FAME: Fault-attack Aware Microprocessor Extensions for Hardware Fault Detection and Software Fault Response

Bilgiday Yuce, Nahid F. Ghalaty, Chinmay Deshpande,
Conor Patrick, Leyla Nazhandali, Patrick Schaumont
Bradley Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, USA
{bilgiday,farhady,chinmay,conorpp,leyla,schaum}@vt.edu

ABSTRACT

Fault attacks are a known serious threat to embedded software security. We propose FAME, a low-cost and flexible approach to defend embedded software against fault attacks.

FAME offers a combination of fault detection in hardware and fault response in software. A hardware fault detection unit continuously monitors the system status. When a fault injection is detected, an alarm signal triggers a secure trap mechanism that passes the control to a software trap handler. The trap handler applies a suitable fault response policy, which may include a broad variety of responses such as clearing sensitive data or issuing system-wide alerts. This enables a targeted, fast fault detection as well as an application-dependent, user-defined fault response.

FAME requires much lower overhead than traditional countermeasure techniques in software or hardware. We demonstrate a prototype implementation of FAME using a modified LEON3 processor, and we analyze the hardware and software overhead to thwart setup-time violation attacks. The hardware area overhead is 7.4% and 14.2% in the number of LUTs and registers, respectively. The overhead of the software trap handler on top of an AES-128 program is 0.59%–0.71% in footprint and 1.01%–2.35% in performance, depending on the security policy. In contrast, traditional countermeasures that use redundant hardware or software against similar faults have at least double overhead.

CCS Concepts

•**Security and privacy** → **Embedded systems security; Hardware attacks and countermeasures;** *Software and application security;*

Keywords

Fault-attack Aware Microprocessor Extensions; Embedded Software Security

1. INTRODUCTION

With the tremendous growth of embedded and pervasive applications, we are increasingly putting trust and confidence in a range of embedded systems from smartcards to pay-TV units. This trend expands the threat model of secure embedded applications from software into hardware. In this research, we investigate an important class of hardware attacks against embedded software, which uses fault injection as a hacking tool. Over the past 15 years, *fault attacks* have grown from a crypto-engineering curiosity into a systematic adversarial technique [1, 2]. Fault attacks use well-chosen, targeted fault injection combined with clever system response analysis to break the security of a system. This includes extracting the key material, weakening the cryptographic strength, and bypassing the security.

Defending software against fault attacks introduces the additional difficulty that the faults do not originate in the software, but rather in the underlying processor hardware. Moreover, modern embedded systems need to satisfy various performance and flexibility requirements. Therefore, modern embedded systems need *low-cost* and *flexible* mechanisms for fault detection and response [3–5].

In this research, we propose *FAME*, a fault-attack aware microprocessor suited for embedded, constrained systems. To protect embedded applications against fault attacks, FAME provides generic microprocessor extensions to unprotected processor hardware and user software. As shown in Fig. 1a, it partitions fault detection and response over hardware and software. Hardware fault detection immediately notifies FAME of a fault attack. Software fault response enables flexible fault handling. At the hardware level, FAME uses fault detectors to capture environmental anomalies such as clock glitches or failed error checksums in memory. A fault control unit ensures that fault recovery information is maintained in a set of Fault Response Registers (FRRs). When a fault is detected, control is passed to a software trap handler to implement

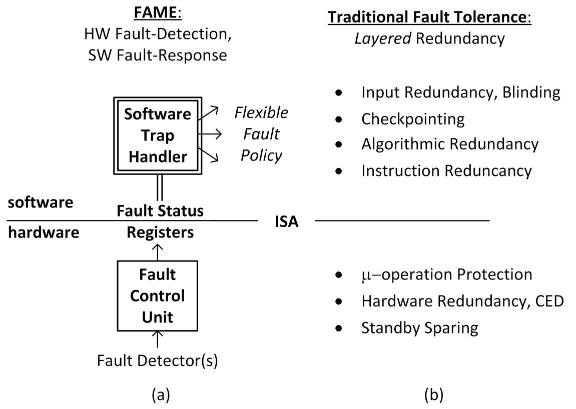


Figure 1: Comparison of (a) FAME and (b) traditional fault-tolerance

HASP 2016, June 18 2016, ,
© 2016 ACM. ISBN 978-1-4503-4769-3/16/06. . . $15.00
DOI: http://dx.doi.org/10.1145/2948618.2948626

a user-defined, application-specific fault response policy. The fault control unit, as well as the software trap handler, are fault-resistant designs.

FAME allows low-cost, performance-efficient, and flexible mitigation of fault attacks. FAME achieves *cost-efficiency* using redundancy to protect a small subset of the processor state (i.e, FRRs) and a small portion of the embedded software (i.e, trap handler). From a *performance* point of view, FAME extensions cause negligible timing overhead on the processor hardware. On the software side, FAME affects the performance *only if* a fault is detected. The actual software overhead depends on the complexity of the trap handler, but will be smaller than the inherently-redundant software-only techniques [6]. FAME provides *flexibility* through user-defined fault response policy, which can be adjusted for the security and performance needs of the application. Furthermore, FAME is *backward compatible* with existing software as the fault response is kept separate from the application.

The remainder of this paper is organized as follows. Section 2 summarizes the motivation behind this work. Section 3 explains the generic FAME extensions. Section 4 provides a detailed case study of the components of FAME on a LEON3 implementation. Section 5 presents the related work. Section 6 shows the results and overhead of FAME and compares it with conventional software countermeasures. Section 7 concludes the paper.

2. MOTIVATION

Today, *a great variety of fault injection and analysis methods* are available to attack all forms of embedded systems. Traditional fault attacks assume a fault model derived from the fault injection technique and they infer internal system secrets by analyzing the observed faulty system response [2]. More recently, faults have also been recognized as a source of side-channel leakage. These so-called biased fault attacks detect the onset of faults as a function of fault injection intensity and internal secret variables [1, 7, 8]. They then test the value of the internal secrets using hypothesis testing. Biased fault attacks use fault models that are less strict than traditional fault attacks. This variety of attacks underlines the growing need for a generic and efficient countermeasure against fault attacks.

