IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2020, No. 2, pp. 272–299. DOI:10.13154/tches.v2020.i2.272-299

FEDS: Comprehensive Fault Attack Exploitability Detection for Software Implementations of Block Ciphers

Keerthi K[1], Indrani Roy[1], Chester Rebeiro[1], Aritra Hazra[2] and Swarup Bhunia[3]

[1] Indian Institute of Technology Madras, India
{keerthi,indrani,chester}@cse.iitm.ac.in
[2] Indian Institute of Technology Kharagpur, India aritrah@cse.iitkgp.ac.in
[3] University of Florida swarup@ece.ufl.edu

Abstract. Fault injection attacks are one of the most powerful forms of cryptanalytic attacks on ciphers. A single, precisely injected fault during the execution of a cipher like the AES, can completely reveal the key within a few milliseconds. Software implementations of ciphers, therefore, need to be thoroughly evaluated for such attacks. In recent years, automated tools have been developed to perform these evaluations. These tools either work on the cipher algorithm or on their implementations. Tools that work at the algorithm level can provide a comprehensive assessment of fault attack vulnerability for different fault attacks and with different fault models. Their application is, however, restricted because every realization of the cipher has unique vulnerabilities. On the other hand, tools that work on cipher implementations have a much wider application but are often restricted by the range of fault attacks and the number of fault models they can evaluate.

In this paper, we propose a framework, called FEDS, that uses a combination of compiler techniques and model checking to merge the advantages of both, algorithmic level tools as well as implementation level tools. Like the algorithmic level tools, FEDS can provide a comprehensive assessment of fault attack exploitability considering a wide range of fault attacks and fault models. Like implementation level tools, FEDS works with implementations, therefore has wide application. We demonstrate the versatility of FEDS by evaluating seven different implementations of AES (including bitsliced implementation) and implementations of CLEFIA and CAMELLIA for Differential Fault Attacks. The framework automatically identifies exploitable instructions in all implementations. Further, we present an application of FEDS in a Fault Attack Aware Compiler, that can automatically identify and protect exploitable regions of the code. We demonstrate that the compiler can generate significantly more efficient code than a naïvely protected equivalent, while maintaining the same level of protection.

Keywords: Fault Attacks, Automatic Fault Attack Evaluation, Security Aware Compilers, Differential Fault Attacks

1 Introduction

Fault attacks are a potent form of physical attacks, where the attacker injects a fault during the execution of a cipher to determine its secret key. The fault, typically induced by a glitch in the voltage or clock lines of the device, or by an optical beam, momentarily disturbs encryption, producing a *faulty ciphertext*. The attacker analyzes the faulty ciphertext with the correct version to determine the secret key. Since the pioneering work of Biham and

Received: 2019-10-15 Accepted: 2019-12-15 Published: 2020-03-02

[1] Shamir in [BS97], there has been a large body of work on such fault injection attacks. These works considerably enhanced the power of fault injection attacks by exploiting cipher properties, such as its differential [BS97] and algebraic characteristics [CN10], key schedule [TFY07], and diffusion functions [Muk09]. The most powerful fault attacks, for instance, require a single, precisely timed fault injection to reveal the entire secret key of ciphers such as the AES, within a few milliseconds [TMA11].

Evaluating the vulnerability of a cipher to fault injection attacks is important to design efficient countermeasures. A naïve implementation could potentially overuse countermeasures resulting in large overheads in design cost, speed, and area. A significant reason for these design overheads is the fact that developers are unaware of exactly which parts of the design are vulnerable to fault attacks. Therefore, they tend to apply countermeasures for the entire design. Identifying the vulnerable regions in the design is non-trivial and very specific, not only to the cipher algorithm but also to its implementation.

In the past, fault attack vulnerability evaluation was a tedious manual task, with no formal guarantees of completeness. More recently, automated tools have been proposed that can quickly and efficiently evaluate fault attack vulnerabilities. Some of these tools can output results that are both sound and complete. Most of these tools [ZGZ+16, KRH17, SKMD17, SMD18, SJP+19] take a high-level representation of the cipher as input and identify the vulnerable operations. We call these tools *High-Level Evaluation* (HLE) tools. The high-level representation helps abstract low-level implementation details and also specify various cipher related properties, such as linear and non-linear functions, the cipher's differential and algebraic properties, and diffusion functions. The high-level specification represents various operations in the cipher as nodes in an information flow graph. For instance, each operation in AES outputs 16 bytes, thus each operation forms 16 nodes in the information flow graph. The AES information flow graph therefore has 640 $(= 16 + (9 \times 4 \times 16) + (3 \times 16))$ nodes present.The small number of nodes makes it feasible to have a comprehensive fault attack evaluation for a variety of complex fault injection attacks such as differential fault attacks (DFA) [TMA11], impossible differential fault attacks (IDFA) [DFL11], algebraic fault attacks (AFA) [CN10], etc. The approach can also cater to many fault models, including random faults and multiple fault scenarios, data and instruction corruption, and instruction skip. The HLE approach, however, has limited application because it is restricted to the cipher's algorithm and cannot assess implementations. This is a huge limitation because each cipher algorithm can be implemented in many different ways. Each implementation has a unique set of vulnerabilities and it becomes an onus of the user to bridge the gap between the high-level fault analysis and the implementation.

An alternate direction of research is to build automatic vulnerability analysis tools that work directly with the cipher's implementation rather than its high-level representation. We call such tools *Implementation Level Evaluation* (ILE) tools. Tools such as those proposed by Breier et al. [BHL18] and Hou et al. [HBZL19] analyze assembly code, while tools such as those proposed by Agosta et al. [ABMP13, ABPS14] work on a compiler generated intermediate representation. These tools have wide applicability since they can be applied to every software realization of the block cipher. The tools proposed, build graphs from the given source code, with nodes representing operations in the implementation and edges representing the information flow in the code. These graphs can become considerably large. For instance, the graph for an AES implementation has around 6,000 nodes and 9,000 edges [BHL18, HBZL19]. Such large graphs restrict the amount of processing done and the information extracted by a tool. Due to these limitations, some of the ILE tools are restricted to very simple fault attack models, such as bit-flips and cannot evaluate more practical fault attack situations such as random byte fault or multiple byte fault situations [ABPS14]. Further, all ILE tools [BHL18, HBZL19, ABMP13, ABPS14] utilize minimal features in the evaluation of complex cipher properties. Properties such as

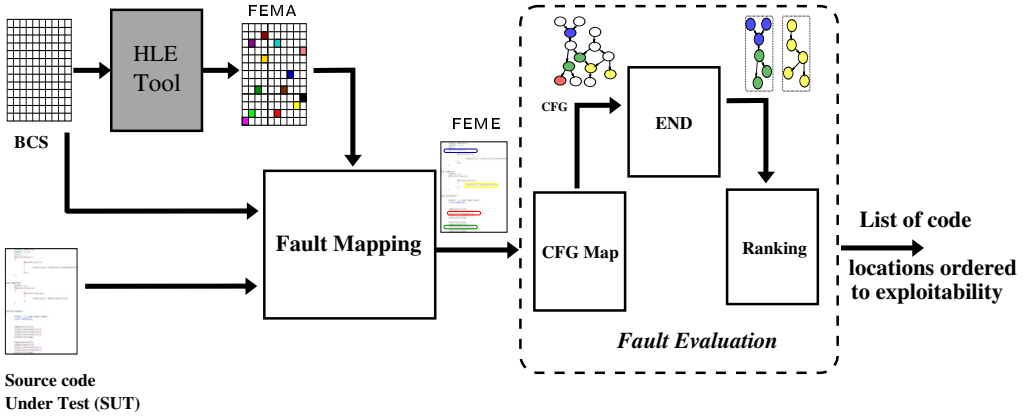


Figure 1: Overview of FEDS Framework to detect exploitability in software implementation. High-Level Evaluation (HLE) tools outputs Fault Exploitable Map in Algorithms (FEMA) from the high-level representation of the block cipher. *Fault Mapping* maps FEMA to FEME (Fault Exploitable Map in Implementation). *Fault Evaluation* works on the Intermediate Representation of the block cipher implementation and determines all fault exploitable instructions in the program.

differential and algebraic properties of the `S-Boxes` and reversibility of its key schedule, that can greatly influence the success of a fault attack, are not considered. This significantly narrows the scope of the fault vulnerability assessment to very simple attack situations.

Our Contribution. In this paper, we present a fault-attack exploitability detection framework called FEDS that has the best features of both the HLE and the ILE tools. Similar to ILE tools, FEDS works at the implementation level and can pinpoint vulnerable locations in the source code. This makes FEDS widely applicable and usable for different implementations of a cipher. Similar to the HLE tools, FEDS promises to provide a comprehensive fault attack vulnerability evaluation and can identify vulnerabilities in complex fault attack situations. It can consider the cryptographic properties of the cipher, such as the differential and impossible differential properties of the `S-Boxes`, reversibility of the key expansion algorithm, etc. It can cater to both random faults as well as single bit-flips. FEDS, thus, provides a comprehensive fault attack evaluation for software programs, having the advantages of both HLE and ILE tools.

At a high-level, FEDS takes as input a high-level representation of the cipher and the Source code Under Test (SUT) as shown in Figure 1. It first runs an HLE tool, such as XFC [KRH17] or ExpFault [SKMD17], on the cipher's high-level representation, to obtain a list of fault attack exploitable operations present in the cipher. A fault injected in any of these operations is exploitable. FEDS then maps these exploitable operations onto the SUT, thus identifying regions in the source code where these exploitable operations are implemented. For each of these exploitable source code regions, FEDS determines the cone-of-influence – thereby identifying the precise instructions, which are exploitable. The output of FEDS is a list of instructions, where a fault if injected, can be exploited.

A high-level overview of FEDS is provided in Figure 1. It consists of three main phases: *(1)* HLE, *(2)* `Fault Mapping`, and *(3)* `Fault Evaluation`. FEDS is agnostic about the HLE tool used; it simply requires compatibility with the input and the output representation. In this work, we adopt a Block Cipher Specification Language (BCSL) [RRHB19] to represent the input and the output to the HLE. This HTML-like language captures operations of the block cipher and the information flow from plaintext to ciphertext. The output of FEDS depends on the capabilities of the HLE tool. For example, if the underlying HLE

tool can evaluate ciphers only for Differential Fault Analysis [TMA11], then the output of FEDS is a set of instructions that are exploitable by Differential Fault Analysis alone. The `Fault Mapping` phase uses a model checker to map the Block Cipher Specification (BCS) onto the SUT. Every operation present in the BCS is mapped to its equivalent region of code in the SUT. Thus, the exploitable operations in the block cipher are mapped to code regions in the SUT. The `Fault Evaluation` phase identifies all the instructions, that when disturbed by a fault, can create a state that is exploitable. The output of FEDS is a list of these instructions. To summarize, the contributions of this paper are as follows:

- We propose the first comprehensive fully automated fault attack evaluation framework for block cipher source code which can handle complex fault attack situations and consider cipher properties such as differential and impossible differential properties of the `S-Box`.
- We evaluate the scalability, efficacy of the FEDS framework with seven different implementations of AES-128, and implementations of CLEFIA-128 and CAMELLIA-128. These implementations vary with respect to the size and number of look-up tables used and the realization of the operations. We also evaluate a bit-sliced implementation of AES-128, showing the scalability of FEDS to non-traditional implementations.
- We incorporate FEDS in the LLVM compiler and demonstrate an application to automatically insert fault-attack countermeasures during compilation. Thus, we propose the first Fault-Attack Aware Compiler. The countermeasures are targeted to specific locations in the source code that are found to be exploitable. We demonstrate that these countermeasures are far more efficient compared to a naïve approach.

The paper is organized as follows: Section 2 provides the necessary background. Section 3 includes the recent works for fault vulnerability detection. Section 4 includes the representations used in FEDS for block cipher algorithms and the implementation. Section 5 gives the detailed description of the FEDS framework. Section 6 includes the evaluation of FEDS on different implementations of AES and block ciphers CLEFIA and CAMELLIA. Section 7 discusses a use case of FEDS, to design a Fault Attack Aware Compilers framework. Section 8 concludes the paper.

2 Preliminaries

In this section, we provide an overview of fault attacks. The section also introduces the intermediate representation format used in LLVM compilers [LA04] and a brief description of Model Checking and Equivalence Mapping tools.

2.1 Fault Attacks

In a fault attack, the attacker corrupts the output of an operation by injecting a fault during the cipher execution [BDL97]. The fault propagates through the cipher resulting in a faulty ciphertext. The attacker uses this faulty ciphertext to retrieve bits of the secret key. Several fault injection techniques have been used in the past, such as injecting glitches in voltage and clock sources, electromagnetic radiation, and laser beams.

