IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 2, pp. 212–240. DOI:10.46586/tches.v2023.i2.212-240

FaultMeter: Quantitative Fault Attack Assessment of Block Cipher Software

Keerthi K and Chester Rebeiro

Indian Institute of Technology Madras, India
{keerthi,chester}@cse.iitm.ac.in

Abstract. Fault attacks are a potent class of physical attacks that exploit a fault injected during device operation to steal secret keys from a cryptographic device. The success of a fault attack depends intricately on (a) the cryptographic properties of the cipher, (b) the program structure, and (c) the underlying hardware architecture. While there are several tools that automate the process of fault attack evaluation, none of them consider all three influencing aspects.

This paper proposes a framework called `FaultMeter` that builds on the state-of-art by not just identifying fault vulnerable locations in a block cipher software, but also providing a quantification for each vulnerable location. The quantification provides a probability that an injected fault can be successfully exploited. It takes into consideration the cryptographic properties of the cipher, structure of the implementation, and the underlying Instruction Set Architecture's (ISA) susceptibility to faults. We demonstrate an application of `FaultMeter` to automatically insert optimal amounts of countermeasures in a program to meet the user's security requirements while minimizing overheads. We demonstrate the versatility of the `FaultMeter` framework by evaluating five cipher implementations on multiple hardware platforms, namely, ARM (32 and 64 bit), RISC-V (32 and 64 bit), TI MSP-430 (16-bit) and Intel x86 (64-bit).

Keywords: Fault Attack, Automatic Fault Attack Evaluation, Quantification Countermeasures

1 Introduction

Cipher implementations are highly vulnerable to a potent class of physical attacks known as fault attacks. These attacks exploit faults injected during the cipher's execution, causing an error that propagates to the output. The flawed output, called *faulty ciphertext*, is then used to extract the secret key using differential, impossible differential, or algebraic properties of the cipher. Several block ciphers including the AES [TMA11], PRESENT [BEG13], Simon [TBM14b], Speck [HZFW15], and CLEFIA [AM13] are vulnerable to fault attacks. A single precisely injected fault in any of these ciphers is sufficient to substantially reduce the entropy of its key.

For software implementations of block ciphers, faults are typically injected in memory components such as registers, flash memory [CN10], SRAM [ZZJ+20], and DRAM [KGGY20]. Alternatively, faults are injected in the processor pipeline, for instance, causing instructions to be skipped [KSV13]. Most faults are injected using glitches in the voltage or clock source of the device or by using optical or electromagnetic radiation. Other faults are injected by exploiting physical properties and the structure of device components. For example, Rowhammer [KDK+14], RAM-Jam [ATG+19], SPOILER [IMB+19], RAMBleed [KGGY20], TRRespass [FVH+20] and Blacksmith [JvdVF+22] utilize the physical properties of memory to inject faults.

Received: 2022-10-15 Accepted: 2022-12-15 Published: 2023-03-06

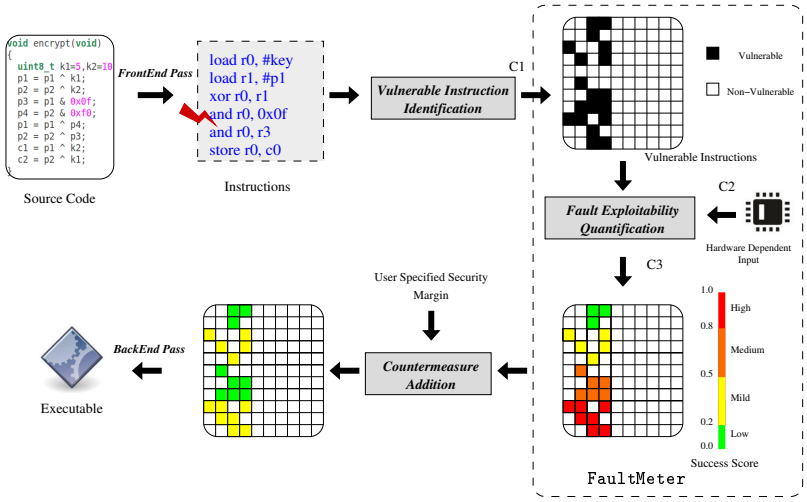


Figure 1: The `FaultMeter` framework, to quantify the vulnerability of block cipher implementations. It has two main modules. The Vulnerable Instruction Identification Module detects vulnerable instructions, while the Fault Exploitability Quantification module quantifies the exploitability. The figure also shows an application of `FaultMeter` to automatically insert countermeasures into the implementation. The countermeasures are tuned to meet the user's security requirement.

While there are a large number of locations in a program where faults can be injected during its execution, only a small portion of these faults are exploitable. There are three requirements that a fault should satisfy to be successfully exploited.

- **Fault should impact vulnerable operations.** The fault should target the small subset of vulnerable operations in the cipher. For instance, prior works such as [KRR+20] show that only 4.98% of instructions in an AES implementation[1] are vulnerable. Faults injected elsewhere in the program cannot be exploited.
- **Corrupt instruction output.** Most fault attacks require that the fault modifies an instruction output and not halt execution. For example, a fault that alters an instruction's opcode can lead to an illegal instruction exception causing the program to terminate. Such a fault is not exploitable because it does not provide the attacker with the faulty ciphertext, which is essential for the attack.
- **Propagate to the output.** The fault at the target instruction should propagate to the ciphertext. This may not always be the case. For instance, if register `r2` is affected by a fault in the instruction `mul r2, r1`, the fault will not propagate if `r1 = 0x0`. Such a fault may not be exploitable[2].

The success of a fault attack is intricately dependent on the cipher algorithm, its implementation, and the underlying hardware. While the vulnerable operations depend on the cipher algorithm, corrupting an instruction output depends considerably on the Instruction Set Architecture (ISA) of the microprocessor, and propagating the fault to the ciphertext depends on the program structure. Understanding the extent to which these factors influence a fault attack would help develop metrics that can be used to compare and evaluate implementations for fault attack resistance. It would help in designing efficient countermeasures and tools that could automatically patch software for fault attack vulnerabilities.

[1] OpenSSL (version 3.0) implementation with 1 lookup table occupying 2048 bytes: https://github.com/openssl/openssl/blob/master/crypto/aes/aes_x86core.c

[2] This paper does not consider SIFA [DEK+18] which target faults that do not affect the output. Incorporation of SIFA in `FaultMeter` is left as future work

The current practice to evaluate fault attack resistance is by empirically subjecting the device to faults. Unfortunately, this is largely a manual process, requiring expensive instruments and considerable time. Recently researchers have introduced tools to automate the process of finding vulnerable instructions in cipher implementations. While tools such as [ABMP13, KRR+20] work on C implementations, [BHL18, HBZL19] operate on assembly code. Many of these tools fail to assess the extent to which an injected fault is exploitable. The output of these tools is binary: either an instruction is exploitable or it is not. Few works like, TADA [HBZL19] additionally identify attacks from vulnerable fault instructions. Most of the tools fail to consider cryptographic properties of the cipher [TMA11, CN10, DFL11] that can significantly impact the attack success. Further, most tools do not consider the impact of the underlying hardware in the fault attack.

Our Contributions. In this paper, we introduce an automated framework, called `FaultMeter`, that not just identifies vulnerable instructions in ciphers but quantitatively evaluates the success with which an injected fault can be transformed into an attack. Figure 1 depicts the flow of `FaultMeter`. Given a block cipher implementation, `FaultMeter` **(C1)** first uses existing tools such as [KRR+20], to identify the vulnerable instructions taking into consideration the cipher's cryptographic properties. Only faults injected in any of these vulnerable instructions are exploitable and can be used to retrieve information about the cipher's key. **(C2)** Then, for each vulnerable instruction, it quantifies the probability that an injected fault can corrupt the instruction's output. To perform this quantification, it captures the sensitivity of the underlying microprocessor's instruction opcodes and data to faults and quantifies the probability of successful instruction skips. **(C3)** It then performs static analysis to capture the probability with which an injected fault can propagate to the program output resulting in a faulty ciphertext. Steps (C2) and (C3) are performed by the Fault Exploitability Quantification module in `FaultMeter`. The output of this module is a success score for every vulnerable instruction. The success score quantifies the fault attack vulnerability of an instruction. A fault injected in an instruction with a high success score is more likely to yield a successful fault attack compared to a fault in an instruction with a low success score. This quantification is different from contemporary fault attack tools [BHL18, HBZL19, KRR+20] that provide the list of vulnerable instructions from cipher implementation. We demonstrate the application of `FaultMeter` in a compiler that generates executables with directed countermeasures automatically inserted to meet user-specified security margins. In addition to the input program, the compiler accepts a user input that specifies the desired security level. The compiler uses the success score from `FaultMeter` to quantify the fault attack threat in the program at an instruction granularity, then applies appropriate countermeasures to minimize performance overheads while adhering to the desired security level. Our contributions can be summarized as follows.

- We present `FaultMeter`, the first automated framework that can quantify the fault attack vulnerability of instructions in block cipher implementations. The vulnerability not just depends on the cipher algorithm and its crypto-properties, but the implementation as well as the underlying hardware.

- We study how the processor's Instruction Set Architectures (ISA) have an influence on a fault attack. For the study, we consider six microprocessors, namely, Intel x86 (64 bit), RISC-V (32-bit and 64-bit), ARM (32-bit and 64-bit), and TI's MSP-430 (16-bit).This results in interesting observations, such as TI's MSP-430 and Intel x86 having highest success score compared to other processors, and RISC-V(32-bit) having the lowest success score.

- To demonstrate that the fault attack vulnerability depends on the implementation, we consider three AES-128 implementations that include a lightweight implementation,

a T-table implementation, and a bitsliced implementation [RSD06]. We also evaluate two other cipher implementations, CLEFIA-128 and CAMELLIA-128, to demonstrate the scalability of `FaultMeter` across ciphers.

- We present an application of `FaultMeter` by using it in a compiler that can automatically tradeoff between security and performance to meet the user's security requirements.

Structure of the Paper. The paper is organized as follows: Section 2 provides the necessary background. Section 3 includes the recent works for automated fault vulnerability detection tools. Section 4 discusses the requirements for a successful fault attack and the `FaultMeter` framework, expanding on steps C2 and C3. Section 5 describes the implementation and evaluates the `FaultMeter` framework on different block cipher implementations and processors. Section 6 provides an application of `FaultMeter` framework, where it used to automatically insert countermeasures based on the user's security requirement. Section 7 provides the limitations of `FaultMeter`. Section 8 includes the discussion and future work. Section 9 concludes the paper.

2 Background

2.1 Fault Attacks

A fault attack has two phases. In the first phase, the attacker injects a fault during the cipher execution that corrupts the output of an operation, causing an error that propagates to the output, resulting in a faulty ciphertext. In the second phase, the attacker uses the faulty ciphertext to reduce the entropy of the secret key. The cipher algorithm critically determines the success of a fault attack. For example, AES is far more vulnerable to fault attacks compared to ciphers like CLEFIA and PRESENT. It takes a single fault during an AES execution to completely reveal its secret key [TMA11], while 8 [AM13] and 18 [BEG13] faults are needed for CLEFIA and PRESENT, respectively. Within a cipher, too, not all operations are equally vulnerable. For example, a fault in the 8-th round of AES reveals the entire secret key, while a fault in the 9-th round only reveals 32-bits of the key. Faults injected before the 7-th round are not exploitable.

Implementations of the cipher also influence the fault attack surface. Keerthi *et al.* [KRR+20] for instance, showed that the percentage of vulnerable instructions in seven different implementations of AES-128 varies from 4.2% to 11.4%. A fault in any of these vulnerable instructions can potentially be exploited. In this paper, we provide quantification for the success of a fault attack. We show how the exploitability of a fault injected in a vulnerable instruction can depend not just on the cipher algorithm and the implementation but also on the underlying Instruction Set Architecture of the microprocessor.