The existing fault countermeasures for embedded software are rooted in fault-tolerant system design, and they do not provide an efficient solution for the requirements of modern embedded systems. The fundamental fault-tolerant method, illustrated in Fig. 1b, is to apply redundancy, to verify the redundant executions for faults, and to restore the correct system state after fault detection. These countermeasures are layered techniques that handle detection and response either completely in software or else completely in hardware [6, 9]. Software fault-tolerant techniques typically incur significant *performance overhead* to establish the detection of a faulty value, and they require specially prepared application software. Hardware fault-tolerant countermeasures need to be self-contained, and as such they incur significant *area overhead*. In addition, fault-tolerant techniques are vulnerable against adaptive adversaries [10, 11].

Considering the requirements of modern embedded systems and issues of fault-tolerant countermeasures, a crucial step for efficient countermeasures is differentiating *fault attack awareness* from *fault tolerance*. Fault tolerance aims at guaranteeing a certain level of correctness under the assumption of a general, often random fault model. The aim of fault-attack awareness is supporting a given security policy against an adversary who applies focused, intelligent faults to extract secret keys. FAME utilizes this difference for a low-cost and flexible countermeasure against fault attacks.

3. FAME EXTENSIONS

This section presents hardware and software extensions provided by FAME for mitigation of fault attacks. These extensions are generic, applicable to any embedded processor and software to protect them against fault attacks. Before presenting the extensions, we will define FAME's threat model and classify the fault types addressed by FAME.

3.1 Threat Model

In our threat model, FAME runs a secure application and contains secret key material. The secure application communicates with the outside world using a predefined protocol. We assume an adversary whose objective is to extract the secret keys by using only the outputs of the application. For this purpose, the adversary actively and adaptively injects faults into execution of the security application, and then, analyzes its response to the fault injection.

For a successful fault attack, the adversary needs to be capable of controlling fault injection, predicting the effects of fault injection on the program's execution, and testing the prediction by observing the actual effects of fault injection on the program's execution. FAME detects faults in the hardware level to prevent an adversary from imposing the expected fault effects on the program's execution. FAME also prevents an adversary from observing the actual fault effects through fault response in software level.

Similar to the adversial model of Lemke-Rust and Paar [12], we exclude invasive adversaries from our threat model. The adversary is not capable of modifying or monitoring the internals of chip, and changing the code of the security application. Numerous solutions can be found in the literature to protect the integrity of embedded hardware [13] and software [14] against invasive adversaries.

3.2 Classification of Faults

Because fault attacks exploit transient faults, FAME focuses on protecting embedded software against them. We separate the transient faults into different categories to specialize FAME's detection and recovery capabilities for the specific needs of different faults.

From the **fault detection point of view**, we partition transient faults into two categories, *synchronous* and *asynchronous* faults. Synchronous faults are detected at well-defined points aligned with the activity of processor's pipeline. The detection of these faults are associated with the execution of an instruction. For example, a synchronous fault in a memory location can be detected while it is being read by a memory-load instruction. Asynchronous faults are detected at arbitrary times, independent of the instructions in the pipeline. For example, asynchronous faults can be detected by monitoring clock signal or supply voltage.

Faults can corrupt data at rest (i.e, in the memory) or in transit (i.e, during computation). From the **fault recovery point of view**, we classify a transient fault as a *computation fault* or a *memory fault*. In case of a memory fault, an adversary injects faults into data being stored in the memory. When this data is used in the future computations, it leads to faulty results. In case of a computation fault, an adversary injects faults into the data being currently processed in the processor's pipeline. Recovering the effects of computation faults requires a different approach than recovering from memory faults. For example, a processor can recover from a computation fault detected during the execution of an instruction by re-executing this instruction [15]. Recovering from a memory fault may require regularly checkpointing the critical state of an application, and then, restarting the application from that point in case of a memory fault [9].

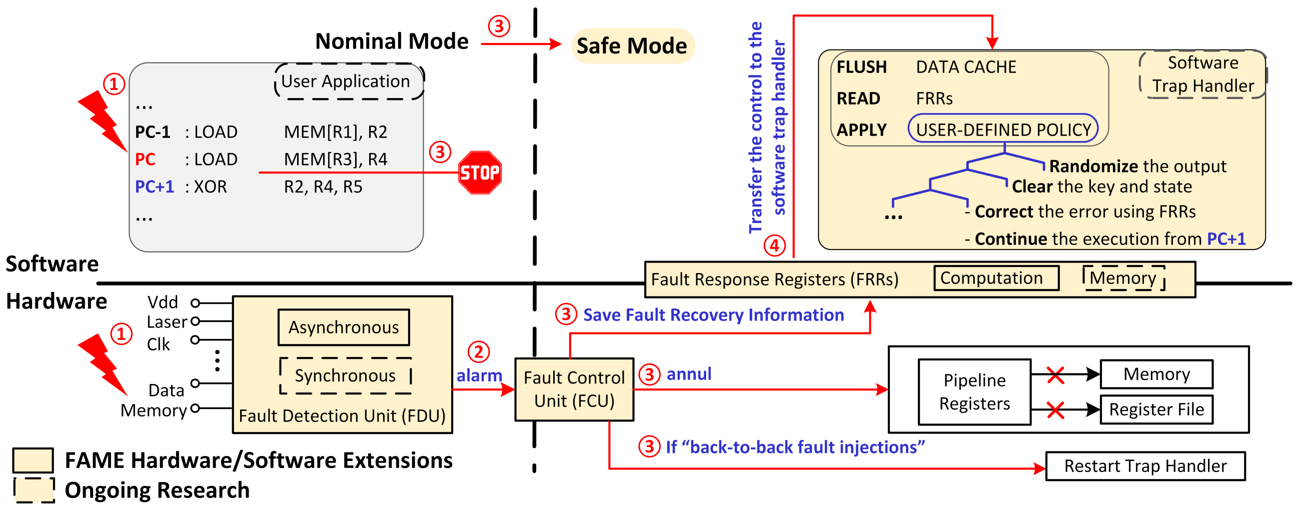


Figure 2: FAME combines fault detection in hardware and fault response in software: ① The adversary injects a fault and disturbs program's execution. ② The FDU raises the alarm. ③ The FCU takes immediate actions in hardware and invokes the trap handler. ④ Trap handler applies a user-defined security policy.

