Fully Automated Differential Fault Analysis on Software Implementations of Cryptographic Algorithms

Xiaolu Hou
School of Comp. Science and Eng.
Nanyang Technological University, Singapore
xlhou@ntu.edu.sg

Jakub Breier
PACE Labs
Nanyang Technological University, Singapore
jbreier@ntu.edu.sg

Fuyuan Zhang
School of Comp. Science and Eng.
Nanyang Technological University, Singapore
fuzh@ntu.edu.sg

Yang Liu
School of Comp. Science and Eng.
Nanyang Technological University, Singapore
yangliu@ntu.edu.sg

ABSTRACT

Emerging technologies with the requirement of small size and portability, such as Internet-of-Things devices, represent a good target for physical attacks, e.g., fault attacks. These attacks often aim at revealing secrets used in cryptographic algorithms, which are the essential building block for communication protocols. Differential Fault Analysis (DFA) is considered as the most popular fault analysis method. While there are techniques that provide a fault analysis automation on the cipher level to some degree, it can be shown that when it comes to software implementations, there are new vulnerabilities, which cannot be found by observing the cipher design specification.

This work bridges the gap by providing a fully automated way to carry out DFA on assembly implementations of symmetric block ciphers. We use a customized data flow graph to represent the program and develop a novel fault analysis methodology to capture the program behavior under faults. We establish an effective description of DFA as constraints that are passed to an SMT solver. We create a tool that takes assembly code as input, analyzes the dependencies among instructions, automatically attacks vulnerable instructions using SMT solver and outputs the attack details that recover the last round key (and possibly the earlier keys). We support our design with evaluations on lightweight ciphers SIMON, SPECK, and PRIDE, and a current NIST standard, AES. By automated assembly analysis, we were able to find new efficient DFA attacks on SIMON, SPECK and PRIDE, exploiting implementation specific vulnerabilities, and a previously published DFA on AES. Moreover, we present a novel DFA on multiplication operation that has never been shown for symmetric block ciphers before. Our experimental evaluation also shows reasonable execution times that are scalable to current cipher designs and can easily outclass the manual analysis.

We note that this is the first work that automatically carries out DFA on cipher implementations without any plaintext or ciphertext information and therefore, can be generally applied to any input data to the cipher.

KEYWORDS

differential fault analysis, cryptographic fault attacks, automation, assembly

1 INTRODUCTION

Internet of Things (IoT) constitutes a significant market that currently comprises over 20 billion devices and is expected to double by 2022 and almost quadruple by 2025, according to IHS [38]. The main requirements for these devices are small size, low cost, and low power profile. Because of these three properties, we are normally looking at low computational power microcontrollers that are sufficient for standard tasks required from IoT platforms, such as reading a sensor data, adjusting settings of home appliances, and communicating with the user. However, because of the connectivity required from IoT devices, security plays a crucial role – no one wants to have their home appliances exposed to the whole world. As stated in [21], IoT security is specific in a way that it includes software, hardware, and network concerns at the same time.

Lightweight cryptography is one of the areas that became crucial with the emergence of IoT. There are numerous algorithms providing sufficient security properties, while keeping the footprint minimal [13]. Some of them work better in hardware, such as SIMON [6] and SKINNY [8], while others aim at software, such as SPECK [6] and ChaCha [9]. However, accessibility of IoT devices and lack of expensive tamper-protection makes them an ideal target for physical attacks, such as Side-Channel Analysis (SCA) and Fault Analysis (FA). These implementation attacks can easily bypass the theoretical security provided on the cipher level. In case of SCA [34], this is done by observing physical characteristics of a device (electromagnetic emanation, timing, etc.) and correlating this information with the values processed in the algorithm. In case of FA [12], the attacker disturbs the computation by intentionally changing the processed values and then gets the secret information by comparing the faulty and the correct outputs.

Differential Fault Analysis (DFA) [12] is normally a method of choice for fault analysis of symmetric key cryptographic algorithms, thanks to its efficiency and simplicity. When properly utilized, the attacker only needs very few encryptions for a secret key recovery. As DFA follows the steps of a reduced-round differential cryptanalysis [11], one can find many attacks that are on the cipher design

