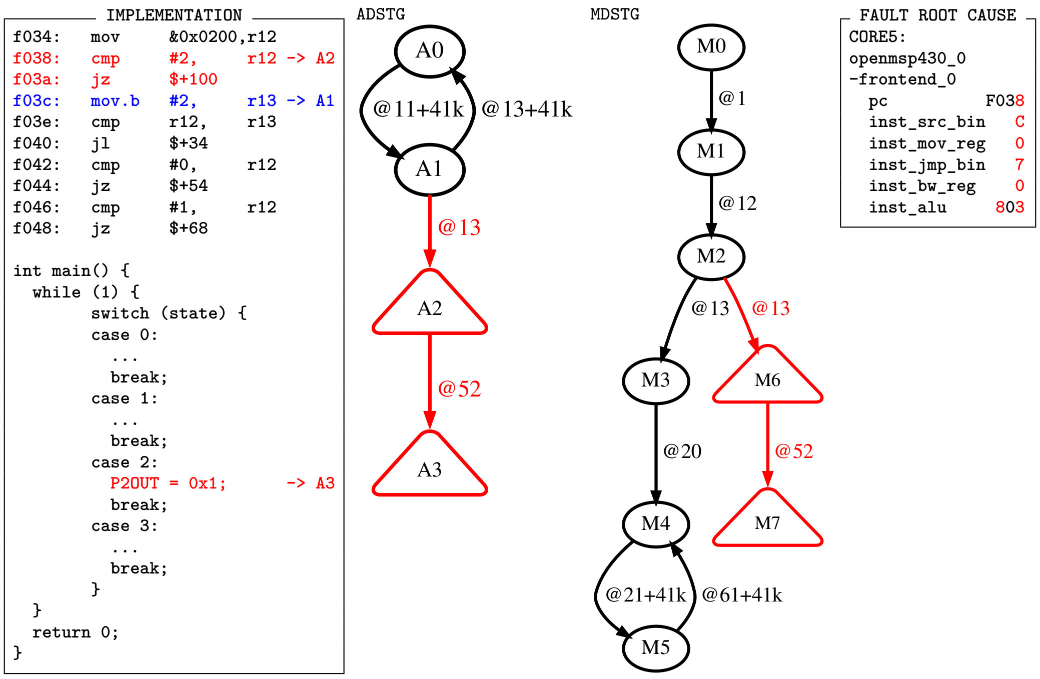
Figure 13: Root-cause of a *tunnel* fault on ASCON SBOX. (Left) Implementation. (Middle) The ADSTG and MDSTG. (Right) Fault Root-cause captured on the scan chain.

resulting an anomalous execution of the `f0b4` instruction. The ADSTG indicates that core 3 takes four clock cycles to recover from the fault, and then returns to a sane state `A3`. Similarly, the MDSTG recovers to a regular micro-architecture state `M4` with a four-cycle lag compared to the correct execution. However, subsequent executions proceed correctly, and the ADSTG of the faulted core 3 matched that of the unfaulted cores thereafter. This time delay manifests as a tunnel pattern in the `ADSTG` and `MDSTG`. At application level, we verified that core 3 continued to produce correct outputs, with the only discernible change being a four-cycle delay.

4.3 Root-cause Analysis on Countermeasure Faults

In this section, we apply FaultDetective onto more traditional cryptographic engineering tasks to explore how faults can impact and propagate within firmware protected by countermeasures. We discuss two cases.

Redundant Simple Two States. We investigate the effectiveness of the Redundant Simple Two States against clock glitch injection. Figure 14 demonstrates how a single glitch can



```
IMPLEMENTATION
f034:   mov     &0x0200,r12
f038:   cmp     #2,     r12 -> A2
f03a:   jz      $+100
f03c:   mov.b   #2,     r13 -> A1
f03e:   cmp     r12,    r13
f040:   jl      $+34
f042:   cmp     #0,     r12
f044:   jz      $+54
f046:   cmp     #1,     r12
f048:   jz      $+68

int main() {
  while (1) {
        switch (state) {
        case 0:
          ...
          break;
        case 1:
          ...
          break;
        case 2:
          P2OUT = 0x1;       -> A3
          break;
        case 3:
          ...
          break;
        }
  }
  return 0;
}
```

```
FAULT ROOT CAUSE
CORE5:
openmsp430_0
-frontend_0
  pc            F038
  inst_src_bin     C
  inst_mov_reg     0
  inst_jmp_bin     7
  inst_bw_reg      0
  inst_alu       803
```