The potency of a fault attack is measured by the number of key bits that a fault can deliver and the offline attack complexity. In the first fault attack on AES, for example, a single bit fault injected just before the final `AddRoundKeys` was used to retrieve one bit of the secret key [Joh03]. Thus, retrieving the entire secret key of AES-128 would require 128 faults. Over the years researchers have identified better locations in the cipher to inject

#	Program Expressions	#	IR Instructions	#	IR Instructions
...	...	...	...	...	...
1	uint16_t t,x	$\mathcal{I}_{305}$	%a0 = get[1]*%state,i64 0,i64 0		c.false :
2	t1 = state[0]∧ state[1];	$\mathcal{I}_{306}$	%0 = load i8, i8* %a0	$\mathcal{I}_{327}$	%8 = load i8, i8* %t1
3	t2 = state[2]∧ state[3];		...	$\mathcal{I}_{328}$	%shl1 = shl i8 %8, 1
4	t = t1∧ t2;	$\mathcal{I}_{310}$	%xor1 = xor i8 %0,%1	$\mathcal{I}_{329}$	br label %c.end
5	x = ((t1)&0x80)	$\mathcal{I}_{311}$	store i8 %xor1, i8* %t1		c.end:
	? ((t1≪1)∧0x1b)	$\mathcal{I}_{312}$	%a1 = get[1]*%state,i64 0,i64 2	...	...
	: ((t1≪1)∧1);		...	$\mathcal{I}_{332}$	Load i8 %8, i8* %x
6	y = x;			$\mathcal{I}_{333}$	store i8%8, i8 %y
7	out[0] = x	$\mathcal{I}_{317}$	store i8 %xor2, i8* %t2	$\mathcal{I}_{334}$	%9 = load i8, i8* %x
	∧ state[0] ∧ t	$\mathcal{I}_{318}$	%4 = load i8, i8* %t1	$\mathcal{I}_{335}$	%a3 = get[1]*%state,i64 0,i64 0]
	∧ *rks++;		...	$\mathcal{I}_{336}$	%10 = load i8, i8* %a3
		$\mathcal{I}_{321}$	%tobool = icmp ne i8 %and, 0	$\mathcal{I}_{337}$	%xor5 = xor i8 %9, %10
		$\mathcal{I}_{322}$	br i1 %bool,label %c.true,label %c.false	$\mathcal{I}_{338}$	%11 = load i8, i8* %t
			c.true:	$\mathcal{I}_{339}$	%xor6 = xor i8 %xor5, %11
		$\mathcal{I}_{323}$	%7 = load i8, i8* %t1		...
		$\mathcal{I}_{324}$	%shl = shl i8 %7, 1	$\mathcal{I}_{344}$	%13 = load i8*, i8** %out.addr
		$\mathcal{I}_{325}$	%xor4 = xor i8 %shl, 27	$\mathcal{I}_{345}$	%a4 = get[1]i8, i8* %13, i64 0
		$\mathcal{I}_{326}$	br label %c.end	$\mathcal{I}_{346}$	%store i8 %xor7, i8* %a4

[1] getelementptr inbounds [16 x i8], [16 x i8]

Figure 2: The software implementation for single byte `MixColumns` operation of AES and corresponding IR Instructions. Different colors represent the mapping of operations in software implementation to IR instructions. The IR instructions are numbered $\mathcal{I}_{305}$ to $\mathcal{I}_{346}$ according to their positions considering the whole AES program.

faults. They also made use of the cipher's properties such as its differential [Muk09, DFL11] and algebraic characteristics [CN10], diffusion, and key schedule [TFY07], to maximize the number of key bits obtained and reduce the offline attack complexity. The most powerful fault attacks today, can retrieve the entire AES key within a few milliseconds with just a single fault injection [TMA11].

The potency of the attack mandates that cipher implementations be evaluated for their vulnerability to fault attacks. Recently, several tools such as [BHL18, HBZL19, ABPS14], were proposed for evaluating fault attack vulnerabilities in software implementations of ciphers. However, all these works are restricted to evaluating simple fault attack models. Complex properties of the cipher such as its differential, algebraic properties, diffusion and key schedule were not considered. In this paper, we use a combination of model checking and compiler techniques to evaluate the exploitability of software implementations considering complex cipher properties.

2.2 Intermediate Representation of a Program

The compiler converts high-level implementations to machine code in different steps. The transformation pass is one among them, which converts a high-level implementation to *Intermediate Representation* (IR) instructions. The IRs used by the LLVM compiler is in *Static Single Assignment* (SSA) form, where variables in every assignment are used only once [RWZ88].

Definition 1. [Static Single Assignment] *Static Single Assignment (SSA), is a format for program representation, where the program variables are assigned exactly once and every variable is defined before its use.*

For example, an assignment $x = x + 1$ is converted to $x_1 = x_0 + 1$. The variable x is renamed to x_0 before x is assigned and the next value of x is replaced with x_1.

Definition 2. [IR Instructions] *IR is an intermediate representation used by the LLVM compiler that is generated during the transformation pass. Each IR instruction is in SSA form, consisting of an opcode, read and write operands. The read operands are defined by the "used variable" set, $\mathbb{U}(n)$ and write operands are defined by the "defined variable" set, $\mathbb{D}(n)$, where n denotes the n-th IR instruction.*

Example 1. The IR instructions for an output byte of the `MixColumns` operation of AES are given in Figure 2. The column, `Program Expressions` show the software implementation for this operation. The second and third columns show the equivalent IR instructions generated by the LLVM compiler. The labeling $\mathcal{I}_{305}$, $\mathcal{I}_{306}, \cdots, \mathcal{I}_{346}$ corresponds to the first output byte of `MixColumns` in the first round.

Example 2. Consider the IR instruction ($\mathcal{I}_{339}$) in Figure 2. The opcode is xor that works on 8 bit integers (i8). The instruction performs %xor6 $\leftarrow$ %xor5 $\oplus$ %11. Thus, $\mathbb{U}(\mathcal{I}_{339}) = \{\%\text{xor5}, \%11\}$, while $\mathbb{D}(\mathcal{I}_{339}) = \{\%\text{xor6}\}$. We refer the reader to the manual[1] for more details about the IR formats.

FEDS converts the IR instructions to Control Flow Graph (CFG) with each instruction in the IR represented as a node in the CFG and edges denote the program flow.

2.3 Model Checking

Model checking [CGP00] is a technique of ensuring the correctness of a system by systematic exploration through its underlying mathematical models. The given model of the system is exhaustively and automatically verified with respect to the given specification. The model checker takes the high-level abstraction of the design and the specification as input. It leverages satisfiability (SAT) solvers [MMZ+01] to ensure the functional correctness of the model with respect to the specification. To do that, it generates a set of constraints and properties from the formal description of the model and the specification which is then evaluated using the in-built SAT solvers [CKL04].

Equivalence checking is a technique to formally prove whether two representations of a model at different levels of abstraction have the same behavior. FEDS uses a model checker to perform equivalence checking between different abstractions of a block cipher; specifically a high-level abstraction with an implementation-level. The model checker generates logical formula from the given input descriptions of the model and checks whether the outputs are equivalent for all possible inputs.

The fundamental intuition behind model checking using SAT solvers is to check for the satisfiability of a set of constraints derived from the model, say $\mathcal{C}$, with another set of constraints derived from the negated properties, say $\neg\mathcal{P}$. Mathematically, if $(\mathcal{C} \wedge \neg\mathcal{P})$ is satisfiable, then it indicates that there are some common behaviors in the model which do not match with the properties – thereby pointing to an error in the model itself. In case of satisfiability, the SAT solvers are also capable of indicating the satisfying assignments of the participating variables. On the other hand, if $(\mathcal{C} \wedge \neg\mathcal{P})$ remains unsatisfiable, then it indicates that the model constraints do not overlap with the negated properties and hence the model may be treated as correct with respect to the properties provided. Note that, in these cases, we assume that the properties are inherently correct (golden) and hence we try to find errors in the model for any satisfaction that is obtained by the SAT solvers for $(\mathcal{C} \wedge \neg\mathcal{P})$.

3 Related Work

Automated fault attack evaluation tools are important to gauge the vulnerability of crypto systems. In the last few years, there have been few works that propose tools for automatic detection of fault attack vulnerabilities in block ciphers. These works can be classified as either *High Level Evaluation* (HLE) tools [KRH17, SKMD17, SJP+19] or *Implementation Level Evaluation* (ILE) tools [ABPS14, BHL18, HBZL19]. Table 1 provides a comparison of these tools.

[1] https://llvm.org/docs/LangRef.html

Table 1: Comparison with the state-of-the-art automatic fault attack evaluation tools.

Tool		Fault Model	Cryptographic properties	Input Type	Output Type
HLE	XFC [KRH17]	Single Byte	Differential Properties of S-Box	High level representation of block cipher	Derivable key bits, Attack complexity
	ExpFault [SMD18]	Single byte	Impossible Differential Properties of S-Box	High level representation of block cipher	Exploitable fault locations, Attack complexity
	AFA [ZGZ+16]	Single Byte	Algebraic Properties of cipher	High level representation of block cipher	Exploitable locations,
ILE	DATAC [BHL18]	Random Byte Instruction skip	N/A	Assembly code	Vulnerable instructions
	TADA [HBZL19]	Random Byte	N/A	Assembly code	Attack Details for last round key
	ADFA [ABPS14]	Bit flip	N/A	LLVM IR	Vulnerable location in LLVM IR
Proposed	FEDS	Similar with HLE Tool	Similar with HLE Tool	Source Code	List of exploitable instructions

HLE tools work on a high-level specification of the block cipher. The specification contains cipher operations and critical characteristics like the differential and algebraic characteristics of the `S-Box`, branch numbers of diffusion layers, and properties of the key-expansion algorithm. One of the first works in this direction was by Khanna, Rebeiro, and Hazra [KRH17], who introduced a framework called XFC to evaluate a block cipher algorithm for vulnerabilities to Differential Fault Attacks (DFA) using a coloring scheme for the functions in the cipher. Saha et al. [SMD18, SJP+19], similarly identified DFA vulnerabilities in block ciphers with the help of data mining and machine learning. Besides DFA, tools could also evaluate cipher algorithms for Impossible Differential Fault Attacks (IDFA) [DFL11]. Zhang et al. [ZGZ+16] used SAT solvers to evaluate ciphers for vulnerabilities to Algebraic Fault Attacks. Their framework converts the cipher algorithm and fault models to algebraic equations, which are solved using the SAT solver.

ILE tools work on implementations of the block ciphers. Agosta et al. [ABPS14] works at the compiler level using an intermediate representation of the program to determine single bit-flip vulnerabilities in software code. The framework aims at determining the vulnerable locations in the cipher but fails to determine the attack complexity. In [BHL18, HBZL19], the assembly code of the block ciphers are represented as a data flow graph (DFG) to identify vulnerable nodes. In [BHL18], exploitable instructions are determined by manual analysis of the vulnerable nodes. Later, Hou et al. [HBZL19] used an SMT solver to solve the final stage equations generated for the last round key.

While HLE tools can comprehensively evaluate ciphers considering different fault models and cipher properties, they have restricted application because they can only be used for algorithm evaluation. ILE tools, can evaluate different realizations of the block cipher algorithm. However, the large state space makes comprehensive evaluation difficult. Hence, most ILE tools are limited to simple fault attack scenarios and do not consider cipher properties that can have a considerable impact on a fault attack. Our approach, named FEDS, has the advantages of both techniques. It can be used to provide a comprehensive evaluation of cipher implementations for a variety of complex fault attack conditions considering various cipher properties. The fundamental idea is to use any existing HLE tool comprehensively evaluate a block cipher algorithm, and then use a fault mapping technique which map the result from the HLE tool to the software realization of the algorithms.