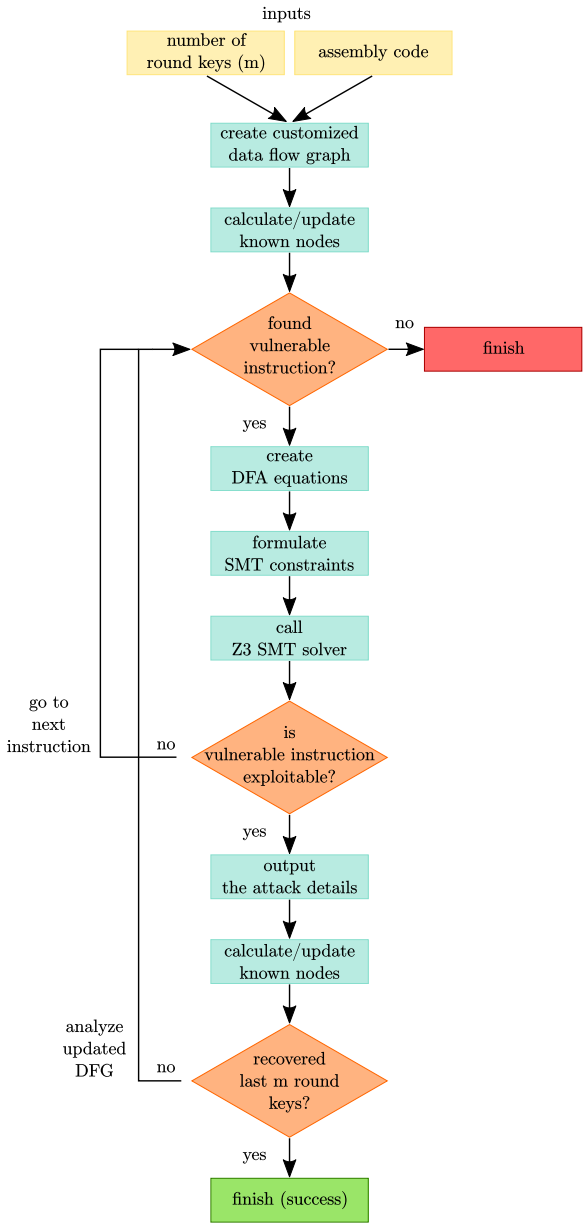

Figure 1: Overview of TADA.

level. Such methods are universal and can normally be applied to any unprotected implementation of a given cipher.

When it comes to the attacks on assembly level, there are not many works in this field, since each implementation is unique and a specific attack on one implementation cannot be generalized to other implementations. However, these implementations can often contain DFA related vulnerabilities that are not visible on the first sight and cannot be identified by simply observing the cipher design [15]. Also, one has to take into account that the number of faults required by DFA on cipher design might be different than when attacking the implementation, making the high-level DFA

Table 1: Examples of linear and non-linear operations.

a	b	$c = a \,\&\, b$	$d = a \oplus b$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

estimates imprecise. For example, in [28], the authors claim they can break SPECK cipher with only 5 ∼ 8 faults, but that is only if the whole cipher state is considered as one large variable. If we have an 8-bit implementation, this number would be 4× bigger in the case of SPECK32/64 and 16× bigger in the case of SPECK128/256.

Furthermore, it is important to do the implementation level analysis since a real attack will always be executed either on assembly level in software or gate level in hardware, by utilizing various fault injection techniques, such as clock/voltage glitch, electromagnetic pulse, or laser pulse [4].

To analyze the vulnerabilities of a software implementation, one has to analyze the assembly code line by line to determine whether it can be exploited by a fault attack. But assembly code of a cryptographic algorithm is normally hundreds to thousands lines long, making it tedious and time consuming for manual analysis.

Shortcomings of current works. As of today, it remains an open problem to automatically find a DFA attack on cryptographic implementation. Current works either focus on cipher level [44] or are not completely automated [15], thus falling short in finding an attack without further manual analysis.

Previous tools (e.g., [44, 50, 51] which all focus on cipher level analysis) search for a possible DFA by enumerating different inputs and then trying to find the key by solving equations for these. Therefore, while the tools are able to find a particular key, they fall short in providing a generic proof that the resulting analysis will work for all keys. Moreover, they also require the knowledge of plaintext in order to carry out the analysis, which is in contrast to most of DFA attacks that do not assume such knowledge.

Our contribution. In this work, we focus on the fully automated DFA attack on software implementations of cryptographic algorithms. We develop a tool that analyzes assembly code statically, constructs an abstract representation of this code, and searches for an attack. In case there is a vulnerability, it outputs the attack procedure that can be used to recover the key by DFA. Unlike aforementioned automated analysis works, our tool does not require any cipher input, such as plaintext and key. Instead, it gives a generic attacking method which can be used to recover any key used for encryption of any plaintext, thus it is aligned with the standard DFA assumptions. This allows us to make the analysis independent of the data, and therefore, to always find an attack if it exists for the given implementation.

We design and implement *TADA – Tool for Automated DFA on Assembly*. An overview of TADA is shown in Figure 1. TADA reads an assembly code from a text file and creates a customized Data Flow Graph (DFG) that records the relations between the variables and identifies the non-linear operations used in the algorithm. It calculates the nodes that can be directly identified from the known data (ciphertext, constants). Then it finds instructions vulnerable to DFA by analyzing the graph and outputs subgraphs and DFA

equations for each of these instructions. DFA equations are passed to SMT solver to analyze. In case the instruction can be attacked by bit flip(s), the attack method is recorded and the graph is updated to capture the result of this attack. Then TADA continues to find next vulnerable instruction. TADA stops either when the correct number of round keys is recovered as required by the user or when no more vulnerable instructions can be found.

We would like to point out that our static analysis method is sound, meaning that a fault attack found by TADA is provably exploitable, i.e., there are no false positives.

We present evaluation on implementations of four well-known block ciphers: SIMON and SPECK are ultra-lightweight algorithms published by NSA [6], AES is the current NIST standard [17], and PRIDE [3] is a lightweight cipher optimized for 8-bit microcontrollers. For SIMON, SPECK and PRIDE, we were able to find novel DFA attacks that are fully implementation specific and provide practical examples of importance of our methodology. In case of AES, we were able to find an equivalent attack that was presented in the literature on the cipher level. Thanks to TADA, we could identify specific instructions that make these attacks possible, which is the highest level of detail that can be provided for an attack on software implementation.

Moreover, we develop a novel attack on multiplication used in block cipher implementations, revealing a vulnerability of such operation. Multiplications with a constant are normally used for more efficient bit shifting, leading to saving a couple of clock cycles. However, thanks to non-linearity of multiplication, it opens a new attack vector that can be exploited by DFA. Such vulnerability can be easily revealed with TADA analysis.

