FortiFix: A Fault Attack Aware Compiler Framework for Crypto Implementations

KEERTHI K and CHESTER REBEIRO, Indian Institute of Technology Madras, India

Fault attacks are one of the most powerful forms of cryptanalytic attack on embedded systems, that can corrupt cipher's operations leading to a breach of confidentiality and integrity. A single precisely injected fault during the execution of a cipher can be exploited to retrieve the secret key in a few milliseconds. Naïve countermeasures introduced into implementation can lead to huge overheads, making them unusable in resource-constraint environments. On the other hand, optimized countermeasures requires significant knowledge, not just about the attack, but also on the (a) the cryptographic properties of the cipher, (b) the program structure, and (c) the underlying hardware architecture. This makes the protection against fault attacks tedious and error-prone.

In this paper, we introduce the first automated compiler framework named FortiFix that can detect and patch fault exploitable regions in a block cipher implementation. The framework has two phases. The *pre-compilation phase* identifies regions in the source code of a block cipher that are vulnerable to fault attacks. The second phase is incorporated as transformation passes in the LLVM compiler to find exploitable instructions, quantify the impact of a fault on these instructions, and finally insert appropriate countermeasures based on user defined security requirements. As a proof of concept, we have evaluated two block cipher implementations AES-128 and CLEFIA-128 on three different hardware platforms such as MSP430 (16-bit), ARM (32-bit) and RISCV (32-bit).

CCS Concepts: • **Security and privacy** → **Security in hardware**; **Hardware attacks**; **Side-channel analysis and countermeasures**.

Additional Key Words and Phrases: Crypto-system, Fault Injection Attacks, Vulnerability Analysis, Countermeasures, Security aware compilers

ACM Reference Format:
Keerthi K and Chester Rebeiro. xxxx. FortiFix: A Fault Attack Aware Compiler Framework for Crypto Implementations. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1 (July xxxx), 18 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Embedded systems have multiple components working in tandem to control and monitor various aspects of our day-to-day life, such as healthcare, automotive, and smart homes. The integrity and confidentiality, within these devices is ensured by the use of crypto-algorithms. Crypto-algorithms, however, are highly vulnerable to a potent class of physical attacks called fault attacks, where the attacker disturbs the encryption process by injecting a fault in the device, to glean information about the secret key [4]. Over the last two decades, fault attacks have been successfully demonstrated in a variety of ciphers, including the AES [21], PRESENT [3], Simon [22], Speck [12], and CLEFIA [2].

In a fault attack, the attacker injects faults while the target device is operational using glitches in the voltage or clock source or by using optical or electromagnetic radiation. The faults typically

Authors' address: Keerthi K, keerthi@cse.iitm.ac.in; Chester Rebeiro, chester@cse.iitm.ac.in, Indian Institute of Technology Madras, P.O. Chennai, Chennai, Tamil Nadu, India, 600036.

1084-4309/xxxx/7-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

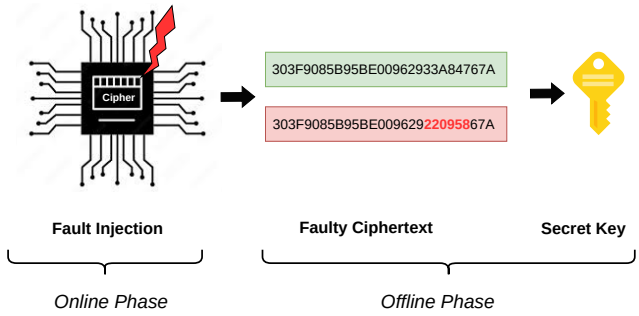


Fig. 1. Faults injected when a device is operational can modify program state and execution. The *offline phase* uses the faulty ciphertext to glean information about the secret key.

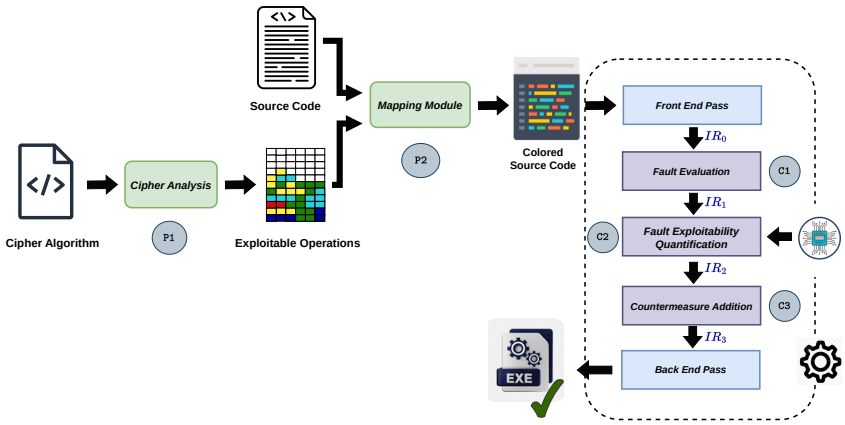


Fig. 2. High Level Overview of the FortiFix Compiler Framework

target memory components such as registers, flash memory [8], SRAM [24], and DRAM [17], corrupting the contents. Alternatively, faults are injected in the processor pipeline, for instance, causing instructions to be skipped [15]. The injected faults force the device to malfunction and result in a corrupted output, called the *faulty ciphertext* as shown in Figure 1. The attacker uses the faulty and fault-free ciphertext to glean information about the secret key. Powerful fault attacks such as the Differential Fault Analysis (DFA) [21] and Impossible Differential Fault Analysis (IDFA) [9] utilize the cipher's cryptographic properties to relax the precision required in the fault injection and reduce the number of faults required.

Our Work. In this paper, we present a fault attack aware compiler framework called FortiFix, that can automatically identity exploitable regions in a block cipher implementation, quantify the vulnerability based on the underlying hardware, and insert appropriate countermeasures based on the user's requirements. Starting with the block cipher algorithm, which is represented in a language called, Block Cipher Specification Language (BCSL), FortiFix evaluates complex properties of the cipher, thus supporting sophisticated attack techniques like DFA [21] and IDFA [9]. FortiFix also provides a vulnerability score for each memory location in a cipher implementation. The score ranges from 0 to 1. A score close to one indicates that the memory location is considerably more vulnerable to fault attacks compared to a location with a score that is close to zero. The vulnerability

score helps to automatically tune countermeasures based on the user's requirement. Thus, a cipher used in a security-critical application can be configured with higher levels of protection introduced compared to a less security-critical application.

Figure 2 depicts the High-Level Overview of the compiler framework. The framework works by first evaluating the block cipher algorithm to identify exploitable cipher operations. A fault in any of these exploitable operations can leak bits of the cipher's key. The *Cipher Analysis* module works at the algorithmic level and uses a coloring scheme that captures the effect of the fault spread during execution and then the threat in terms of key extraction. The output of this step is a list of vulnerable cipher operations and needs to be done once for every cipher algorithm.