4 Block Cipher Algorithms and their Implementations

Block Cipher Algorithms. A block cipher encryption can be defined as a function $E_k = P \rightarrow C$, which maps the plaintext P to ciphertext C based on a secret key k. Decryption is the reverse operation, mapping from ciphertext to plaintext. Block cipher encryption and decryption algorithms have a fixed number of rounds. Each round has a

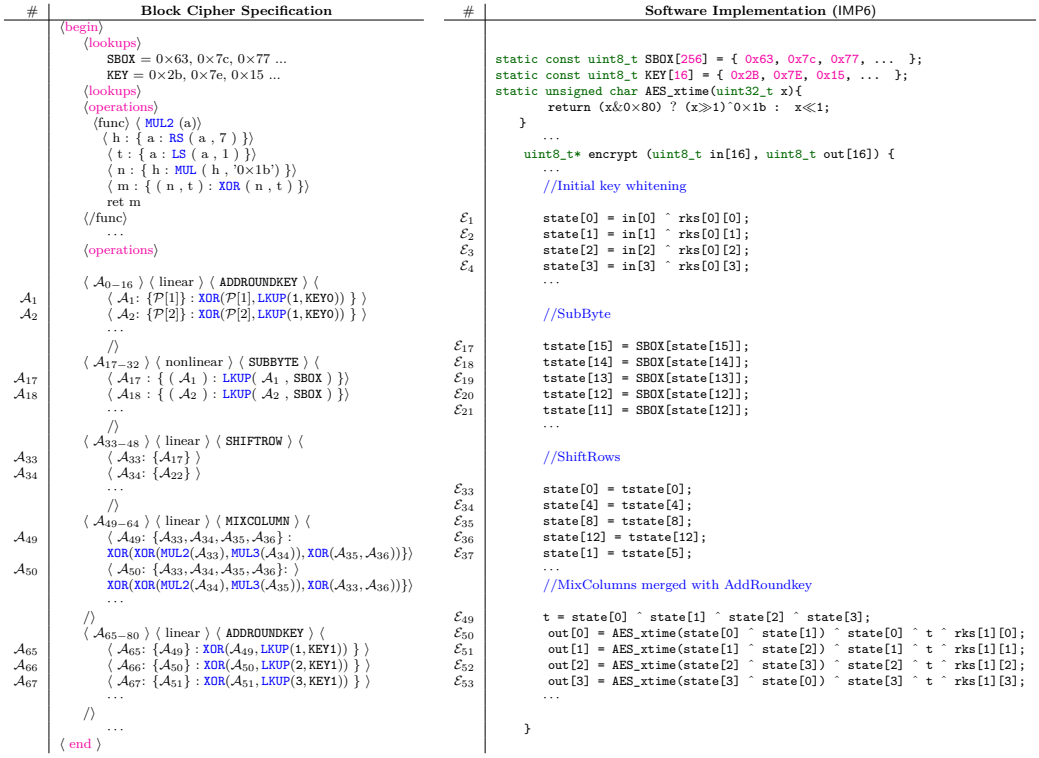


Figure 3: 1^{st} round representations of AES in BCS and Software implementation.

fixed number of operations (N), some of these operations are non-linear such as substitution boxes (`S-Boxes`); others are linear operations such as diffusion or key addition. The entire encryption can be formalized as follows:

$$C = (\mathcal{A}_L \circ \mathcal{A}_{L-1} \circ \ldots \circ \mathcal{A}_3 \circ \mathcal{A}_2 \circ \mathcal{A}_1)(P) \ , \tag{1}$$

where P is the plaintext, C is the ciphertext and $\mathcal{A}_i$ $(1 \leq i \leq L)$ are sub-operations in the cipher. For example, AES-128 [Pub01] has 10 rounds, and each round has 4 different operations. For the first round, $\mathcal{A}_1$ to $\mathcal{A}_{16}$ denotes the initial `Key Whitening` and $\mathcal{A}_{17}$, $\mathcal{A}_{18}$, $\cdots$, $\mathcal{A}_{32}$ the `SubBytes` operation; $\mathcal{A}_{33}$, $\mathcal{A}_{34}$, $\cdots$, $\mathcal{A}_{48}$ the `ShiftRows` and the remaining are the `MixColumns` and `KeyAddition` operations. For AES, the entire sequence will thus have 640 (*i.e.* $L = 640 = 16 + (9 \times 4 \times 16) + (3 \times 16)$ sub-operations, corresponding to initial key whitening, 9 rounds with 4 operations each having 16 sub-operations and the 10^{th} round with 3 operations with 16 sub-operations.

Each sub-operation $\mathcal{A}_i$ $(1 \leq i \leq L)$ is either linear or non-linear and can be represented as

$$out_a \leftarrow \mathcal{A}_i(in_1, in_2, \cdots) \ , \tag{2}$$

with inputs $in_1, in_2 \cdots$ and output out_a. The inputs are either the plaintext or outputs of previous sub-operations.

In [RRHB19], Roy et al. introduced the Block Cipher Specification Language to provide a high-level representation of a cipher algorithm. The Block Cipher Specification (BCS) captures various operations performed by the cipher and the information flow of the plaintext through the algorithm. The left-hand side of Figure 3 shows a snippet of the first round of AES represented in the Block Cipher Specification Language. Each sub-operation is represented in the following format:

$$\langle \text{ sub-operation} : \{\text{dependent sub-operations}\} : operation(operand1, operand2, \cdots) \} \rangle \ .$$

Table 2: Different Implementations of AES (all unrolled).

#	Implementation Types	Description
IMP1	Simple LookUp Table	Implements each AES operation in a different function. Uses 256 byte lookup table for the `SubBytes` and `ShiftRows` follows `SubBytes`
IMP2	Interchanged Operations	Similar to IMP1 except `SubBytes` follows `ShiftRows` in a round
IMP3	T-Tables	OpenSSL Implementation; Uses 4 T-tables, each of 1KB. The T-table combine `SubBytes`, `ShiftRows`, and `MixColumns` The 5-th table is used exclusively for the last round
IMP4	Compressed Table	OpenSSL Implementation; Uses 1 T-table of 2KB. The T-table combines `SubBytes`, `ShiftRows`, and `MixColumns`
IMP5	Merged Operations	Similar to IMP1 except `ShiftRows` done implicitly with `SubBytes`.
IMP6	Running Example (Figure 3)	Similar to IMP1 with order within `SubBytes` and `ShiftRows` is random, `MixColumns` and `AddRoundKeys` are merged
IMP7	Bitsliced Implementation	Bitwise Operations for all sub-operations with `SubByte` `MixColumns` and `AddRoundKeys` are merged

For example, the sub-operation $\mathcal{A}_1$ (Figure 2), specifies the first round key addition. It uses a table to look-up the first key byte and XORs it with first plaintext byte $\mathcal{P}[1]$.

$$\langle\ \mathcal{A}_1 : \{\mathcal{P}[1]\} : \texttt{XOR}(\mathcal{P}[1], \texttt{LKUP}(1, \texttt{KEY0}))\}\ \rangle\ .$$

Block Cipher Implementations. Implementations of block ciphers differ considerably depending on the target application and the underlying platform. For instance, typical AES implementations on 32-bit platforms use 5 T-tables[2], where the operations `SubBytes`, `ShiftRows`, and `MixColumns` are merged and replaced with look-ups to four 1KByte tables. A memory conscious implementation, similarly merges the three AES operations using a single 2KByte table[3]. On 8-bit memory constrained processors, `SubBytes` is implemented using a single 256 byte look-up table, while `ShiftRows` and `MixColumns` are combined[4]. Thus, given a block cipher algorithm, different programs would implement the algorithm in different ways. However all the programs would be functionally equivalent. Table 2 shows seven popular implementations of AES.

To formally capture a block cipher implementation, we represent the implementation as a sequence of program expressions, which realize the block cipher sub-operations $\mathcal{A}_1$, $\mathcal{A}_2$, $\cdots$, $\mathcal{A}_L$ as follows:

$$C = E_k(P) = (\mathcal{E}_1,\ \mathcal{E}_2,\ \mathcal{E}_3, \cdots, \mathcal{E}_{M-2},\ \mathcal{E}_{M-1},\ \mathcal{E}_M)\ , \tag{3}$$

where M is the number of program expressions in the implementation, and each expression $\mathcal{E}_i$ is a combination of arithmetic, logical, or memory operations present in the program. The output of the expression is out_e as shown below:

$$\mathcal{E}_i\ :\ out_e \leftarrow \langle\text{arithmetic/logical/memory expression in implementation}\rangle\ . \tag{4}$$

The expressions and the value of M vary for different implementations of a cipher. The right-hand side of Figure 3 shows a typical C implementation (IMP6, in Table 2) of the first round of AES along with the various expressions marked.

Fault Exploitability Maps. We define a fault exploitability map as the set of *locations* in the block cipher where a fault injected during the cipher operation can be exploited to derive bits of the secret key. The term 'locations' has a different meaning for algorithms and implementations. For block cipher algorithms, the fault exploitability map denoted FEMA, is a set of sub-operations such as

$$\text{FEMA} = \{\mathcal{A}_{i_1},\ \mathcal{A}_{i_2},\ \mathcal{A}_{i_3},\ \cdots\}\ .$$

[2]OpenSSL (version 3.0) implementation with 4 lookup tables, each table occupying 1024 bytes: https://github.com/openssl/openssl/blob/master/crypto/aes/aes_core.c

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

[4]Gladman's implementation of AES for embedded platforms : https://github.com/BrianGladman/aes/blob/master/aestab.c

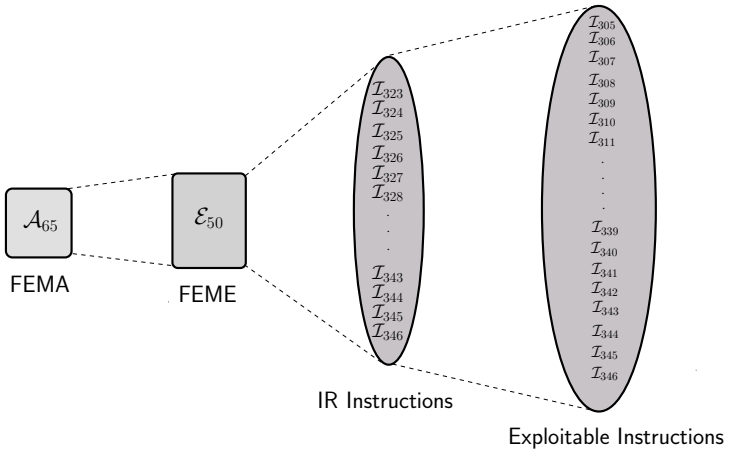


Figure 4: The `Fault Mapping` of FEDS determines mapping from the FEMA to FEME. `Fault Evaluation` maps FEME to corresponding IR Instructions and also determine the all the exploitable instructions, *i.e.* instructions that can corrupt the output of program expressions that are present in FEME.

A fault injected in any of these sub-operations is exploitable. HLE tools such as XFC [KRH17] and ExpFault [SMD18] can automatically identify these exploitable sub-operations for DFA and IDFA respectively. As shown in Table 1, the HLE tools consider cryptographic properties of the cipher to arrive at a comprehensive fault exploitability assessment.

The term 'locations' in block cipher implementations refers to a set of program expressions that are exploitable. The fault exploitability map for implementations denoted FEME, is a set of exploitable program expressions such as

$$\text{FEME} = \{\mathcal{E}_{j_1},\ \mathcal{E}_{j_2},\ \mathcal{E}_{j_3},\ \mathcal{E}_{j_4},\ \cdots\}\ .$$

We thus have two sets. Exploitable sub-operations present in FEMA are mapped to exploitable program expressions in FEME. For example, assume that the sub-operation $\mathcal{A}_{65}$ in the BCS (LHS in Figure 3) is exploitable. The sub-operation $\mathcal{A}_{65}$ is realized through $\mathcal{E}_{50}$ in the implementation (RHS in Figure 3). During compilation, $\mathcal{E}_{50}$ gets converted to one or more IR instructions. Figure 2 shows the IR instructions for $\mathcal{E}_{50}$ comprising of instructions $\mathcal{I}_{323}, \mathcal{I}_{324}, \cdots, \mathcal{I}_{346}$. A fault in any instruction $\mathcal{I}_{305}$ to $\mathcal{I}_{346}$ will manifest as a faulty output of $\mathcal{E}_{50}$. Figure 4 depicts these relationships. Thus, $\mathcal{A}_{65} \in$ FEMA and $\mathcal{E}_{50} \in$ FEME.

Since each block cipher implementation is structurally different, the FEME for each implementation would be different. ILE tools such as DATAC [BHL18], TADA [HBZL19], and ADFA [ABPS14] can pinpoint some of these exploitable instructions. However, since these tools do not consider cryptographic properties of the cipher that may aid the attacker (refer Table 1), the exploitable instructions provided by these tools is incomplete.

In the next section we propose our framework called FEDS, that can comprehensively identify exploitable locations in block cipher implementations. Like the ILE tools, it works on the cipher implementation and therefore can identify exploitable program expressions. Unlike ILE tools, FEDS also considers cryptographic properties of the cipher and therefore provides a comprehensive set of exploitable instructions for the implementation.

5 The FEDS Framework

FEDS is designed based on the fundamental observation that, if a sub-operation in the

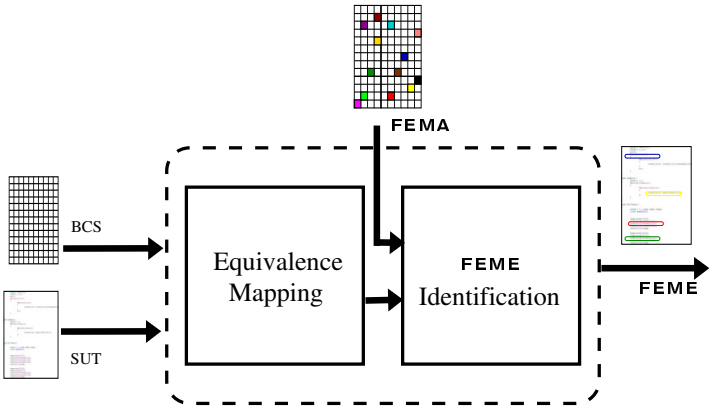


Figure 5: `Fault Mapping` module in FEDS, (1) *Equivalence Mapping* module maps all the sub-operations in BCS to program expression in the SUT, (2) FEME *Identification* maps all the sub-operations in FEMA to exploitable program expression FEME.

block cipher algorithm is exploitable then a fault induced in any of the program expressions that realize this sub-operation is also exploitable. Proposition 1 formally states this.