Execution times for finding attacks on full ciphers fall within reasonable range, considering that the analysis is complete and does not require any human intervention. Lightweight ciphers vary within the range of minutes – the fastest analysis was on PRIDE (4.6 minutes), the slowest was on SIMON (17.2 minutes). Larger ciphers fall within the range of hours, where the analysis of AES needed less than 5 hours.

Organization. The rest of the paper is structured as follows. Section 2 provides preliminaries on symmetric block ciphers, DFA and SMT solvers. Section 3 presents the design and describes the usage of TADA. Section 4 shows experimental evaluations on SIMON, SPECK, AES and PRIDE. Section 5 introduces related work. Section 6 provides the discussion. Finally, Section 7 concludes this work.

2 BACKGROUND

2.1 Symmetric Block Cipher

A symmetric block cipher is an algorithm operating on blocks of data of a pre-defined size. It specifies two processes, encryption and decryption. The encryption takes a plaintext and a secret key as inputs and produces a ciphertext as an output. Similarly, for a ciphertext and a secret key as inputs for decryption, it outputs the plaintext. In the rest of this work, we focus on the encryption to simplify the explanations. However, the proposed method would work the same way on the decryption operation. Block cipher normally consists of several rounds where each round consists of a small number of operations. Those operations scramble the input by using various transformations and adding key-dependent data. A key used in a round is referred to as *round key*. Round keys are derived from the secret key, which is called master key, by a key scheduling algorithm that works as an invertible transformation. Therefore, by getting information about a certain round key, it is possible to get the master key by using an inverse key scheduling algorithm. This is important in context of DFA that normally tries to recover the last round key.

2.2 Differential Fault Analysis

When performing DFA, the attacker first obtains a correct ciphertext, by running the encryption without any disturbance. Then, she runs the algorithm again with the same input values (plaintext, secret key), while injecting a fault into a certain round of the cipher, obtaining a faulty ciphertext. Later, she compares these two ciphertexts and if the attack was successful, she gets an information about the secret key.

As an example, let us consider the operation that takes a and b as inputs and outputs the result $c = a \,\&\, b$, where $a, b \in \{0, 1\}$ and $\&$ is the bitwise AND operator. We assume the output is known to the attacker but values of a, b are unknown. The attacker can then inject fault in b by flipping it to find the value of a: the first three columns of Table 1 show the case when b is flipped. If the output c stays the same, then $a = 0$; otherwise $a = 1$.

Now let us consider the same fault attack on the operation that takes input $a, b \in \{0, 1\}$ and outputs $d = a \oplus b$, where $\oplus$ is bitwise XOR operator. In this case the attack will not work: columns 1,2,4 in Table 1 shows that whenever b is flipped, d will also be flipped.

Operations similar to $\oplus$ are said to be linear and those similar to $\&$ are said to be non-linear. Formally, we give the following definition.

Definition 2.1 (Linear Operation). Let F be an operation with the set of inputs denoted by F^i and the set of outputs denoted by F^o. Suppose the inputs and outputs of F are all binary strings of length n. We say F is *linear* if for any pair a, b ($a \in F^i$ is an input of F, $b \in F^o$ is an output of F), the following condition is always satisfied: for any set of fixed values of $F^i \backslash \{a\}$, $\forall x, y \in \{0, 1\}^n$ such that when $a = x, b = y$, we have if $a = x \oplus \Delta$, then $b = y \oplus \Delta$, $\forall \Delta \in \{0, 1\}^n$, where $\oplus$ is the bitwise XOR operator.

DFA exploits non-linear functions of the cipher. It often works with just a single faulty and correct ciphertext pair from the encryption [47]. The attacker makes use of two sets of equations: one set that describes the correct execution of the encryption; one set corresponds to the faulted encryption process. For a simple example, let us consider the program that takes two binary inputs $a, b \in \{0, 1\}$, calculates $c = a \,\&\, b$ and outputs c. The equation corresponding to the correct execution is $c = a \,\&\, b$. If a fault is injected in b such that b is flipped, the equation corresponding to the faulted execution would be $c' = a \,\&\, b'$, where $b' = b \oplus \delta$ and $\delta = 1$. By calculating the difference of the output $\Delta = c \oplus c'$, the attacker can get the value in a: $a = 0$ if $\Delta = 0$; $a = 1$ if $\Delta = 1$. The two sets of equations corresponding to correct and faulted executions are referred to as *DFA equations*. The change in b, denoted by δ, which is equal to 1 in this case, is called the *fault mask*. The output difference Δ is called the *output mask*.

DFA is usually executed at the final rounds of the cipher, so that there are not too many collisions of the altered values. Otherwise,

it would make the analysis too complex. The most straightforward approaches inject a fault into the last round, usually requiring at least as many faults as the number of non-linear operations in the round. More sophisticated approaches attack 2-3 rounds before the encryption ends, utilizing the permutation layer that distributes the fault into the whole state. Such techniques require lower number of faults, but the number of equations to solve is higher.

In our approach, we first consider fault injections in the last round in order to recover the last round key. If an attack on the last round can be found, it then depends on user decision whether the attack is carried out further on earlier rounds.

2.3 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) [18] is concerned with deciding the satisfiability of first order formulas w.r.t. background theories, e.g. the theory of linear arithmetic over integers, of bit-vectors, of arrays, and so on. Decision procedures for solving SMT problems are called SMT solvers. In program analysis and verification, many problems can be naturally reduced to SMT problems and SMT solvers have been used as back-end engines in many tools for software testing, analysis and verification. The SMT solver we use in TADA is Z3 [19].