The subsequent steps in the framework operate on the implementation of the algorithm. A cipher can be implemented in multiple ways, for example, optimized for performance or size. The vulnerable operations can, therefore, be manifested in different ways in each implementation. Therefore, the next step in the framework is to map the vulnerable cipher operations to the cipher's software implementation, using the *Mapping Module*. This is done using model checking and static analysis tools. The output of this step is the mapped implementation with vulnerable regions are flagged. These implementations are then passed to the compiler framework where all the exploitable intermediate instructions are determined in the *Fault Evaluation* pass. A fault in any of these instructions can be exploited to reveal bits of the cipher's secret key. The success of the attack, however, varies based on the program structure and the underlying microprocessor. The next pass, *Fault Exploitability Quantification*, quantifies the exploitability of each vulnerable fault that can be inserted by evaluating the program's structure and fault propagation properties of the microprocessor. The final stage, *Countermeasure Addition*, incorporates appropriate countermeasures into the implementation. The amount of protection added by the compiler can be controlled by threshold set by the user. For example, the user can set a lower threshold for more sensitive applications, therefore protecting a larger portion of the vulnerable instructions in the implementation. On the other hand, a higher threshold would cause a smaller portion of the vulnerable instructions protected.

Contributions. This paper discusses the end-to-end design of FortiFix highlighting the implementation aspects. The core idea of individual modules in FortiFix have been published before. Namely, the Cipher Analysis module is published in [16], the Mapping Module in [14], and Fault Exploitability Quantification Module in [13].

The contributions of this paper is summarized as follows:

- We provide the end-to-end design of FortiFix, that takes as input a source code of a block cipher and uses compiler passes to generate an executable that is protected against fault attacks.
- Fault attack vulnerabilities not just depend on the cipher algorithm but also on the way it is implemented, and the underlying processor. Thus, to demonstrate the functionality of FortiFix, we use two AES-128 implementations and an implementation of CLEFIA-128. One of the AES-128 implementations is optimized for memory constraint devices, while the other uses large tables for look-ups. We consider three microprocessors, namely, ARM (32-bit), RISC-V(32-bit), and TI's MSP-430 (16-bit).
- The compiler framework strategically inserts countermeasures based on the user's requirements. A security-critical application can be compiled with more protection inserted, compared to a less critical application. This allows users to trade-off between security and performance.
- The entire source code is made open-source and is available here: (https://bitbucket.org/keerthikamal/fortifix/src/master/).

Structure of the Paper. The paper is organized as follows: Section 2 provides the necessary background. Section 3 includes the recent works for automated fault vulnerability detection tools.

Section 4 discusses FortiFix framework, expanding different module, and the evaluation results. Section 5 includes the usage details and use-case of FortiFix. Section 6 concludes the paper.

2 BACKGROUND

2.1 Fault Attacks

In a fault attack, the attacker corrupts the output of an operation by injecting a fault during the cipher execution [5]. The fault propagates through the cipher resulting in a faulty ciphertext. The attacker uses this faulty ciphertext to retrieve bits of the secret key.

However, not all faulty ciphertexts are exploitable. The location of the faults induced is critical to the success of a fault attack. For example, in AES, a fault induced before the 7^{th} round [21] is not easily exploitable by a differential fault attack, while a fault induced after the 8^{th} round cannot retrieve all bits of the secret key. The optimal locations are in the 8^{th} round, where a fault induced can be used to recover all bits of the secret key.

Extracting key bits by a fault injected in the 8th round of AES requires strategies such as the *Differential fault analysis* (DFA). In DFA, the faulty ciphertext and its fault-free are used to build equations in either one of the following forms:

$$S\ (x \oplus k) \oplus S\ (x' \oplus k) = \delta \tag{1}$$

$$S^{-1}\ (y \oplus k) \oplus S^{-1}\ (y' \oplus k) = \delta \tag{2}$$

In the above equations, S refers to the cipher's S-box operation; x and y are respectively, the input and output; and x' and y' are the input and output of the same S-box when a fault occurred. A typical differential fault attack involves solving difference equations (either of the form (1) or (2)) to obtain part of the cipher's round key k.

Fault Models. Fault attacks are influenced by the type of disturbance induced. For example, certain fault attacks have a strict requirement of a stuck-at-zero or stuck-at-one fault; while others require a fixed number of bits to be affected. Moreover, most attacks require that the injected fault be transient, meaning that it is momentarily applied. In this paper, we use a single random byte fault model and can evaluate block ciphers for differential fault attacks.

2.2 Intermediate Representation (IR)

The LLVM compiler converts high-level implementations to machine code in different stages. The *front end pass* includes lexical analysis, syntax analysis, and semantic analysis that converts the high-level the implementation to *Intermediate Representation* (IR) instructions. Many of the modules in FortiFix framework (Figure 2) are designed to work of these IR instructions. Each IR instruction is in Static Single Assignment (SSA) form, consisting of an opcode, read and write operands.

Definition 2.1. **[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.*

Example 2.2. 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.

The *Back End Pass* of the compiler includes code optimization and code generation to convert the IR instructions to executable object codes.

3 RELATED WORK

Manually analyzing fault attack vulnerabilities is a tedious task that requires considerable expertise. Naïvely inserting countermeasures could potentially result in huge overheads. A significant reason for

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

Tools	Input Type	Output	Cryptographic properties	Target Processor	Automated Countermeasure Insertion
XFC [16]	High-Level	Exploitable Operations	Differential Properties of S-Box	✗	✗
ExpFault [20]	High-Level	Exploitable Operations	Impossible Differential Properties	✗	✗
AFA [23]	High-Level	Exploitable Operations	Algebraic Properties of Cipher	✗	✗
ExploreFault [10]	High-Level	Exploitable Operations	Differential Properties	✗	✗
DATAC [6]	Assembly code	Vulnerable instructions	N/A	Atmel AVR	✗
TADA [11]	Assembly code	Attack Details for last round	N/A	AVR ATmega	✗
ADFA [1]	LLVM IR	Vulnerable IR Instructions	N/A	✗	✗
Proposed - FortiFix	Source Code	Executables with countermeasure inserted	Differential Properties of S-Box	TI MSP 430 (16-bit) ARM(32-bit) RISC-V(32-bit)	✓

these overheads is the fact that developers are unaware of exactly which parts of the implementation are vulnerable to fault attacks. In a typical cipher implementation, out of the millions of faults that can be injected, very few of them can be exploited. Identifying the vulnerable regions or nodes in the implementation is non-trivial and very specific to the cipher algorithm and its implementation. Further, the impact of an injected fault introduced at a specific location varies based on the underlying processor architecture.