FAME envisions efficient detection and recovery of transient faults injected with a malicious intention. However, we will focus on detection of asynchronous faults and recovery of computation faults in this work. Integration of low-cost mechanisms for synchronous-fault detection and memory-fault recovery into FAME is our ongoing research.

3.3 Overview of Operation

Fig. 2 shows the overall architecture and operation of FAME, which relies on hardware/software extensions to an unprotected processor and user application.

A fault injection attempt is detected in hardware by the *Fault Detection Unit (FDU)*. Fault handling is achieved by a *secure trap* mechanism. The *Fault Control Unit (FCU)* and *Fault Response Registers (FRRs)* provide hardware support for this mechanism. Upon detection of a fault, the FCU initiates the transition to a software trap handler. FRRs provide an interface between the processor hardware and the software trap handler. The software trap handler applies a user-defined, application-specific security policy in a *safe mode*. Next, we will explain the details of FAME's components and operation, of which steps are labeled with ①–④ in Fig. 2.

3.4 Fault Detection Unit (FDU)

The FDU includes a set of fault detectors and monitors the processor to detect anomalies. During the normal operation, an application runs in the nominal mode and no overhead is accrued. Upon detection of a fault, the FDU asserts an *alarm* signal to notify the processor of a potential fault attack (① and ② in Fig. 2).

The FDU derives the fault status for the overall processor by combining different fault detectors. To detect asynchronous faults, the FDU uses dedicated sensors such as clock/voltage glitch detectors [16] and electromagnetic pulse detectors [17]. To detect synchronous faults in the datapath, the FDU uses concurrent error detection methods [18] and shadow latches [19]. For the synchronous faults in the memory, the FDU uses error detection codes [20]. The detector configuration of FAME depends on the application domain and the desired level of fault sensitivity. In this paper, we detect faults that originate from timing violations. Given the error detection mechanisms enumerated above, however, it should be clear that the FDU mechanism is generic and that it can handle a multitude of fault injection mechanisms.

3.5 Fault Control Unit (FCU)

The FCU acknowledges the alarm signal of the FDU and takes immediate actions in the hardware level (③ in Fig. 2).

It saves the fault recovery information into FRRs, annuls the instructions being executed in the pipeline, and disables write operations into the register file as well as memory. This enables two essential capabilities. First, the faulty parts of the software-visible state, which are contaminated before the alarm is raised, can be recovered by the trap handler. Second, no more faulty results will be committed to the architectural state of the processor after the alarm is raised.

Meanwhile, the FCU stops the execution of the application, initiates a non-maskable secure trap, and switches the processor from nominal mode to safe mode. This switching is done immediately at the next clock cycle. In safe mode, the processor is aware of the fault injection, and it can handle the faults through software trap handler.

It is mandatory that the FCU detects further fault injections and prevents fault attacks on the trap handling mechanism. In this work, the FCU restarts the software trap handler if a fault injection is detected during safe mode. This guarantees that FAME cannot exit from safe mode without completing the user-defined security policy.

3.6 Fault Response Registers (FRRs)

FRRs maintain the fault recovery information, the minimum information required for recovering fault effects on the software-visible state and returning from the trap handler.

To recover from computation faults, FRRs need to contain (*i*) return address to the interrupted program; (*ii*) status register of the processor; and (*iii*) the register-file inputs of the write-back stage, at minimum. The software trap handler can use this information to restore the software-visible state back to its correct, pre-fault status. FAME uses redundancy to guarantee the correctness of the content of FRRs.

Recovering from memory faults requires to keep the correct, pre-fault values of the critical memory and register file locations. This is ongoing research and future work for us. Our challenges in this

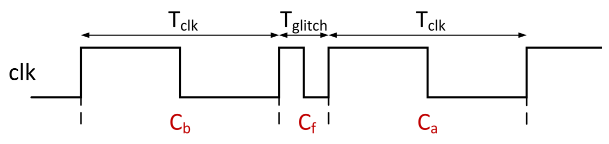


Figure 3: The effect of glitch injection on the clock signal

task are determining the minimum content to maintain, identifying the minimal set of hardware support, and designing a low-cost mechanism.

3.7 Software Trap Handler

The final step of fault handling is passing control to the trap handler (④ in Fig. 2). It applies a user-defined, application-specific security policy in software, which enables a flexible and adaptive fault response.

The trap handler first flushes the data cache to wipe out possibly faulty data. Then it recovers the processor state just before the fault injection by accessing FRRs. Finally, it applies the security policy and returns back to the nominal mode. The security of the trap handler is provided by traditional fault-tolerant countermeasures [6, 21]. As the size of the trap handler is much smaller than the size of the application, the cost of using traditional countermeasures is affordable. The security policy can be adjusted for the cost and security requirements of the application.

In summary, FAME uses fault detectors that are combined into a processor-level alarm signal. The alarm signal initiates a software trap to decide on the further course of action. FAME provides hardware-level support to maintain the fault recovery information. It is up to the trap handler to decide if it is safe to continue execution or not. FAME ensures that the trap handling mechanism itself is protected from faults. Next, we provide a proof-of-concept implementation for FAME.

4. FAME PROTOTYPE

To evaluate the FAME processor, we implemented a prototype by enhancing an existing processor with FAME extensions. The prototype aims at protecting embedded software against setup-time violation attacks. For this purpose, it employs FAME extensions for detecting asynchronous faults in hardware-level and recovering from computation faults in software-level.