Proposition 1. Let $\mathcal{A}_i$ $(1 \leq i \leq L)$ be a sub-operation in a block cipher and $\mathcal{E}_j$ $(1 \leq j \leq M)$ the realization of $\mathcal{A}_i$ in a block cipher implementation. If $\mathcal{A}_i \in$ FEMA then, the expressions $\mathcal{E}_j \in$ FEME.

At a high-level FEDS, uses an HLE tool to identify the fault exploitable map FEMA for the block cipher. It then uses the `Fault Mapping` module to map the exploitable sub-operations in the FEMA to program expressions of the Source code Under Test (SUT). This gives the fault exploitable map FEME for the implementation.

The output of an exploitable program expression can be corrupted in multiple ways. For example, the output of expression $\mathcal{E}_{50}$ can be corrupted by inducing a fault in expressions $\mathcal{E}_{50}$, $\mathcal{E}_{49}$, `state` variables and instructions in `AES_xtime` function etc. (refer Figure 3). In the third phase, FEDS searches for all possible ways by which faults can be induced in a given exploitable expression as shown in Figure 4. This search is implemented using the `Fault Evaluation` present in FEDS (refer Figure 1), which constructs a control flow graph from the compiler generated IR instructions of the SUT. The output of the module is a list of IR instructions that can induce a fault for a given sub-operation.

5.1 High-Level Exploitability Tool

FEDS uses the Block Cipher Specification Language (refer Figure 3) [RRHB19] to provide a high-level representation of the cipher. This specification is the input to the HLE tool, which identifies all exploitable sub-operations of the cipher. While any HLE tool can be used with FEDS, in this work, we use tools called XFC [KRH17] and ExpFault [SMD18]. While XFC uses a coloring scheme to evaluate a block cipher for vulnerabilities to Differential Fault Attacks, ExpFault can also evaluate a block cipher's vulnerabilities to Impossible Differential Fault Analysis (IDFA). The output of both these tools is the set FEMA comprising of exploitable sub-operations of the cipher.

Fault Models and Fault Attacks Evaluated. FEDS is limited by the fault model and the fault attacks that the HLE tool supports. For instance, XFC [KRH17] is restricted to a single random byte fault model and can evaluate block ciphers for differential fault attacks. If XFC is used in FEDS for the high-level evaluation, then the fault model for FEDS is also restricted to evaluating single random byte faults and Differential Fault

Attacks. It may be noted that multiple HLE tools can be used with FEDS to obtain a stronger evaluation. For example, creating the FEMA from the union of outputs obtained from tools like XFC [KRH17], ExpFault [SMD18] and [ZGZ+16], would enable FEDS to evaluate Differential Fault Attacks (supported by XFC), Impossible Differential Fault Attacks (supported by ExpFault) and Algebraic Fault Attacks (supported by [ZGZ+16]).

5.2 Fault Mapping

This module has two components as shown in Figure 5. The first, called `Equivalence Mapping`, maps the block cipher specification to program expressions in the SUT. The second module called FEME `Identification`, then uses the set FEMA obtained from the HLE to construct the set of exploitable program expressions (*i.e.* the set FEME). This set can differ from one implementation to another as described in Section 4. Further complexities arise since programmers may use different strategies to realize the same functionality. Operations may be merged, shuffled, or interleaved as described in Table 2. However, we observe that even though programs can look very different, they typically have the following common underlying properties:

P1. *Completeness.* Any implementation of a block cipher must realize all the operations $\mathcal{A}_1$ to $\mathcal{A}_L$. This is mandatory to ensure that the implementation is complete.

P2. *Merge.* All operations within a round can be merged. For instance, IMP3 and IMP4 in Table 2, combine `SubBytes`, `ShiftRows`, and `MixColumns` operations using T-tables. Similarly, IMP5 combines the `SubBytes` and `ShiftRows`.

P3. *Interchange.* The positions of the functions may be interchanged. For example in AES, one implementation may perform `ShiftRows` followed by `SubBytes` (IMP2 in Table 2), while another may perform these (appropriately altered) operations in the reverse order.

We leverage these properties to map the sub-operations ($\mathcal{A}_1$, $\mathcal{A}_2$, $\cdots$, $\mathcal{A}_L$) from the block cipher specification to program expressions in the SUT ($\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3$, $\cdots$, $\mathcal{E}_M$). The mapping is done using a model checking tool that establishes equivalence between sub-operations and program expressions.

Equivalence Mapping. The *Equivalence Mapping* module in FEDS takes the Block Cipher Specification (BCS) and the Source code Under Test (SUT) as input. The only assumption we make is that loops in the SUT are unrolled. Figure 3 shows the code for BCS and the first round of a software implementation of AES (IMP6 in Table 2), where the order of `SubBytes` and `ShiftRows` are permuted compared to the BCS, and `MixColumns` is merged with `AddRoundKey`. The objective of Equivalence Mapping is to map the sub-operations in the Block Cipher Specification (*i.e.* $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$, $\cdots$, $\mathcal{A}_L$) to program expressions (*i.e.* $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3$, $\cdots$, $\mathcal{E}_M$). The module outputs a map that contains all the sub-operations ($\mathcal{A}$) in the BCS and the corresponding expressions ($\mathcal{E}$) in software code (SUT).

The major challenge in mapping is that the SUT can be implemented in many different ways as shown in Table 2. Operations can be merged and/or interchanged as presented in **P2** and **P3**. Thus, for a given pair of BCS and SUT, sub-operations to expressions can have a one-to-one, one-to-many, many-to-one and many-to-many mapping. For example, for the BCS and SUT given in Figure 3, the sub-operations $\mathcal{A}_1$ to $\mathcal{A}_{16}$ have one-to-one mapping with $\mathcal{E}_1$ to $\mathcal{E}_{16}$, while $\mathcal{A}_{49}$ and $\mathcal{A}_{65}$ have a many-to-many mapping with $\mathcal{E}_{49}$ and $\mathcal{E}_{50}$. Also $\mathcal{A}_{17}$ has a one-to-many mapping with $\mathcal{E}_{17}$, $\mathcal{E}_{18}$, $\cdots$,$\mathcal{E}_{32}$, and $\mathcal{A}_{50}$, $\mathcal{A}_{66}$ have a many-to-one mapping with $\mathcal{E}_{51}$. For the one-to-many and many-to-many mappings, the algorithm that we present records only the final program expression. The other expressions

Algorithm 1: `Equivalence_Mapper`

Input: BCS : $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \cdots, \mathcal{A}_L$
SUT : $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \cdots, \mathcal{E}_M$
Output: `E_Map` : List of mapped segments. Each entry comprises of sub-operations in the BCS that map to program expressions in the SUT.

1 **Define:** $\mathsf{MAX}_{\mathcal{A}}$ and $\mathsf{MAX}_{\mathcal{E}}$ are the maximum number of $\mathcal{A}$ and $\mathcal{E}$ that need to be considered.
2 Set $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \cdots, \mathcal{A}_L$ as unmapped
3 Set $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \cdots, \mathcal{E}_M$ as unmapped
4 `E_Map` ← NULL
5 **begin**
6 **while** *true* **do**
7 Let $i \geq 1$ be the smallest index such that $\mathcal{A}_i$ is unmapped
8 Let $j \geq 1$ be the smallest index such that $\mathcal{E}_j$ is unmapped
9 Let MA = $(\mathcal{A}_i, \mathcal{A}_{i+1}, \mathcal{A}_{i+2}, \cdots \mathcal{A}_{i+l}) : 0 \leq l \leq \mathsf{MAX}_{\mathcal{A}}$ and $(i+l) \leq L$
10 Let MI = $(\mathcal{E}_j, \mathcal{E}_{j+1}, \mathcal{E}_{j+2}, \cdots \mathcal{E}_{j+m}) : 0 \leq m \leq \mathsf{MAX}_{\mathcal{E}}$ and $(j+m) \leq M$
11 Find smallest values of l and m such that `EquivalenceChecker`(MA,MI) returns *true* (*i.e.* map is found)
12 **if** *Map found* **then**
13 **if** $l > 0$ **then**
14 //many-to-one and many-to-many mapping
15 Using the BCS, find the information flow path to $\mathcal{A}_{i+l}$. Let the path be represented as $(\mathcal{A}_{i_1}, \mathcal{A}_{i_2}, \mathcal{A}_{i_3}, \cdots, \mathcal{A}_{i_k}, \cdots \mathcal{A}_{i+l})$ such that all $\mathcal{A}_{i_k}$ are unmapped and $i \leq i_k$.
16 //$(\mathcal{A}_{i+l}, \mathcal{A}_{i_1}, \mathcal{A}_{i_2}, \cdots, \mathcal{A}_k)$ map to the same program expressions.
17 Insert $(\mathcal{A}_{i_1}, \mathcal{A}_{i_2}, \mathcal{A}_{i_3}, \cdots, \mathcal{A}_{i+l}) \mapsto (\mathcal{E}_{j+m})$ into `E_map`
18 Mark $\mathcal{A}_{i_1}, \mathcal{A}_{i_2}, \mathcal{A}_{i_3}, \cdots, \mathcal{A}_{i+l}$ as mapped.
19 Mark $\mathcal{E}_{j+m}$ as mapped.
20 **else**
21 //one-to-one and one-to-many mapping
22 Insert $(\mathcal{A}_i, \mathcal{A}_{i+1}, \mathcal{A}_{i+2}, \cdots \mathcal{A}_{i+l} \mapsto (\mathcal{E}_{j+m})$ into `E_map`
23 Mark $\mathcal{A}_i, \mathcal{A}_{i+1}, \mathcal{A}_{i+2}, \cdots \mathcal{A}_{i+l}$ as mapped.
24 Mark $\mathcal{E}_{j+m}$ as mapped.
25 **else**
26 break
27 **if** *there exist some* $\mathcal{A}_i$ *is unmapped* **then**
28 **return** `Failure`
29 **return** `E_Map`

in these maps are identified by a control flow graph constructed from the program during the Fault Evaluation. This is discussed in Section 5.3.

The algorithm used for `Equivalence Mapping` is given in Algorithm 1. At a high-level, the mapping algorithm iterates over the BCS and the SUT and uses an `EquivalenceChecker` to identify segments that are equivalent. The `EquivalenceChecker` extracts sub-operations and program expressions from the respective inputs and verifies that the output out_a from sub-operations (Equation 2) is equivalent to out_e from program expression (Equation 4) for all the possible combinations of input. The `EquivalenceChecker` internally invokes a model checker that first converts the sub-operations and the program expressions to static single assignment (SSA) form and then uses a SAT solver to verify for equivalence. More details about this is presented in Section 6.2.

Algorithm 1 works incrementally starting from the first sub-operation $\mathcal{A}_1$ and first program expression $\mathcal{E}_1$. For each iteration of the `while` loop (Lines 6 to 26), the algorithm finds the smallest contiguous sequences MA and MI (Line 9-10) comprising of cipher sub-operations from the BCS and program expressions from the SUT, which have not been mapped previously and for which the `EquivalenceChecker` returns *true*, indicating that the l sub-operations in MA maps to the m expressions in MI.

Depending on the software implementation, multiple mappings are possible. If the

Table 3: Iteration wise result of `Equivalence_Mapper` (Algorithm 1) for the software implementation (IMP6) given in Figure 3. Column 1 `#` shows the iterations count.