In the last few years, researchers have introduced fault attack vulnerability assessment tools that can quickly and efficiently evaluate fault attack vulnerabilities. Most of the tools [23, 20, 16, 10] work at the algorithmic level, taking a high-level representation of the cipher as input to identify the vulnerable cipher-operations. While this approach, can evaluate vulnerabilities due to powerful fault attacks like the DFA and IDFA, they have limited application because they work at the algorithm level and cannot assess implementations. 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 [6, 11, 1, 14] rather than its high-level representation. However, properties such as differential and algebraic properties of a block cipher's internal operations, like the S-Boxes, can greatly influence the success of a fault attack. These internal features and characteristics of the block cipher are not considered. Further, the output of these tools is binary, only denoting whether an instruction is exploitable or not. The tools cannot quantify the exploitability.

Table 1 includes the state-of-the-art techniques to detect fault attack vulnerability from crypto implementations. Unlike these works, FortiFix works on the implementation level, to find vulnerabilities by considering cryptographic properties of the cipher such as differential and impossible differential properties, and also quantifies the vulnerabilities based on the underlying processor architectures. Further more the framework is incorporated as transformation passes at the LLVM compiler that can be used to automatically insert optimized countermeasures and that are tuned based on the security requirements. Finally, it generates executables with the countermeasures inserted.

4 FORTIFIX: FAULT ATTACK AWARE COMPILER

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

- **Fault should impact vulnerable operations.** The fault should target the small subset of vulnerable operations in the cipher. Faults injected elsewhere in the program cannot be exploited.
- **Corrupt instruction output.** Most fault attacks require that the fault modifies an instruction output and not halt execution.
- **Propagate to the output.** The fault at the target instruction should propagate to the ciphertext.

FortiFix considers all the three influencing aspects of fault attacks to detect and patch the exploitable instructions. To analyze cipher implementations considering complex cryptographic properties, program structure and the underlying hardware, we propose a two-phase compiler framework as depicted in Figure 2. (1) *Pre-Compiler Phase*: that runs outside the LLVM compiler and identifies the fault-exploitable operations from a block cipher algorithm, and then maps the exploitability to different cipher implementations. (2) *Compiler Phase*: is incorporated as transformation passes within the LLVM compiler that can find the exploitable instructions from the IR instructions of the cipher, quantifies the exploitability based on the underlying hardware, incorporate countermeasures to meet the security requirements, and finally generates executables that are protected from fault attacks.

4.1 Pre-Compiler Phase

This phase works outside the compiler module and determines the exploitable operations at the high-level representation of the block cipher. The *Pre-Compiler Phase* has two modules as depicted in Figure 2. The (P1) *Cipher Analysis* module finds the fault exploitable operations from a high-level representation of the cipher and the (P2) *Mapping Module* that maps the exploitable operations from high-level representation to different implementations of the cipher.

P1: Cipher Analysis. Module P1 named *Cipher Analysis* finds the exploitable operations in the cipher algorithm. It also provides the attack complexity and number of key bits that can be recovered from differential fault attacks. The input to the module is the high-level representation of the cipher,

⟨ begin ⟩ ⟨ linear ⟩ ⟨ KeyWhitening ⟩
⟨$\mathcal{A}_1$⟩ : {$\mathcal{P}$1 : XOR (P[1]),LKUP(1,KEY0)}
⟨$\mathcal{A}_2$⟩ : {$\mathcal{P}$2 : XOR (P[2]),LKUP(2,KEY0)}
⟨ nonlinear ⟩⟨ SubByte ⟩
⟨$\mathcal{A}_3$⟩ : {$\mathcal{A}_1$: LKUP($\mathcal{A}_1$,SBOX)}
⟨$\mathcal{A}_4$⟩ : {$\mathcal{A}_2$: LKUP($\mathcal{A}_2$,SBOX)}
⟨ linear ⟩⟨ Swap ⟩
⟨$\mathcal{A}_5$⟩ : {$\mathcal{A}_4$}
⟨$\mathcal{A}_6$⟩ : {$\mathcal{A}_3$}
⟨ linear ⟩⟨ Diffusion ⟩
⟨$\mathcal{A}_7$⟩ : {⟨$\mathcal{A}_5, \mathcal{A}_6$⟩ : XOR(MUL3($\mathcal{A}_5, \mathcal{A}_6$}
⟨$\mathcal{A}_8$⟩ : {⟨$\mathcal{A}_5, \mathcal{A}_6$⟩ : XOR(MUL3($\mathcal{A}_6, \mathcal{A}_5$}
⟨ linear ⟩⟨ KeyAddition ⟩
⟨$\mathcal{A}_9$⟩ : {$\mathcal{A}_7$: XOR ($\mathcal{A}_7, \mathcal{A}_8$),LKUP(1,KEY1)}
⟨$\mathcal{A}_{10}$⟩ : {$\mathcal{A}_8$: XOR ($\mathcal{A}_7, \mathcal{A}_8$),LKUP(2,KEY1)}
⟨ end ⟩

Fig. 3. BCSL representation of the Toy Cipher

named BCSL [16]. The Block Cipher Specification Language (BCSL) captures various operations performed by the cipher and the information flow of the plaintext through the algorithm. Figure 3 shows the BCSL representation of a toy cipher with two byte input, that performs a few operations such as KeyAddition, SubByte, Swap and Diffusion, and the sub-operations are denoted from $\mathcal{A}_1$, $\mathcal{A}_2 \cdots \mathcal{A}_{10}$, and each sub-operations performs a byte operation in the cipher.

The module analyzes how the fault is propagated within the block cipher using a coloring scheme. When a fault is induced into a block cipher byte, we assign a new color. We then trace the propagation of the fault through the subsequent functions. Each time a colored byte propagates through a non-linear function (e.g. SubByte), it changes to a new color. Every output of a linear function is colored the same as the input if all its inputs have the same color. On the other hand, if the inputs have more than one color, the output is assigned a new color.

Table 2. Output of P1: *Cipher Analysis* Module.

Cipher	Round Number	#Derived Keys	Offline Complexity
AES	1-7	0	N/A
	7-8	128	2^8
	8-9	32	2^8
	9-10	0	N/A
CLEFIA	13-14	32	2^8
	14-17	32	$2^{4.76}$

Example 4.1. The figure below shows the coloring technique for the toy cipher given in Figure 3. The fault is injected in before the SubByte operation, where a new color is assigned. When it propagates through a non-linear function (e.g., SubByte), a new color is assigned, and for linear function (e.g., MixColumn), the color remains the same. When an input of a function takes multiple colors, the output is assigned with a new color irrespective of whether it's linear or non-linear.

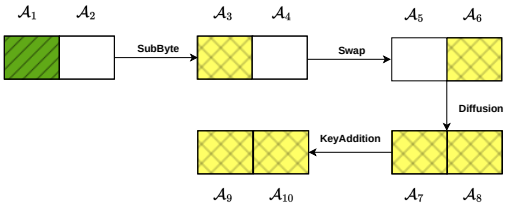


Fig. 4. Fault Propagation in KeyWhitening of the toy cipher given in Figure 4.