Consider the case of attacking $c = a \,\&\, b$ in the above mentioned DFA. Let ψ denote the following formula, which specifies the DFA equations as well as the equations for fault mask and output mask:

$$(c = a \,\&\, b) \wedge (c' = a \,\&\, b') \wedge (b' = b \oplus \delta) \wedge (\Delta = c \oplus c').$$

To find a fault attack for $c = a \,\&\, b$ amounts to finding a mapping between the value of Δ and a. To this end, we use an SMT solver to check the satisfiability of the following two formulas, where V denotes the set of variables in ψ:

1) $\forall V \backslash \{\delta\} : ((\psi \wedge \Delta = 1) \Rightarrow a = 1) \wedge ((\psi \wedge \Delta = 0) \Rightarrow a = 0)$
2) $\forall V \backslash \{\delta\} : ((\psi \wedge \Delta = 1) \Rightarrow a = 0) \wedge ((\psi \wedge \Delta = 0) \Rightarrow a = 1)$

Notice that δ is the only free variable in both formulas. We explain the first formula briefly. Since δ is the only free variable in formula 1), checking the satisfiability of formula 1) amounts to ask whether we can find a value for the fault mask δ such that it is always the case that $a = 1$ if $\Delta = 1$ and $a = 0$ if $\Delta = 0$. By calling an SMT solver, we know that formula 1) is satisfiable because the formula evaluates to true when $\delta = 1$. Therefore, we can perform DFA by using $\delta = 1$. This result is consistent with the fault analysis given in the above section. On the other hand, formula 2) is unsatisfiable.

3 TADA METHODOLOGY

In this section we present the methodology that was used when implementing TADA. Section 3.1 describes the fault models we consider. Section 3.2 details attacks on target instructions. Requirements on assembly code are stated in Section 3.3. Section 3.4 provides the design overview of TADA and details the automated analysis steps. Finally, Section 3.5 explains the analysis carried out by the SMT solver module.

3.1 Fault Models

An assembly implementation of a cryptographic algorithm, say $\mathcal{F}$, is a finite sequence of instructions. An instruction f consists of four parts: sequence number, mnemonic, the set of input operands and the set of output operands. Sequence number is the index of f as an element of $\mathcal{F}$. The mnemonic of f is the operation that f uses. We say an instruction f is *linear* if its mnemonic is a linear function by Definition 2.1.

A single fault attack on an assembly implementation $\mathcal{F}$ can be modeled by a fault injected in one of the instructions in $\mathcal{F}$ such that it either affects the mnemonic/input operands/output operands of this instruction or it deletes this instruction from the sequence (instruction skip) [40].

We are interested in *single fault adversarial* model, meaning that the attacker can inject exactly one fault during the execution of an assembly implementation of the algorithm at a time. However, she can repeat the execution as many times as she wants, with different faults. We consider *bit flip* fault model, therefore for a register length n, there are n possibilities of flipping a bit. A bit flip changes one bit in one of the input/output operands of an instruction in an assembly implementation $\mathcal{F}$. The change is referred to as a *fault mask*, which can be chosen by the attacker. Bit flip model is the most precise fault model for DFA[1] and it is usually the model of choice when attacking cryptographic algorithms with binary non-linear operations (e.g. addition-rotation-xor based ciphers). This model was previously shown to be practically achievable either by laser fault injection [2] or Rowhammer attack [10].

As most DFA attacks, we assume known ciphertext attack without the knowledge of the plaintext.

3.2 Attacks on Target Instructions

In general, DFA aims at attacking non-linear instructions. Up to now, there have been various attacks exploiting the following operations: bitwise AND [48], bitwise OR [15], addition [7, 29, 48], and table lookup [31, 43, 47], often used for Sbox calculation. Even though the attack varies for different ciphers, the main principle behind the attack of a particular operation stays the same [32].

In our work, we focus on these operations and moreover, we present a novel attack on multiplication with a constant. To the best of our knowledge, this is the first attack on multiplication used in a cryptographic implementation. Multiplications are not used in symmetric block cipher designs, however they can be efficient for performing logical bit shifts. For example, in the implementation of SPECK that we analyzed, shift by 3 bits to the left was done by multiplication with a constant value of `0x08`.

The analysis module of TADA focuses on instructions that implement the aforementioned operations. Here, we explain the generic idea for attacking each of the operations.

Bitwise AND, bitwise OR. The attack on bitwise AND operator follows the description in Section 2.2 (c.f. Table 1 and corresponding discussions).

The attack on bitwise OR is similar. Suppose we have a program that takes two binary inputs $a, b \in \{0, 1\}$, calculates $c = a \mid b$, and outputs c. The relations between a, b, c are as follows:

a	b	$c = a \mid b$
0	0	0
0	1	1

a	b	$c = a \mid b$
1	0	1
1	1	1

[1] Bit sets/resets, although being more precise than bit flips, are used for other methods, such as ineffective fault analysis and are out of scope of DFA.