#	MA	MI	IR
	«AddRoundKey(0)»		
1	$\mathcal{A}_1$	$\mathcal{E}_1$	$\mathcal{I}_1 - \mathcal{I}_{12}$
2	$\mathcal{A}_2$	$\mathcal{E}_2$	$\mathcal{I}_{13} - \mathcal{I}_{24}$
...	...	...	...
15	$\mathcal{A}_{15}$	$\mathcal{E}_{15}$	$\mathcal{I}_{167} - \mathcal{I}_{180}$
16	$\mathcal{A}_{16}$	$\mathcal{E}_{16}$	$\mathcal{I}_{181} - \mathcal{I}_{192}$
	«SubByte»		
17	$\mathcal{A}_{17}$	$\mathcal{E}_{32}$	$\mathcal{I}_{193} - \mathcal{I}_{197}$
18	$\mathcal{A}_{18}$	$\mathcal{E}_{31}$	$\mathcal{I}_{198} - \mathcal{I}_{202}$
...	...	...	...
31	$\mathcal{A}_{31}$	$\mathcal{E}_{16}$	$\mathcal{I}_{262} - \mathcal{I}_{267}$
32	$\mathcal{A}_{32}$	$\mathcal{E}_{17}$	$\mathcal{I}_{268} - \mathcal{I}_{272}$

#	MA	MI	IR
	«ShiftRows»		
33	$\mathcal{A}_{33}$	$\mathcal{E}_{33}$	$\mathcal{I}_{273} - \mathcal{I}_{274}$
34	$\mathcal{A}_{34}$	$\mathcal{E}_{37}$	$\mathcal{I}_{275} - \mathcal{I}_{276}$
...	...	...	...
47	$\mathcal{A}_{47}$	$\mathcal{E}_{42}$	$\mathcal{I}_{301} - \mathcal{I}_{302}$
48	$\mathcal{A}_{48}$	$\mathcal{E}_{45}$	$\mathcal{I}_{303} - \mathcal{I}_{304}$
	«MixColumn+AddRoundKey(1)»		
49	$\mathcal{A}_{49}$, $\mathcal{A}_{65}$	$\mathcal{E}_{50}$	$\mathcal{I}_{323} - \mathcal{I}_{346}$
50	$\mathcal{A}_{50}$, $\mathcal{A}_{66}$	$\mathcal{E}_{51}$	$\mathcal{I}_{347} - \mathcal{I}_{370}$
...	...	...	...
63	$\mathcal{A}_{63}$, $\mathcal{A}_{79}$	$\mathcal{E}_{67}$	$\mathcal{I}_{717} - \mathcal{I}_{740}$
64	$\mathcal{A}_{64}$, $\mathcal{A}_{80}$	$\mathcal{E}_{68}$	$\mathcal{I}_{741} - \mathcal{I}_{764}$

value of l is zero, it indicates that MA has only one sub-operation present ($\mathcal{A}_i$). Thus a one-to-one, one-to-many mapping to MI is possible. Lines 20 to 24 handles these situations. Similarly, if $l > 0$ multiple sub-operations are present, then a many-to-one or many-to-many mapping is possible. Lines 13 to 19 handles these situations. After all the sub-operations are mapped to expressions in the SUT, the algorithm returns `E_Map`, which comprises of a list of mapped segments. The Algorithm also takes two additional inputs $\mathsf{MAX}_{\mathcal{A}}$ and $\mathsf{MAX}_{\mathcal{E}}$, which are respectively the maximum number of sub-operations and program expressions that the `EquivalenceChecker` considers. Thus, in any iteration, the algorithm attempts to match at most $\mathsf{MAX}_{\mathcal{A}}$ sub-operations in the BCS to at most $\mathsf{MAX}_{\mathcal{E}}$ program expressions in the implementation. Failure to find any equivalence will result in a failure of the algorithm. The values of $\mathsf{MAX}_{\mathcal{A}}$ and $\mathsf{MAX}_{\mathcal{E}}$ are found empirically and would depend on the BCS and SUT.

Table 3 shows the result of Algorithm 1 for the BCS and SUT inputs shown in Figure 3. $\mathsf{MAX}_{\mathcal{A}}$ and $\mathsf{MAX}_{\mathcal{E}}$ are both set to 32 for this example. In the example, the initial `AddRoundKey` in BCS has a one-to-one mapping with that of SUT. Hence in the first 16 iterations of the algorithm, the 16 sub-operations in BCS key addition are mapped to the 16 program expression as seen in the table. In each of these 16 iterations, a one-to-one mapping of sub-operations is found. Thus $l = 0$ and $m = 0$ in each of these iterations.

Notice that in the SUT shown in Figure 3, the operations `SubBytes` and `ShiftRows` are interchanged compared to that of BCS. Thus, in the 17-th iteration of Algorithm 1, the smallest MA and MI segments that are mapped are $\mathcal{A}_{17}$ to $(\mathcal{E}_{17}, \mathcal{E}_{18}, \cdots \mathcal{E}_{32})$. This is a one-to-many mapping and handled by Lines 20 to 24 in the algorithm. `E_Map` is appended with $\mathcal{A}_{17}$ mapping to $\mathcal{E}_{32}$ because the `EquivalenceChecker` only records the final program expression in MI, in this case $\mathcal{E}_{32}$. Subsequent iterations, consider the unmapped nodes from BCS and SUT. For example, in the 18-th iteration the smallest MA and MI segments that can be mapped are $\mathcal{A}_{18}$ and $(\mathcal{E}_{17}, \mathcal{E}_{18}, \cdots, \mathcal{E}_{31})$, thus $\mathcal{A}_{18}$ mapped to $\mathcal{E}_{31}$ is appended in `E_Map`. The algorithm determines all the sub-operations and program expression for `ShiftRows` and `MixColumns` in the same way. Iterations 19 to 48 are also one-to-many maps and the mappings are found in a similar manner as shown in Table 3.

For the SUT given in Figure 3, the operations of `MixColumns` and `AddRoundKey` are merged. Thus, $\mathcal{A}_{49}$, $\mathcal{A}_{65}$ map to $\mathcal{E}_{49}$ and $\mathcal{E}_{50}$. However, since MA is chosen in a sequential manner, in the 49^{th} iteration, the algorithm would identify MA and MI to be $(\mathcal{A}_{49}, \mathcal{A}_{50}, \cdots \mathcal{A}_{65})$ and $(\mathcal{E}_{49}, \mathcal{E}_{50})$ respectively. This is a many-to-many mapping, which is handled by Lines 13 to 19 in the algorithm. The information flow path in MA from the last sub-operation (*i.e.* $\mathcal{A}_{65}$) is considered (Line 15). This corresponds to $\mathcal{A}_{49}$, $\mathcal{A}_{65}$. Hence sub-operations ($\mathcal{A}_{49}$, $\mathcal{A}_{65}$) is mapped to program expression ($\mathcal{E}_{50}$). Iterations 50 to 64 are

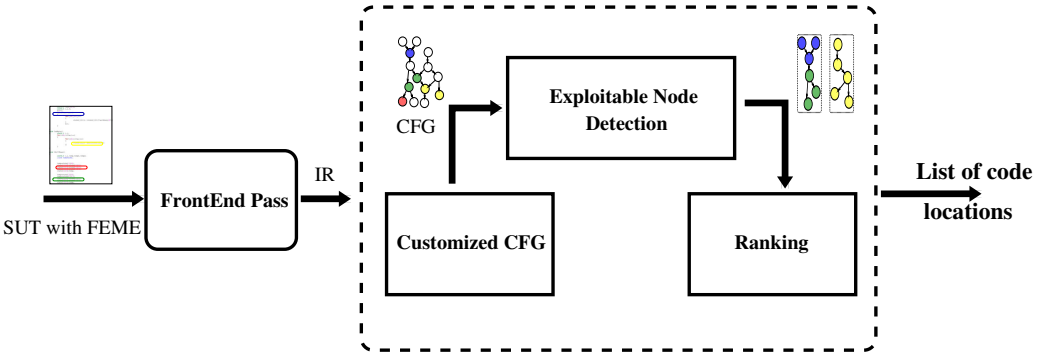


Figure 6: `Fault Evaluation` in FEDS convert the SUT to CFG, then determine all the exploitable IR instructions from the given SUT.

also many-to-many maps and the mappings are found in a similar manner, as shown in iteration 49 to 64 of Table 3. The algorithm terminates either when no equivalence is found for the given $\mathsf{MAX}_{\mathcal{A}}$ and $\mathsf{MAX}_{\mathcal{E}}$, or all BCS sub-operations have been mapped.

Complexity. Let the number of sub-operations in the BCS be L and the program expression in the SUT be M. The worst case complexity of the algorithm is $L \times M$. The worst case occurs when the order of program expressions is inverted compared to that of the BCS.

FEME **Identification.** The first input to this module (Figure 5) is `E_Map`, the output of Algorithm 1, which holds the mapping from various sub-operations in the BCS to program expressions in the SUT. The second input is the set of exploitable sub-operations in the BCS (FEMA), which are obtained from the HLE tool. Given these inputs, Proposition 1 can be used to identify the program expressions that are exploitable. The collection of all such exploitable program expressions form FEME.

5.3 Fault Evaluation

Given the FEME, this module determines all the IR instructions that are susceptible to a fault attack. The module uses the LLVM compiler and has a flow as shown in Figure 6. The input to the module is the given SUT along with FEME obtained from `Fault Mapping` (Section 5.2).

`Fault Evaluation` has the following components: (1) *Front-End Pass*, converts the given SUT into IR instructions (for example, refer Figure 2). (2) *Customized CFG*, converts the IR instructions to a control flow graph (CFG). (3) *Exploitable Node Detection*, for each exploitable program expression in the FEME, identifies all the IR instructions that can corrupt the output of the program expression. The output is a list of all exploitable IR instructions from the given SUT.

5.3.1 IR to Control Flow Graph

The IR instructions are represented as a sequence of instructions $(\mathcal{I}_1, \mathcal{I}_2, \cdots, \mathcal{I}_N)$ arranged in basic blocks. A *Basic Block* contains an instruction sequence ending with a branch or an exit instruction. For example, the IR shown in Figure 2, is divided into four basic blocks: $(\mathcal{I}_{305}, \mathcal{I}_{306}, \cdots, \mathcal{I}_{322})$, $(\mathcal{I}_{323}, \mathcal{I}_{324}, \cdots, \mathcal{I}_{326})$, $(\mathcal{I}_{327}, \mathcal{I}_{328}, \mathcal{I}_{329})$, and $(\mathcal{I}_{330}, \mathcal{I}_{331}, \cdots, \mathcal{I}_{346})$.

The generated IR is transformed into a CFG $\mathcal{G}(V,E)$. Since we assume that the SUT is unrolled, the corresponding CFG will not have any cycles. Each instruction in the IR forms a vertex in the graph. An edge from vertex $\mathcal{I}_i$ to $\mathcal{I}_j$ is present if there is a control

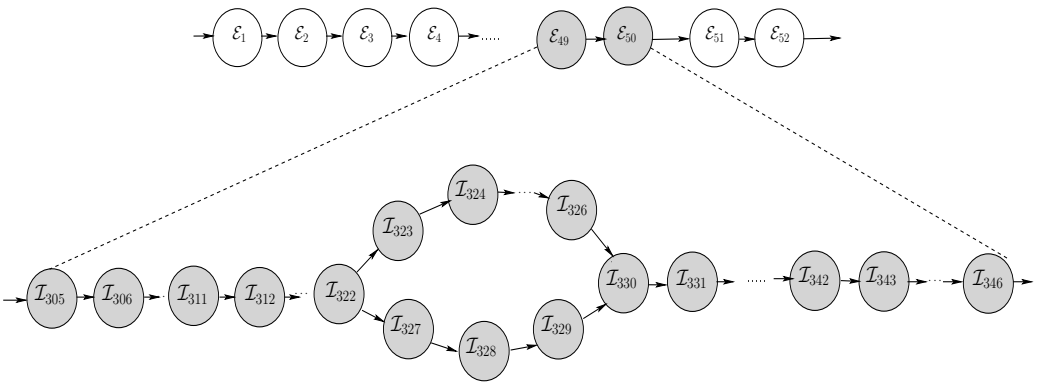


Figure 7: Control flow graph $\mathcal{G}(V,E)$ corresponding to the IR instruction for program expressions $\mathcal{E}_{49}$, $\mathcal{E}_{50}$ given in Figure 3.

flow such that, instruction $\mathcal{I}_j$ executes immediately after instruction $\mathcal{I}_i$. The vertex $\mathcal{I}_j$ is called the *successor* of $\mathcal{I}_i$ and $\mathcal{I}_i$ is called the *predecessor* of $\mathcal{I}_j$. It may be noted that a vertex $\mathcal{I}$ may have zero or more successors and zero or more predecessors. We denote the set of successors and predecessors by $\mathbb{S}(\mathcal{I})$ and $\mathbb{P}(\mathcal{I})$ respectively.

Example 3. Control Flow Graph $\mathcal{G}(V,E)$ for the IR in Figure 2 is given in Figure 7. These IR instructions correspond to program expressions $\mathcal{E}_{49}$ and $\mathcal{E}_{50}$. The CFG has 42 vertices ($\mathcal{I}_{305}$, $\mathcal{I}_{306}$, ..., $\mathcal{I}_{346}$) equal to the number of instructions in the IR. Instruction $\mathcal{I}_{322}$ is a branch instruction which can move to either `true` ($\mathcal{I}_{323}, \mathcal{I}_{324}, \cdots, \mathcal{I}_{326}$) or `false` ($\mathcal{I}_{327}, \mathcal{I}_{328}, \mathcal{I}_{329}$), hence vertex $\mathcal{I}_{322}$ has two successors $\mathcal{I}_{323}$ and $\mathcal{I}_{327}$. Similarly, vertex labelled $\mathcal{I}_{330}$ has 2 predecessors, $\mathcal{I}_{326}$ and $\mathcal{I}_{329}$. Thus, $\mathbb{S}(\mathcal{I}_{322}) = \{\mathcal{I}_{323}, \mathcal{I}_{327}\}$ and $\mathbb{P}(\mathcal{I}_{330}) = \{\mathcal{I}_{326}, \mathcal{I}_{329}\}$.

5.3.2 Exploitable Node Detection

A corrupted output of an exploitable program expression can be used by an attacker to glean bits of the secret key. For a given exploitable expression, this module identifies all instructions that can corrupt the output of the expression. Said another way, a fault injected in any of the identified instructions could result in an incorrect output of the program expression, which in turn can be exploited in a fault attack.

This module starts with the FEME, which comprises of all exploitable program expressions. Algorithm 2 provides details about identifying relevant instructions that can corrupt the expressions present in FEME. Besides FEME, the algorithm takes as input `E_Map` (the output of Algorithm 1) and the CFG (Section 5.3.1). The algorithm outputs a set of instructions in `E_Nodes` that can corrupt the output of one or more program expressions present in FEME.