The coloring scheme captures the flow of information and relationships between intermediate operations. Outputs of operations with the same color are linearly related, while outputs with different colors are not related. We use the color scheme to build equations of the form (e.g., Equation 1 and 2). These equations can be solved to derive parts of the secret key. The number of such equations formed is associated with a search space that represents the offline complexity of determining the corresponding key parts.

Table 2 shows the output of the P1 module for two block ciphers. Each row shows the round number of the cipher and the vulnerability to a single random fault temporally injected in any of these round operations. For example, if a fault is injected in any operations between the round 7 and 8 of AES, then 128-bit key bits can be derived with an offline complexity of 2^8. On the other hand, a fault injected between the 8-th and 9-th round of AES, can be used to derive 32-bits of the key with a similar offline complexity. Faults injected in any other rounds are not exploitable by the Differential Fault Analysis (DFA). More details about the Cipher Analysis module can be found in [16] and [18].

P2: Mapping Module. The implementations of block ciphers differ considerably depending on the target application and the underlying platform. For instance, typical AES-128 implementations on 32-bit platforms use 5 T-tables, where the operations SubBytes, ShiftRows, and MixColumns are merged and replaced with look-ups to four 1KByte tables. On 8-bit memory-constrained processors, SubBytes is implemented using a single 256-byte look-up table. Thus, given a block cipher algorithm, different programs would implement the algorithm in different ways. However all the implementations are functionally equivalent, and the output of each round would be the same, irrespective of how the round is implemented (for example merged operations using T-tables or 256-byte lookup-table S-box). The module P2 uses this property to map the cipher algorithm to a given implementation. Once mapped, the vulnerable cipher operations discovered in P1 can be identified in the cipher implementation.

The *Mapping Module* takes as input a C implementation of the cipher that is synthesized from a BCSL cipher definition [19], where exploitable operations identified from `P1` are annotated. This implementation acts as a golden reference. It maps each operation in the golden reference to program expressions (statements) in the given source code. The main challenge is that the cipher operations in the source code can be merged, interchanged, or swapped compared to the golden reference. Hence, the mapping module should be able to handle all possible mapping cases.

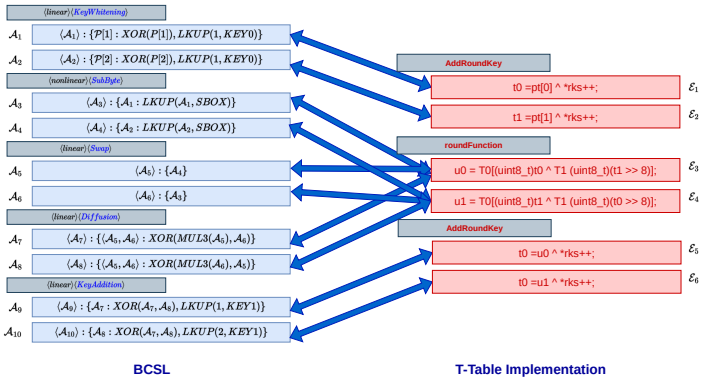


Fig. 5. Mapping from BCSL to Implementation

To effectively map the sub-operations to expressions in implementation, we use a model checking tool named *CBMC* [7]. The model checker extracts the sub-operations from the golden reference and program expressions from the source code, and checks the equivalence with the help of an underlying SAT solver. The mapping module parses the BCSL and the implementation until the equivalence is found. If an unmapped node is found in the golden reference, the module generates the information flow graph within the BCSL and performs a reverse information flow analysis to determine the merged and interchanged operations. The parser continues running until all the sub-operations in BCSL are mapped to that of the implementation. The output of the Mapping Module is the mapped implementation with exploitable expressions of the source code colored.

Example 4.2. Figure 5 shows the mapping of the toy cipher shown in Figure 3. The objective of the mapping module is to map the sub-operations in the Block Cipher Specification (*i.e.* $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$, $\cdots$, $\mathcal{A}_{10}$) to program expressions (*i.e.* $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3$, $\cdots$, $\mathcal{E}_6$). The mapping below shows the output of the *Mapping Module*, for the toy cipher implementation given in Figure 5.

$$\begin{array}{ccc} (\mathcal{A}_1) & \leftrightarrow & (\mathcal{E}_1) \\ (\mathcal{A}_2) & \leftrightarrow & (\mathcal{E}_2) \\ (\mathcal{A}_3,\mathcal{A}_5,\mathcal{A}_7) & \leftrightarrow & (\mathcal{E}_3) \\ (\mathcal{A}_4,\mathcal{A}_6,\mathcal{A}_8) & \leftrightarrow & (\mathcal{E}_4) \\ (\mathcal{A}_9) & \leftrightarrow & (\mathcal{E}_5) \\ (\mathcal{A}_{10}) & \leftrightarrow & (\mathcal{E}_6) \end{array}$$

This mapping can be interpreted as follows: operations $\mathcal{A}_1$ and $\mathcal{A}_2$ in the golden reference (BCSL definition) are realized by the program expression $\mathcal{E}_1$ and $\mathcal{E}_2$ respectively. The operations $\mathcal{A}_3$, $\mathcal{A}_5$, $\mathcal{A}_7$ are realized by the program expression $\mathcal{E}_3$. This is an example where operations are merged. Similarly, operations may be shuffled, or swapped within a round. The use of the model checker to determine equivalence ensures these implementation variations can be efficiently handled.

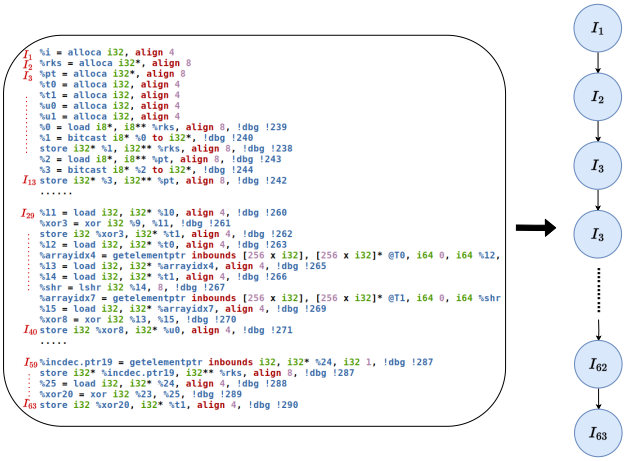


Fig. 6. Control Flow Graph for the toy cipher implementation given in Figure 5.

Table 3 shows the result of the mapping module for two different AES implementations and CLEFIA. The interesting observation is that the number of expressions varies based on the implementation, and hence, the time for mapping also varies based on how the cipher is implemented. More information about the Mapping Module can be found in [14].

Table 3. Output of the P1 : *Mapping Module* for two AES implementations, and CLEFIA.