Figure 14: Root-cause of countermeasure faults on Redundant Simple Two States with a single glitch. (Left) Implementation. (Middle) The ADSTG and MDSTG. (Right) Fault Root-cause captured on the scan chain.

break the Redundant Simple Two States. The implementation on the left summarizes the C implementation of the Redundant Simple Two States FSM using a case statement. The clock glitch forces the FSM into `case 2`, causing the application to output a `1` (ADSTG state `A3`), flagged as a fault. The fault root-cause analysis explains what happens. The glitch at cycle 13 affects multiple registers in the `frontend_0` instruction decoder. The program counter (PC) is modified to `f038` but after the instruction fetch of `f03e`. In this case, the clock glitch causes the program to jump *backward*. Upon inspecting the Verilog and the signal activities, we concluded that the execution semantics of `cmp #2, r12` are altered, and intermixed with the execution semantics of the `f03e` instruction, namely `cmp r12, r13`. Thus, the net effect is that the instruction at `f038` appears to execute as `cmp r12, r12`. This sets the zero flag and eventually leads the case statement falls directly into the `case 2` entry, thereby effectively bypassing the redundancy. Of course, one could argue that the implementation of Redundant Simple Two States in C itself is not redundant. The instruction group `f034 - f048` are all related to the evaluation of `switch(state)`, which is not redundant. But the fault root-cause analysis using FaultDetective highlights how and why this application can fail.

VerifyPIN5. A second example of root-cause analysis on fault countermeasures explains how `VerifyPIN5` fails in both hardware and software. Figure 15 illustrates the setup. The C code at the bottom right of Figure 15 provides an overview of the PIN checker implementation. At the beginning of the PIN verification process, a trial counter is decremented, and the PIN checking proceeds only if the counter contains a non-zero value. We inject a glitch right at the start of the PIN verification, during the section of code that tests the trial counter `g_ptc`. The glitch occurs in the middle of a byte-compare instruction (at `f0e2`), abruptly terminating that instruction. The scan chain differential shows that multiple registers in `frontend_0` are affected, including the program counter

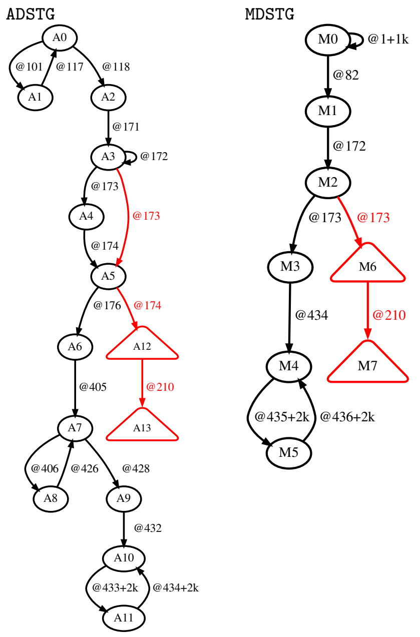
```
IMPLEMENTATION
# if (g_ptc > 0)
f0da: mov.b  &0x0209,r12
f0de: sxt    r12
f0e0: clr.b  r13
f0e2: cmp.b  &0x0209,r13 -> A3
f0e6: jl     $+6         -> A5

# return (BOOL_FALSE)
f0e8: clr.b  r12         -> A12
f0ea: jmp    $+66

verifyPIN5() {
 if (g_ptc > 0) {
   g_ptc--;

  if(byteArrayCompare(..)) {
   if(byteArrayCompare(..)) {
    g_ptc = 3;
    g_authenticated = BOOL_TRUE;
    return BOOL_TRUE;
   } else countermeasure();
  }

}
return BOOL_FALSE;
}
```



```
FAULT ROOT CAUSE
CORE5:
openmsp430_0
-frontend_0
  pc           F0E6
  int_type        1
  inst_sz         0
  inst_jmp_bin    0
  inst_dest_bin   9
  inst_bw_reg     0
  inst_as        01
  inst_alu      20B
  inst_ad        00
  e_state         D
```