4.1 Fault Injection to the Processor

To understand the operation of FAME, we need to carefully define the sequence of events leading to a fault. In a fault attack, an adversary waits until a program reaches a specific point in its execution. Then, he injects the fault into the program at this point. Finally, the adversary observes the fault effects after this point. In this work, we use *fault cycle* (C_f) to denote the clock cycle in which the fault occurs. We use *before-fault cycle* (C_b) and *after-fault cycle* (C_a) for the clock cycles before and after the fault, respectively. The program's execution is fault-free before C_f, and faulty from C_f.

The embodiment of C_b,C_f,C_a depends on the fault injection method. Fig. 3 describes the situation when we use clock glitch injection. A clock glitch will temporarily shorten the length of a clock cycle from T_{clk} to T_{glitch}, thereby causing a timing violation during C_f.

We will assume that the adversary does not have physical access to the on-chip bus and memories. Therefore, the adversary uses standard communication channels to observe the effects of faults.

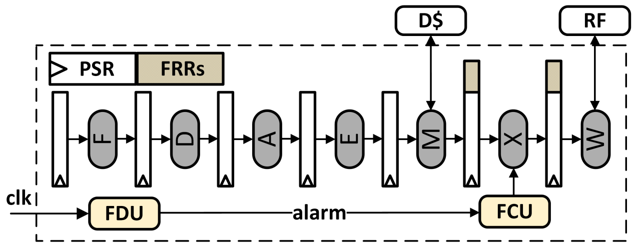


Figure 4: 7-stage LEON3 pipeline with FAME extensions

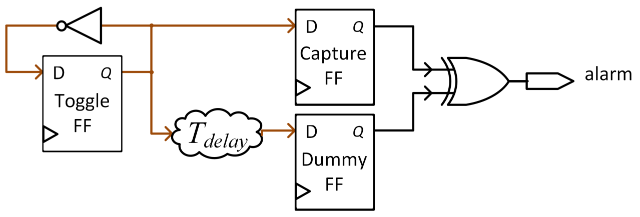


Figure 5: Block diagram of the Fault Detection Unit

4.2 LEON3 Processor Overview

Our prototype implementation is based on LEON3 [22], which is an open-source, 32-bit, SPARCv8-compilant, RISC processor with a 7-stage pipeline. To incorporate the FAME extensions, we modified a base LEON3 configuration, which includes a 64-KB on-chip RAM memory as well as 4-KB direct-mapped caches for Instructions and Data.

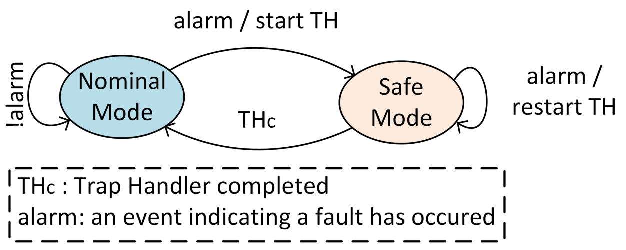
Fig. 4 shows the 7-stage pipeline of LEON3 with FAME extensions. The pipeline consists of *fetch (F), decode (D), register access (A), execute (E), memory (M), exception (X),* and *write-back (W)* stages. We use a clock monitor as the FDU. We integrated the FCU into the *X stage*. FRRs provide fault recovery information for some parts of the *X* and *W* stages because of two reasons. First, the Register File (RF) and Processor Status Register (PSR) are updated in the *W stage*. Second, the return address for the trap handler is computed in the *X stage*. Next, we will provide implementation details of the FAME prototype.

4.3 Fault Detection Unit (FDU)

In our prototype, the FDU detects setup time violations by using a delay chain [16,17] and a dummy flip-flop (FF) [19]. Fig. 5 shows the FDU, which includes three FFs, a delay chain, a NOT, and an XOR gate. The delay chain consists of buffers adjusted such that its propagation delay T_{delay} is slightly greater than the critical path $T_{critical}$ of the design. In this work, we determine the delay of the chain using static timing analysis with worst-case conditions (STA-WC). We use STA-WC as the proof-of-concept because of its simplicity. More advanced techniques, such as representative critical path synthesis [23], can be used to capture process variations in the synthesis of the delay chain. Toggle FF toggles its value every cycle which then arrives at Capture FF immediately and at Dummy FF after T_{delay}. In normal operation (T_{clk}), the inputs of both Capture and Dummy FFs toggle to the new value before the next clock edge. In case of glitchy clock T_{glitch}, Capture FF latches the new value whereas the Dummy FF latches the old value as the T_{glitch} is not enough for delay chain to make transition to new value. Therefore, the XOR gate generates the alarm signal at the next upgoing clock edge.

4.4 Fault Control Unit (FCU)

In our prototype, the FCU uses the state machine shown in Fig. 6



alarm / start TH
!alarm
Nominal Mode
THc
Safe Mode
alarm / restart TH
THc : Trap Handler completed
alarm: an event indicating a fault has occured

Figure 6: State diagram of the Fault Control Unit

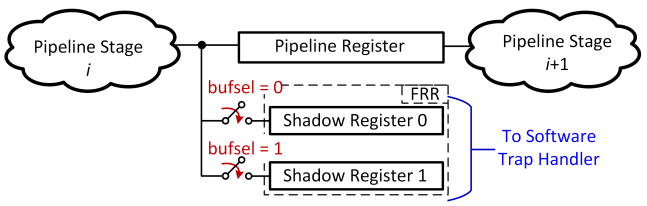
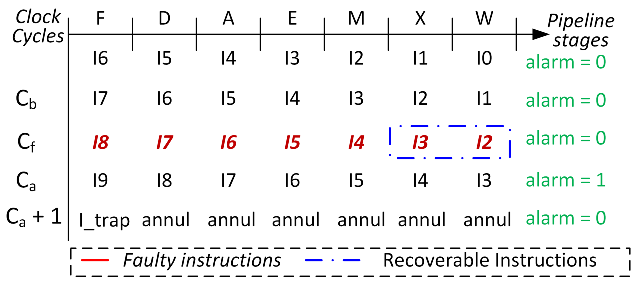


Figure 7: Ping-Pong buffering for FRRs: Only one shadow register is updated at a time. Content of FRR is frozen in case of an alarm.