We inject a fault δ in b and we have the following equations:

DFA equations		Fault mask	Output mask
$c = a \mid b$	$c' = a \mid b'$	$b' = b \oplus \delta$	$\Delta = c \oplus c'$

Take $\delta = 1$, the value of a can be obtained from the value of output mask Δ:

$$\Delta = 1 \Longrightarrow a = 0; \quad \Delta = 0 \Longrightarrow a = 1. \tag{1}$$

Note that if we let $\texttt{out} = 1, \texttt{var}_0 = 1, \texttt{var}_1 = 0$, then Δ & $\texttt{out} = \texttt{var}_0$ or $\texttt{var}_1$, and the following is equivalent to equation (1):

$$\Delta \,\&\, \texttt{out} = \texttt{var}_0 \Longrightarrow a = 0; \;\; \Delta \,\&\, \texttt{out} = \texttt{var}_1 \Longrightarrow a = 1.$$

Addition. For addition, we have

a	b	$c = a + b$
0	0	00
0	1	01

a	b	$c = a + b$
1	0	01
1	1	10

We inject a fault δ in b and we have the following equations:

DFA equations		Fault mask	Output mask
$c = a + b$	$c' = a + b'$	$b' = b \oplus \delta$	$\Delta = c \oplus c'$

Take $\delta = 1$, the value of a can be obtained from the value of output mask Δ: if $\Delta = 01$ then $a = 0$ and if $\Delta = 11$ then $a = 1$. Equivalently, let $\texttt{out} = 11, \texttt{var}_0 = 01, \texttt{var}_1 = 11$, then Δ & out is either $\texttt{var}_0$ or $\texttt{var}_1$, and Δ & $\texttt{out} = \texttt{var}_0 \Longrightarrow a = 0$; Δ & $\texttt{out} = \texttt{var}_1 \Longrightarrow a = 1$.

Addition with carry. In case there is a carry bit for addition calculation, DFA needs to take the value of the carry bit into consideration. We look at the program that takes three inputs $a, b, \texttt{carry} \in \{0, 1\}$, calculates $c = a + b + \texttt{carry}$ and outputs c in binary format. We have:

a	b	carry	c
0	0	0	000
0	1	0	001
1	0	0	001
1	1	0	010

a	b	carry	c
0	0	1	001
0	1	1	010
1	0	1	010
1	1	1	011

We inject a fault δ in b and we have the following equations:

DFA equations		Fault mask	Output mask
$c = a + b + \texttt{carry}$	$c' = a + b' + \texttt{carry}$	$b' = b \oplus \delta$	$\Delta = c \oplus c'$

For $\texttt{carry}= 0$, $\Delta = 001 \Longrightarrow a = 0$ and $\Delta = 011 \Longrightarrow a = 1$. For $\texttt{carry}= 1$, $\Delta = 011 \Longrightarrow a = 0$ and $\Delta = 001 \Longrightarrow a = 1$. Let $\texttt{out} = 011, \texttt{var}_0 = 001, \texttt{var}_1 = 011$ if $\texttt{carry}= 0$ and let $\texttt{out} = 011, \texttt{var}_0 = 011, \texttt{var}_1 = 001$ if $\texttt{carry}= 1$, then Δ & $\texttt{out} = \texttt{var}_0$ or $\texttt{var}_1$ and Δ & $\texttt{out} = \texttt{var}_0 \Longrightarrow a = 0$; Δ & $\texttt{out} = \texttt{var}_1 \Longrightarrow a = 1$. Note that the choices of value for variables out, $\texttt{var}_0, \texttt{var}_1$ are not unique, taking $\texttt{out} = 111, \texttt{var}_0 = 001, \texttt{var}_1 = 011$ for $\texttt{carry}= 0$ and let $\texttt{out} = 010, \texttt{var}_0 = 010, \texttt{var}_1 = 000$ for $\texttt{carry}= 1$ also gives the same result. Furthermore, we can see that for DFA, it is necessary to consider both values of the carry bit.

Table lookup for Sbox. Sbox (substitution-box) is a basic nonlinear component in cipher designs, mostly used in SPN (Substitution-Permutation Network) ciphers. Sbox is responsible for the confusion property in encryption modules defined by Shannon [46]. It is a permutation function on integers with values $0, 1, 2, \ldots, 2^n - 1$, where n is referred to as the number of bits of the Sbox. An Sbox can be described as an array: for example, $\{1, a, 0, 2, 3, 4, 5, 6, e, b, c, 8, 7, 9, d, f\}$ is a 4–bit Sbox such that $\text{Sbox}(0) = 1, \text{Sbox}(1) = a \ldots, \text{Sbox}(f) = f$ (integers are in hexadecimal format). It can either be implemented as a lookup table in the memory or in Algebraic Normal Form (ANF) that can be calculated as a series of arithmetic and logic operations. We provide analysis of both - lookup Sbox in case of AES implementation and algebraic Sbox in case of PRIDE. For any Sbox, there is an associated *difference distribution table* (DDT) [11], where the (Δ, δ)–entry consists of the values x such that $\text{Sbox}(x) \oplus \text{Sbox}(x') = \Delta$, where $x' = x \oplus \delta$. As we only consider bit flip fault model, for 4–bit Sboxes, the fault mask δ takes only 4 values: 1, 2, 4, 8. The DDT for the above mentioned 4–bit Sbox with only bit flip fault masks is as follows:

$\Delta \backslash \delta$	1	2	4	8
1		0 2	a e	1 9
2	2 3 e f	5 7 8 a	0 4 9 d	
3	6 7	9 b		
4	a b		3 7	4 c
5	8 9		2 6	
6		4 6 d f		
7	4 5		b f	
8		1 3		6 e
9			8 c	7 f
a		c e		3 b
b	0 1			
c				2 a
d				5 d
e	c d		1 5	
f				0 8

By observing the output mask and fault mask pairs, the attacker can get the value of the input. For example, if the 4 fault mask and output mask pairs are $(1, b), (2, 2), (4, 2), (8, f)$ then the input is uniquely identified to be 0.

Multiplication with a constant. Consider a program that takes input $a \in \{0, 1, 2, 3, 4\}$ and outputs the product, denoted by c, of a with constant 2. The strategy is to inject fault in the constant operand and get value of a.