2.2 Countermeasures for Fault Attacks

Several countermeasures [BG13, GST12, LRT12, GK13, TBM14a, ML08] have been introduced to protect cipher implementations from fault attacks. Most countermeasures detect fault injection using techniques like redundancy, parity, or error correction codes [BBK+03, GK12, KWMK02, KKG03, WKKG04]. If a fault is detected, the countermeasure either aborts the encryption operation or masks the output of the operation to make the fault unexploitable. Other countermeasures makes use of infection techniques that diffuse faults, making them unexploitable [LRT12, GST12, BG13, TBM14a]. Naïvely inserting either of these countermeasures has considerable overheads, often degrading performance by over two times. In the paper we show how `FaultMeter` can be used to automatically insert targeted countermeasures during compilation. The countermeasures are tuned to meet the application's security and performance requirements.

Table 1: Comparison with the state-of-the-art fault attack automation tools. FaultMeter is the only tool that works at the implementation level and considers the cryptographic properties, different hardware to quantify the vulnerability.

Tools	Output	
	Detect	Quantify
Algorithm		
XFC [KRH17]	✓	✓
ExpFault [SMD18]	✓	✗
FaultDroid [RRHB21]	✓	✓
Hardware		
SOLOMON [SSR+20]	✓	✗
SoFi [WLR+21]	✓	✗
Verfi [AWMN20]	✓	✗
FIVER [RSS+21]	✓	✓
[DPdC+15]	✓	✓

Tools	Output		Input Type	Crypto Properties	Hardware Fault Analysis
	Detect	Quantify			
Software					
DATAC [BHL18]	✓	✗	Assembly	N/A	N/A
TADA [HBZL19]	✓	✓	Assembly	N/A	N/A
FEDS [KRR+20]	✓	✗	Source Code	✓	N/A
Lazart [PMPD14]	✓	✓	IR	N/A	N/A
ARMORY [HSP21]	✓	✗	Assembly	N/A	ARM
[RPL+14]	✓	✓	Source Code and Assembly	N/A	N/A
[RBLC15]	✓	✓	IR	N/A	N/A
[LFB+21]	✓	✓	IR	N/A	N/A
[BHE+19]	✓	✓	Binary	N/A	ARM
[GJL20]	✓	✓	IR	N/A	N/A
[HKR+15]	✓	✓	Assembly	N/A	ARM
FaultMeter (This Work)	✓	✓	Source Code and IR Analysis	✓	ARM (32/64) RISC-V (32/64) TI-MSP430 Intel x86(64bit)

2.3 Intermediate Representation (IR)

The LLVM compiler converts the high-level representation to machine code using different compiler passes. The transformation pass converts the high-level representation to *Intermediate Representation* (IR) instructions. FaultMeter uses LLVM's generated IR instructions for the analysis. These instructions are represented in the Static Single Assignment form as defined below.

Definition 1. [Static Single Assignment] *Static Single Assignment (SSA), is a format for program representation, where variables in every assignment are used only once* [RWZ88].

Below are a few examples of IR instructions.

%t0 = load %i8, i8 ∗ %p0	:	read from memory pointed by %p0 to variable %t0
%xor = xor i8 %x0, %x1	:	%xor ← %x0 ⊕ %x1 (works on 8 bit integers (i8))
store i8 %xor, i8 ∗ %s0	:	write %xor to memory pointed by %s0
%a = alloca i8	:	allocate a 8 bit integer on the stack
br i1 %cond, label %L1, label %L2	:	if condition %cond is true jump to %L1, else %L2

3 Related Work

3.1 Automated Fault Attack Vulnerability Detection

Evaluating the security of cipher implementations against fault attacks is a tedious and manual task. Recently a few tools were introduced to automate the fault attack assessment process. Tools like [KRH17, SKMD17, RRHB21] work at the algorithm level to determine vulnerable operations in a cipher and compute the attack complexity. These tools work directly on the algorithm and do not consider implementation aspects, which can significantly influence the attack success.

Another class of tools [SSR+20, AWMN20, GJL20, RSS+21, WLR+21] work on hardware implementations of ciphers, typically taking RTL or netlist of the design as input to detect fault vulnerable gates. Any fault injected in these vulnerable gates can result in a successful fault attack. FIVER [RSS+21], for instance, determines effective and ineffective faults on a gate-level netlist while [GJL20] bridges the gap between hardware and software faults.

The third set of tools [BHL18, HBZL19, HSP21, KRR+20, PMPD14, RPL+14, RBLC15, LFB+21, HKR+15, BHE+19] detects fault attack vulnerable locations in software implementations. Software tools work either at the assembly level [BHL18, HBZL19, HSP21, HKR+15], at the source code [KRR+20] or at a compiler-generated intermediate representation of the program [PMPD14, RPL+14, RBLC15, LFB+21]. Some of these tools [HBZL19, PMPD14,

Table 2: Comparison with automated fault attack countermeasure insertion tools. Unlike the other tools, only `FaultMeter` can work with optimized code and tune the countermeasure to trade off between performance and security.

Tools	Work Optimized Code	Provide Security × Performance Trade offs	Security Estimate
SAFARI [RRHB20]	✗	✓	✓
FEDS [KRR+20]	✓	✗	✗
`FaultMeter`(This work)	✓	✓	✓

RPL+14, RBLC15, LFB+21] quantify the vulnerability and can determine the attack success based on specific fault models. They, however, do not consider cryptographic properties of the cipher, such as its differential [TMA11] impossible differential [DFL11], and algebraic properties [CN10]. A cipher's cryptographic properties significantly abet fault attacks. `FaultMeter`, on the other hand, builds on existing tools and can evaluate cipher implementations considering complex cipher properties.

The underlying Instruction Set Architecture of the platform greatly affects fault induction. Except for [HSP21], [BHE+19], and [HKR+15], none of the other software tools take into consideration the impact of the fault in the underlying processor. While [HSP21, BHE+19, HKR+15] evaluates faults in the ARM processor, `FaultMeter` considers a range of processors from 16-bit to 64-bit, RISC and CISC architectures. This analysis brings out interesting results, such as some ISAs are more vulnerable to fault attacks compared to others. Further, `FaultMeter` computes the probability that a disturbed instruction output can propagate to the ciphertext. Such quantification helps to customize countermeasures as per the user's requirement.

3.2 Automated Fault Attack Countermeasure Insertion

Automatic countermeasure insertion was first proposed in SAFARI [RRHB20], which synthesized hardware and software programs based on a high-level specification of the cipher algorithm and a user-defined security margin. While the generated programs had fault-attack countermeasures inserted automatically, the programs were generic and could not be optimized to suit specific platforms and requirements. For example, SAFARI would synthesize the same code for an IoT edge device as well as a server.

Rather than synthesizing countermeasures like SAFARI, FEDS [KRR+20] can insert countermeasures in any cipher implementation, thus supporting optimized codes in hand-written assembly. However, FEDS cannot tune countermeasures to meet the user's security requirements. For example, a user developing a highly sensitive application such as an electronic voting machine would require high security guarantees and would not mind the additional performance overheads. On the other hand, less security critical applications, such as a smart-clock, would value performance and energy consumption over security. FEDS would be ignorant of the difference in requirements and provides the same countermeasures for both applications.

Similar to FEDS, `FaultMeter` can produce highly optimized implementations of block ciphers, however unlike FEDS, it can support countermeasures that can be added automatically based on the user's security requirements. Thus `FaultMeter` would likely provide a stronger countermeasure for the electronic voting machine and weaker countermeasures for the smart clock. The weaker countermeasures would result in lower performance overheads and energy requirements. A critical aspect in `FaultMeter` that enables such application-specific operations is the ability to quantify the success of converting an injected fault into an attack.

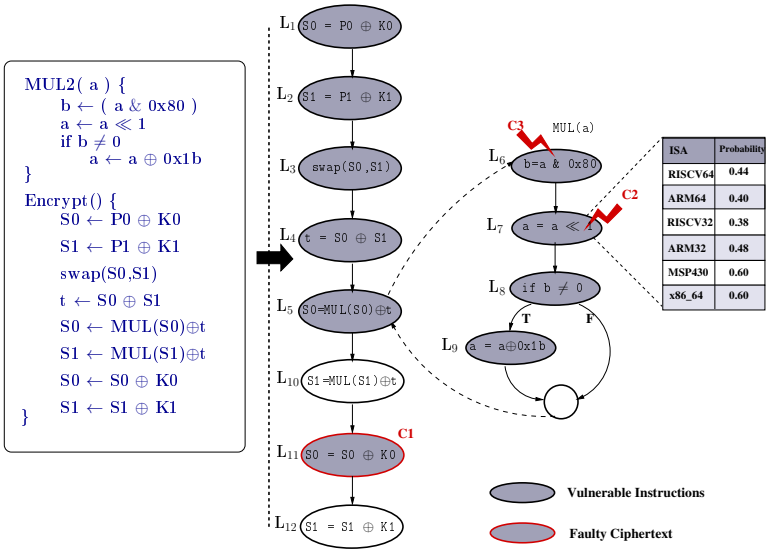


Figure 2: The pseudo-code of a toy cipher, where `(P0,P1)` and `(K0,K1)` are pairs of plaintext and secret key bytes, respectively. The output of the cipher is the ciphertext bytes `(S0, S1)`. The figure also shows the Control Flow Graph (CFG) for the code with vulnerable nodes with respect to K0 highlighted. The vertices in the CFG represent the operations of the cipher, and the edges represent the program control flow. A fault in vulnerable instructions can potentially alter its output. The probability that this happens intricately depends on the underlying hardware.

4 Quantifying the Success of Injected Fault

The probability that an injected fault can be exploited to create a successful attack depends on the (1) cipher algorithm, (2) its implementation, and (3) the underlying hardware.

`FaultMeter` uses FEDS, to detect instructions in a program that are vulnerable to fault attacks. In this section, we provide a quantification of the vulnerability that can be used to distinguish between less vulnerable and more vulnerable fault injections. The quantification depends considerably on the underlying hardware architecture and program structure. In this section, we provide the basis for the quantification.

Fault Model. Fault injection can either modify the data flow or control flow of the program. We consider a single transient fault injected in the device during the cipher's execution. The fault either corrupts an instruction or the associated data during the program execution. Alternatively, the fault can be inserted in the program counter altering the sequence of instructions executed, *i.e.* the control flow. After the fault is injected, it propagates towards the output. The fault model considered is a fault injected in code, data, or program counter to randomly alter it.

Requirements for a fault attack exploit. To exploit the fault, requires three conditions to be satisfied. We discuss these requirements using a toy cipher shown in Figure 2.

C1. [**Fault in vulnerable instructions**] Only faults injected in certain locations can yield a successful attack. For example, only faults inserted in the shaded nodes in the Control Flow Graph (CFG) in Figure 2 can be used to recover key `K0`. These are the vulnerable instructions with respect to `K0`. Faults injected anywhere else in the program do not yield any information about `K0`. `FaultMeter` identifies these vulnerable instructions with the help of existing tools in the Vulnerable Instruction Identification module (refer Figure 1).

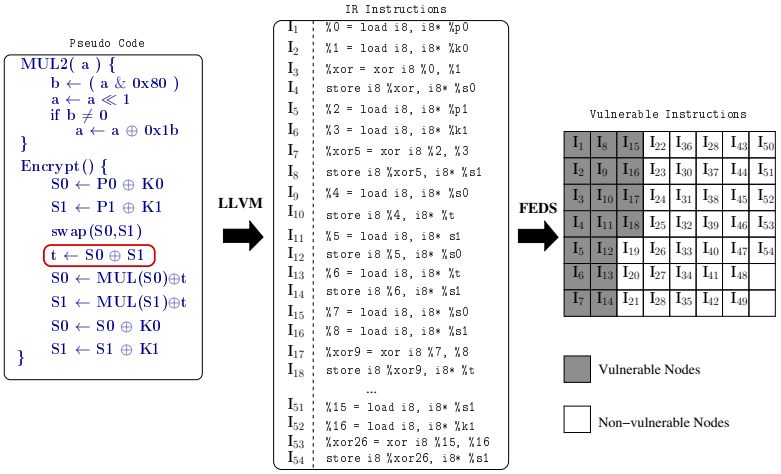


Figure 3: Vulnerable Instruction Identification. The Intermediate Representation obtained from the LLVM compiler for the pseudo code is used by FEDS to identify vulnerable instructions.