Clock Cycles	F	D	A	E	M	X	W	Pipeline stages
	I6	I5	I4	I3	I2	I1	I0	alarm = 0
C_b	I7	I6	I5	I4	I3	I2	I1	alarm = 0
C_f	*I8*	*I7*	*I6*	*I5*	*I4*	*I3*	*I2*	alarm = 0
C_a	I9	I8	I7	I6	I5	I4	I3	alarm = 1
C_a + 1	I_trap	annul	annul	annul	annul	annul	annul	alarm = 0

Faulty instructions
Recoverable Instructions

Figure 8: Fault effect on the pipeline. Fault is injected during C_f. The alarm is raised during C_a. The first trap handler instruction is fetched after C_a.

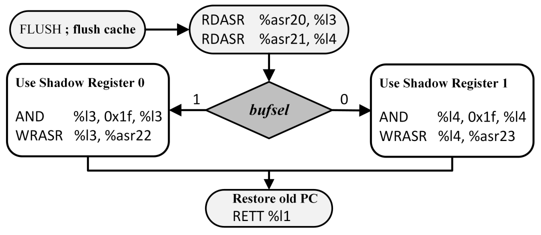


Figure 9: Trap Handler Flowchart for Invoking Resume Security Policy

to manage the secure trap mechanism. If the FCU detects an alarm signal while the processor is in the nominal mode, it switches the processor to safe mode. During this transition, the FCU (*i*) annuls all instructions in the pipeline; (*ii*) disables all memory and register file transfers of the user application; (*iii*) saves the fault recovery information into FRRs; and (*iv*) resumes execution with the first instruction of the trap handler. If the trap handler completes its execution without another fault attack detection, the FCU switches the processor back to the nominal mode. If the FDU detects a fault while the processor is in safe mode, the FCU restarts the trap handler and stays in safe mode. This guarantees that FAME cannot exit from safe mode without completing the user-defined security policy.

To initiate the fault processing after an alarm is raised, we extend the *X stage* of LEON3. The *X stage* supports precise trap handling, and transitions between processor modes. These extensions enable two crucial elements of our fault handling method. First, secure traps of FAME are immediately handled when the alarm is asserted. Second, FAME saves the fault recovery information into FRRs and provides this information to the software trap handler for correct execution. Next, we explain our selection and security strategies for the content of FRRs.

4.5 Fault Response Registers (FRRs)

FRRs keep the part of the processor state that is updated in C_b, just before the fault injection in C_f. The software trap handler can restore this processor state back and resume the execution of the application.

Fig. 7 shows the principle of our FRR implementation. The FRR keeps the previous value of the original pipeline register in one of its shadow registers; while keeping the new value in the other shadow register. Every clock cycle, only one of the shadow registers is updated. The shadow register to be updated is selected by a 1-bit signal *bufsel*. If Shadow Register 0 is updated during the *before-fault cycle* C_b, Shadow Register 1 is updated during C_f. Therefore, it is guaranteed that the fault occuring in C_f cannot contaminate the correct value within both shadow registers. When the alarm is asserted (in C_a), the update of the shadow registers are frozen until the trap handler is successfully completed. This prevents the correct FRR content from being overwritten after C_f.

We determine the content of FRRs by analyzing the effect of the fault injection on the execution of the pipeline. Fig. 8 shows the effect of a fault on the LEON3 pipeline. In Fig. 8, clock cycles run from the top, and pipeline stages run from left to right. In C_f, up to seven instructions, *I2 – I7*, will potentially be faulty. During C_f, two instructions could commit their results to the software-visible state of the processor. First, instruction *I4* could write a faulty value to the data cache. Second, instruction *I2* could update the Processor Status Register (PSR) and the register file with a faulty value. Both of these updates need to be intercepted and corrected by the software trap handler. Then the execution can be resumed from the next valid instruction (*I3* in Fig. 8). Therefore, FRRs keep (*a*) the register-file write address, write data, and write enable fields of the write-back stage registers; (*b*) the flags field of PSR; and (*c*) the address of the instruction being executed in the X stage in C_f.

After control is passed to the software trap handler, it reads the frozen content (*a*)–(*c*) of FRRs. At the minimum, the trap handler will restore the correct processor state (using (*a*) and (*b*)), and resume execution (using (*c*)).

Our prototype implementation of FAME in LEON3 integrates FRRs as follows. During the transition from nominal mode to safe mode, the processor hardware writes the program counter to the local register `%l1` of the software trap handler. We pack the remaining bits of FRRs into two pairs of LEON3's Ancillary State Registers (ASRs), `%asr20-21` and `%asr22-23`. The software handler can access these registers using `RDASR` instruction of LEON3. The frozen value of *bufsel* is also written into `%asr20-21`. Then the trap handler can know which shadow register contains the correct value.

4.6 Software Trap Handler

The trap handler is given control of the Processor once a fault alert triggers. The first instruction in the handler is a memory flush. This ensures all of the invalidated memory in the cache is dumped and not used. Then, the software trap handler can provide different options for handling the fault. One option can be to use the contents of FRR and resume the program under attack from the point of fault injection. In this scenario, we follow the flowchart in Fig. 9. First, the trap handler reads `%asr20-21` using `RDASR`. Then it checks the *bufsel* bit in `%asr20-21` to know which shadow registers of FRRs contains the correct value. If *bufsel* is a one, then the content of Shadow Register 0 is valid. If *bufsel* is not set, then Shadow Register 1 must be used. Next, the valid FRR is bitmasked to get the register index from it. This register index is the last register that was written to the register file and could have been affected by the fault in C_f. This is written back to the register file through a `WRASR` instruction of LEON3. Our control hardware will use this register index to restore the respective register to its last correct value. Finally, the trap handler restores the PC and returns to the nominal mode for resuming the program.