DFA equations		Fault mask	Output mask
$c = a \times 2$	$c' = a \times \texttt{const}$	$\texttt{const} = 2 \oplus \delta$	$\Delta = c \oplus c'$

When a bit flip fault is injected in 2, we get either 0 or 3. Representing the integers in binary format, we have:

a	$c = a \times 2$
00	0000
01	0010
10	0100
11	0110

a	$c' = a \times 0$
00	0000
01	0000
10	0000
11	0000

a	$c' = a \times 3$
00	0000
01	0011
10	0110
11	1001

We can see that if the fault mask $\delta = 2$, then the output mask $\Delta = c \oplus c' = a$. Similarly, if fault mask $\delta = 1$, a can also be identified by value Δ.

In real DFA attacks, the output value of a vulnerable instruction normally cannot be observed directly, but the output mask propagates to the ciphertext and can be analyzed. However, in assembly implementations, it is not easy to track which register value gives the information of the output mask. TADA constructs a customized data flow graph to capture the propagation of the fault, then it utilizes SMT solver to prove whether the above techniques can be applied to the vulnerable instructions.

Table 2: Assembly implementation $\mathcal{F}_{ex}$ for example cipher.

#	Instruction	#	Instruction	#	Instruction
0	LD r0 X+	3	LD r3 key1+	6	EOR r1 r3
1	LD r1 X+	4	AND r0 r1	7	ST x+ r0
2	LD r2 key1+	5	EOR r0 r2	8	ST x+ r1

3.3 TADA Usage

There are few requirements regarding the assembly implementations that have to be addressed before the analysis, detailed below.

We do not assume any annotations in the assembly code, however, certain naming conventions are required for important variables so that TADA could identify them correctly. More specifically, round keys have to be identified by the word *"key"* followed by the round number. Ciphertext variables then have to be identified by a small letter *"x"*.

The analyzed implementations are unrolled – without loops and jumps. We discuss this requirement more in Section 6.

The parsing subsystem is currently capable of reading assembly files written for AVR ATmega microcontrollers[2]. However, the analysis is done on an intermediate representation and therefore, after creating a new parsing module, TADA can be reused on any other instruction set (e.g. Thumb-2 or LLVM).

3.4 TADA Design

Implementation. TADA was implemented in Java (static analysis part) and F# (Z3 SMT solver part) programming languages; each of the following steps corresponds to one module.

Customized data flow graph. TADA constructs a customized data flow graph in a static single assignment form from an assembly implementation. The data flow graph represents the instructions as edges and it takes input and output operands of the instructions as nodes. Each node in the data flow graph corresponds to one unit of data storage in the architecture. We refer to the nodes that correspond to registers storing round key values as *key nodes*. Similarly, nodes that represent ciphertext words are called *ciphertext nodes*. For an instruction f, the nodes corresponding to its input operands are called the *input nodes* of f and the nodes corresponding to its output operands are called the *output nodes* of f. Both input and output nodes are referred to as *nodes* of f. If a is an output node of f, we say f *generates* a.

Example 3.1. Let us consider a toy block cipher implemented in AVR assembly, stated in Table 2. The example cipher has one round. It takes a 16-bit plaintext input. The first 8 bits are XORed with an 8-bit key word and give the first 8 bits of the ciphertext. Bitwise AND operation is applied on the two parts of the plaintext, then the result is XORed with another 8-bit key to give the last 8 bits of the ciphertext. The customized data flow graph generated by TADA for this example cipher implementation is given in Figure 2. As the registers have 8 bits, the 16–bit ciphertext is stored in two ciphertext words. Nodes `r3(3)` and `r2(2)` are the key nodes; `x(8)` and `x(7)` are the ciphertext nodes. Instruction 4 has input nodes `r0(0)`,`r1(1)` and output node `r0(4)`. We say that instruction 4 generates node `r0(4)`.

[2]https://www.microchip.com/webdoc/avrassembler/avrassembler.wb_instruction_list.html

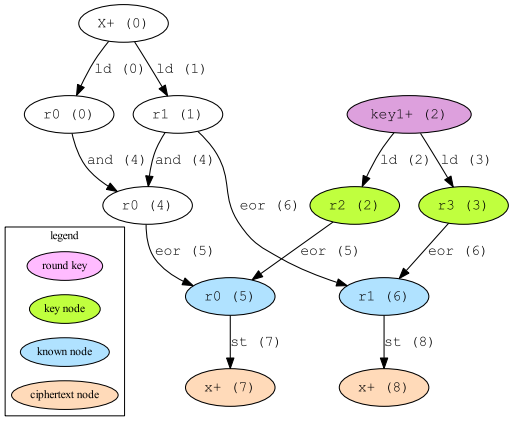

Figure 2: Data flow graph generated by TADA for example cipher implementation $\mathcal{F}_{ex}$ from Table 2.

Known nodes and constants. Before further analysis, TADA does a pre-examination of the nodes to find the *known nodes*. Since we assume the attacker knows the ciphertext, the ciphertext nodes are marked as known nodes. Tracing up from the graph, some nodes can also be easily identified as known nodes. Moreover, a node that represents a constant is marked as both a known node and a constant, the value of the constant is also stored.

Example 3.2. In Figure 2, the value of `r0(5)` is equal to that of `x(7)` because they are respectively the input node and output node of a store instruction, hence `r0(5)` is marked as a known node. Similarly, `r1(6)` is also a known node.