C2. **[Corrupting the output of vulnerable instructions]** When a fault is injected in the instruction, it causes bits in the opcode to toggle. Similarly, faults injected in data can change the values stored in memory or registers, and faults injected in the program counter can alter the sequence of instructions executed. However, not all faults would result in a wrong output. For example, the fault may result in an undefined instruction or get interpreted as another instruction. In the former case, the undefined instruction would result in an exception, causing the program to terminate. Such faults cannot be exploited because the faulty ciphertext is not available. In the latter case, there is a chance that the output of the instruction is not affected by the fault. For example, if a fault in the `swap(S0, S1)` instruction transforms it to `swap(S1, S0)`, the output of the instruction is unaffected. The opcode encoding significantly impacts the probability that an instruction is corrupted by a fault. Figure 2 shows the probability that a randomly injected fault corrupts the output of the instruction $\texttt{a} = \texttt{a} \ll \texttt{1}$ in the six different platforms namely, ARM (32-bit and 64-bit), RISC-V (32-bit and 64-bit), TI's MSP-430 (16-bit) and Intel x86 (64-bit) microcontroller. `FaultMeter` learns these probabilities offline for each microprocessor. Section 4.2 provides further details about how these probabilities are computed.

C3. **[Fault propagation to the ciphertext.]** The structure of the program can influence if the fault propagates to the ciphertext. For example, consider the instruction $\texttt{L}_6$ (Figure 2) resulting in a non-zero value for `b`. If a fault induced in this instruction changes the value of the byte `b` to another non-zero value, then the fault will not propagate to the ciphertext byte due to the condition statement in $\texttt{L}_8$. Thus, only a fault that changes `b` to zero would propagate to the output. `FaultMeter` uses the Fault Exploitable Quantification module (refer Figure 1) to compute the fault propagation probabilities. Section 4.3 provides more details.

4.1 Identifying Vulnerable Instructions in an implementation (C1)

Only faults injected in vulnerable instructions can yield a successful attack. The percentage of vulnerable instructions varies based on the algorithm as well as the implementation characteristics. Recently, researchers introduced tools [BHL18, HBZL19, KRR+20, HSP21] that could automatically identify vulnerable instructions in implementations. The tools

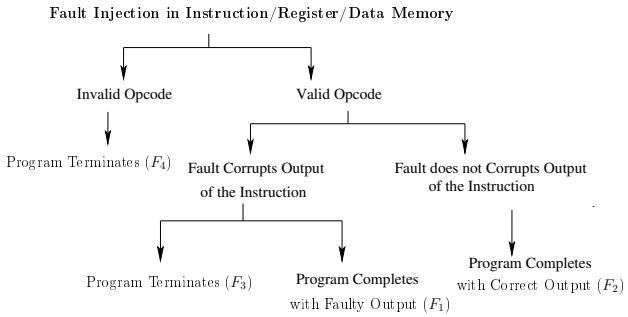


Figure 4: Manifestation of a fault injected in instruction or data may either alter or leave unaltered the instruction output. It may also result in a program termination.

take as input block cipher implementation either in the form of assembly or in a high-level language like C and outputs the list of instructions that are vulnerable to fault attacks. Typically, each tool handles a subset of fault attacks. For example, [KRR+20] can detect regions of an implementation that are vulnerable to Differential Fault Analysis [TMA11] and Impossible Differential Fault Analysis [DFL11]. Similarly, the DATAC [BHL18] tool identifies locations that are vulnerable to instruction skip fault injections.

The first stage of `FaultMeter` (Figure 1) uses one of these tools to determine vulnerable instructions in a program. Only a few instructions in a cipher implementation are exploitable by a fault attack. In this paper, we make use of the open-source tool FEDS[3] [KRR+20] that uses the LLVM Intermediate Representation (IR)[4] of the program to identify vulnerable program instructions. FEDS takes the source code of a block cipher as input and outputs the list of exploitable instructions in the implementation by mapping the known vulnerable instructions as shown in Figure 3. A fault in any of these 'vulnerable instructions' is exploitable.

The input to FEDS is a compiler generated Intermediate Representation (IR) obtained from the LLVM compiler. FEDS converts the IR to a Control Flow Graph (CFG), where the instructions form the vertices of the graph and edges are added based on the program flow. To perform the analysis, FEDS performs backward dataflow analysis on the CFG to identify the vulnerable nodes in the graph. Figure 3 depicts the list of vulnerable instructions ($\texttt{I}_1$ to $\texttt{I}_{18}$) that can induce a fault in the output of the operation $\texttt{t} \leftarrow \texttt{S0} \oplus \texttt{S1}$.

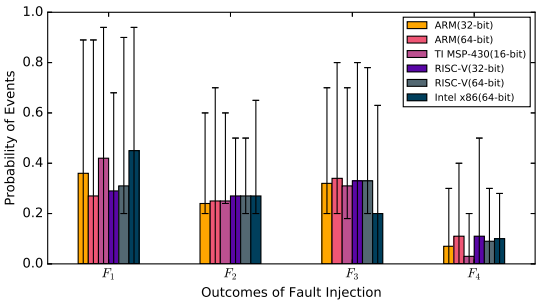
The result from FEDS is binary. Either an instruction is vulnerable, or it is not. For each vulnerable instruction identified by FEDS, `FaultMeter` provides a score between 0 and 1. A score close to 1 indicates that a fault injected in that instruction is more likely to result in a successful attack. For non-vulnerable instructions (not identified by FEDS), `FaultMeter` results a score of 0.

4.2 Quantifying the probability of fault-induced instruction corruption (C2)

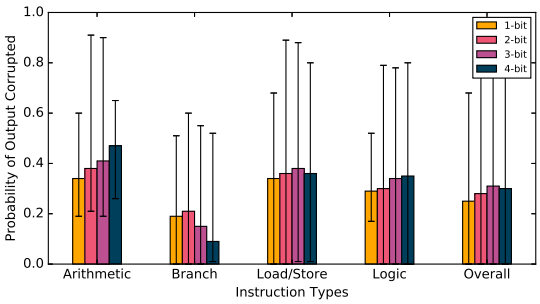
When a single fault is transiently injected during an instruction execution, it can manifest by either altering or leaving unaltered the instruction output, or terminating the program, as shown in Figure 4. A fault due to the altered instruction output may propagate, resulting in a faulty ciphertext. We classify the fault manifestations into four classes:

F_1. **Fault is activated**: The induced fault alters the instruction execution resulting in an incorrect instruction output.

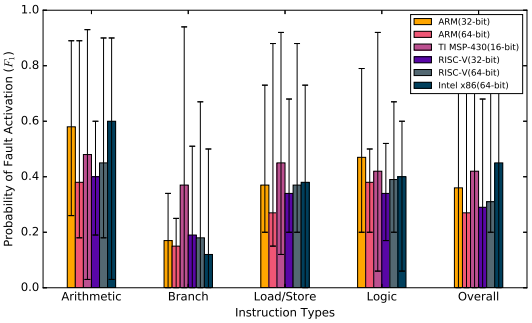
[3] https://bitbucket.org/casl/faultanalysis/src/master/FEDS/
[4] https://llvm.org/



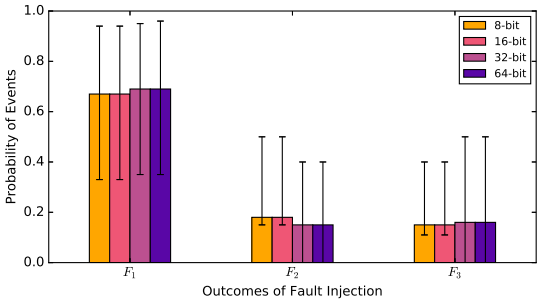
(a) Outcome of a fault injection in instructions on different architectures.



(b) Probability of producing an incorrect output for different fault models in RISC-V (32-bit)



(c) Probability of producing incorrect output on different instruction classes.



(d) Probability of events when a transient fault is injected in a register or data memory operand

Figure 5: Probability on different microprocessors of instructions producing incorrect outputs when a transient fault is injected during execution.

F_2. **Fault is not activated**: The induced fault alters the instruction execution but does not change the instruction output. For example, a fault in the instruction `swap(S0, S1)` which swaps the contents of registers `S0` and `S1`.

F_3&F_4. **Program is terminated**: The induced fault leads to an illegal operation causing the program to terminate.

The faults in set F_2, F_3 and F_4 cannot induce a successful fault attack, as the outcomes do not provide the faulty ciphertext that is necessary to carry out the attacks. In this section, we quantify the probability that an injected fault leads to a faulty output. We consider three types of fault injections. First, we consider faults injected in an instruction affecting the opcodes. Second, faults injected in operands (for example registers), and third, faults injected in the program counter affecting the flow of the program. Each subsection considers one of these fault injections.

4.2.1 Fault Injection in instructions

When an injected fault changes bits in an opcode, it can result in a valid or invalid instruction (see Figure 4). An invalid instruction opcode results in program termination (F_4), while a valid instruction can have any of the remaining three (*i.e.* F_1, F_2, or F_3) outcomes. The probability of these outcomes depends not just on the type of instruction but also on the encoding. They are thus unique to each Instruction Set Architecture. To understand these probabilities, we consider six microprocessors, namely, Intel x86 (64-bit),

TI's MSP-430 (16-bit)[5], ARM (32-bit and 64-bit)[6]and RISC-V(32-bit and 64-bit)[7], to identify the reliance of fault injection on the underlying architecture. For each of these microprocessors, we generate random programs[8], cross compile and execute the binary multiple times in a simulator[9]. In each execution, faults are injected in an instruction using simulation tools, such as by modifying the instruction memory and then observing the instruction output. The result of the fault falls in one of the four classes *i.e.* F_1, F_2, F_3, or F_4. Figure 5 shows the results from the simulation. These probabilities were computed based on 50 randomly generated programs, with over 25,000 instructions and about a million injected faults in each platform. Figure 5a shows the probability of each fault class on the six microprocessors. Of the four classes, the probability that the fault is activated, *i.e.* F_1, is interesting for evaluating fault attacks. Figure 5b shows the impact of 1-bit, 2-bit, 3-bit, and 4-bit fault injections on an instruction. We observe that the probability of F_1 occurring, does not vary much based on the fault model. Thus, to simplify evaluation, we consider only 1-bit fault injections.

Instruction encoding and fault outcome. The instruction encoding plays a critical role in the outcome when a fault is injected. Reduced Instruction Set Computing processors have fixed length instructions. Most instructions have two components: an opcode and operands. The opcode defines if the instruction is an arithmetic, logic, branch, or a memory operation. Instructions have zero or more operands. The operands, if present, in an instruction can hold registers, immediate values, or memory addresses. In the following, we evaluate the outcome of faults injected in the opcodes and operands.

Fault resulting in program termination due to invalid opcode (F_4). A fault in the opcode can either change the instruction or result in an invalid opcode. This depends on the density of instructions in the instruction set. An instruction set is considered dense if a fault injected in the opcode transforms it to another valid instruction with a significant probability. Among the five processors considered, TI MSP-430 has the highest instruction density. Thus for TI MSP-430, the probability of program termination due to an invalid opcode (F_4) is the lowest. For instance, the most significant four bits of double operand instructions in TI MSP-430[5] holds the opcode of the instruction. There are 15 valid opcodes and one invalid opcode. Thus, a fault in any of these four bits is more likely to change the opcode to a valid opcode than an invalid one.

Among the 32-bit processors considered, RISC-V has a lower instruction density compared to ARM. This is because RISC-V has considerably large number of unused opcodes compared to ARM, hence low density and higher chances that F_4 occurs. RISC-V 64-bit has a higher instruction density compared to the 32-bit variant. This is because of the additional instructions supported in the 64-bit and not 32-bit. This marginally increases instruction density, lowering the chances that F_4 occurs.