#	# No of Sub-operation	# No of Expression	Time in mins.
AES-128 (Look-Up)	640	780	59.33
AES-128 (T-Table)		196	98
CLEFIA	1184	1425	205.56

4.2 Compiler Phase

The front-end of compilers take the source code as input and generates Intermediate Representation (IR), while the compiler's back-end pass uses the IR to generate the executable. FortiFix uses transformation passes that work on the IR to analyze and modify it. The advantage of using transformation passes is that the IRs allow to work at a low level of abstraction. It also provides interoperability, as the IR is hardware independent.

The input to the compiler is the source code with exploitable expressions colored. The FortiFix compiler phase is depicted in Figure 2. The colored source code is parsed through the *Front-End pass* of the compiler to generate the IR instructions. The IR is then parsed through three transformation passes of FortiFix, C1 *Fault Evaluation*, finds the fault exploitable instructions in the IR instructions, C2 *Fault Exploitable Quantification*, quantifies the exploitability based on the underlying hardware C3 *Countermeasure Addition*, adds the countermeasure based on the user's security requirements, by instrumenting code at the IR level.

C1: Fault Evaluation. The input to the C1 pass is the IR instructions where the initial set of exploitable expressions are colored by the pre-compiler stage. The C1 pass converts the IR instructions to a *Customized Flow Graph (CFG)*, where 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 flow such that instruction $\mathcal{I}_j$ executes immediately

after instruction $\mathcal{I}_i$. Figure 6 depicts the IR instructions and the corresponding CFG generated by the C1 pass of the FortiFix compiler for the toy cipher implementation given in Figure 5.

A corrupted output of a colored IR instruction can be used by an attacker to glean bits of the secret key. For a given IR instruction, C1 identifies all predecessor instructions that can influence the colored IR. To identify the exploitable instructions, C1 performs a reverse data flow analysis on the control flow graph to determine the list of predecessor nodes that can influence the exploitable IR node. The output is a list of all exploitable IR instructions. A fault injected in any of these exploitable instructions could result in an incorrect output, which in turn can be successfully exploited in a fault attack.

Example 4.3. To corrupt the output of the node I_{63}, the fault can be injected in the I_{63} node or propagated from the predecessor node. The C2 pass gives the list of exploitable nodes. i.e., $(I_1 \cdots I_{31}, I_{41} \cdots I_{49}, I_{57} \cdots I_{63})$. A fault induced in any of these instruction can corrupt the output of the node I_{63}. More details about the Fault Evaluation module can be obtained from [14].

Table 4. Statistics of step C1: *Fault Evaluation* for 2 AES implementations, and CLEFIA.

#	# IR Instruction in the CFG	% of exploitable instructions.	Time in secs.
AES-128 (Look-Up)	7206	6.56	38.2
AES-128 (T-Table)	4299	3.71	15.5
CLEFIA	1026	6.54	105.5

Table 4 shows the results of the C1 pass of the compiler. The interesting observation is that, the percentage of exploitable instructions varies based on how the cipher is implemented and not just based on the algorithm. A fault in any of these exploitable nodes can be used to mount a fault attack. Comparing different implementations of AES-128, the T-Table implementation has the lesser number of exploitable instructions.

C2: Fault Exploitable Quantification. The input to the C2 pass is the IR instructions where all the exploitable IR instructions are colored. However, the color does not signify the scale of the exploitability. The C2 pass quantifies the exploitability based on the underlying hardware. The C2 pass takes an additional input called the *Hardware Fault Probability*, which is the probability of fault-induced instruction corruption. To study the effect of fault-corrupting instruction on different hardware, we have analyzed three different processor, namely, TI-MSP430(16-bit), RISCV(32-bit), and ARM(32-bit).

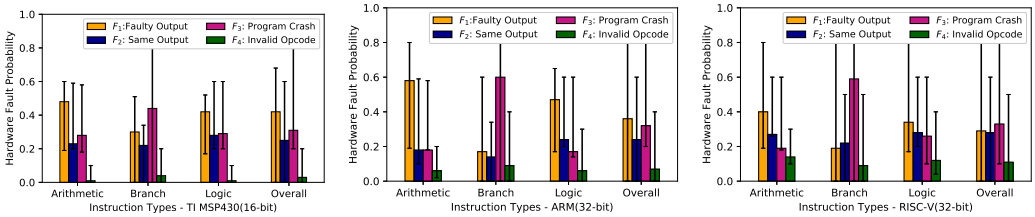
Hardware Fault Probability: When a single fault is transiently injected during an instruction execution, it can manifest by either altering, leaving unaltered the instruction output, or terminating the program. A fault due to the altered instruction output may propagate, resulting in a faulty ciphertext.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

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

F_2. **Fault is not activated**: The induced fault alters the instruction execution but does not change the instruction output.

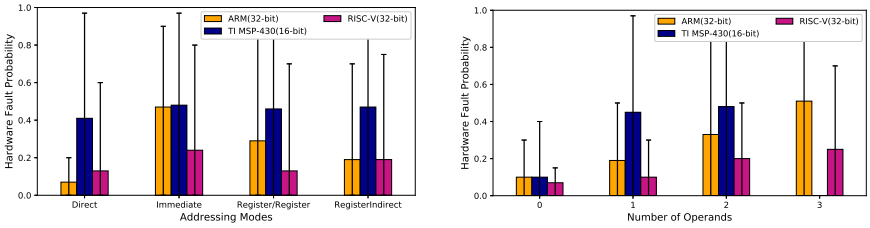
F_3&F_4. **Program is terminated**: The induced fault leads to an illegal operation, causing the program to terminate.

The faults in set F_2, F_3, and F_4 cannot induce a successful fault attack, as the outcomes does not provide the faulty ciphertext that is necessary to carry out the attacks. When an injected fault changes



(a) (b) (c)

Fig. 7. Outcome of fault injection on (a) TI-MSP430(16-bit) , (b)ARM (32-bit) and (c) RISC-V(32-bit).



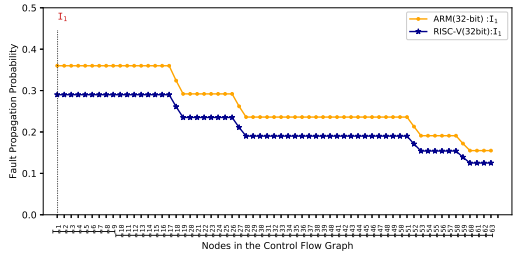
(a) (b)

Fig. 8. (a) Outcome of fault injection on different addressing modes , (b) Outcome of fault injection on different number of operands).