Data propagation is captured within the graph. Each edge has a property that says whether it is linear or non-linear according to the instruction it represents. A node x *affects* a node y if the following two conditions are satisfied: the sequence number of the instruction that generates y is bigger than the sequence number of the instruction that generates x; furthermore, changing the value of x would influence the value of y (the second condition is equivalent as saying y is a child of x in the directed graph). The number of non-linear operations between a node x and each node y affected by x is recorded as *distance* between them. A node x and a key node y are *linearly related* to each other if x does not affect y and there is another node z such that both x and y affect z with distance 0. A node x is *linearly related* to a round key if it is linearly related to a key node of this round key.

Example 3.3. In Figure 2, `r1(1)` affects `r0(5)` with distance 1; it also affects `x(8)` with distance 0. `r1(1)` is linearly related to key node `r3(3)` and `r0(4)` is linearly related to key node `r2(2)`.

Vulnerable instruction. The goal of an attack on cryptographic implementation is normally the recovery of the master key. For some ciphers, the recovery of the last round key is sufficient (e.g. AES). For other ciphers, the attacker needs more than one round key to get the master key. For example, for SPECK, SIMON, and PRIDE, the last round key and the second last round key are both needed to get the master key. In view of this, we allow a user input, *Number of target round keys*, which indicates how many round keys

Algorithm 1: Check if an instruction is vulnerable.

Input : f: an instruction of a program $\mathcal{F}$, DFG: customized data flow graph corresponding to $\mathcal{F}$, key: target round key
Output: boolean: is f vulnerable?

```
if Mnemonic of f ∈ {AND, OR, ADD, ADC, LPM, MUL} then
    if Mnemonic of f =MUL then
        boolean vul = false;
        for a: input nodes of f do
            if a is a constant then
                vul=true;
        if vul=false then
            return false;
    for a: output nodes of f do
        for x: known nodes affected by a do
            if distance(a, x) > 0 then
                return false;
            for b: nodes of f do
                if b is linearly related to key then
                    return true;
return false;
```

are supposed to be retrieved, counting from the last round key. Thus if *Number of target round keys* = 1, TADA would only aim for the recovery of the last round key. If *Number of target round keys* = 2, TADA would work on the attack to obtain the keys from last two rounds. During the execution, the round key which is under analysis is referred to as the *target round key*. An instruction is considered vulnerable by TADA if the following conditions are satisfied:

1. The instruction is one of the operations as described in Section 3.2. In AVR assembly, these include operations with mnemonics `AND`, `OR`, `ADD`, `ADC`, `LPM`, `MUL`, which are respectively bitwise AND, bitwise OR, addition, addition with carry, table lookup and multiplication. For multiplication, we further check if one of the input nodes is a constant.
2. For each output node of the instruction, the distance from it and each of its affected known nodes is = 0. Thus, there is only one non-linear instruction between the input nodes of this instruction and the known nodes, which is the instruction under analysis. This ensures that there is only one non-linear equation to solve. Furthermore, this enables us to derive the SMT constraints based on the generic attacking method described in Section 3.2.
3. At least one of its nodes is linearly related to a key node that stores the value of the *target round key*, which is the round key under analysis during the execution.

The algorithm for checking if an instruction is vulnerable is outlined in Algorithm 1.

Remark 1. *As explained in the discussion before Definition 2.1, when a fault is injected in a linear instruction, the output mask does not give information of the inputs as it is always equal to the fault mask. Similarly, if we have a series of linear instructions before a non-linear instruction, injecting a fault in one of the linear instructions is equivalent to injecting a fault in the non-linear instruction. That is why we put our focus on non-linear instructions only.*

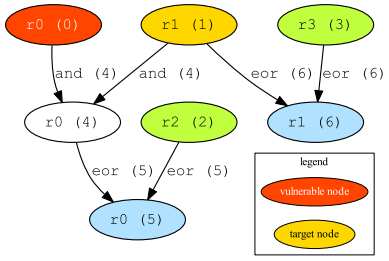

Figure 3: Subgraph generated by TADA for target node `r1(1)` and vulnerable node `r0(0)` from vulnerable instruction 4 of example cipher $\mathcal{F}_{ex}$ in Table 2.

Table 3: DFA Equations generated for subgraph in Figure 3.

Correct execution	Faulted execution	Fault mask
(a) $r0(4) = r0(0)$ & $r1(1)$	(d) $r0(4)' = r0(0)'$ & $r1(1)$	$r0(0)' = r0(0) \oplus \delta$
(b) $r0(5) = r0(4) \oplus r2(2)$	(e) $r0(5)' = r0(4)' \oplus r2(2)$	
(c) $r1(6) = r1(1) \oplus r3(3)$		

Target node and vulnerable node. For a vulnerable instruction, each of its input nodes that is not known can be a *target node*. Each of the input nodes can be a *vulnerable node* (which can be the same as the target node). Recall that by selection of vulnerable instructions, at least one of the nodes of the instruction is linearly related to the target round key nodes. A DFA on the vulnerable instruction injects fault in one of the vulnerable nodes, hoping to get information about the target node, hence revealing information about the linearly related key nodes.

Subgraphs and DFA equations generation. For each pair of target node and vulnerable node, TADA extracts a subgraph of the full data flow graph that includes the vulnerable instruction and the nodes affected by it. The subgraph stops at the known nodes.

Example 3.4. For example cipher from Table 2, one of the vulnerable instructions found by TADA is instruction 4. Figure 3 shows the subgraph for target node `r1(1)` and vulnerable node `r0(0)`.