Fault resulting in a valid opcode but the program terminates (F_3). In most cases, these appear due to faults in the operand of branch and memory instructions. For example, a fault changes the branch offsets stored as part of a branch instruction leading to an illegal branch target. Similarly, faults may modify the address of load/store instructions leading to an invalid memory operation. Arithmetic and logic instructions can also experience these events. For example, the fault changes the destination register to either stack pointer or program counter, potentially setting illegal values to these registers. In a few cases, faults in the opcode of instructions can also trigger the event F_3. For example, a fault changing an arithmetic/logic opcode to a branch instruction.

[5] https://www.ti.com/tool/MSP430-GCC-OPENSOURCE
[6] https://developer.arm.com/architectures/instruction-sets
[7] https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf
[8] Random programs generated using Csmith https://embed.cs.utah.edu/csmith/
[9] QEMU (https://www.qemu.org/) is used for ARM and RISC-V, GDB[5] is used for MSP-430 and x86

Faults that result in program completion (events F_1 and F_2). A fault that results in program completion can either produce a correct output or a faulty output. For such faults, the correct output is produced in 25% of the cases on average across all processors. Some examples where the output of the program does not change in spite of fault injection are provided here:

- ARM supports conditional execution of instructions, where an instruction is executed only if certain conditional flags are set. We found that in many cases, a fault injected in the condition bit present in the instruction did not alter the output.
- Often, multiple registers may hold the same data. Few bits in the instructions specify the operands to be used for the source and destination registers. A fault that changes the source register to another holding the same data would not affect the output.
- Certain faults were observed to change the arithmetic and logic opcodes in a way that does not alter the outputs. For example, a fault that changes the opcode for `add` to the signed equivalent `adds` in ARM may not always alter the output.
- Compare operations have outputs of True or False; hence with high probability, the output remains the same even after fault injection.
- Faults that alter the memory address of load instructions in a way that the new address holds the same data as the original do not alter the program output.

4.2.2 Fault in data memory and registers

Faults injected in data memory, or registers can influence the output of an instruction. With respect to Figure 4, a disturbance in data memory or registers can cause in instruction to provide a wrong output (F_1) or cause program termination (F_3). In some cases, the fault would go unaffected (F_2). However, such faults in data or registers cannot result in an invalid opcode (*i.e.* F_4). The probability of the events F_1, F_2, and F_3 depend not just on the type of instruction but also on the width of the registers.

To understand the probabilities of these events, we consider faults injected in 8-bit, 16-bit, 32-bit, and 64-bit registers. For each register size, we generate random C programs, compile, simulate random fault injections in registers, and observe the outputs of each instruction. The event F_3 is observed when the fault modified registers are used to hold addresses for branch, load, or store instructions. The modified registers result in invalid instructions causing program termination.

In arithmetic and logic instructions, these faults result in either a wrong output (*i.e.* F_1), and in some cases do not affect the output (*i.e.* F_2). For example, in a conditional branch such as '`if (a < b)`' a fault in either `a` or `b` does not alter the output in half of the executions. Arithmetic instructions like multiplication `mul a,b` do not alter the output if one of the operands is zero. Similarly, in 32-bit platforms, the output of `and a,b` is not altered when one of the operands is $0\times$`FFFFFFFF`.

4.2.3 Faults in the Program Counter

Unlike faults in instruction and data, the effect of a fault in the Program Counter (PC) is influenced by the control flow graph of the program. If the fault modifies the PC in such a way that the new address falls outside the control flow graph, *i.e.* an address outside the program, then the program is likely to terminate (F_3) (refer Figure 4). On the other hand, if the fault modifies the PC such that the new address lies in the control flow graph, then either events F_1 or F_2 are likely. These faults either skip instructions or repeat the execution of instructions. The former generally occurs when the fault causes the PC to be incremented, while the latter generally occurs when the fault decrements the PC. Not all

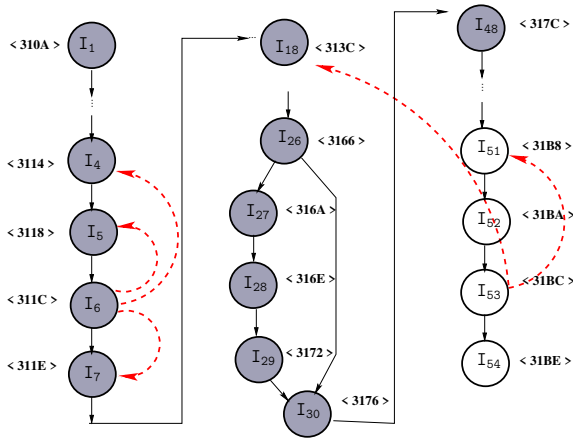


Figure 6: Control Flow Graph (CFG) with addresses ($\langle \cdot \rangle$) for the intermediate representation of the pseudo-code given in Figure 2. The addresses correspond to an implementation on TI's MSP-430, while the shaded nodes are vulnerable. A fault in the program counter can skip instructions or repeat the execution of instructions as shown by the dashed lines.

F_1 faults are exploitable. The exploitable F_1 faults are restricted to those where one or more vulnerable nodes in the program are skipped or executed more than the expected number of times.

To understand the probabilities of these events for a given cipher implementation, we generate the control flow graph of the program with vulnerable nodes marked. These nodes are identified by the Vulnerable Instruction Identification module (Section 4.1). Fault injections are simulated in the PC for each node of the CFG and the flow of the program is observed after the fault injection. The events F_1, F_2, and F_3 are counted to compute the probabilities of occurrence.

The Control Flow Graph (CFG) for the LLVM intermediate representation (IR) in Figure 3 is depicted in Figure 6. The cipher is implemented in the 16-bit TI MSP-430 and the address of each IR instruction is also shown in Figure 6. If a single bit fault is injected in the program counter, for instance in node $\mathtt{I}_6$ with the address is `<311C>`, the PC can take 16 possible values due to the fault injection, 11 of these values result in a PC outside the program causing the program to terminate. The valid PCs after fault injection are `<3114>`, `<3118>`, `<311E>`, `<313C>`, and `<319C>` as these addresses fall within the CFG. Of these addresses, `<311E>`, `<313C>`, and `<319C>` result in a forward jump, skipping vulnerable nodes (shaded in Figure 6). These faults result in exploitable F_1 events. The addresses `<3114>` and `<3118>` result in a backward jump, causing the re-execution of vulnerable nodes. These too result in exploitable F_1 events. There are no F_2 events in this example. Hence the probability of F_1, F_2, and F_3 events for the fault in the PC corresponding to node $\mathtt{I}_6$ are: 0.31, 0.0, and 0.69 respectively. In a similar way, faults injected in the PC corresponding to the $\mathtt{I}_{53}$ results in probabilities 0.06, 0.25, and 0.69 respectively. Among all faults, only one that modifies the PC from `<31BC>` to `<313C>` resulting in the re-execution of the vulnerable instruction $\mathtt{I}_{18}$ which is a vulnerable instructions and hence marked exploitable.

4.2.4 Computing the probability that an injected fault causes for an instruction

Of the four events, F_1, F_2, F_3, and F_4, only event F_1 is useful in a fault attack because only in this case the fault induces an error in the program and does not terminate it. We denote the probability that the output of the j-th instruction can be faulted by

$$\mathcal{P}(C^*_{2,j}) = \mathcal{P}(F_1) \ , \tag{1}$$

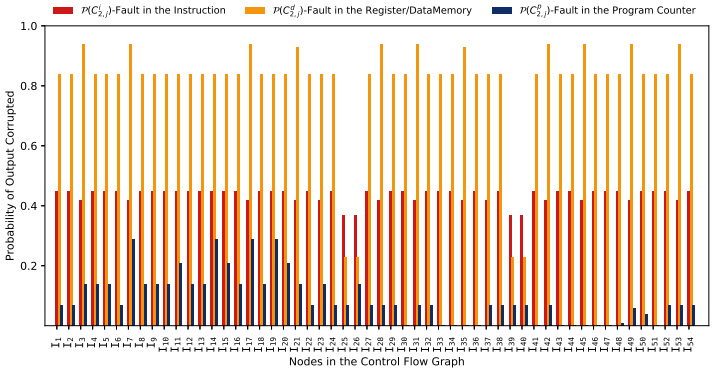


Figure 7: The probability of a fault corrupting the output of instructions given in Figure 3 on an TI MSP-430 processor. Faults can be induced either in the instruction, memory, or in the program counter.

where $\mathcal{P}(F_1)$ is the probability of event F_1 occurring when a fault is injected and '$*$' denotes any of the fault injections *i.e.* in the instruction, memory, or program counter.

Figure 7 depicts these probabilities for each IR instruction for the pseudo-code shown in Figure 3 with the three fault injections corresponding to a fault in the instruction, data/register, or the program counter. These probabilities are represented as $\mathcal{P}(C^i_{2,j})$, $\mathcal{P}(C^d_{2,j})$, and $\mathcal{P}(C^p_{2,j})$ respectively. To generate these probabilities, the source code written in C is first compiled using the LLVM compiler to generate a binary and also the Intermediate Representation (IR).

Using the probabilities obtained in Section 4.2.1 and 4.2.2, each instruction and the operands in the generated executable are analyzed to determine the corresponding $\mathcal{P}(C^*_{2,j})$ (where $*$ is either i or d). These probabilities are at the instruction level and is the only hardware dependent step. We map these probabilities onto the corresponding machine-independent IR instructions generated by the LLVM compiler. Unlike opcodes and operands, faults in the PC are directly evaluated using the control flow graphs (CFGs) generated from the IR instructions as discussed in Section 4.2.3.

The graph in Figure 7 shows that the faults injected in registers or memory have a higher probability of corrupting the instruction output compared to the faults injected in the opcode or the program counter. This is because faults injected in opcodes and the program counter are more likely to terminate the program due to invalid opcodes F_4 or an invalid program counter.

4.3 Fault propagation from the instruction to the ciphertext (C3)

The output of an instruction in the program can be corrupted either by a fault injected in that instruction (discussed in Section 4.2.1) or a fault injected in a previous instruction that propagates to the given instruction. The latter depends on the program structure. For example, consider the instruction $\mathtt{L_6}$ (refer Figure 2) resulting in a non-zero value of `b`. If a fault induced in this instruction changes the value of the byte `b` to another non-zero value, then the fault will not propagate to the ciphertext due to the conditional statement in $\mathtt{L_8}$. Thus, only a fault that changes `b` to zero would propagate to the output. We denote the probability that the fault propagates through a sequence of instructions $\mathtt{I}_i$, $\mathtt{I}_{i+1}$, ..., $\mathtt{I}_{i+n}$ as $\mathcal{P}(C_{3,(i,i+1,\cdots,i+n)})$.

Fault propagation can be done in two ways. The first is through registers, where the output of one instruction is used as an input to another. Alternatively, faults can propagate through memory operations. For instance, by a store of faulted data to memory, followed by a subsequent load from the same address. To compute $\mathcal{P}(C_{3,(*)})$, `FaultMeter` classifies instructions into three different classes based on the model proposed by Guanpeng et al.

in [LPH+18]. The first class considers fault propagation through registers. The second class considers fault propagation from a corrupted store to a subsequent load, while the third class considers control flow instructions.