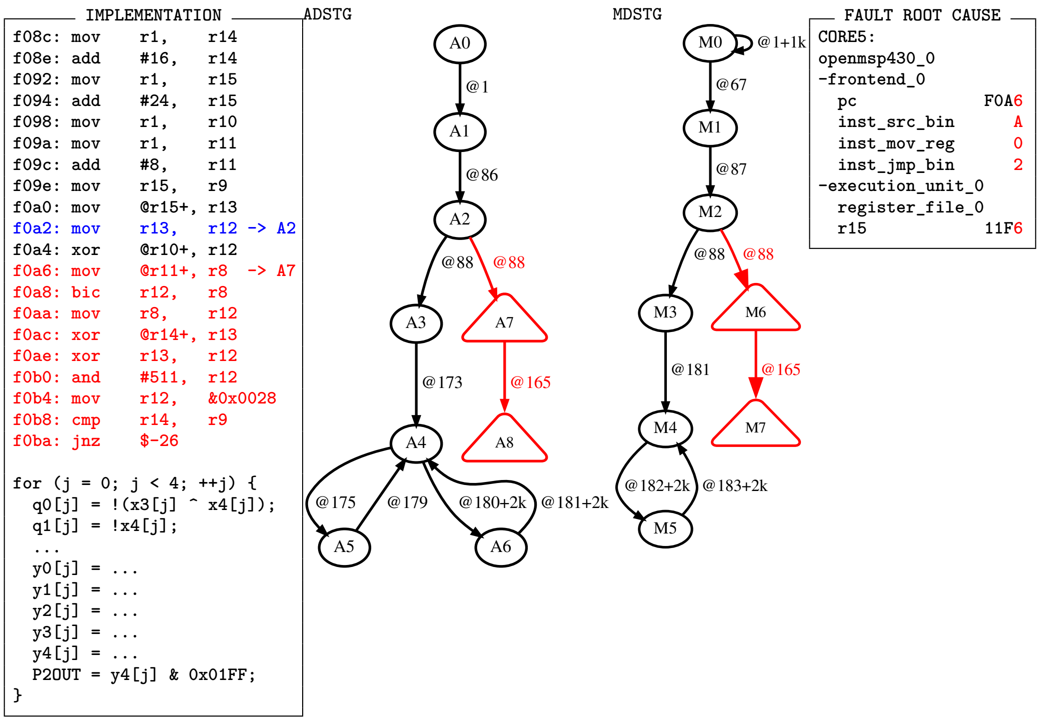
Figure 15: Root-cause of countermeasure faults on VerifyPIN5 with a single glitch. (Left) Implementation. (Middle) The ADSTG and MDSTG. (Right) Fault Root-cause captured on the scan chain.

`pc` and the instruction sequencer `e_state`. Subsequently, there is a behavior that can only be understood at the micro-architecture level. The fault in `e_state` causes the sequencer to enter two non-functional (non-executing) states before resuming execution at `f0ea`. As a result, both `jl $+6` and `clr.b r12` are not executed. Skipping the first instruction does not compromise the countermeasure; it only prematurely terminates the trial counter test. However, skipping the second instruction poses a problem. It changes the interpretation of `return BOOL_FALSE` to the last value held by `r12`, which in this case would be `return g_ptc`. Since `g_ptc` retains a positive value, the PIN verifier evaluates it as `BOOL_TRUE`, thereby compromising the countermeasure. The PIN verifier in the main program does not test for a hardened Boolean, but only for a non-zero return value.

4.4 Root-cause Analysis on Classic Ciphertext Faults

Our final example demonstrates how FaultDetective analyzes the fault root-cause of an application that produces a sensitive output. We used a bit-sliced ASCON SBOX as the driving example.

ASCON Sbox Bit-slicing Implementation. Figure 16 illustrates the fault root-cause analysis following a clock glitch injection in a bit-sliced ASCON SBOX. The glitch occurred just after the start of the second of four loop iterations, while executing the `xor @r10+, r12` instruction at `f0a4`. Surprisingly, this glitch affected all subsequent outputs (`P2OUT`) of the loop, despite each iteration being coded independently with `j` iterations. The root-cause analysis explains the sequence of events after the fault is injected: the core 5 follows a wormhole pattern with a modified register `r15`, as shown in Figure 16 (right). The fault in `r15` is caused by the auto-increment operation of instruction `ffa0`, which executes during the fault injection and causes an unintended double increment. Given that the the compiler has used incremental pointer arithmetic to access the cipher state