For each program expression ($\mathcal{E}$) in FEME, the corresponding IR instructions are identified and added to the set `E_Nodes`. A program expression may be mapped to multiple instructions. In such a case it is sufficient to evaluate only the final instruction ($\mathcal{I}_{end}$), all other instructions would be handled by the algorithm. Thus, starting at $\mathcal{I}_{end}$, the algorithm works its way up to the start of the CFG. During this process (Lines 8 to 19), all unmapped instructions that can influence the output of expression $\mathcal{E}$ are identified and added to the set `E_Nodes`. Mapped instructions do not need to be added to `E_Nodes` because they are either unexploitable or the corresponding expression would be present in FEME, therefore handled in a different iteration. To identify, if an instruction can influence

Algorithm 2: `Exploitable_Node_Detection`

Input: `FEME` : $\mathcal{E}_{i1}, \mathcal{E}_{i2}, \mathcal{E}_{i3} \cdots \mathcal{E}_{ij}$
`CFG` : *customized* control flow graph $\mathcal{G}(V,E)$
`E_Map` : Mapping Set from $\mathcal{A}$ to $\mathcal{E}$
Output: `E_Nodes` : Set of the exploitable instructions in the SUT

1 **begin**
2 `E_Nodes` $\leftarrow \varnothing$
3 **for** *each* $\mathcal{E} \in$ FEME **do**
4 Let $\mathcal{E}$ be mapped to the IR instructions ($\mathcal{I}_{i1}$, $\mathcal{I}_{i2}, \cdots, \mathcal{I}_{in}(= \mathcal{I}_{end})$)
5 $\mathbb{R}_c(\mathcal{I}_{end}) \leftarrow \mathbb{U}(\mathcal{I}_{end})$
6 `E_Nodes` $\leftarrow$ `E_Nodes` $\cup \{\mathcal{I}_{end}\}$
7 `pred_Nodes` $\leftarrow \mathbb{P}(\mathcal{I}_{end})$
8 **for** *each* $\mathcal{I}$ *in* `pred_Nodes` **do**
9 Compute $\mathbb{D}(\mathcal{I})$ and $\mathbb{U}(\mathcal{I})$ sets for the node $\mathcal{I}$
10 $\mathbb{R}_c(\mathcal{I}) \leftarrow (\bigcup_{m \in \mathbb{S}(\mathcal{I})} \mathbb{R}_c(\mathcal{I}_m))$
11 Let $\mathcal{I}$ be an instruction in program expression $\mathcal{E}_i$
12 Find if $\mathcal{E}_i$ present in `E_Map`
13 **if** *not found* **or** $\mathcal{E} = \mathcal{E}_i$ **then**
14 // $\mathcal{I}$ is not mapped or mapped to the same $\mathcal{E}$
15 **if** $\mathbb{D}(\mathcal{I}) \cap \mathbb{R}_c(\mathcal{I}) \neq \varnothing$ **then**
16 $\mathbb{R}_c(\mathcal{I}) \leftarrow \mathbb{R}_c(\mathcal{I}) \setminus \mathbb{D}(\mathcal{I})$
17 $\mathbb{R}_c(\mathcal{I}) \leftarrow \mathbb{R}_c(\mathcal{I}) \cup \mathbb{U}(\mathcal{I})$
18 `E_Nodes` $\leftarrow$ `E_Nodes` $\cup \{\mathcal{I}\}$
19 `pred_Nodes` $\leftarrow$ Append with all elements of $\mathbb{P}(\mathcal{I})$
20 **return** `E_Nodes`

the output of an expression $\mathcal{E}$, we track the data flow using the set $\mathbb{R}_c$. For any instruction ($\mathcal{I}$), $\mathbb{R}_c(\mathcal{I})$ contains all the used variables that can influence the output of $\mathcal{E}$.

Complexity. Let the number of exploitable expression in FEME be `M` and number of nodes in the control flow graph be `N`, then the worst case complexity for running the algorithm is `M` $\times$ `N`.

Example 4. Consider expression $\mathcal{E}_{50}$ from Figure 2 which maps to the instructions $\mathcal{I}_{334}$, $\mathcal{I}_{335}$, $\cdots$, $\mathcal{I}_{339}$. Assume that the instructions $\mathcal{E}_{50}$ is in FEME, $\mathcal{I}_{332}$ and $\mathcal{I}_{333}$ are unmapped and $\mathcal{I}_{331}$ belongs to another expression which is in `E_Map`. Algorithm 2 starts execution from $\mathcal{I}_{339}$ which is $\mathcal{I}_{end}$. Table 4 traces the algorithm execution through every iteration. $\mathbb{R}_c(\mathcal{I}_{339})$ is the set of used variables in $\mathcal{I}_{end}$. i.e $\{\%\text{xor5}, \%11\}$. A fault in any of these variables can disturb the output of $\mathcal{I}_{339}$. In the first iteration, we consider the predecessor nodes of $\mathcal{I}_{339}$, *i.e.* $\mathcal{I}_{338}$. We see that $\mathcal{I}_{338}$ has %11 defined, which is also present in $\mathbb{R}_c(\mathcal{I}_{339})$. Thus $\mathbb{R}_c(\mathcal{I}_{338})$ is a copy of $\mathbb{R}_c(\mathcal{I}_{339})$ with %11 replaced with the used variables of $\mathbb{R}_c(\mathcal{I}_{338})$, in this case $\%t$. This is done in Lines 16 to 18 in the algorithm. Note that a fault, in $\%t$ affects %11, which in turn affects the result of $\mathcal{I}_{339}$. In this way, every iteration updates the corresponding $\mathbb{R}_c$ with used variables that can influence the result of $\mathcal{I}_{339}$. In $\mathcal{I}_{331}$, however, update of $\mathbb{R}_c$ is not done because $\mathcal{I}_{331}$ belongs to an expression which is in `E_Map`. Thus, this instruction is either not exploitable or would be handled in another iteration of the algorithm. Thus, `E_Nodes` contains $\mathcal{I}_{334}$ to $\mathcal{I}_{339}$.

6 Using FEDS to Evaluate Implementations of Block Ciphers

To demonstrate the working of FEDS, we choose seven different implementations of AES-128, which are summarized in Table 2, CLEFIA and CAMELLIA. Some of these implementations are from standard libraries, for example OpenSSL's AES implementation

Table 4: Result of `Exploitable_Node_Detection` (Algorithm 2) for instructions $\mathcal{I}_{331}$, $\mathcal{I}_{332}$, $\cdots$, $\mathcal{I}_{339}$ given in Figure 2.

#	Instruction ($\mathcal{I}$)	$\mathbb{D}(\mathcal{I})$ and $\mathbb{U}(\mathcal{I})$	$\mathbb{R}_c(\mathcal{I})$	E_Nodes
$\mathcal{I}_{339}$	%xor6 ← %xor5 ⊕ %11	$\mathbb{D}(\mathcal{I}_{339}) \leftarrow \{\%xor6\}$ $\mathbb{U}(\mathcal{I}_{339}) \leftarrow \{\%xor5, \%11\}$	$\mathbb{R}_c(\mathcal{I}_{339}) \leftarrow \{\%xor5, \%11\}$	✓
$\mathcal{I}_{338}$	%11 ← Load %t	$\mathbb{D}(\mathcal{I}_{338}) \leftarrow \{\%11\}$ $\mathbb{U}(\mathcal{I}_{338}) \leftarrow \{\%t\}$	$\mathbb{R}_c(\mathcal{I}_{338}) \leftarrow \{\%xor5, \%t\}$	✓
$\mathcal{I}_{337}$	%xor5 ← %9 ⊕ %10	$\mathbb{D}(\mathcal{I}_{337}) \leftarrow \{\%xor5\}$ $\mathbb{U}(\mathcal{I}_{337}) \leftarrow \{\%9, \%10\}$	$\mathbb{R}_c(\mathcal{I}_{337}) \leftarrow \{\%9, \%10, \%t\}$	✓
$\mathcal{I}_{336}$	%10 ← Load %a3	$\mathbb{D}(\mathcal{I}_{336}) \leftarrow \{\%10\}$ $\mathbb{U}(\mathcal{I}_{336}) \leftarrow \{\%a3\}$	$\mathbb{R}_c(\mathcal{I}_{336}) \leftarrow \{\%9, \%a3, \%t\}$	✓
$\mathcal{I}_{335}$	%a3 ← Get Element state[0]	$\mathbb{D}(\mathcal{I}_{335}) \leftarrow \{\%a3\}$ $\mathbb{U}(\mathcal{I}_{335}) \leftarrow \{state[0]\}$	$\mathbb{R}_c(\mathcal{I}_{335}) \leftarrow \{\%9, state[0], \%t\}$	✓
$\mathcal{I}_{334}$	%9 ← Load %x	$\mathbb{D}(\mathcal{I}_{334}) \leftarrow \{\%9\}$ $\mathbb{U}(\mathcal{I}_{334}) \leftarrow \{\%x\}$	$\mathbb{R}_c(\mathcal{I}_{334}) \leftarrow \{\%x, state[0], \%t\}$	✓
$\mathcal{I}_{333}$	%y ← Store %8	$\mathbb{D}(\mathcal{I}_{333}) \leftarrow \{\%y\}$ $\mathbb{U}(\mathcal{I}_{333}) \leftarrow \{\%8\}$	$\mathbb{R}_c(\mathcal{I}_{333}) \leftarrow \{\%x, state[0], \%t\}$	✗
$\mathcal{I}_{332}$	%8 ← Load %x	$\mathbb{D}(\mathcal{I}_{332}) \leftarrow \{\%8\}$ $\mathbb{U}(\mathcal{I}_{332}) \leftarrow \{\%x\}$	$\mathbb{R}_c(\mathcal{I}_{332}) \leftarrow \{\%x, state[0], \%t\}$	✗
$\mathcal{I}_{331}$	%x ← Store %cond	$\mathbb{D}(\mathcal{I}_{331}) \leftarrow \{\%x\}$ $\mathbb{U}(\mathcal{I}_{331}) \leftarrow \{\%cond\}$	$\mathbb{R}_c(\mathcal{I}_{331}) \leftarrow \{\%x, state[0], \%t\}$	✗

with 4 T-tables (IMP3) and compressed tables (IMP4). Other implementations such as IMP2, IMP5, and IMP6 have been crafted to demonstrate the versatility of FEDS in handling operations that are swapped and merged. IMP7 is a bitsliced implementation of AES that performs 64 encryptions in a bit-wise manner. In this section we provide details about the implementation aspects of FEDS and the results for these implementations.

6.1 High Level Exploitability Tool

FEDS uses HLE tools to determine the fault exploitable sub-operations of the block cipher. FEDS can use any HLE tool like [KRH17, SMD18, SJP+19, ZGZ+16], provided the input-output representations are compatible. We demonstrate FEDS using XFC [KRH17] as the HLE tool. The tools take as input the Block Cipher Specification (refer LHS of Figure 3) and determines the sub-operations in the cipher that are vulnerable to Differential Fault Attacks. The output comprises of a list of round keys that can be derived from a single random fault injected in the specified sub-operation. XFC also provides the offline complexity in deriving the key bits after the fault is injected. We also use ExpFault [SMD18] as the HLE tool to determine the sub-operations in AES-128 that are vulnerable to Impossible Differential Fault Attacks. Table 5 shows the output of XFC and ExpFault for the different block cipher implementations. Each row represents a sequence of sub-operations and the vulnerability to a single random fault temporally injected in any of these operations. For example, if a fault is injected in any sub-operation between $\mathcal{A}_{433}$ and $\mathcal{A}_{496}$, then 128-bit key bits can be derived with an offline complexity of 2^8. Note that these HLE results are dependent on the cipher structure and properties. They are, however, independent of the cipher implementations. The output of this phase is the set FEMA comprising of sub-operations that are exploitable. For AES-128, the FEMA comprises of sub-operations $\mathcal{A}_{433}$ to $\mathcal{A}_{560}$ based on the HLE tool XFC (Refer Table 5). Table 6 shows a summary of the output of the HLE, the size of FEMA, and the time taken.

6.2 Fault Mapping

The `Fault Mapping` module described in Section 5.2 uses `Equivalence Mapping` to map the fault exploitable sub-operations from FEMA to corresponding program expressions in the SUT. The `Equivalence Mapping` module includes three main operations: (1) generate information flow graph (IFG) from the BCS, (2) extract the clauses for equivalence checking from the sub-operations and expressions of BCS and SUT respectively, (3) use of a model checker to check for equivalence between extracted clauses.

Table 5: Output of HLE Tools. XFC was used to evaluate ciphers for Differential Fault Attacks, ExpFault was used to evaluate AES for Impossible Differential Fault Attacks.