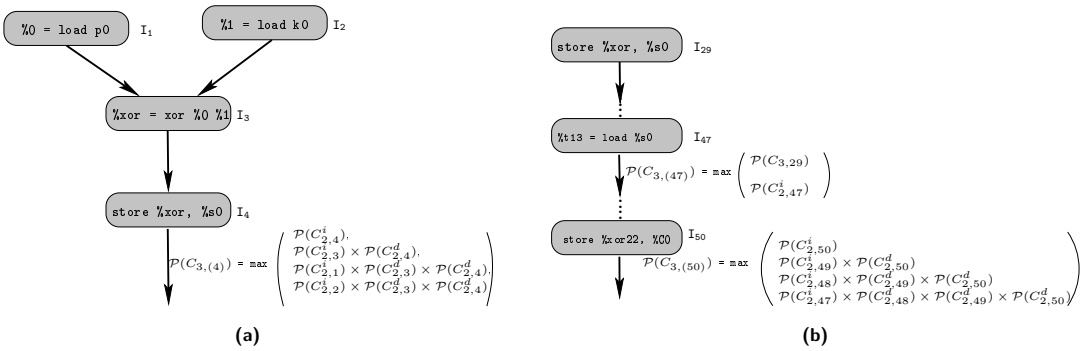


Figure 8: The instructions in (a) shows fault propagation data dependant instruction sequence corresponding to IR instructions implementing S0 ← P0 ⊕ K0 in Figure 3. For instance, at $\texttt{I}_4$, a fault could occur at $\texttt{I}_4$ or one of the previous nodes, and propagate forward. (b) shows fault propagation in memory dependent load and store instructions. A corrupted store in $\texttt{I}_{29}$ will propagate to the load in $\texttt{I}_{47}$, which propagates further to influence the output in $\texttt{I}_{50}$.

Fault propagation through registers. We consider a sequence of instructions, where the output of instruction is the input to another, and the data is transferred through registers. This data-dependent sequence of instructions has two properties: **(a)** it ends with a store or a control flow instruction, and **(b)** the output of every instruction in the sequence flows to the input of a subsequent instruction in the sequence. As an example the first four IR instructions ($\texttt{I}_1$, $\texttt{I}_2$, $\texttt{I}_3$, and $\texttt{I}_4$) from Figure 3 have two data-dependent sequences, namely, ($\texttt{I}_1$, $\texttt{I}_3$, $\texttt{I}_4$) and ($\texttt{I}_2$, $\texttt{I}_3$, $\texttt{I}_4$) as shown in Figure 8a. Note that $\texttt{I}_4$ is a `store` instruction that marks the end of the data sequence.

For each control flow or store instruction, `FaultMeter` computes the maximum probability that the output of the instruction is corrupted. To do this, it identifies all possible data-dependent sequences that terminate with the given control flow or store instruction, and computes the maximum probability that the output of the instruction is corrupted. The choice of maximum probability gives the highest success with which the output of an instruction can be corrupted. For instance, the maximum probability that $\texttt{I}_4$ (Figure 8a) is corrupted is given by

$$\mathcal{P}(C_{3,(4)}) = \max\left(\mathcal{P}(C^i_{2,4}), \mathcal{P}(C_{3,(3,4)}), \mathcal{P}(C_{3,(1,3,4)}), \mathcal{P}(C_{3,(2,3,4)})\right) \quad . \tag{2}$$

$\mathcal{P}(C^i_{2,4})$ is the probability that a fault injected in instruction $\texttt{I}_4$ and corrupts its output. All other probabilities correspond to a fault injected in a predecessor instruction (either $\texttt{I}_1$, $\texttt{I}_2$, or $\texttt{I}_3$) that propagates, corrupting the output of $\texttt{I}_4$. To compute these probabilities, we take an example of the sequence of instructions $\texttt{I}_1$, $\texttt{I}_3$ and $\texttt{I}_4$. If a fault is injected in the instruction $\texttt{I}_1$ then, assuming independence between instructions, the fault propagation probability at the end of the sequence is computed as

$$\mathcal{P}(C_{3,(1,3,4)}) = \mathcal{P}(C^i_{2,1}) \times \mathcal{P}(C^d_{2,3}) \times \mathcal{P}(C^d_{2,4}) \ , \tag{3}$$

where $\mathcal{P}(C^i_{2,1})$ is the probability of a fault injected in instruction $\texttt{I}_1$ (refer Section 4.2.1). This fault propagates through $\texttt{I}_3$ and $\texttt{I}_4$ due to the data dependent path. We quantify this by considering the fault in the data in the corresponding instructions, *i.e.* $\mathcal{P}(C^d_{2,3})$ and $\mathcal{P}(C^d_{2,4})$ (refer Section 4.2.2). In a similar manner we quantify the fault propagation probability in each instruction sequence $\mathcal{P}(C_{3,(3,4)})$ and $\mathcal{P}(C_{3,(2,3,4)})$.

Fault propagation in memory dependent instructions. To compute the fault propagation through memory operations, we keep track of load and store dependencies.

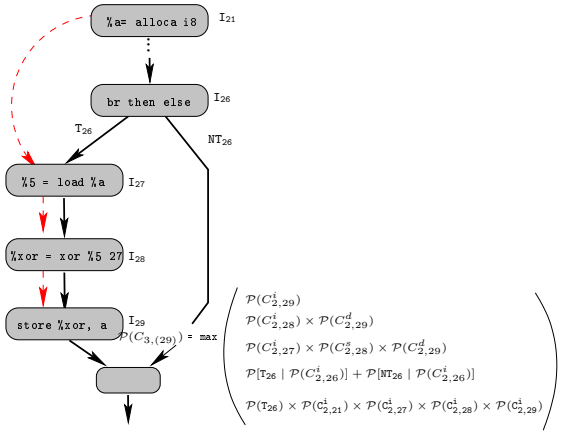


Figure 9: Fault propagation across the control flow instruction can corrupts the store instruction $\mathtt{I}_{29}$ either through data-dependent sequence (marked in red) or by a fault in branch $\mathtt{I}_{26}$.

For example, in Figure 8b, the store in $\mathtt{I}_{29}$ and the subsequent load in $\mathtt{I}_{47}$ are to the same address. Thus a corrupted store in $\mathtt{I}_{29}$ corrupts the load in $\mathtt{I}_{47}$. From the corrupted load, the fault propagates through registers in data-dependent instruction sequences. For example, the next data dependant instruction sequence is from $\mathtt{I}_{47}$ to $\mathtt{I}_{50}$, which produces an output. It should be noted that the current version of the work assumes that the addresses can be resolved statically. The support for load and store operations that can only be resolved at runtime is left as future work.

Fault propagation in control flow instructions. Instructions such as branch and call, can propagate the fault to another location or change the control flow path of the program. For example, in Figure 9, if a fault is injected in instruction $\mathtt{I}_{21}$, it can influence the instructions within the control flow path of the program ($\mathtt{I}_{27}$ to $\mathtt{I}_{29}$) only if the branch is taken. On the other hand, if a fault is injected in $\mathtt{I}_{26}$, it can change a taken branch to a not-taken branch or vice-versa. We analyze these two cases separately.

Fault does not change the control flow path. For example, the fault in $\mathtt{I}_{21}$ can influence the store in $\mathtt{I}_{29}$ only if the branch is taken. Thus, the probability of fault propagation from $\mathtt{I}_{21}$ to $\mathtt{I}_{29}$ also depends on the probability that the branch is taken. To compute these probabilities, we determine all the store instructions along the control flow path and then evaluate fault propagation to the stored memory location using the data-dependant sequence (discussed earlier in this section). Thus, the probability that a fault injected in $\mathtt{I}_{21}$ can propagate to the memory location used in the store in $\mathtt{I}_{29}$ is computed as follows:

$$\mathcal{P}(C_{3,(21,27,28,29)}) = \mathcal{P}(\mathtt{T}_{26}) \times \mathcal{P}(C^i_{2,21}) \times \mathcal{P}(C^d_{2,27}) \times \mathcal{P}(C^d_{2,28}) \times \mathcal{P}(C^d_{2,29}) \ , \tag{4}$$

where $\mathtt{T}_{26}$ is the probability that the branch in $\mathtt{I}_{26}$ is taken. To generalize this approach when there are multiple stores in both taken as well as not-taken branches, we independently compute probability of fault propagation to each store using the data-dependant sequence analysis as discussed earlier, and then consider the maximum probability.

Fault changes the control flow path of the program. For example, a fault in instruction $\mathtt{I}_{26}$ can cause the taken branch to be not-taken (or vice-versa) and can result in the store instruction at $\mathtt{I}_{29}$ to be illegally executed (or not executed). In either case, the corresponding memory operation in the store instruction is corrupted. To compute the fault propagation probability from the control flow instruction to the corrupted store, we would need to consider the conditional probability that the branch is illegally taken or not-taken due to the fault in the branch instruction. The probability that the branch is

illegally not-taken due to the fault in the control flow instruction that alters the output of the instruction and is given by

$$\mathcal{P}[\mathtt{NT}_{26} \mid F_1] = \mathcal{P}(\mathtt{T}_{26}) \times \mathcal{P}(C^i_{2,26}) \ . \tag{5}$$

This fault corresponds to the event F_1 defined in Section 4.2, where the fault is activated. In Equation 5, $\mathcal{P}(\mathtt{T}_{26})$ is the probability that a branch is taken, and $\mathcal{P}(C^i_{2,26})$ is the probability that the output of the 26-th instruction (the branch) is corrupted by the fault. The probability of the branch taken or not taken is determined empirically. For example, a corrupted output of a branch instruction could make a not-taken branch taken or vice-versa. Similarly, we can compute the probability that the branch is illegally taken due to the fault, *i.e.* $\mathcal{P}[\mathtt{T}_{26} \mid F_1]$, in a similar manner.

Considering Figure 9, the probability that the memory location used in the store instruction $\mathtt{I}_{29}$ is corrupted due to a fault in $\mathtt{I}_{26}$ can either result in execution or non execution of the store. Assuming that the branch at $\mathtt{I}_{26}$ is taken, the fault can force the branch to be not-taken, resulting in the execution of the instruction $\mathtt{I}_{29}$ corrupting the corresponding stored memory. On the other hand, assuming that the branch at $\mathtt{I}_{26}$ is not-taken, the fault can force the branch to be taken, skipping the store instruction at $\mathtt{I}_{29}$. This too corrupts the corresponding memory. Thus, the probability that the memory used in the store is corrupted is given by

$$\begin{aligned} \mathcal{P}(C_{3,(26,27,28,29)}) &= \mathcal{P}\left(\mathtt{T}_{26} \;\middle|\; (C^i_{2,26})\right) + \mathcal{P}\left(\mathtt{NT}_{26} \;\middle|\; (C^i_{2,26})\right) \\ &= \mathcal{P}(\mathtt{NT}_{26}) \times \mathcal{P}(C^i_{2,26}) + \mathcal{P}(\mathtt{T}_{26}) \times \mathcal{P}(C^i_{2,26}) \\ &= (1 - \mathcal{P}(\mathtt{T}_{26})) \times \mathcal{P}(C^i_{2,26}) + (\mathcal{P}(\mathtt{T}_{26}) \times \mathcal{P}(C^i_{2,26}) = \mathcal{P}(C^i_{2,26}) \ . \end{aligned} \tag{6}$$

4.3.1 The FaultMeter Algorithm

The FaultMeter algorithm (Algorithm 1) takes three parameters. The first parameter, CFG is the control flow graph of the implementation under test (IUT). The second is V_list, which is a list of vulnerable instructions in the IUT. These vulnerable instructions are obtained from the Fault Vulnerable Identification module and the only locations in the IUT where an injected fault can be exploited. The third parameter is a processor specific lookup table, TPF, comprising of instructions and the fault activation probabilities that were obtained empirically as discussed in Section 4.2. The fault can be injected in instruction, memory/register, or in the program counter. Without loss of generality, Algorithm 1 considers faults injected only in instructions. The algorithm assigns a probability to each node in V_List. This probability, SuccessScore, denotes the success with which injected faults propagate to the output.

The algorithm starts executing from Main(Line 1-7). For each vulnerable location (*i.e.* each element in V_List), it creates a data-dependent graph (DDG). These graphs are acyclic and show the propagation of a fault from $\mathtt{I}_l$ to the output. For example, in Figure 3, 18 out of the 54 instructions present are vulnerable, hence the algorithm would have 18 different DDGs. For each of these graphs $\mathtt{DDG}[\mathtt{I}_l]$, function ComputeP is invoked. The second parameter passed to ComputeP holds the fault activation probability for the instruction $\mathtt{I}_l$. The function returns the probability with which a fault in instruction $\mathtt{I}_l$ propagates to the output.