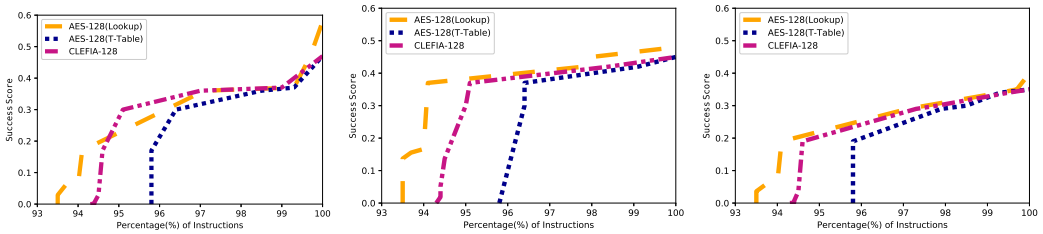
bits in an opcode, it can result in a valid or invalid instruction. An invalid instruction opcode results in program termination (F_4), while a valid instruction can have any of the remaining three (*i.e.* F_1, F_2, or F_3) outcomes. The probability of these outcomes depends not just on the type of instruction but also on the instruction encoding. They are thus unique to each Instruction Set Architecture. To understand these probabilities, we consider three microprocessors, namely, TI's MSP-430 (16-bit), ARM (32-bit), and RISCV (32-bit), to identify the reliance of fault injection on the underlying architecture. For each of these microprocessors, we generate random programs [1], cross-compile and execute the binary multiple times in a simulator. In each execution, faults are injected in an instruction using simulation tools, such as by modifying the instruction memory and then observing the instruction output. The result of the fault falls in one of the four classes *i.e.* F_1, F_2, F_3, or F_4. Figure 7 shows the probability of producing incorrect output for different instruction types on three microprocessors.

The C2 pass takes the above hardware fault probability as input and performs the quantification analysis on the IR. The output of an instruction in the program can be corrupted either by a fault injected in that instruction or a fault injected in a previous instruction that propagates to the given instruction. The latter depends on the program structure. Fault propagation can be done in two ways. The first is through registers, where the output of one instruction is used as an input to another. Alternatively, faults can propagate through memory operations. For instance, by a store of faulted data to memory, followed by a subsequent load from the same address. We determine the fault propagation probability by dividing the instructions into three different classes (1) fault in data-dependent instructions, (2) fault in control-dependent instructions, and (3) fault in the memory-dependent instructions. The individual instructions level probability is taken and propagated through the data-dependent paths to the ciphertext, and the maximum probability at each node considering the three cases is taken as the success score of the node.

[1] Csmith(https://embed.cs.utah.edu/csmith/)



Example 4.4. Figure above shows the fault propagation probability for the toy cipher implementation given in Figure 5. Figure shows the how the fault propagation probability changes on two different architecture when the fault is induced on instruction I_1

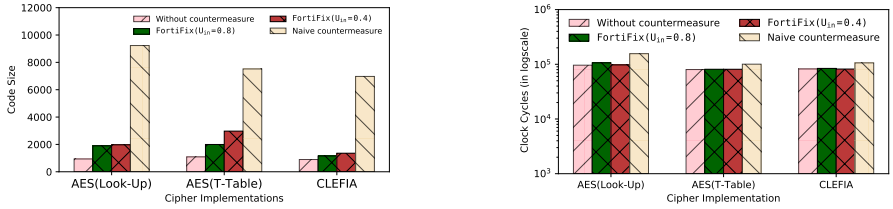


(a) ARM(32-bit) (b) MSP430(16-bit) (c) RISC-V(32-bit)

Fig. 9. SuccessScore of three different ciphers across three different architectures.

During our analysis, we observed that the opcode encoding in the microprocessor plays an important role in the vulnerability. Figure 7 shows the probability of producing incorrect output for different instruction classes on three microprocessors. This brings out interesting observations, such as TI MSP-430 is most vulnerable to fault attacks compared to other processors evaluated. This is because TI MSP-430 has the highest instruction density. Thus, the probability of program termination due to fault in the opcode is the lowest. Among the 32-bit processors considered, ARM is more vulnerable to fault injection than RISC-V, as RISC-V has a considerably large number of unused opcodes, hence low density and a higher chance of program termination due to fault. Figure 9(a) shows the percentage of exploitable instruction of AES-128 (LookUp) table based implementation along with the success score on three different hardware. The graph shows that the success score varies based on the underlying hardware. Figure 8 shows how the hardware fault probability vary based on the addressing mode and number of operands. The probability is higher for immediate addressing mode across the architecture and the probability increases as the number of operands increases. Figure 9 shows the success score of three ciphers on TI-MSP430(16-bit). AES-128 (T-table) based implementation is the least vulnerable, and the Look-Up table based implementation is most vulnerable to fault attack.

C3**: Countermeasure Addition.** Incorporating fault attack countermeasures is expensive. It can increase run time overheads by over 100% and memory requirements by over 800%. These overheads are unacceptable for several applications, especially where time and resources are critical. We incorporate the countermeasures based on the security requirement. For instance, a block cipher used in critical infrastructure would require much more secure implementations compared to an application in a consumer device. Thus, for such applications, designers typically would want to prioritize security in lieu of performance. Such trade-offs would be less acceptable for the consumer device, especially in a resource-constraint device, where each byte and each clock cycle is valuable.



(a) Code Size (b) Execution Time

Fig. 10. Comparison of unprotected implementations, naively protected implementations with FortiFix based automatic protection.

The input to the C3 pass is the IR instructions with all the exploitable instructions, along with the success score of corrupting the output of the program. FortiFix introduced redundancy-based countermeasure at the IR level, where each exploitable instruction is duplicated and then performs a check at the end of each instruction to determine whether the output of the instruction is corrupted or not.

Example 4.5. 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 T1[0].

```
%74 = getelementptr inbounds [256 x i32], [256 x i32]* @T1, i64 0, i64 %shr12
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 T0[256] given in the instruction is duplicated in T1[256]; (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 violet are the replicated operations, while the instructions in blue perform the comparison between the original and replicated operations. The executable generated would thus be protected against the differential fault attacks.

```
%74 = getelementptr inbounds [256 x i32], [256 x i32]* @T1, i64 0, i64 %shr12
store i8 %67, i8* %74
%75 = getelementptr inbounds [256 x i32], [256 x i32]* @T01,i64 0,i64 %shr121
store i8 %67, i8* %75
%76 = getelementptr inbounds [256 x i32], [256 x i32]* @T0,i64 0,i64 %shr12
%77 = load i8, i8* %76
%78 = zext i8 %77 to i32
%79 = getelementptr inbounds [256 x i32], [256 x i32]* @T01,i64 0,i64 %shr121
%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:
     ...
```

The *BackEnd Pass* of the compiler converts the instrumented IR instructions to executable where countermeasures are already incorporated. Figure 10 shows the results of C3 stage the compiler for two different implementations of AES and CLEFIA. Compared to naively incorporating fault attack

countermeasures, the quantification based on the underlying microprocessor reduces overheads by 50%-120%, which can be further reduced based on the user's security requirements.

5 IMPLEMENTATION AND USAGE CASE OF FORTIFIX

Our framework FortiFix is open source and is publicly available[2]. The FortiFix framework has two phases: the pre-compiler phase, implemented using the CBMC model checking tool and python scripts; the compiler phase is developed using the LLVM Compiler. The configuration of the tools used for the development of FortiFix framework is as follows:

Configuration :

```
CBMC --version 5.6 or greater
Python --version 2.7
LLVM -- version 7.0.0svn
Clang -- version 7.0.0
```

5.1 Execution Steps

Before using the framework, the underlying architecture of the target processor and the user's security requirements need to be mentioned in the configure.py file.