The importance of the flexibility provided by the software trap handler becomes more significant at higher abstraction levels such as protocol or algorithm level. For example, let us assume that the processor is busy with multiple transactions over a standard established connection to a server. These transactions contain several sessions. Each session has an encryption function with a predefined key. In case of detecting a fault during a session, the software handler can take several actions. A low-level security policy could allow the notification of the two parties. A medium-level security policy could restore the correct status of the processor before fault injection and continue the encryption algorithm. Higher security levels might require changing the session key and restarting the session or aborting the connection. Similarly, the security requirements can be adjusted dynamically depending on the severity of the fault injection. For example, lower-level security policies can be applied at the first invocation of the trap handler, while higher-level security policies are applied at the following invocations of the trap handler.

5. RELATED WORK

We summarize previous work on processor-oriented fault countermeasures in three categories.

Software Countermeasures against Fault Attacks: These techniques rely on redundancy such as Instruction Duplication (ID) [21], Instruction Triplication (IT) [21], replacing each instruction with a fault-tolerant instruction sequence (IS) [24], and parity checking [6]. Theißing et al. [6] and Barenghi et al. [21] provide a comprehensive list and analysis of the software countermeasures. The main drawback of these techniques is their large overhead in performance and footprint.

Fault-Tolerant Design: A generic and well-studied solution for faults is fault-tolerant design. Fault-tolerant design relies on redundancy in hardware or software such as modular redundancy, standby sparing, or N-version programming [9]. The fault-tolerant design deals with random sporadic faults. However, in case of a fault attack, the faults are intentionally injected by an intelligent adversary. Thus, the fault-tolerant design is not sufficient for the fault attack resistance. For example, the adversary can manipulate the original and redundant hardware and retrieve the secret key [10]. In addition, the nature of the fault attack problem enables significant optimizations in the cost of redundancy. For example, FAME uses redundancy for a small part of the processor, FRRs.

Secure Processors: Researchers have proposed various secure processors to provide hardware-level information security [25]. In their security model, a system is partitioned into an on-chip trusted region and an off-chip untrusted region [26]. They assume that the on-chip state of the processor (registers and caches) cannot be tampered with; while the off-chip components can be observed and modified by an adversary. Thus, they provide techniques to protect integrity and confidentiality of the off-chip memories, such as hash trees and memory encryption. FAME is different as it provides security against fault attacks, in which an adversary can inject faults into the on-chip components. Therefore, FAME is complementary to the existing secure processors and it can be integrated into them for on-chip security.

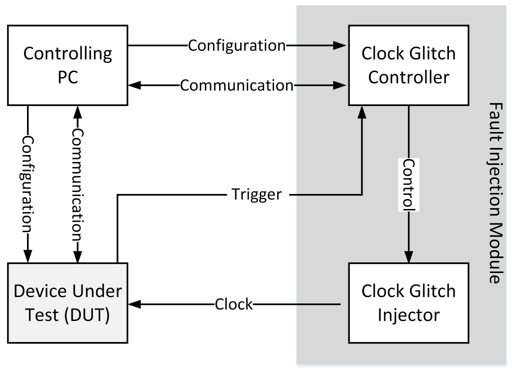


Figure 10: High-level block diagram of fault injection and analysis setup.

6. EXPERIMENTAL RESULTS

In this section, we provide experimental results for hardware/software overhead and fault injection evaluation for the prototype of FAME.

6.1 Fault Injection and Analysis Setup

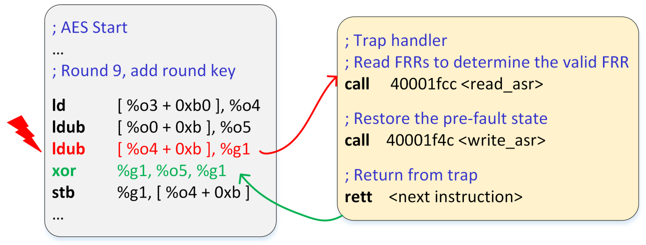
Figure 10 shows an overview of our fault injection and analysis setup. It consists of a controlling PC, a device under test (DUT), a clock glitcher module, and an oscilloscope. We implement the glitcher module and DUT on a SAKURA-G board [27].

The PC manages the fault injection process by controlling and configuring both the glitcher module and DUT. In this work, DUT is the FAME prototype or unprotected LEON3 processor. It executes a target program, which is an AES-128 implementation for this work. DUT also sends trigger to the glitcher for fault injection. The controlling PC communicates with DUT via a debug monitor. The glitcher module takes a glitch-free clock signal as an input from the pulse generator and generates a glitchy clock signal as an output. We dynamically set the glitch parameters via commands from the controlling PC.

6.2 Fault Injection Evaluation

We implemented the prototype of FAME on the main FPGA (Xilinx Spartan6 XC6SLX75) of the SAKURA-G board to evalute its operation under fault injection. We used an AES-128 implementation as the target program. In this experiment, our objective is to obtain the secret key of the AES-128 in the unprotected LEON3 and FAME prototype. First, we injected faults into the target program while it is running on the unprotected LEON3 and obtained the faulty ciphertetxs to launch a fault attack. Then, we injected faults into the target program running on FAME prototype and observed its fault response.

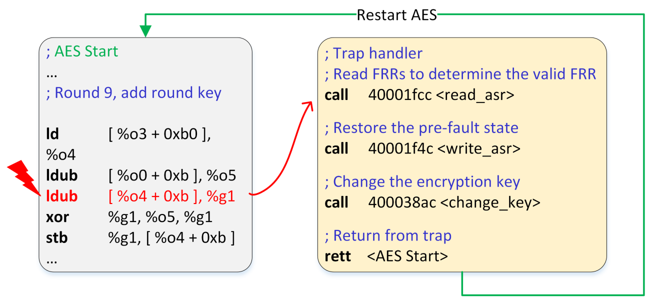
We mounted a recent biased fault attack, DFIA [7], on the output of AES round 9 to extract a byte of the secret key. We considered the `AddRoundKey` function as the last step in the execution of round 9. The reference implementation in C shows 16 statements of the



```
; AES Start
...
; Round 9, add round key

ld      [ %o3 + 0xb0 ], %o4
ldub    [ %o0 + 0xb ], %o5
ldub    [ %o4 + 0xb ], %g1
xor     %g1, %o5, %g1
stb     %g1, [ %o4 + 0xb ]
...
```

```
; Trap handler
; Read FRRs to determine the valid FRR
call    40001fcc <read_asr>

; Restore the pre-fault state
call    40001f4c <write_asr>

; Return from trap
rett    <next instruction>
```