HLE Tool	Cipher	Sub-operations	#Derived Keys	Offline Complexity
XFC [KRH17]	AES	$\mathcal{A}_1$ - $\mathcal{A}_{432}$	0	N/A
		$\mathcal{A}_{433}$ - $\mathcal{A}_{496}$	128	2^8
		$\mathcal{A}_{497}$ - $\mathcal{A}_{560}$	32	2^8
		$\mathcal{A}_{561}$ - $\mathcal{A}_{640}$	0	N/A
	CLEFIA	$\mathcal{A}_{625}$ - $\mathcal{A}_{673}$	32	2^8
		$\mathcal{A}_{673}$ - $\mathcal{A}_{743}$	32	$2^{4.76}$
	CAMELLIA	$\mathcal{A}_{956}$ - $\mathcal{A}_{1004}$	13 or 14	2^8
		$\mathcal{A}_{1005}$ - $\mathcal{A}_{1052}$	5 or 6	2^8
ExpFault [SMD18]	AES	$\mathcal{A}_{368}$ - $\mathcal{A}_{432}$	32	2^{32}

Table 6: Statistics of outputs from the three different steps of FEDS for 7 AES implementations given in Table 2, CAMELLIA and CLEFIA. The entire framework was tested using an Ubuntu 16.04 Linux machine on a quad-core Intel i5-3340 CPU @ 3.10GHz.

#	HLE Tool (XFC)			Fault Mapping			Fault Evaluation		
	BCS	Size of FEMA	Time (ms.)	No. of Expressions	Size of E_Map	Time (mins.)	Nodes in CFG	% Exploitable Instructions	Time (sec.)
IMP1	640 sub-ops	128	< 20	780	640	59.33	7206	6.56	38.2
IMP2				780	640	72.16	7206	6.20	38.7
IMP3				196	84	98	4299	3.71	15.5
IMP4				123	44	97.5	4132	4.98	9.2
IMP5				580	480	40.5	7076	5.97	37.7
IMP6				790	496	79	7318	5.36	36.1
IMP7				2277	336	2560	8256	11.45	215.7
CAMELLIA	2144	128	< 36	400	2144	94.46	1475	23.2	99.3
CLEFIA	1184	256	< 24	1425	1184	205.56	1026	6.54	105.5

While any model checking tool would suffice, we chose to perform `Equivalence Mapping` using a model checker called CBMC [CKL04]. We chose this tool due to its ease-of-use, stability and our prior experience working with it. The model checker generates constraints $\mathcal{C}$ and properties $\mathcal{P}$ from the extracted clauses and then invokes the SAT solver to solve satisfiable equations of the form $\mathcal{C} \wedge \neg\mathcal{P}$. If a solution is found, *false* is returned. This means that the BCS and SUT clauses are not equivalent. If a solution is not found, *true* is returned. This means that the BCS and SUT clauses are equivalent.

The `EquivalenceChecker` invoked in Line 11 of Algorithm 1, takes two inputs MA and MI corresponding to a sequence of sub-operations in the BCS and SUT respectively. To understand how CBMC finds equivalence, we take an example with

$$\text{MA} = \mathcal{A}_1 : \{\{\texttt{P[1]}\} : \texttt{XOR}\,(\texttt{P[1]}, \texttt{LKUP(1, KEY0)})\}\ ,$$
$$\text{MI} = \mathcal{E}_1 : \{\texttt{state[0]} = \texttt{in[0]} \wedge \texttt{rks[0][0]}\}\ .$$

Figure 8 shows the four steps involved in verifying equivalence between MA and MI.

Step 1. The sub-operations in MA are converted to valid C-code for equivalence checking. This is needed because CBMC is a C-based model checker and requires C-code as input. No conversion is required for the SUT expressions, which are written in C. Valid assertions are added to check for equivalence.

Step 2. Valid assert conditions are added to the C code to check for equivalence.

Step 3. The generated conditions are transformed to SSA form (Refer Definition 1).

Step 4. The SSA statements are converted to a set of constraints ($\mathcal{C}$) and set of properties ($\mathcal{P}$). CBMC then invokes a SAT solver to solve $\mathcal{C} \wedge \neg\mathcal{P}$. If a solution is not found, then sub-operations in MA are functionally equivalent to the program expressions in MI.

The output of Algorithm 1 is the map `E_Map`, which includes the mapping set of $\mathcal{A}$ to $\mathcal{E}$ (refer: Table 3). Table 6 shows the number of program expression in each implementation,

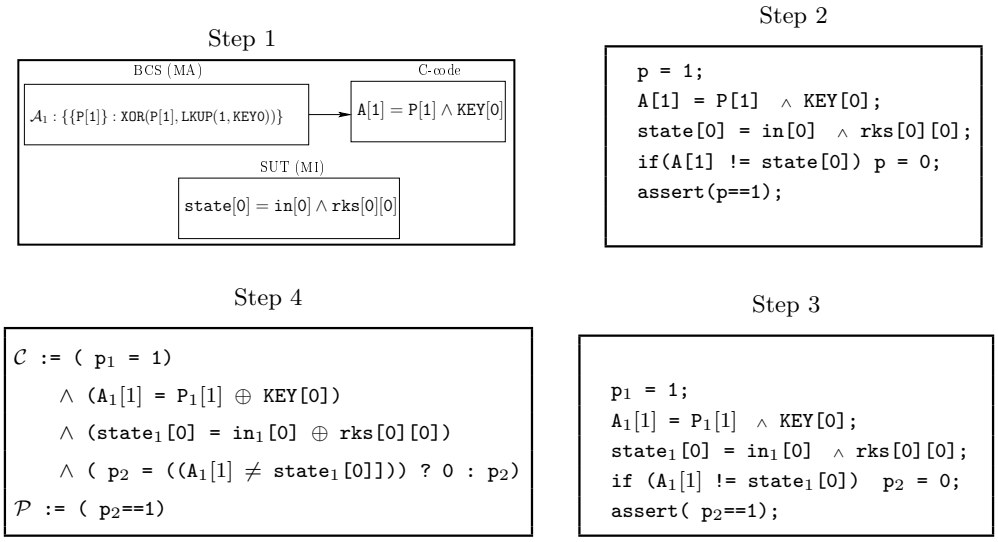


Figure 8: Steps involved for verifying equivalence between MA and MI.

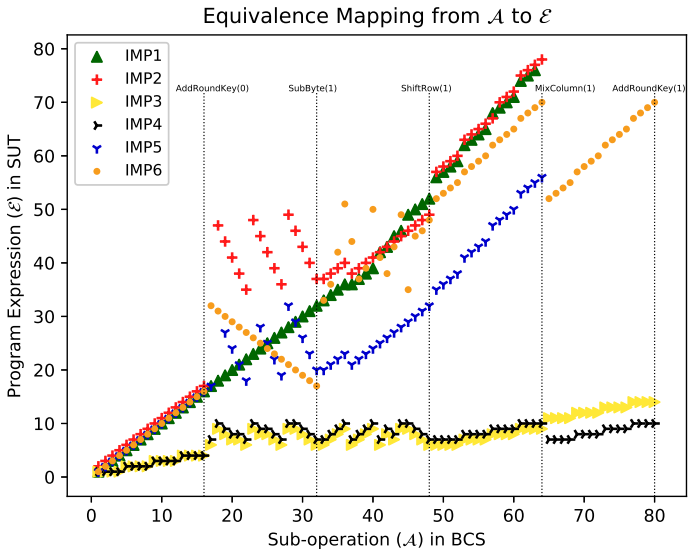


Figure 9: *Equivalence Mapping* output for a single round of the AES implementations given in Table 2. The x-axis denotes the sub-operations in BCS and y-axis is the corresponding program expression in SUT.

the size of `E_Map`, and the average time taken to map all the sub-operations in the BCS to corresponding expressions in the SUT for each block cipher implementation considered. Finding equivalence for all sub-operations in the BCS to expressions in the SUT also proves the correctness of the implementation with respect to the given specification.

Figure 9 depicts the mapping from BCS to various SUTs. The x-axis denotes the sub-operations in the BCS and the y-axis denotes the mapped program expressions for the first round of AES. Implementations IMP3 and IMP4 use T-tables in which the AES operations in a round are merged by the T-table accesses. These merged operations result

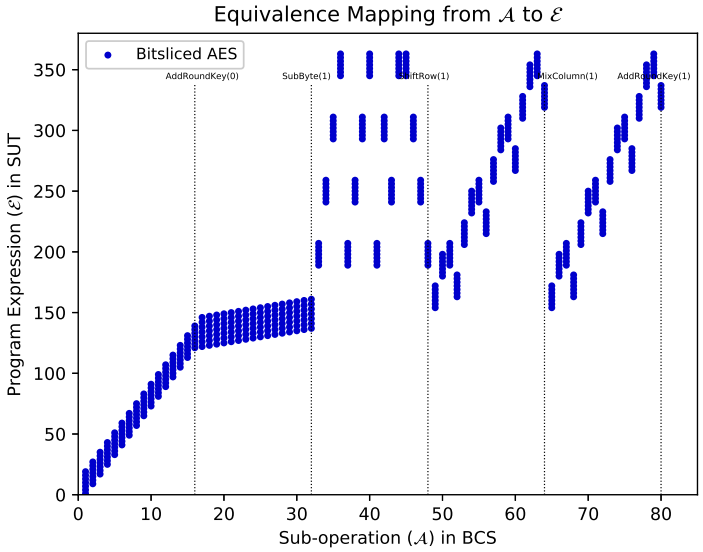


Figure 10: *Equivalence Mapping* output for a single round of the bitsliced AES implementations (IMP7) given in Table 2. The x-axis denotes the sub-operations in BCS and y-axis is the corresponding program expression in SUT.

in traces that are parallel to the x-axis. For example $\mathcal{A}_{10}$, $\mathcal{A}_{11}$, $\mathcal{A}_{12}$, and $\mathcal{A}_{13}$ all map to the program expression $\mathcal{E}_3$ therefore these points are parallel to the x-axis. In IMP2 and IMP6, we notice that the mapping of sub-operations in `SubBytes` and `ShiftRows` do not follow a trend similar to the other implementations. This is because, in these implementations, `SubBytes` and `ShiftRows` are implemented in an order different from the BCS.

Mapping BCS to Bitsliced SUT. For bitsliced implementations such as IMP7 [RSD06], where all the operations are bit-wise, FEDS needs additional user inputs to determine how to interpret a byte in the SUT. The `Extract` function needs to be specified as follows:

$$\text{MI} = \{\texttt{Extract}(b_j \mid ... \mid b_k)\} \ ,$$

where $b_j \cdots b_k$ $(j < k)$ are the bits in the bit-sliced representation, which `Extract` converts to a byte. Figure 10 depicts the mapping from the BCS to bitsliced implementation of AES (IMP7), given in Table 2. Due to the bitsliced representation [RSD06], one sub-operation in BCS is mapped to 8 expressions in the SUT of the implementation.

Mapping BCS to Other Block Ciphers. To demonstrate the scalability of FEDS, we have evaluated two of the block ciphers CLEFIA-128 [SSA+07] and CAMELLIA-128 [AIK+00]. Table 6 includes the size of `E_Map`, and average time to map sub-operations to program expression for these implementations.

After the equivalence mapping is obtained, the FEME can be constructed. Essentially, every program expression that is mapped from an exploitable sub-operation in the FEMA, is added to the FEME.

6.3 Fault Evaluation

Each component of the `Fault Evaluation` module is implemented as a transformation pass in the LLVM Clang compiler[5] Version 7.0. `Fault Evaluation` works by first converting

[5] https://llvm.org/

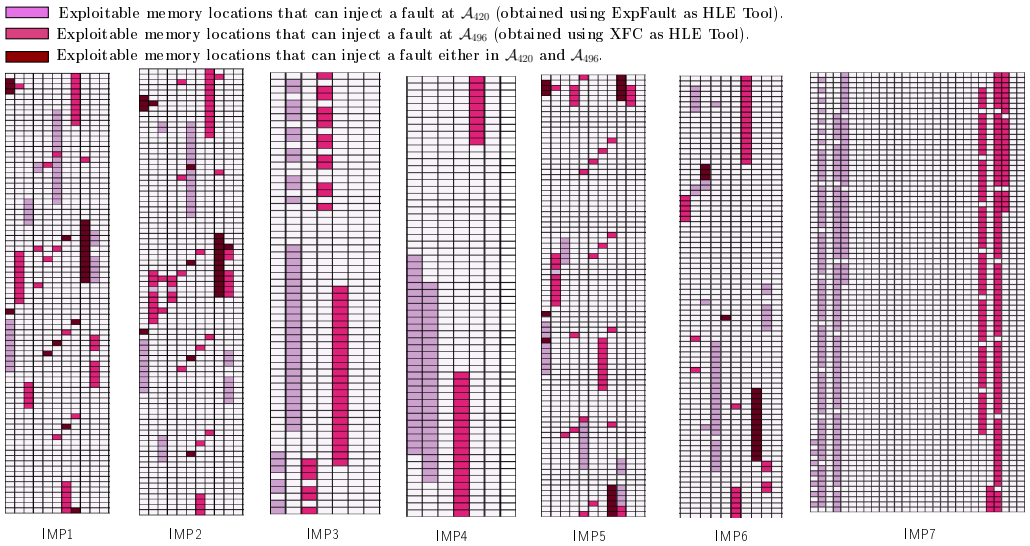


Figure 11: `Fault Evaluation` output for different implementations of AES given in Table 2 for fault injected at locations $\mathcal{A}_{496}$ and $\mathcal{A}_{420}$. The fault evaluation for $\mathcal{A}_{496}$ was done using he HLE tool XFC [KRH17], while the fault evaluation for $\mathcal{A}_{420}$ was done using the HLE tool ExpFault [SMD18]. Each cell denotes an instruction in the IR. Starting from the top left corner, instructions are represented column wise from top to bottom. The last instruction is in the bottom right corner. The colored cells denote the exploitable instructions based on HLE tool XFC and ExpFault.

from IR to a control flow graph to find the dependencies between instructions (refer Section 5.3.1). Each node in the CFG is an instruction in the IR. After the CFG is created, Algorithm 2 uses the program expressions in the FEME to determine all IR instructions where a fault, if injected, can change the result of the program expression. The total number of nodes in the CFG for the different block cipher implementations is shown in Table 6 along with the percentage of exploitable nodes and the execution time of the tool. A fault in any of these exploitable nodes can be used to mount a fault attack. In the next part of the section, we take an example of two exploitable AES sub-operations and study their exploitability over the seven AES implementations.