```
$ Configure the architecture, user inputs etc. in configure.py
$ python FortiFix.py
```

The `FortiFix.py` framework will take a few minutes to complete the executions and generate a set of intermediate outputs at each step. The following section includes details about each step in the framework and the input/output format at each step.

5.2 Input and Output Interpretation at each step of FortiFix

The input/output interpretation of the pre-compiler and compiler modules of FortiFix are as follows:

P1 *Cipher Analysis*: The input to the module is the BCS representation as shown in Figure 11(b). Each line shows a byte operation in the function, along with the linear and nonlinear functions. The module performs a comprehensive analysis and gives the list of exploitable operations as shown in Figure 11(a). For example, if we inject a byte fault on the first byte of function `F28`, then 128 bits of the secret key can be determined from the correct and fault ciphertext pairs. Hence, the operation `F28[1]` is marked exploitable.

P2 *Mapping Module*: The input to the module is the source code (refer Figure 12(a)) and the BCS (refer Figure 11(b)) of the given crypto algorithm. The module maps the operations in BCS to expressions in source code using the CBMC model-checking tool. Figure 12(a) is the T-Table implementation of AES, and (b) shows the result of the mapping module for the first round of AES, where operations are mapped to expression. For example, `G32`, the operation in line 32 of BCS representation, maps to `E734`, the expression in line 734 of the implementation.

C *Compiler Phase*: Once the exploitable operations are mapped, the next phase is the compiler module. The Front End Pass generates the IR from the source code, where exploitable instructions are marked. Figure 13 shows the output of different stages of the compiler module. The `C1` module finds all the exploitable nodes from the CFG. The `C2` module quantifies the vulnerability and gives the exploitability score for each node based on the architecture configured. The `C3` module adds

[2]FortiFix (https://bitbucket.org/keerthikamal/fortifix/src/master/)

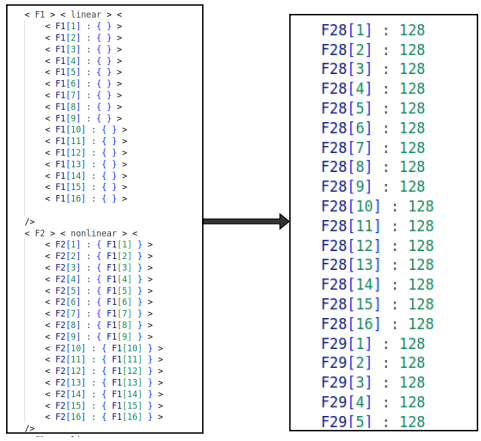


Fig. 11. The cipher Analysis Phase takes BCS as input and determine the exploitable operations from BCS.

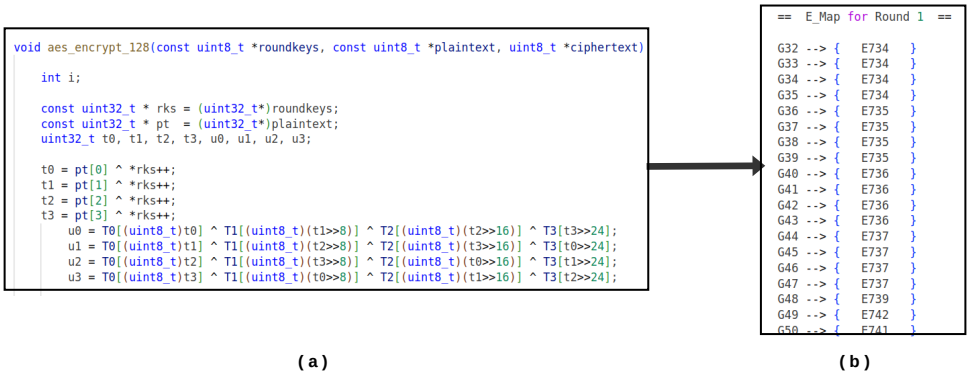


Fig. 12. The mapping module takes the crypto implementation and output of `P1` to determine all possible mapping. Operation in BCS (`G32`) maps to expression (`E734`) in the implementation.

the countermeasure based on the requirements from the user inputs configured. For example, once

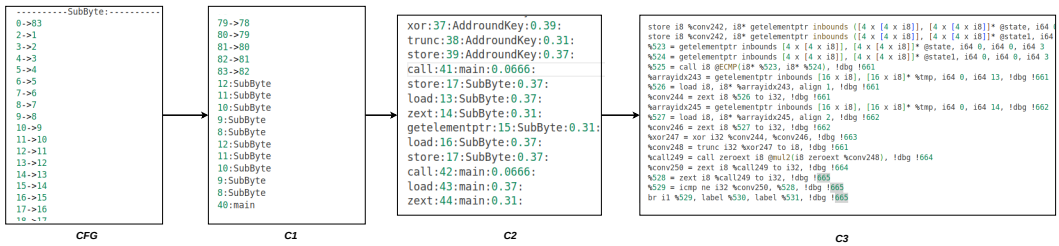


Fig. 13. Output of different stages of the compiler.

the CFG is generated from the intermediate representation (IR) as shown in Figure 13. `C1` takes

CFG and IR as input and outputs lists of exploitable nodes from the CFG. In Figure 13, the output `12:SubByte` indicates that node 12 of the `SubByte` function is marked as exploitable by the `C1` pass. The input to C2 is the modified IR and the list of exploitable nodes. `C2` outputs the vulnerability score for each node. For example, `load:16:SubByte:0.37` indicates that node 16 of SubByte has a vulnerability score of 0.37. Once the vulnerability score of all the nodes is determined, then the `C3` pass of the compiler adds countermeasures based on the user security requirements configured.

5.3 Execution Time and Coverage

Table 5 shows the execution time of the FortiFix framework for two AES implementations and a CLEFIA implementation. It also shows the security coverage the framework provides after inserting countermeasures for security requirements 0.4 and 0.8. Notice that the security coverage increases with the user's security input.

Table 5. Execution Time and Security Coverage that FortiFix provides for two AES implementations and an implementation of CLEFIA.

#	Time in mins.	Security Requirement	Coverage (%)
AES-128 (Look-Up)	59.96	0.4	79
		0.8	84
AES-128 (T-Table)	98.25	0.4	86
		0.8	90
CLEFIA	207.56	0.4	81
		0.8	85

6 CONCLUSION

Our compiler framework FortiFix, can automatically determine the exploitable instructions in the cipher implementations, quantify and patch the vulnerability to generate fault-attack resistant executables. We show the exploitability and the fault propagation probability depends on the cipher algorithm, its implementation, as well as the Instruction Set Architecture (ISA) of the processor. Our evaluation of three cipher implementations on three hardware platforms brings out interesting observations. For instance, TI MSP 430 (16-bit) is the most vulnerable to fault attacks. Comparing the 32-bit RISC processors, ARM is more vulnerable to fault injection than RISC-V. Comparing different implementations of AES, the T-table implementation is the most secure against fault attacks. The quantification that the framework provides can be used to strategically used to choose the right countermeasure in block cipher implementations to meet the security requirements. Currently, FortiFix cannot determine the vulnerability on an FPGA as ciphers implemented in RTL. Providing such a framework for FPGAs would be an interesting further direction.

REFERENCES