Figure 11: The target program, attack model, and trap handler for FAME-Resume



```
; AES Start
...
; Round 9, add round key

ld      [ %o3 + 0xb0 ],
%o4
ldub    [ %o0 + 0xb ], %o5
ldub    [ %o4 + 0xb ], %g1
xor     %g1, %o5, %g1
stb     %g1, [ %o4 + 0xb ]
...
```

```
; Trap handler
; Read FRRs to determine the valid FRR
call    40001fcc <read_asr>

; Restore the pre-fault state
call    40001f4c <write_asr>

; Change the encryption key
call    400038ac <change_key>

; Return from trap
rett    <AES Start>
```

Figure 12: The target program, attack model, and trap handler for FAME-KeyChange

following form, with `state` the state variable, and `Roundkey` an array with AES round key values.

```
(*state)[i][j] ^= RoundKey[K];
```

The LEON3 assembly code for this C includes 5 instructions. The target for fault injection is affecting register `%g1`. Therefore, we injected clock glitches during the execution of `ldub [ %o4 + 0xb ], %g1` instruction.

```
ld    [ %o3 + 0xb0 ], %o4 //*state
ldub  [ %o0 + 0xb ], %o5 // RoundKey[K]
ldub  [ %o4 + 0xb ], %g1 //(*state)[i][k]
xor  %g1, %o5, %g1
stb  %g1, [ %o4 + 0xb ]
```

We implemented two different security policies to protect the AES software. The first policy (Fig. 11) restores the pre-fault status of the program and resumes the encryption. We call this policy *FAME-Resume*. The second policy (Fig. 12) is called *FAME-KeyChange*. It changes the secret key and starts a new encryption with a fresh key. In this work, we used a fixed, known value as the fresh key. Thus, we expect to see correct results for execution of the target program on the FAME prototype. In contrast, we expect faulty outputs from the target program running on the unprotected LEON3 because of fault injection.

During our experiments, we injected 68 clock glitches with different glitch widths. We changed glitch width from 16*ns* to 5*ns* with 162*ps* step size. Figure 13 shows the glitchy clock and the alarm signal for the glitch width of *6.8ns*. As it is seen, the alarm is raised in the cycle that follows the glitch. In the unprotected prototype, we successfully captured 10 faulty ciphertexts, and then, extract the key byte by launching the DFIA attack. Our trap handlers successfully recovered the fault effects and output the correct results for every fault injection. Therefore, we were not able to collect faulty ciphertexts required for DFIA. As a result, we were not able to extract the key when we used FAME.

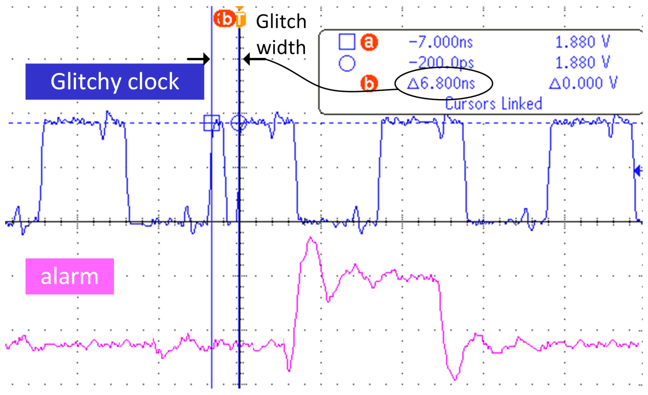


Figure 13: Clock glitch injection and *alarm* signal generation

Table 1: Hardware Overhead of FAME Extensions

	# Slice LUTs		# Slice Regs	
	Total	Overhead	Total	Overhead
LEON3 (Unprotected)	3435	-	1275	-
LEON3 + FCU,FRRs	3691	7.4%	1456	14.2%
LEON3 + FCU,FRRs,FDU	3744	8.9%	1459	15.2%

6.3 Hardware Overhead

To evaluate the hardware overhead of FAME, we mapped our FAME design into a Xilinx Spartan6 (XC6SLX75) FPGA. We obtained area and timing results for (*i*) a LEON3 implementation without FAME extensions; (*ii*) a LEON3 implementation with the FCU and FRRs; and (*iii*) a LEON3 implementation with the FCU, FRRs, and FDU.

The maximum operating frequency of all implementations is 62.5 MHz. This shows that FAME extensions incur no timing overhead. Table 1 lists the obtained area results. Adding the FCU and FRRs causes 7.4% and 14.2% increase in the number of LUTs and FFs, respectively. Adding the FDU incurs an additional 1.5% and 1% increase in the number of LUTs and FFs, respectively. As a result, FAME provides fault-attack resistance without any timing overhead and with a low area overhead.

6.4 Software Trap Overhead

Table 2 displays the overhead of the trap handler in terms of clock cycles and code footprint (Bytes). The results were taken for encryption a block of data (16 Bytes) with AES algorithm. The footprint includes the *text*, *data* and *bss* sections of the program.

For each detected fault injection attempt, the overhead of FAME-Resume in performance and footprint is 1.01% and 0.59%, respectively. The performance and footprint overhead of FAME-KeyChange is 2.35% and 0.71%, respectively. On the other hand, researchers have shown that the performance overhead of the well-known software countermeasures for full protection of AES (ID, IT and IS) is 97%–239% [6, 24]. The footprint overhead of these countermeasures is 89.9%–200% [6, 24]. Therefore, the overhead of FAME is much lower compared to other software techniques. Furthermore, the code redundancy in other techniques is always executed even if no fault injection attempt happens. However, the FAME trap handler is only invoked when the FDU detects a fault.