IMPLEMENTATION

```
f08c: mov    r1,    r14
f08e: add    #16,   r14
f092: mov    r1,    r15
f094: add    #24,   r15
f098: mov    r1,    r10
f09a: mov    r1,    r11
f09c: add    #8,    r11
f09e: mov    r15,   r9
f0a0: mov    @r15+, r13
f0a2: mov    r13,   r12 -> A2
f0a4: xor    @r10+, r12
f0a6: mov    @r11+, r8  -> A7
f0a8: bic    r12,   r8
f0aa: mov    r8,    r12
f0ac: xor    @r14+, r13
f0ae: xor    r13,   r12
f0b0: and    #511,  r12
f0b4: mov    r12,   &0x0028
f0b8: cmp    r14,   r9
f0ba: jnz    $-26

for (j = 0; j < 4; ++j) {
  q0[j] = !(x3[j] ^ x4[j]);
  q1[j] = !x4[j];
  ...
  y0[j] = ...
  y1[j] = ...
  y2[j] = ...
  y3[j] = ...
  y4[j] = ...
  P2OUT = y4[j] & 0x01FF;
}
```

FAULT ROOT CAUSE

```
CORE5:
openmsp430_0
-frontend_0
  pc                F0A6
  inst_src_bin         A
  inst_mov_reg         0
  inst_jmp_bin         2
-execution_unit_0
  register_file_0
  r15               11F6
```

Figure 16: Fault Root-cause analysis of ASCON SBOX. (Left) Implementation. (Middle) The ADSTG and MDSTG. (Right) Fault Root-cause captured on the scan chain.

elements, a fault in the pointer `r15` consequently affects all subsequent ciphertext.

4.5 Simulation Performance

Table 3 summarizes the simulation complexities of the previously discussed test cases. A test case involves a compiled firmware subjected to a specific fault injection method. Table 3 includes six test cases, each subjected to a single fault injection method: either clock glitching or laser injection. A fault simulation is a simulation of a single fault injection in the context of a test case. A fault campaign is the collection of all fault simulations needed to cover a test case. Each test case firmware is characterized by two important properties: the number of instructions subjected to fault simulation and the number of clock cycles subjected to fault simulation. Table 3 also documents the fault campaign size per test case. For clock glitching, the fault campaign size is equal to the number of clock cycles subjected to glitching in the test case firmware. For example, `VerifyPIN0` runs for 177 clock cycles, resulting in 177 testbenches and consequently 177 fault simulations, each involving a single unique clock glitch. For laser fault injection, the fault campaign size equals the number of selected layout locations. For example, by selecting a 15 μm diameter for the laser spot, there are 2,606 non-overlapping regions across the entire layout (Figure 8), resulting in a total of 2,606 testbenches resulting in 2,606 fault simulations, each affecting one to three registers. Additionally, fault simulations for root-cause analysis use constant data inputs hardcoded within each test case firmware. For example, verifyPIN0 has both the secret PIN and a mismatched trial PIN hardcoded in the test firmware. Therefore, a single fault injection per firmware execution is sufficient. Finally, Table 3 shows the number of detected faults in the firmware, i.e. fault injections that result in the firmware detecting a difference among the redundant cores. The root-case analysis examples in Section 4 are all selected from injections that ended in a detected fault.

Table 3: Fault Simulation Complexity.

Test Case	Firmware	Fault Method	Firmware Instructions	Firmware Cycles	Campaign Size (# Injections)	Detected Faults
1	verifyPIN0	clock	65	177	177	21
2	Red. Two States	laser	19	28	2,606	67
3	ASCON SBOX	laser	43	96	2,606	192
4	Red. Two States	clock	19	28	28	5
5	VerifyPIN5	clock	110	294	294	10
6	ASCON SBOX	clock	43	96	96	35

Table 4: Fault Simulation Time (in seconds).

Test Case	Firmware	Setup	Fault Sim per Injection	ADSTG/MDSTG Generation	Parallel Fault Campaign
1	verifyPIN0	<1	133	3	3,600
2	Red. Two States	5	93	1	30,600
3	ASCON SBOX	5	116	4	19,800
4	Red. Two States	<1	193	1	3,600
5	VerifyPIN5	<1	221	5	5,400
6	ASCON SBOX	<1	159	4	3,600

Table 4 lists the simulation time per test case, measured in seconds. The simulation setup time is the time needed to generate all testbenches for the fault campaign. The fault simulation time is the average time needed for a fault simulation over the complete fault campaign. The ADSTG/MDSTG generation time is the average time needed to construct the ADSTG and MDSTG, scan state differential, and to record the fault root-cause of a fault simulation for the fault campaign. The parallel fault campaign time is the time needed for the entire fault campaign using parallel fault simulation. For a six-core design, each fault simulation completes in a few minutes on a Xeon Gold 6248 CPU. The numbers in Table 4 use 10x parallel runs for test case 2 and 20x parallel runs for all other cases. Notably, the fault simulation time is dominating the overall duration of the experiment on all test cases.