[1] Giovanni Agosta et al. "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*. 2014, p. 137. DOI: 10.1145/2659651.2659709. URL: https://doi.org/10.1145/2659651.2659709.

[2] S. Ali and D. Mukhopadhyay. "Improved Differential Fault Analysis of CLEFIA". In: *FDTC*. 2013, pp. 60–70.

[3] Nasour Bagheri, Reza Ebrahimpour, and Navid Ghaedi. "New differential fault analysis on PRESENT". In: *EURASIP J. Adv. Signal Process.* 2013 (2013), p. 145. DOI: 10.1186/1687-6180-2013-145. URL: https://doi.org/10.1186/1687-6180-2013-145.

[4] 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*. 1997, pp. 513–525. DOI: 10.1007/BFb0052259. URL: https://doi.org/10.1007/BFb0052259.

[5] 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*. 1997, pp. 37–51. DOI: 10.1007/3-540-69053-0_4. URL: https://doi.org/10.1007/3-540-69053-0%5C_4.

[6] Jakub Breier, Xiaolu Hou, and Yang Liu. "Fault Attacks Made Easy: Differential Fault Analysis Automation on Assembly Code". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 96–122. DOI: 10.13154/tches.v2018.i2.96-122. URL: https://doi.org/10.13154/tches.v2018.i2.96-122.

[7] 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*. 2004, pp. 168–176.

[8] 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.

[9] 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*. 2011, pp. 274–291.

[10] Hao Guo et al. "ExploreFault: Identifying Exploitable Fault Models in Block Ciphers with Reinforcement Learning". In: *60th ACM/IEEE Design Automation Conference, DAC 2023, San Francisco, CA, USA, July 9-13, 2023*. IEEE, 2023, pp. 1–6. DOI: 10.1109/DAC56929.2023.10247953. URL: https://doi.org/10.1109/DAC56929.2023.10247953.

[11] Xiaolu Hou et al. "Fully Automated Differential Fault Analysis on Software Implementations of Cryptographic Algorithms". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 3 (2019), pp. 1–29. DOI: 10.13154/tches.v2019.i3.1-29. URL: https://doi.org/10.13154/tches.v2019.i3.1-29.

[12] Yuming Huo et al. "Improved Differential Fault Attack on the Block Cipher SPECK". In: *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2015, Saint Malo, France, September 13, 2015*. Ed. by Naofumi Homma and Victor Lomné. IEEE Computer Society, 2015, pp. 28–34. DOI: 10.1109/FDTC.2015.15. URL: https://doi.org/10.1109/FDTC.2015.15.

[13] Keerthi K and Chester Rebeiro. "FaultMeter: Quantitative Fault Attack Assessment of Block Cipher Software". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.2 (Mar. 2023), pp. 212–240. DOI: 10.46586/tches.v2023.i2.212-240. URL: https://tches.iacr.org/index.php/TCHES/article/view/10282.

[14] Keerthi K. et al. "FEDS: Comprehensive Fault Attack Exploitability Detection for Software Implementations of Block Ciphers". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.2 (2020), pp. 272–299. DOI: 10.13154/tches.v2020.i2.272-299. URL: https://doi.org/10.13154/tches.v2020.i2.272-299.

[15] Dusko Karaklajic, Jörn-Marc Schmidt, and Ingrid Verbauwhede. "Hardware Designer's Guide to Fault Attacks". In: *IEEE Trans. Very Large Scale Integr. Syst.* 21.12 (2013), pp. 2295–2306. DOI: 10.1109/TVLSI.2012.2231707. URL: https://doi.org/10.1109/TVLSI.2012.2231707.

[16] 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*. 2017, 8:1–8:6. DOI: 10.1145/3061639.3062340. URL: https://doi.org/10.1145/3061639.3062340.

[17] Andrew Kwong et al. "RAMBleed: Reading Bits in Memory Without Accessing Them". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 695–711. DOI: 10.1109/SP40000.2020.00020. URL: https://doi.org/10.1109/SP40000.2020.00020.

[18] Indrani Roy et al. "FaultDroid: An Algorithmic Approach for Fault-Induced Information Leakage Analysis". In: *ACM Trans. Design Autom. Electr. Syst.* 26.1 (2021), 2:1–2:27. DOI: 10.1145/3410336. URL: https://doi.org/10.1145/3410336.

[19] Indrani Roy et al. "SAFARI: Automatic Synthesis of Fault-Attack Resistant Block Cipher Implementations". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39.4 (2020), pp. 752–765. DOI: 10.1109/TCAD.2019.2897629. URL: https://doi.org/10.1109/TCAD.2019.2897629.

[20] Sayandeep Saha, Debdeep Mukhopadhyay, and Pallab Dasgupta. "ExpFault: An Automated Framework for Exploitable Fault Characterization in Block Ciphers". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 242–276. DOI: 10.13154/tches.v2018.i2.242-276. URL: https://doi.org/10.13154/tches.v2018.i2.242-276.

[21] 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*. 2011, pp. 224–233. DOI: 10.1007/978-3-642-21040-2_15.

[22] Harshal Tupsamudre, Shikha Bisht, and Debdeep Mukhopadhyay. "Differential Fault Analysis on the Families of SIMON and SPECK Ciphers". In: *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014, Busan, South Korea, September 23, 2014*. Ed. by Assia Tria and Dooho Choi. IEEE Computer Society, 2014, pp. 40–48. DOI: 10.1109/FDTC.2014.14. URL: https://doi.org/10.1109/FDTC.2014.14.

[23] Fan Zhang et al. "A Framework for the Analysis and Evaluation of Algebraic Fault Attacks on Lightweight Block Ciphers". In: *IEEE Trans. Information Forensics and Security* 11.5 (2016), pp. 1039–1054. DOI: 10.1109/TIFS.2016.2516905. URL: https://doi.org/10.1109/TIFS.2016.2516905.

[24] Fan Zhang et al. "Persistent Fault Attack in Practice". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.2 (2020), pp. 172–195. DOI: 10.13154/tches.v2020.i2.172-195. URL: https://doi.org/10.13154/tches.v2020.i2.172-195.