7. CONCLUSIONS

FAME combines fault detection in hardware and fault response in software. This allows low-cost, performance-efficient, and flex-

Table 2: Software Overhead of FAME Extensions

	# Cycles		**Footprint(Bytes)**	
	Total	**Overhead**	**Total**	**Overhead**
AES (Unprotected)	17631	-	25964	-
FAME-Resume	17810	1.01%	26116	0.59 %
FAME-KeyChange	18045	2.35%	26148	0.71%

ible integration of hardware and software techniques to mitigate fault attack risk. FAME is a generic solution, applicable to existing embedded processors.

FAME is low-cost as it uses redundancy to protect only a small subset of the processor state (i.e, FRRs) and a small portion of the embedded software (i.e, trap handler). FAME extensions do not bring any timing overhead on the processor hardware. On the software side, FAME affects the performance only if a fault injection is detected. FAME enables a flexible and application-specific trap handler, which can be adjusted for the security needs of the application.

8. ACKNOWLEDGMENT

This research was supported in part through the National Science Foundation Grant 1441710, and in part through the Semiconductor Research Corporation.

9. REFERENCES

[1] B. Yuce, N. F. Ghalaty, and P. Schaumont, "Improving fault attacks on embedded software using RISC pipeline characterization," in *Proc. of FDTC'15*, pp. 97–108, 2015.

[2] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012.

[3] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM TECS*, vol. 3, no. 3, pp. 461–491, 2004.

[4] B. Robisson, M. Agoyan, P. Soquet, S. Le Henaff, F. Wajsbürt, P. Bazargan-Sabet, and G. Phan, "Smart security management in secure devices," tech. rep., Cryptology ePrint Archive, Report 2015/670, 2015. http://eprint. iacr. org, 2015.

[5] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, L. Bossuet, and R. Vaslin, "Reconfigurable hardware for high-security/high-performance embedded systems: the safes perspective," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 2, pp. 144–155, 2008.

[6] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks," in *Proc. of DATE'13*, pp. 404–409, 2013.

[7] N. F. Ghalaty, B. Yuce, M. Taha, and P. Schaumont, "Differential fault intensity analysis," in *Proc. of FDTC'14*, pp. 34–43, 2014.

[8] Y. Li, K. Sakiyama, S. Gomisawa, T. Fukunaga, J. Takahashi, and K. Ohta, "Fault sensitivity analysis," in *Proc. of CHES'10*, pp. 320–334, 2010.

[9] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Morgan Kaufmann, 2010.

[10] S. Patranabis, A. Chakraborty, P. H. Nguyen, and D. Mukhopadhyay, "A biased fault attack on the time redundancy countermeasure for AES," in *Proc. of COSADE'15*, pp. 189–203, 2015.

[11] S. Endo, N. Homma, Y.-i. Hayashi, J. Takahashi, H. Fuji, and T. Aoki, "A multiple-fault injection attack by adaptive timing control under black-box conditions and a countermeasure," in *in Proc. COSADE'14*, pp. 214–228, Springer, 2014.

[12] K. Lemke-Rust and C. Paar, "An adversarial model for fault analysis against low-cost cryptographic devices," in *Proc. of FDTC'06*, pp. 131–143, 2006.

[13] O. Kömmerling and M. G. Kuhn, "Design principles for tamper-resistant smartcard processors," *USENIX Workshop on Smartcard Technology*, pp. 9–20, 1999.

[14] M. Milenković, A. Milenković, and E. Jovanov, "Hardware support for code integrity in embedded processors," in *Proc. of CASES'05*, pp. 55–65, 2005.

[15] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, *et al.*, "Razor: circuit-level correction of timing errors for low-power operation," *IEEE Micro*, no. 6, pp. 10–20, 2004.

[16] N. Selmane, S. Bhasin, S. Guilley, T. Graba, and J.-L. Danger, "WDDL is protected against setup time violation attacks," in *Proc. of FDTC'09*, pp. 73–83, 2009.

[17] L. Zussa, A. Dehbaoui, K. Tobich, J.-M. Dutertre, P. Maurine, L. Guillaume-Sage, J. Clediere, and A. Tria, "Efficiency of a glitch detector against electromagnetic fault injection," in *Proc. of DATE'14*, pp. 1–6, 2014.

[18] X. Guo, D. Mukhopadhyay, C. Jin, and R. Karri, "Security analysis of concurrent error detection against differential fault analysis," *Journal of Cryptographic Engineering*, pp. 1–17, 2014.

[19] T. Sato and Y. Kunitake, "A simple flip-flop circuit for typical-case designs for DFM," in *Proc. of ISQED'07*, pp. 539–544, 2007.

[20] S. Gregori, A. Cabrini, O. Khouri, and G. Torelli, "On-chip error correcting techniques for new-generation flash memories," *Proceedings of the IEEE*, vol. 91, no. 4, pp. 602–616, 2003.

[21] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures against fault attacks on software implemented AES: effectiveness and cost," in *Proc of WESS'10*, pp. 1–10, 2010.

[22] "LEON3 processor." http://www.gaisler.com/index.php/products/processors/leon3. [Online; accessed 09-May-2016].

[23] Q. Liu and S. S. Sapatnekar, "Capturing post-silicon variations using a representative critical path," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, no. 2, pp. 211–222, 2010.

[24] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, "Formal verification of a software countermeasure against instruction skip attacks," *Journal of Cryptographic Engineering*, vol. 4, no. 3, pp. 145–156, 2014.

[25] R. Kannavara and N. G. Bourbakis, "Surveying secure processors," *Potentials, IEEE*, vol. 28, no. 1, pp. 28–34, 2009.

[26] G. E. Suh et al., "AEGIS: architecture for tamper-evident and tamper-resistant processing," in *Proc. of ICS'03*, pp. 160–171, 2003.

[27] "SAKURA hardware security project." http://satoh.cs.uec.ac.jp/SAKURA/index.html. [Online; accessed 10-May-2016].