We consider faults injected in sub-operations $\mathcal{A}_{420}$ and $\mathcal{A}_{496}$. While $\mathcal{A}_{420}$ corresponds to the 5-th byte `ShiftRows` operation in the 7-th round, $\mathcal{A}_{496}$ corresponds to the 16-th byte `ShiftRows` operation in the 8-th round. A fault in $\mathcal{A}_{496}$ is exploitable by Differential Fault Attacks and can uncover the entire AES key with an offline complexity of 2^8 [TMA11]. A fault in $\mathcal{A}_{420}$ is exploitable by Impossible Differential Fault Attacks [DFL11] and can uncover 32-bits of the AES key with a complexity of 2^{32}. While XFC [KRH17] can identify the vulnerability at $\mathcal{A}_{496}$, it cannot identify that $\mathcal{A}_{420}$ as vulnerable. ExpFault [SMD18] on the other hand can identify the vulnerability due to IDFA in $\mathcal{A}_{420}$.

For each of the seven AES implementations (Table 2), Figure 11 shows the vulnerable memory locations for these two faults. Each cell in the memory layout represents an instruction in the IR. A colored cell indicates that the instruction is exploitable by a fault in either sub-operation $\mathcal{A}_{496}$ or $\mathcal{A}_{420}$. All cells with the same color are equally exploitable and result in a fault attack that uncovers the same key bytes with the same complexity. The figures show some memory locations that are exploitable by both faults. These especially occur due to functions in the implementations that are invoked multiple times during the execution of the program. For example, function `AES_xtime` given in Figure 3, is invoked in program expressions $\mathcal{E}_{50}$, $\mathcal{E}_{51}$, $\mathcal{E}_{52}$, $\mathcal{E}_{53}$ and is also invoked in other rounds

of the implementation. Hence, the IR instructions in function `AES_xtime` can influence multiple operations of AES. Figure 11 shows that common vulnerable instructions are present in implementations IMP1, IMP2, IMP4, and IMP5, where the round operations are implemented as functions. For the T-Table implementations such as IMP3 and IMP4, and the bitsliced implementation, IMP7, no function invocations are present within a round, hence these implementations do not have common vulnerable instructions.

From the figure, it is evident that the number of exploitable instructions varies based on implementation. IMP3 and IMP4 have the least number of exploitable instructions compared to other implementations. This means that an attacker has lesser options to inject faults in $\mathcal{A}_{496}$ and $\mathcal{A}_{420}$ compared to the other implementations. Table 6 includes the results of Fault Evaluation for the various implementations of AES given in Table 2. It also has the results for other block ciphers, namely, CLEFIA and CAMELLIA. The percentage of exploitable instructions given in Table 6 is the union of all the exploitable instructions in each implementation considering all exploitable sub-operations given in Table 5. The percentage of exploitable instructions varies from 3.71% (IMP3) to 23.2% (CAMELLIA) of the total instructions are exploitable. For implementation like IMP3, with 4299 number of nodes, 3.71% ($\approx$ 160) instructions are exploitable. CAMELLIA, with 1475 nodes in CFG, has 23.2% $\approx$ 343 instructions are exploitable. Hence the exploitable instruction also depends on how the block cipher is implemented. We have tested the results by simulating faults on the exploitable instructions and noted that the output of the targeted sub-operation in the cipher was corrupted. Deriving the secret key from these faulty operations is well documented in literature [DFL11, TMA11, CWF07, AM13].

7 Using FEDS to Design a Fault Attack Aware Compiler

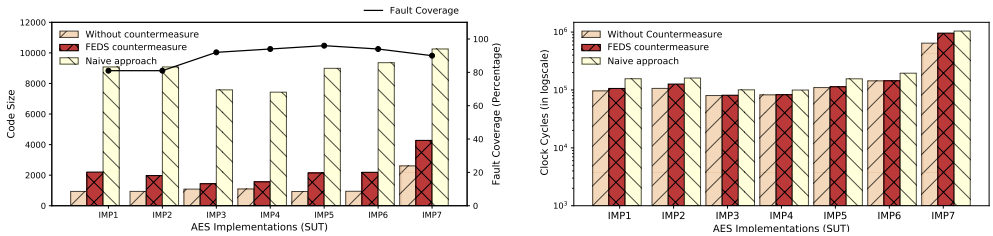
As seen in Table 6, a very small percentage of the instructions in the program are exploitable. To secure against fault attacks requires only these instructions to be protected. Identifying and protecting these instructions manually requires considerable expertise. Thus, naïve protection schemes would simply insert countermeasures for the entire program, leading to considerable overheads.

A `Fault Attack Aware Compiler` automatically protects only the exploitable instructions in the code. The FEDS framework can be first invoked to identify exploitable instructions. To protect these instructions, several countermeasure strategies are available [KWMK02, MSY06]. Each countermeasure differs in the overheads and fault coverage. To demonstrate the use of FEDS in a `Fault Attack Aware Compiler`, we use a simple redundancy countermeasure that duplicates the exploitable instructions and program variables and performs the required checks to determine if the fault occurred or not.

Example 5. Assume that the following IR instruction is found to be exploitable. The instruction store takes the value from location pointed by %67 and stores the result in a variable state[3][2].

```
%74 = getelementptr inbounds [4x[4xi8]], [4x[4xi8]]* @state, i64 0, i64 3, i64 2
store i8 %67, i8* %74
```

Redundancy countermeasure duplicates this exploitable instruction. This requires (1) duplication of program variables accessed by the instruction. For example, the two-dimensional array state[4][4] given in the instruction is duplicated in state1[4][4]; (2) the duplicated instructions are checked for equality with the actual instruction to determine the occurrence of a fault. These redundant instructions are automatically inserted by the compiler. In the instruction snippet shown below, the instructions in green are the replicated operations, while the instructions in blue perform the comparison between the original and replicated



(a) Code Size and Fault Coverage **(b)** Execution Time

Figure 12: Comparison of unprotected implementations, naïvely protected implementations with FEDS based automatic protection. The graph also shows the fault coverage, which is same for both approaches. Results are taken from a 64-bit Ubuntu 16.04 Linux machine on a quad-core Intel i5-3340 CPU @ 3.10GHz.

operations. The executable generated would thus be protected against the specific class of Differential or Impossible Differential Fault Attacks in the specific fault model considered by the HLE Tools.

```
%74 = getelementptr inbounds [4x[4xi8]], [4x[4xi8]]* @state, i64 0,i64 3,i64 2
store i8 %67, i8* %74
%75 = getelementptr inbounds [4x[4xi8]], [4x[4xi8]]* @state1,i64 0,i64 3,i64 2
store i8 %67, i8* %75
%76 = getelementptr inbounds [4x[4xi8]], [4x[4xi8]]* @state,i64 0,i64 3,i64 2
%77 = load i8, i8* %76
%78 = zext i8 %77 to i32
%79 = getelementptr inbounds [4x[4xi8]], [4x[4xi8]]* @state1,i64 0,i64 3,i64 2
%80 = load i8, i8* %79
%81 = zext i8 %80 to i32
%82 = icmp ne i32 %78, %81
br i1 %82, label %83, label %87

; <label>:83:
call void @exit(i32 0)
br label %87
; <label>:87:
....
```

Figure 12 compares a FEDS protected AES implementation with a naïvely protected equivalent and unprotected implementation. While the FEDS protected implementations have redundancy countermeasures inserted only for exploitable instructions, the naïvely protected implementations have countermeasures inserted for all instructions. As seen from the graphs, performance as well as code-size is considerably improved for the FEDS implementations. These advantages are achieved without any compromise in the fault coverage. For evaluating the fault coverage, we have induced faults in all possible vulnerable locations obtained from FEDS and determined the percentage of faults that can be captured after adding the required countermeasures.

8 Conclusion

FEDS is the first automated framework that can determine the fault exploitability in software implementations considering complex cipher properties and fault models. FEDS, therefore, has the advantages of both, the HLE as well as ILE tools for fault attack assessment. The evaluation with multiple AES implementations and block cipher CLEFIA and CAMELLIA for Differential Fault Attacks demonstrates the ability of FEDS to

uniquely identify exploitable instructions. Our results show that the exploitability of each implementation differs. The number of exploitable instructions varies between 3.71% to 23.2% depending on the implementation. The OpenSSL T-table based implementation being the most secure, while the CAMELLIA implementation is the most vulnerable. The runtime for the entire framework is less than 2565 minutes on a standard Intel i5 desktop with 16GB RAM.

The FEDS based fault attack aware compiler presented in this paper, can generate executables that have fault attack countermeasures automatically inserted. The targeted countermeasure insertion results in executables that are leaner and have less performance overheads compared to executables where countermeasures were naïvely incorporated.

An inherent limitation of FEDS is that it is restricted by the capabilities of the HLE tools used. Another limitation of FEDS is that it cannot evaluate implementations that have fault attack protection already incorporated. The fault attack aware compiler currently only supports the spatial redundancy countermeasure. While we do not foresee any implementation that FEDS cannot handle, the scalability of the model checking tool may prevent certain implementations to be evaluated. In our future work, we plan to address these limitations.

Acknowledgements

The authors would like to thank the Ministry of Electronics and Information Technology (MeitY), India, DIST-FIST Grant Program 2016, from Department of Science and Technology, India and Cisco Grant Program 2018 for partially funding this project. We would also like to thank Dr. Rupesh Nasre for his valuable inputs and suggestions and the anonymous reviewers for their valuable comments and constructive suggestions which have led to significant improvements in the contents of this work.

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.

[ABPS14] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. Differential fault analysis for block ciphers: an automated conservative analysis. In *Proceedings of the 7th International Conference on Security of Information and Networks, Glasgow, Scotland, UK, September 9-11, 2014*, page 137, 2014.

[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.

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

[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.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, pages 513–525, 1997.

[CGP00] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *10th International Conference, TACAS 2004, Held as Part of the Joint ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 168–176, 2004.

[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.

[CWF07] Hua Chen, Wenling Wu, and Dengguo Feng. Differential fault analysis on clefia. In *Proceedings of the 9th International Conference on Information and Communications Security*, ICICS'07, pages 284–295, Berlin, Heidelberg, 2007. Springer-Verlag.

[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.

[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.

[Joh03] Johannes Blömer and Jean-Pierre Seifert. Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In *7th International Conference on Financial Cryptography*, pages 162–181, 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.

[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.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[MMZ+01] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Effcient SAT Solver. In *Proceedings of 38th Annual Design Automation Conference*, pages 530–535, 2001.

[MSY06] Tal Malkin, François-Xavier Standaert, and Moti Yung. A comparative cost/security analysis of fault attack countermeasures. In *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings*, pages 159–172, 2006.

[Muk09] Debdeep Mukhopadhyay. An improved fault based attack of the advanced encryption standard. In *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, pages 421–434, 2009.

[Pub01] FIPS Pub. 197: Advanced encryption standard (aes). federal information processing standards publication, us department of commerce. *Information Technology Laboratory (ITL), National Institute of Standards and Technology (NIST), Gaithersburg, MD, USA*, 2001.

[RRHB19] Indrani Roy, Chester Rebeiro, Aritra Hazra, and Swarup Bhunia. Safari: Automatic synthesis of fault-attack resistant block cipher implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[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.

[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.

[SJP+19] Sayandeep Saha, Dirmanto Jap, Sikhar Patranabis, Debdeep Mukhopadhyay, Shivam Bhasin, and Pallab Dasgupta. Automatic characterization of exploitable faults: A machine learning approach. *IEEE Trans. Information Forensics and Security*, 14(4):954–968, 2019.

[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.

[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.

[TFY07] Junko Takahashi, Toshinori Fukunaga, and Kimihiro Yamakoshi. DFA mechanism on the AES key schedule. In *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007*, pages 62–74, 2007.

[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.

[ZGZ+16] Fan Zhang, Shize Guo, Xinjie Zhao, Tao Wang, Jian Yang, François-Xavier Standaert, and Dawu Gu. A framework for the analysis and evaluation of algebraic fault attacks on lightweight block ciphers. *IEEE Trans. Information Forensics and Security*, 11(5):1039–1054, 2016.