The ComputeP function extracts the path from $\mathtt{I}_l$ to $\mathtt{I}_n$ such that $\mathtt{I}_n$ is a store, a branch instruction, or an output instruction. If there exists a branch between $\mathtt{I}_l$ and $\mathtt{I}_n$ in the CFG, then the probability of $\mathtt{I}_n$ being executed is determined (Line 11-14). This depends on the probability that the branch is taken or not-taken. For example, in Figure 9, the data dependant path is $\mathtt{I}_{21}$, $\mathtt{I}_{27}$, $\mathtt{I}_{28}$ and $\mathtt{I}_{29}$. The probability that $\mathtt{I}_{29}$ executes is the probability that the branch at $\mathtt{I}_{26}$ is taken. Line 15, computes the fault propagation probability from $\mathtt{I}_l$ to $\mathtt{I}_n$.

Lines 16-28 identify the next sequence of instructions to evaluate. If $\mathtt{I}_n$ is a branch, it identifies the probability that store instructions in the taken and not-taken paths are

Algorithm 1: The FaultMeter Algorithm

Input: CFG: Control Flow Graph of Implementation and V_List : List of the vulnerable instructions and TPF: Processor dependent fault activation probability table
Output: SuccessScore : Success Score of the vulnerable instructions

```
1  Function Main():
2      begin
3          Create the Data Dependent Graph (DDG) for each node in V_List
4          for each graph with I_l as the start node do
5              p ← Fetch P(C^i_{2,l}) of I_l from TPF
6              SuccessScore[I_l] ←ComputeP(DDG[I_l], p)
7          return SuccessScore
8  Function ComputeP(DDG[I_l], p):
9      begin
10         SScore ← ∅
11         ⟨I_l, ··· , I_n⟩ ← Extract path from I_l to I_n of DDG_l, where I_n is the first be store, branch,
             or output instruction in the path.
           /* instruction sequence ⟨I_l, I_{l_1}, I_{l_2} ··· , I_n⟩                    */
12         b ← True, if ∃ a branch between I_l and I_n in CFG, else b ← False
13         if b = True then
14             p ← p × (Probability of I_n being executed)
           /* Computing fault propagation probability of data-dependent sequence.      */
15         p ← p × P(C^d_{2,l_1}) × ··· × P(C^d_{2,n})
16         if I_n is a branch then
17             ⟨store⟩ ← Find all stores in the taken and not-taken path of branch I_n.
18             for each I_s in ⟨store⟩ do
19                 ⟨I_s, p_s⟩ ← (I_s, p)
20             for each ⟨I_s, p_s⟩ do
21                 Mem ← Find the memory location in the store instruction
22                 SScore[Mem]← ComputeP(DDG[I_s], p_s)
23             P(C_{3,(I_l,···,I_n)}) ← max(SScore[Mem])
24         else if I_n is output then
25             P(C_{3,(I_l,···,I_n)}) ← p
26         else if I_n is a store then
               /* Instruction I_n to I_l has the Store to Load Dependency              */
27             I_l ← Next Memory Dependent instruction of store (I_n) in the DDG[I_l]
28             P(C_{3,(I_l,···,I_n)}) ← ComputeP(DDG[I_l], p)
29         return P(C_{3,(I_l,···,I_n)})
```

corrupted as described in Equation 6. For each of these store instructions, the algorithm recursively invokes ComputeP with I_s as the start node of the DDG (DDG[I_s]) and p_s the probability of the fault corrupting the output of I_s (Line 20-22). The maximum probability of all the stores is returned if there is a write to the same memory locations. ComputeP terminates if I_n is an output instruction then $\mathcal{P}(C_{3,(I_l,\cdots,I_n)})$ is assigned with a probability p (Line 25).

Suppose the instruction I_n is a store, then a store to load dependency (I_n to I_l) is determined to propagate the fault further (described in Section 4.3). ComputeP is recursively invoked (Line 28) with the load instruction I_l as the fault activation location of the DDG[I_l] and p, the probability of fault corrupting the output of I_l (refer example in Figure 8b). The function returns with the value of the fault propagation probability for I_l, when it finds an output instruction.

Figure 10 shows the fault propagation for 2 locations I_1, I_{15} for two processors TI MSP-430(16-bit) and RISC-V(32-bit) for the toy cipher given in Figure 3. The SuccessScore for these locations is computed using Algorithm 1. The x-axis highlights the IR instructions I_1 to I_{54}, and the y-axis shows the fault propagation probability. The graph shows that

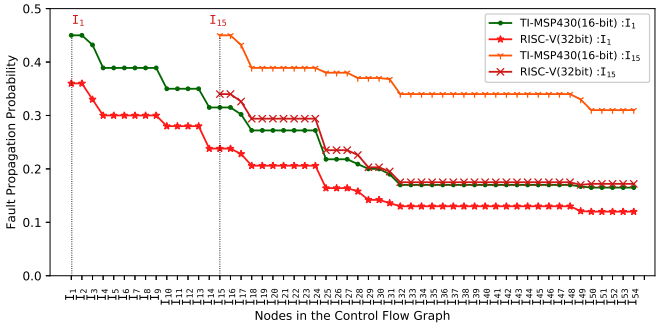


Figure 10: The fault propagation probability of instructions I_1, I_{15} for the processors TI MSP-430(16-bit) and RISC-V(32-bit) for the toy cipher implementation given in Figure 3.

Table 3: We invoke FEDS [KRR+20] in the Vulnerable Instruction Identification module for the five implementations considered. This step is hardware independent.

Block Cipher	#IR Instruction in the CFG	% of Vulnerable Instructions	Time (in Secs)
AES-128(LookUp Table)	7206	6.56	38.2
AES-128(T Table)	4299	4.2	15.5
AES-128(BitSliced)	8256	11.45	215.7
CAMELLIA-128	1475	23.2	99.3
CLEFIA-128	1024	6.54	105.5

the fault propagation probability varies based on the processor and the location of fault activation.

5 Implementation and Evaluation

In this section we provide details about the implementation of `FaultMeter` and present the results on the cipher implementations. We consider five cipher implementations. Three implementations are realizations of AES-128 based on LookUp Tables, where the `SubBytes` operation is implemented using a single 256 byte look-up table[10], T-Tables, where the operations `SubBytes`, `ShiftRows`, and `MixColumns` are merged and replaced with look-ups[11], and BitSliced [RSD06], where all the operations are bitsliced. We also consider implementations of CLEFIA-128 [SSA+07] and CAMELLIA-128 [AIK+00]. The `FaultMeter` framework is implemented as transformation passes in the LLVM Clang compiler[4] Version 7.0.

The probability that the output of an instruction is corrupted due to an injected fault, event F_1, is computed beforehand on the ARM (32 and 64 bit), RISC-V(32 and 64 bit), TI MSP-430 (16 bit), and Intel x86(64-bit) architectures. Next, we discuss the results at the output of each stage of `FaultMeter`.

5.1 Vulnerable Instruction Identification

The Vulnerable Instruction Identification module (refer Figure 3) takes the block cipher source code as input and marks the fault vulnerable instructions from the implementation.

`FaultMeter` uses the FEDS [KRR+20] framework for this stage. FEDS determines all the exploitable instructions from IR instructions that are susceptible to fault attack. The Vulnerable Instruction Identification module works by converting the IR instructions to control flow graph and also finds the dependencies between the instructions using a reverse data flow analysis on the control flow graph (refer Section 4.1). Table 3 shows the

[10] https://github.com/BrianGladman/aes/blob/master/aestab.c
[11] https://github.com/openssl/openssl/blob/master/crypto/aes/aes_core.c

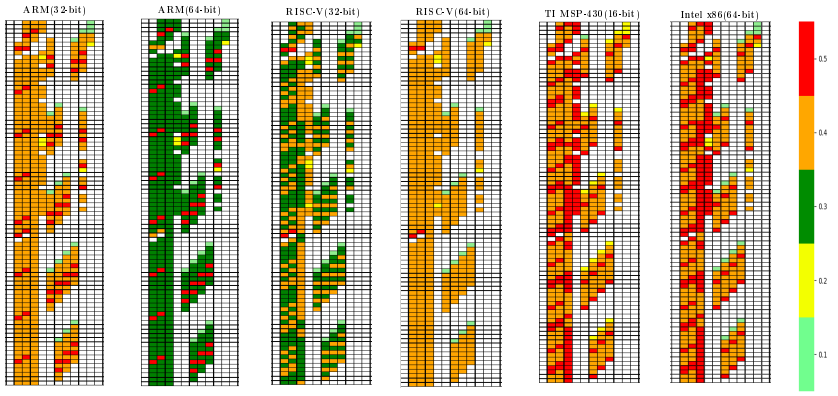


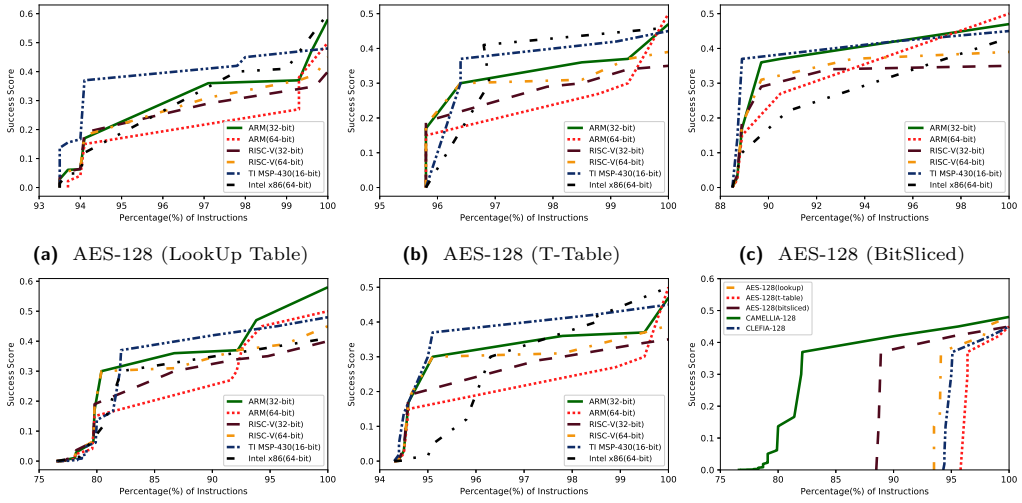
Figure 11: `SuccessScore` of vulnerable instruction from an AES(LookUp Table) based implementation on six different processors. Each cell represents an instruction and the color code represents the `SuccessScore`.

output of the Vulnerable Instruction Identification module. The percentage of exploitable instructions varies from 6.56% AES-128(LookUp Table) to 23.2% (CAMELLIA-128) from the total instruction in the control flow graph. As an example, the AES (T-Table) implementation has 4.2% instructions that are exploitable to fault attack from the total of 4299 IR instructions present. A fault induced in any of these vulnerable instructions can result in a successful fault attack. These results just depend on the cipher implementation and are agnostic of the underlying hardware used.

5.2 Fault Exploitability Quantification

Fault Exploitability Quantification module takes the Control Flow Graph with marked vulnerable nodes as input and processor dependent fault activation probability and quantifies the vulnerability using Algorithm 1. The module considers faults injected in instruction opcodes, memory, registers, and in the program counter. In each case, the probability of the injected fault propagating to the output is computed. This probability depends on the underlying hardware architecture and the program structure.