5 Conclusion

FaultDetective demonstrates the ability to identify the earliest onset of a fault in hardware by capturing the state of the design using a scan chain. This capability allows us to precisely analyze how applications fail. We extend Dullien's *sane machine/weird machine* framework into a layered structure that separates the Intended FSM, the architecture, and the micro-architecture. By tracking the scan state over time, we can monitor the behavior of the weird machine and its impact on software execution. Using a set of test cases, we illustrated that fault root-cause analysis is feasible and that it is possible to bridge the gap between fault simulation based on uniform fault models and fault testing based on real experiments. We are presently testing a physical implementation of the test ASIC described in this paper, and we aim to demonstrate that the scan chain sampling method can be applied to physical fault attacks as well.

Acknowledgements

This research was support in part by National Science Foundation Award 2219810.

References

[ACD$^+$24] Ihab Alshaer, Brice Colombier, Christophe Deleuze, Vincent Beroulle, and Paolo Maistri. Microarchitectural insights into unexplained behaviors under clock glitch fault injection. In Shivam Bhasin and Thomas Roche, editors, *Smart Card Research and Advanced Applications*, pages 3–22, Cham, 2024. Springer Nature Switzerland.

[AHS09] Kahraman D. Akdemir, Ghaith Hammouri, and Berk Sunar. Non-linear error detection for finite state machines. In Heung Youl Youm and Moti Yung, editors, *Information Security Applications*, pages 226–238, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[AWMN20] Victor Arribas, Felix Wegener, Amir Moradi, and Svetla Nikova. Cryptographic fault diagnosis using verfi. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020*, pages 229–240. IEEE, 2020.

[BBG+19] Arthur Beckers, Josep Balasch, Benedikt Gierlichs, Ingrid Verbauwhede, Saki Osuka, Masahiro Kinugawa, Daisuke Fujimoto, and Yuichi Hayashi. Characterization of em faults on atmega328p. In *2019 Joint International Symposium on Electromagnetic Compatibility, Sapporo and Asia-Pacific International Symposium on Electromagnetic Compatibility (EMC Sapporo/APEMC)*, pages 1–4, 2019.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

[BGX+21] Lauren Biernacki, Mark Gallagher, Zhixing Xu, Misiker Tadesse Aga, Austin Harris, Shijia Wei, Mohit Tiwari, Baris Kasikci, Sharad Malik, and Todd M. Austin. Software-driven security attacks: From vulnerability sources to durable hardware defenses. *ACM J. Emerg. Technol. Comput. Syst.*, 17(3):42:1–42:38, 2021.

[DBC+18] Jean-Max Dutertre, Vincent Beroulle, Philippe Candelier, Stephan De Castro, Louis-Barthelemy Faber, Marie-Lise Flottes, Philippe Gendrier, David Hély, Regis Leveugle, Paolo Maistri, Giorgio Di Natale, Athanasios Papadimitriou, and Bruno Rouzeyre. Laser fault injection at the cmos 28 nm technology node: an analysis of the fault model. In *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–6, 2018.

[DPP+16] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. FISSC: A fault injection and simulation secure collection. In *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*, pages 3–11, 2016.

[Dul20] Thomas Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2):391–403, 2020.

[Gir17] Olivier Girard. openmsp430. https://opencores.org/projects/openmsp430, 2017. Accessed Online on 4/15/2024.

[GS21] Jacob Grycel and Patrick Schaumont. Simplifi: Hardware simulation of embedded software fault attacks. *Cryptography*, 5(2), 2021.

[GYG$^+$18] Marjan Ghodrati, Bilgiday Yuce, Surabhi Gujar, Chinmay Deshpande, Leyla Nazhandali, and Patrick Schaumont. Inducing local timing fault through EM injection. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 142:1–142:6. ACM, 2018.

[HSP21] Max Hoffmann, Falk Schellenberg, and Christof Paar. ARMORY: fully automated and exhaustive fault simulation on ARM-M binaries. *CoRR*, abs/2105.13769, 2021.

[KDD21] Vanthanh Khuat, Jean-Luc Danger, and Jean-Max Dutertre. Laser fault injection in a 32-bit microcontroller: from the flash interface to the execution pipeline. In *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 74–85, 2021.