`FaultMeter` evaluation on different architectures. To demonstrate that the fault propagation varies based on program structure, we have considered five cipher implementations (given in Table 3) and five RISC microprocessors: ARM (32 and 64 bit), RISC-V (32 and 64 bit), and TI MSP-430 (16 bit). We also considered the Intel x86 (64-bit) CISC architecture. Figure 11 shows the memory layout of exploitable instructions for the AES-128 LookUp Table based implementation (Table 3). Each colored cell shows the exploitable instructions, with the color indicating the probability that a fault in that instruction can corrupt the ciphertext and result in a successful attack. Notice that on each processor architecture, the vulnerable instructions are the same. However, the difference in color across the architectures indicates that the exploitability of the instruction differs from one architecture to another. TI's MSP430 (16-bit) RISC processor and Intel x86 (64-bit) processor have instructions with high fault susceptibility. This means that a fault injected in these processors have a high chance of disturbing the execution compared to a fault in the same instruction in the other processors. This is because, comparatively, TI MSP430 has a densely packed instruction set (refer Section 4.2); therefore, there is a smaller probability of obtaining an invalid opcode. This results in high $\mathcal{P}(C_2)$. Similarly, the large number of instructions in Intel's x86 platforms, owing to the CISC architecture, provides a similar impact.



(a) AES-128 (LookUp Table) **(b)** AES-128 (T-Table) **(c)** AES-128 (BitSliced)

(d) CAMELLIA-128 **(e)** CLEFIA-128 **(f)** All Ciphers on TI MSP-430

Figure 12: Percentage of vulnerable instructions for different cipher implementations. To understand these graphs, consider for example in Figure (d), when $x = 85\%$, $y = 0.38$ on TI MSP-430(16-bit). This means that 85% of CAMELLIA-128 instructions have `SuccesScore` ≤ 0.38.

`FaultMeter` evaluation on different software implementations. Figures 12a to 12e shows the output of the Fault Exploitability Quantification module for the five cipher implementations on the six processors. For AES-128 cipher implementations (Figures 12a to 12c), the percentage of vulnerable instructions varies based on how the cipher is implemented, and the final `SuccessScore` of vulnerable instructions varies based on the program structure and the underlying architecture. For example, for AES-128 (LookUp Table), Figure 12a, 0.5% of instructions (≈ 72) instructions have maximum `SuccessScore` is 0.5 across different architecture, whereas for AES-128 (T-Table) Figure 12b and AES-128 (BitSliced) Figure 12c, the percentage of vulnerable instructions with similar `SuccessScore` is 0.4% (≈ 18) and 1.5%($\approx$124) respectively. Of the three AES implementations, the BitSliced implementation is most prone to fault attacks because it has the highest percentage of vulnerable instructions (Table 3), and it has a higher fault propagation probability across all the architectures (Figure 12). Similarly, the T-Table implementation is the most secure of the AES-128 implementations considered.

Figure 12f compares the fault propagation probability of the cipher implementations (given in Table 3) on the TI MSP-430 (16-bit) processor. From the figure, it is clear that CAMELLIA-128 is more vulnerable to fault injection attacks as it has the highest percentage of vulnerable instructions. For CAMELLIA-128, $\approx 85\%$ of instructions have `SuccesScore` ≤ 0.4. AES-128 (T-Table) is least vulnerable on TIs MSP-430, where $\approx 96\%$ of vulnerable instructions have `SuccesScore` ≤ 0.4.

6 Applying `FaultMeter` to automatically insert countermeasures based on User Specified Security Input (U_{in})

Incorporating fault attack countermeasures is expensive. It can increase run time overheads by over 100% and memory requirements by over 800%. These overheads are unacceptable for several applications, especially where time and resources are critical. One approach to address this issue is to provide just sufficient countermeasures to meet an application's security requirement. This is motivated by the fact that a program's security requirements vary considerably based on the application. For instance, a block cipher used in critical

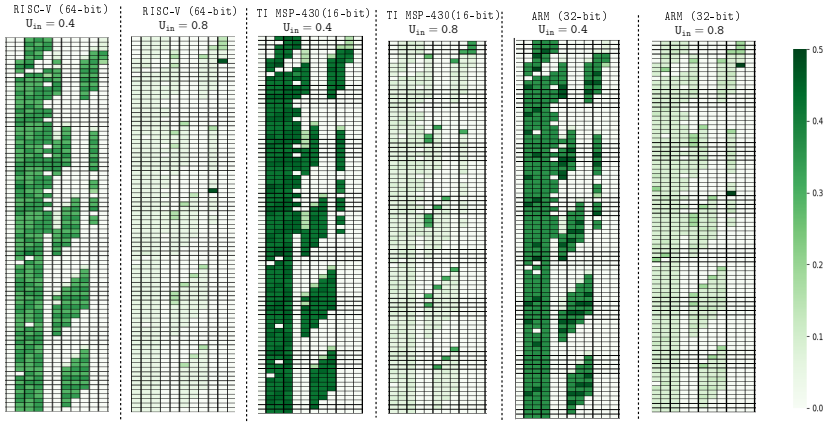


Figure 13: `SuccessScore` of vulnerable instructions (AES-128 (LookUp Table)) after inserting countermeasures on three processors with U_{in} 0.4 and 0.8. Each cell represents an instruction.

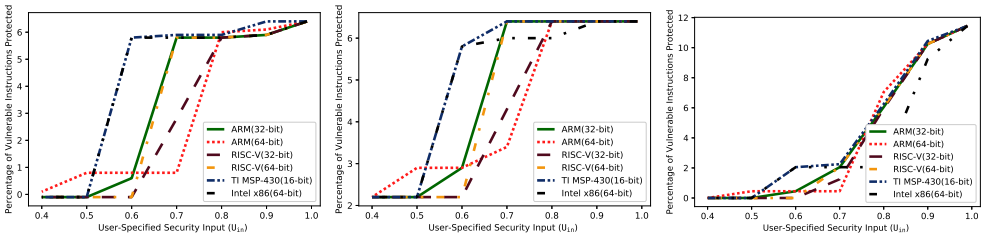
infrastructure would require much more secure implementations compared to an application in a consumer device. Thus for such applications, designers typically would want to prioritize security in lieu of performance. Such trade-offs would be less acceptable for the consumer device, especially in a resource-constraint device, where each byte and each clock cycle is valuable.

Table 4: Comparison of unprotected and naïvely protected implementations with `FaultMeter` based countermeasure addition. The table shows the percentage increase in code size, clock, along with fault coverage. The comparison is made on a TI MSP-430 (16-bit) and ARM(64-bit) processors with different user requirements.

Block Cipher	Original		Naive Approach		User Input (U_{in})	FaultMeter (TI MSP430(16-bit))		Coverage (%)	FaultMeter (ARM (64-bit))		Coverage (%)
	Code Size	Clock Cycles	% Increase in Code Size	% Increase in Execution Time (Clock Cycles)		% Increase in Code Size	% Increase in Execution Time (Clock Cycles)		% Increase in Code Size	% Increase in Execution Time (Clock Cycles)	
AES-128 (LookUp Table)	936	95347	886.12	157.80	0.8	110.68	23.01	84	110.03	20.00	84
					0.6	108.54	18.75	82	80.05	15.12	75
					0.4	102.56	10.40	79	75.32	10.27	74
AES-128 (T-Table)	1089	79141	590.02	25.75	0.8	173.55	3.54	90	173.00	3.50	90
					0.6	165.47	2.65	89	150.00	2.50	84
					0.4	120.75	2.48	86	100.00	2.40	80
AES-128 (BitSliced)	2606	643348	288.14	91.73	0.8	64.08	48.98	91	64.02	48.00	91
					0.6	60.85	48.80	90	59.32	44.32	88
					0.4	58.36	48.39	86	47.12	40.00	81
CAMELLIA-128	1045	108996	831	42.05	0.8	21.05	13.21	95	21.03	13.20	95
					0.6	15.86	12.14	83	13.23	10.30	81
					0.4	10.35	12.05	80	5.25	10.01	75
CLEFIA-128	895	81311	679.01	21.40	0.8	40.11	5.25	85	40.05	5.20	85
					0.6	34.07	3.20	83	30.12	3.12	80
					0.4	29.21	2.17	81	20.18	1.5	70

In this section, we demonstrate the use of `FaultMeter` to cater to the diverse security requirements of applications. We use `FaultMeter` to automatically insert countermeasures based on the user's input, which is a number between 0 and 1 defining the extent to which security is important in the application. A value close to 1 implies that the user prioritizes security over performance, while a value close to 0 implies that performance is critical. The `SuccessScore` produced by `FaultMeter` is used to tune between security and performance. For instance, if the user input is U_{in}, then all the instructions where `SuccessScore` $\geq (1 - U_{in})$ are protected by automatically inserting countermeasures only in these locations. All other locations with lesser `SuccessScore` are not protected.

To demonstrate automatic countermeasure insertion, we use the spatial redundancy countermeasure as a case study. The countermeasure addition module (Figure 1) works at the IR level. It takes the vulnerable instructions along with the `SuccessScore` of each instruction and replicates instructions where `SuccessScore` $\geq (1 - U_{in})$. Additional variables are defined as required, and instructions are inserted to compare with redundant results.



(a) AES-128 (LookUp Table) **(b)** AES-128 (T-Table) **(c)** AES-128 (BitSliced)

Figure 14: Percentage of vulnerable instructions protected for different user input ($\texttt{U}_{\texttt{in}}$) across six different processors.

Table 4 shows the percentage increase in code size and clock cycles for different ciphers realized in the TI MSP-430 (16-bit) and ARM (64-bit) processors for a naïvely protected executable and an executable generated with `FaultMeter` based countermeasures. We observe that **(a)** the percentage increase in code size and execution clock cycles is far less with `FaultMeter` based protection. Unlike the naïve approach where all instructions in the executable are protected, with the `FaultMeter` based protection, only instructions with a $\texttt{SuccessScore} \geq (1 - \texttt{U}_{\texttt{in}})$ are protected. **(b)** The increase in code size and execution varies directly with the value of $\texttt{U}_{\texttt{in}}$ and inversely with fault coverage.

The TI MSP-430(16-bit) is the most vulnerable processor compared to other architectures, hence the percentage increase in code size is higher compared to ARM (64-bit) processor. From the table it is evident that the performance overhead after the countermeasure addition varies depends on the implementation as well as the underlying architecture.

Figure 13 shows the heat map of vulnerability of instructions for two different User Input ($\texttt{U}_{\texttt{in}}$) values for the AES-128(Look-Up Table) (Table 3) on three platforms after the countermeasure is inserted. From the figure, it is evident that a higher $\texttt{U}_{\texttt{in}}$ results in more protected implementations. The countermeasures inserted too is different in each platform. The measure of `SuccessScore` after the countermeasure inserted is computed independently of the previous experiments.

Figure 14 shows the percentage of instructions protected for different user inputs. For example, for AES-128 (T-Table) on a TI MSP-430, when $\texttt{U}_{\texttt{in}} = 0.7$, 4% of vulnerable instructions are protected, while for an ARM 64-bit, only 1% of vulnerable instructions are protected

7 Limitations

In its current form, there are two limitations of `FaultMeter`.

- Fault Vulnerable Identification module used in `FaultMeter` identifies the vulnerable instructions, while `FaultMeter` quantifies the vulnerability. If an instruction is identified incorrectly as not vulnerable by the Fault Vulnerable Identification module, `FaultMeter` will not be able to quantify it. Similarly, if an instruction is marked incorrectly as vulnerable, the output of `FaultMeter` is also incorrect.

- `FaultMeter` currently works with unprotected implementations of block ciphers. It needs to be extended to support implementations where the protection is already incorporated. In order to do this, FaultMeter would need to distinguish instructions that are present due to the countermeasure. Distinguishing these countermeasure related instructions from other instructions is challenging at the compiler's intermediate representation level and therefore left as future work.

8 Discussion

Non-Cryptographic Applications: Besides cryptography, `FaultMeter` can be used for other security applications, such as information flow analysis [SM03], safety-critical applications etc. The major challenge is to find the sensitive locations from the application software.

9 Conclusion

`FaultMeter` is an automated framework that can quantify the success with which an injected fault can be exploited. We show that this success probability depends on the cipher algorithm, its implementation, as well as the Instruction Set Architecture (ISA) of the processor. Our evaluation of five cipher implementations on six hardware platforms brings out interesting observations. For instance, TI MSP 430 (16-bit) and Intel x86 (64-bit) are the most vulnerable to fault attacks. Comparing the 32-bit RISC processors, ARM is more vulnerable to fault injection than RISC-V. On the other hand, the 64-bit variant of RISC-V is more vulnerable than the equivalent ARM variant. Further, the smaller TI MSP-430 processor is the most vulnerable amongst all processors considered. Comparing different implementations of AES, the T-table implementation is the most secure against fault attacks. The quantification that `FaultMeter` provides can be used to strategically used to choose the right countermeasure in block cipher implementations to meet the application's security requirements as we demonstrated in the paper.

References

[ABMP13] Giovanni Agosta, Alessandro Barenghi, Massimo Maggi, and Gerardo Pelosi. Compiler-based side channel vulnerability analysis and optimized countermeasures application. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 81:1–81:6, 2013.

[AIK+00] Kazumaro Aoki, Tetsuya Ichikawa, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Junko Nakajima, and Toshio Tokita. Camellia: A 128-bit block cipher suitable for multiple platforms - design and analysis. In *Selected Areas in Cryptography, 7th Annual International Workshop, SAC 2000, Waterloo, Ontario, Canada, August 14-15, 2000, Proceedings*, pages 39–56, 2000.

[AM13] S. Ali and D. Mukhopadhyay. Improved Differential Fault Analysis of CLEFIA. In *FDTC*, pages 60–70, 2013.

[ATG+19] Md. Mahbub Alam, Shahin Tajik, Fatemeh Ganji, Mark Mohammad Tehranipoor, and Domenic Forte. Ram-jam: Remote temperature and voltage fault attack on fpgas using memory collisions. In *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2019, Atlanta, GA, USA, August 24, 2019*, pages 48–55. IEEE, 2019.

[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)*, pages 229–240, 2020.

[BBK+03] Guido Bertoni, Luca Breveglieri, Israel Koren, Paolo Maistri, and Vincenzo Piuri. Error analysis and detection procedures for a hardware implementation of the advanced encryption standard. *IEEE Trans. Computers*, 52(4):492–505, 2003.

[BEG13] Nasour Bagheri, Reza Ebrahimpour, and Navid Ghaedi. New differential fault analysis on PRESENT. *EURASIP J. Adv. Signal Process.*, 2013:145, 2013.

[BG13] Alberto Battistello and Christophe Giraud. Fault analysis of infective AES computations. In Wieland Fischer and Jörn-Marc Schmidt, editors, *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*, pages 101–107. IEEE Computer Society, 2013.

[BHE+19] Jean-Baptiste Bréjon, Karine Heydemann, Emmanuelle Encrenaz, Quentin Meunier, and Son-Tuan Vu. Fault attack vulnerability assessment of binary code. In *Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems*, CS2 '19, page 13–18, New York, NY, USA, 2019. Association for Computing Machinery.

[BHL18] Jakub Breier, Xiaolu Hou, and Yang Liu. Fault attacks made easy: Differential fault analysis automation on assembly code. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):96–122, 2018.

[CN10] Ware D Courtois NT, Jackson K. Fault-algebraic attacks on inner rounds of des. In *e-Smart '10 Proceedings: The Future of Digital Security Technologies*, 2010.

[DEK+18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: exploiting ineffective fault inductions on symmetric cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):547–572, 2018.

[DFL11] Patrick Derbez, Pierre-Alain Fouque, and Delphine Leresteux. Meet-in-the-middle and impossible differential fault analysis on AES. In *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, pages 274–291, 2011.

[DPdC+15] Louis Dureuil, Marie-Laure Potet, Philippe de Choudens, Cécile Dumas, and Jessy Clédière. From code review to fault injection attacks: Filling the gap using fault model inference. In Naofumi Homma and Marcel Medwed, editors, *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, volume 9514 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 2015.

[FVH+20] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P*, May 2020. Best Paper Award, Pwnie Award for Most Innovative Research, IEEE Micro Top Picks Honorable Mention.

[GJL20] Thomas Given-Wilson, Nisrine Jafri, and Axel Legay. Combined software and hardware fault injection vulnerability detection. *Innov. Syst. Softw. Eng.*, 16(2):101–120, 2020.

[GK12] Xiaofei Guo and Ramesh Karri. Invariance-based concurrent error detection for advanced encryption standard. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 573–578. ACM, 2012.

[GK13] Xiaofei Guo and Ramesh Karri. Invariance-based concurrent error detection for advanced encryption standard. *IACR Cryptol. ePrint Arch.*, 2013:603, 2013.

[GST12] Benedikt Gierlichs, Jörn-Marc Schmidt, and Michael Tunstall. Infective computation and dummy rounds: Fault protection for block ciphers without check-before-output. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, volume 7533 of *Lecture Notes in Computer Science*, pages 305–321. Springer, 2012.

[HBZL19] Xiaolu Hou, Jakub Breier, Fuyuan Zhang, and Yang Liu. Fully automated differential fault analysis on software implementations of cryptographic algorithms. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 3:1–29, 2019.

[HKR+15] Andrea Höller, Armin Krieg, Tobias Rauter, Johannes Iber, and Christian Kreiner. Qemu-based fault injection for a system-level analysis of software countermeasures against fault attacks. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, pages 530–533. IEEE Computer Society, 2015.

[HSP21] Max Hoffmann, Falk Schellenberg, and Christof Paar. ARMORY: fully automated and exhaustive fault simulation on ARM-M binaries. *IEEE Trans. Inf. Forensics Secur.*, 16:1058–1073, 2021.

[HZFW15] Yuming Huo, Fan Zhang, Xiutao Feng, and Li-Ping Wang. Improved differential fault attack on the block cipher SPECK. 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 28–34. IEEE Computer Society, 2015.

[IMB+19] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative load hazards boost rowhammer and cache attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 621–637, Santa Clara, CA, August 2019. USENIX Association.

[JvdVF+22] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable Rowhammering in the Frequency Domain. In *S&P*, May 2022.

[KDK+14] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 361–372. IEEE Computer Society, 2014.

[KGGY20] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 695–711. IEEE, 2020.

[KKG03] Ramesh Karri, Grigori Kuznetsov, and Michael Gössel. Parity-based concurrent error detection of substitution-permutation network block ciphers. In

Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 113–124. Springer, 2003.

[KRH17] Punit Khanna, Chester Rebeiro, and Aritra Hazra. XFC: A framework for exploitable fault characterization in block ciphers. In *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*, pages 8:1–8:6, 2017.

[KRR+20] Keerthi K., Indrani Roy, Chester Rebeiro, Aritra Hazra, and Swarup Bhunia. FEDS: comprehensive fault attack exploitability detection for software implementations of block ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):272–299, 2020.

[KSV13] Dusko Karaklajic, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware designer's guide to fault attacks. *IEEE Trans. Very Large Scale Integr. Syst.*, 21(12):2295–2306, 2013.

[KWMK02] Ramesh Karri, Kaijie Wu, Piyush Mishra, and Yongkook Kim. Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(12):1509–1517, 2002.

[LFB+21] Guilhem Lacombe, David Féliot, Etienne Boespflug, Marie-Laure, and Potet. Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities. 2021.

[LPH+18] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael B. Sullivan, and Timothy Tsai. Modeling soft-error propagation in programs. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 27–38. IEEE Computer Society, 2018.

[LRT12] Victor Lomné, Thomas Roche, and Adrian Thillard. On the need of randomness in fault attack countermeasures - application to AES. In Guido Bertoni and Benedikt Gierlichs, editors, *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012*, pages 85–94. IEEE Computer Society, 2012.

[ML08] Paolo Maistri and Régis Leveugle. Double-data-rate computation as a countermeasure against fault analysis. *IEEE Trans. Computers*, 57(11):1528–1539, 2008.

[PMPD14] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 213–222. IEEE Computer Society, 2014.

[RBLC15] Lionel Rivière, Julien Bringer, Thanh-Ha Le, and Hervé Chabanne. A novel simulation approach for fault injection resistance evaluation on smart cards. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, pages 1–8. IEEE Computer Society, 2015.

[RPL+14] Lionel Rivière, Marie-Laure Potet, Thanh-Ha Le, Julien Bringer, Hervé Chabanne, and Maxime Puys. Combining high-level and low-level approaches to evaluate software implementations robustness against multiple fault injection attacks. In Frédéric Cuppens, Joaquín García-Alfaro, A. Nur Zincir-Heywood, and Philip W. L. Fong, editors, *Foundations and Practice of Security - 7th International Symposium, FPS 2014, Montreal, QC, Canada, November 3-5, 2014. Revised Selected Papers*, volume 8930 of *Lecture Notes in Computer Science*, pages 92–111. Springer, 2014.

[RRHB20] Indrani Roy, Chester Rebeiro, Aritra Hazra, and Swarup Bhunia. SAFARI: automatic synthesis of fault-attack resistant block cipher implementations. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(4):752–765, 2020.

[RRHB21] Indrani Roy, Chester Rebeiro, Aritra Hazra, and Swarup Bhunia. Faultdroid: An algorithmic approach for fault-induced information leakage analysis. *ACM Trans. Design Autom. Electr. Syst.*, 26(1):2:1–2:27, 2021.

[RSD06] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. Bitslice implementation of AES. In *Cryptology and Network Security, 5th International Conference, CANS 2006, Suzhou, China, December 8-10, 2006, Proceedings*, pages 203–212, 2006.

[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.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 12–27, 1988.

[SKMD17] Sayandeep Saha, Ujjawal Kumar, Debdeep Mukhopadhyay, and Pallab Dasgupta. An automated framework for exploitable fault identification in block ciphers - A data mining approach. In *PROOFS@CHES 2017, 6th International Workshop on Security Proofs for Embedded Systems, Taipei, Taiwan, Friday September 29th, 2017*, pages 50–67, 2017.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 21(1):5–19, 2003.

[SMD18] Sayandeep Saha, Debdeep Mukhopadhyay, and Pallab Dasgupta. Expfault: An automated framework for exploitable fault characterization in block ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):242–276, 2018.

[SSA+07] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-bit blockcipher CLEFIA (extended abstract). In *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, pages 181–195, 2007.

[SSR+20] Milind Srivastava, Patanjali SLPSK, Indrani Roy, Chester Rebeiro, Aritra Hazra, and Swarup Bhunia. SOLOMON: an automated framework for detecting fault attack vulnerabilities in hardware. In *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*, pages 310–313. IEEE, 2020.

[TBM14a] Harshal Tupsamudre, Shikha Bisht, and Debdeep Mukhopadhyay. Destroying fault invariant with randomization - A countermeasure for AES against differential fault attacks. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 93–111. Springer, 2014.

[TBM14b] Harshal Tupsamudre, Shikha Bisht, and Debdeep Mukhopadhyay. Differential fault analysis on the families of SIMON and SPECK ciphers. In Assia Tria and Dooho Choi, editors, *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014, Busan, South Korea, September 23, 2014*, pages 40–48. IEEE Computer Society, 2014.

[TMA11] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the advanced encryption standard using a single fault. In *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication - 5th IFIP WG 11.2 International Workshop, WISTP 2011, Heraklion, Crete, Greece, June 1-3, 2011. Proceedings*, pages 224–233, 2011.

[WKKG04] Kaijie Wu, Ramesh Karri, Grigori Kuznetsov, and Michael Gössel. Low cost concurrent error detection for the advanced encryption standard. In *Proceedings 2004 International Test Conference (ITC 2004), October 26-28, 2004, Charlotte, NC, USA*, pages 1242–1248. IEEE Computer Society, 2004.

[WLR+21] Huanyu Wang, Henian Li, Fahim Rahman, Mark M. Tehranipoor, and Farimah Farahmandi. Sofi: Security property-driven vulnerability assessments of ics against fault-injection attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2021.

[ZZJ+20] Fan Zhang, Yiran Zhang, Huilong Jiang, Xiang Zhu, Shivam Bhasin, Xinjie Zhao, Zhe Liu, Dawu Gu, and Kui Ren. Persistent fault attack in practice. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):172–195, 2020.