[MBCB05] Debdeep Mukhopadhyay, Shibaji Banerjee, Dipanwita Roy Chowdhury, and Bhargab B. Bhattacharya. Cryptoscan: A secured scan chain architecture. In *14th Asian Test Symposium (ATS 2005), 18-21 December 2005, Calcutta, India*, pages 348–353. IEEE Computer Society, 2005.

[MWE$^+$03] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 29–40, 2003.

[NFM$^+$19] Adib Nahiyan, Farimah Farahmandi, Prabhat Mishra, Domenic Forte, and Mark Tehranipoor. Security-aware fsm design flow for identifying and mitigating vulnerabilities to fault attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(6):1003–1016, 2019.

[NOV$^+$22] Pascal Nasahl, Miguel Osorio, Pirmin Vogel, Michael Schaffner, Timothy Trippel, Dominic Rizzo, and Stefan Mangard. SYNFI: pre-silicon fault analysis of an open-source secure element. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):56–87, 2022.

[OGM15] Sébastien Ordas, Ludovic Guillaume-Sage, and Philippe Maurine. EM injection: Fault model and locality. In Naofumi Homma and Victor Lomné, editors, *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2015, Saint Malo, France, September 13, 2015*, pages 3–13. IEEE Computer Society, 2015.

[PG23a] George Papadimitriou and Dimitris Gizopoulos. Anatomy of on-chip memory hardware fault effects across the layers. *IEEE Transactions on Emerging Topics in Computing*, 11(2):420–431, 2023.

[PG23b] George Papadimitriou and Dimitris Gizopoulos. Silent data corruptions: Microarchitectural perspectives. *IEEE Transactions on Computers*, 72(11):3072–3085, 2023.

[PHB$^+$19] Julien Proy, Karine Heydemann, Alexandre Berzati, Fabien Majéric, and Albert Cohen. A first isa-level characterization of em pulse effects on superscalar microarchitectures: A secure software perspective. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, New York, NY, USA, 2019. Association for Computing Machinery.

[PV16] Frank Piessens and Ingrid Verbauwhede. Software security: Vulnerabilities and countermeasures for two attacker models. In Luca Fanucci and Jürgen Teich, editors, *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 990–999. IEEE, 2016.

[RSG23] Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. Revisiting fault adversary models - hardware faults in theory and practice. *IEEE Trans. Computers*, 72(2):572–585, 2023.

[RSS$^+$21] Jan Richter-Brockmann, Aein Rezaei Shahmirzadi, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. FIVER - robust verification of countermeasures against fault injections. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):447–473, 2021.

[SMB$^+$21] Chad Spensky, Aravind Machiry, Nathan Burow, Hamed Okhravi, Rick Housley, Zhongshu Gu, Hani Jamjoom, Christopher Kruegel, and Giovanni Vigna. Glitching demystified: Analyzing control-flow-based glitching attacks and defenses. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 400–412, 2021.

[SRVL$^+$20] M. Sultan M. Siddiqui, Sharma Ruchi, Loi Van Le, Taegeun Yoo, Ik-Joon Chang, and Tony Tae-Hyoung Kim. Sram radiation hardening through self-refresh operation and error correction. *IEEE Transactions on Device and Materials Reliability*, 20(2):468–474, 2020.

[Sto16] Ko Stoffelen. Optimizing s-box implementations for several criteria using sat solvers. In *FSE*, pages 140–160. Springer, 2016.

[TAC$^+$22] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. Exploration of fault effects on formal risc-v microarchitecture models. In *2022 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 73–83, 2022.

[TAC$^+$23] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. μarchifi: Formal modeling and verification strategies for microarchitectural fault injections. In Alexander Nadel and Kristin Yvonne Rozier, editors, *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, pages 101–109. IEEE, 2023.

[THN$^+$24] Simon Tollec, Vedad Hadzic, Pascal Nasahl, Mihail Asavoae, Roderick Bloem, Damien Couroussé, Karine Heydemann, Mathieu Jan, and Stefan Mangard. Fault-resistant partitioning of secure cpus for system co-verification against faults. *IACR Cryptol. ePrint Arch.*, page 247, 2024.

[VMDB20] Raphael Andreoni Camponogara Viera, Philippe Maurine, Jean-Max Dutertre, and Rodrigo Possamai Bastos. Simulation and experimental demonstration of the importance of ir-drops during laser fault injection. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(6):1231–1244, 2020.

[WH10] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective.* Addison-Wesley Publishing Company, USA, 4th edition, 2010.

[YGS15] Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. Tvvf: Estimating the vulnerability of hardware cryptosystems against timing violation attacks. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 72–77, 2015.