For each subgraph (i.e. each pair of target node and vulnerable node), TADA constructs one set of DFA equations and one equation for fault mask. The DFA equations describe the relation from the vulnerable instruction until the known nodes. The equation for fault mask indicates that the change in the vulnerable node is equal to δ. Input to SMT solver module also indicates which variables involved in the DFA equations represent known nodes.

Example 3.5. Equations (in the human readable form) generated by TADA for subgraph in Figure 3 are given in Table 3 (color scheme corresponds to Figures 2,3). The real format of the equations is in the form of SMT solver module input. The list of known nodes, which is `{r0(5),r1(6)}` is also passed to SMT solver module.

SMT solver and graph update. For each pair of vulnerable node and target node, the SMT solver module of TADA designs constraints to describe the corresponding DFA attack and calls SMT solver to output the attack details in case the attack is successful.

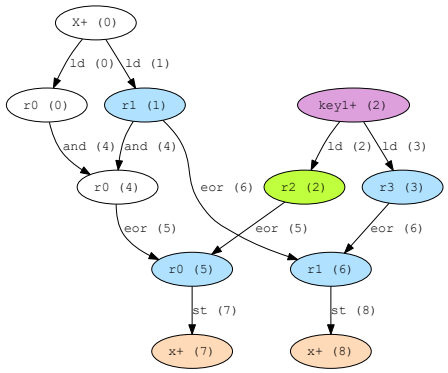

Figure 4: Updated graph generated by TADA after a successful attack on target node `r1(0)` from vulnerable instruction 4 of example cipher $\mathcal{F}_{ex}$ in Table 2.

The details of this module are presented in Section 3.5. After each successful attack on a target node, TADA updates the known nodes in the graph.

Example 3.6. The attack on vulnerable instruction 4 with target node `r1(1)` and vulnerable node `r0(0)` as described in Example 3.5 gives 8 bits of `r1(1)`. Since `r1(6)` is a known node, TADA updates the key node `r3(3)` as known node. The updated graph is shown in Figure 4. Furthermore, for the same vulnerable instruction with target node `r0(0)` and vulnerable node `r1(1)`, TADA recovers 8 bits of `r1(1)`. At this point, both of the key bytes are retrieved and the cipher is broken. (The final graph is shown in Appendix A Figure 9).

In case the target node cannot be obtained from attacking one vulnerable node, TADA tries to obtain the same target node with a different vulnerable node. The analysis of one instruction stops when either all the input nodes are retrieved or when there is no more target node which can be obtained. The details are explained in Algorithm 2. Note that SMT solver module stores the attack details in separate files even if the attack is not successful (i.e. less than 8 bits are retrieved). TADA will only retrieve the corresponding file and output the attack details if SMT solver module returns `true`, which indicates a successful attack (lines 5-7 of Algorithm 2).

After the attack on one vulnerable instruction is finished, TADA analyzes the new graph to find another vulnerable instruction. TADA stops when the required number of round keys is found or when there is no vulnerable instruction that can be attacked.

Remark 2. *In practical DFA, the attack is also considered successful if not all of the bits of the key can be recovered but the brute force complexity of recovering the key is acceptable. Taking this into consideration, TADA allows the user to specify the least number of bits that need to be recovered to consider an attack as successful. In case the number is less than* 8*, TADA records the number of bits missing and outputs the total brute force complexity in the end (see Remark 4).*

Algorithm 2: The analysis of one vulnerable instruction

```
   Input  : f: a vulnerable instruction.
   Output : boolean variable exploit.
1  boolean exploit = false;
2  for a: input nodes of f do
3      if a is known then
4          continue;
5      boolean target = false;
6      for b: input nodes of f do
7          run SMT solver module with inputs: DFA equations, fault
             mask equation, constraint on fault mask, the list of known
             nodes;
8          if SMT solver module returns true then
9              output the attack details;
10             update graph;
11             target = true;
12             exploit = true;
13             break;
14         if target then
15             break;
16 return exploit;
```

3.5 SMT Solver Module

As mentioned earlier, for each pair of target node and vulnerable node, the input of SMT solver module is the target node, vulnerable node, the corresponding DFA equations, equation for fault mask and a list of known nodes involved in the DFA equations (e.g. Example 3.5). In this section, we detail how other constraints and the satisfiability problems are designed for each of the operations described in Section 3.2.

Depending on the mnemonics of the operation, TADA executes different algorithms. Since we are considering 8−bit architecture and bit flip fault model, in case the mnemonic is not `LPM`, TADA generates the following constraint for the fault mask:

$$(\delta = 1) \vee (\delta = 2) \vee (\delta = 4) \vee (\delta = 8) \vee (\delta = 16) \vee (\delta = 32) \vee (\delta = 64) \vee (\delta = 128). \quad (2)$$

Bitwise AND, bitwise OR. The algorithm for attacking bitwise AND, and bitwise OR, is outlined in Algorithm 3. For each known node x, TADA generates an equation for output mask Δ (line 2). Then it tries to attack each bit of target node a (line 4). Line 5 specifies that for some variables `out`, $\texttt{var}_0$ and $\texttt{var}_1$, the output mask Δ has one of the two patterns: Δ & `out` = $\texttt{var}_0$ or Δ & `out` = $\texttt{var}_1$. Line 6 specifies if Δ is of the first pattern then the kth bit of the target node is 0. Line 7 says if Δ is of the second pattern, the kth bit of the target node is 1. Line 10 tests if there exist valuations to variables `out`, δ, $\texttt{var}_0$ and $\texttt{var}_1$ such that the above mentioned constraints are all satisfiable. In case it is satisfiable, the 6−tuple $(k, \delta, x, \texttt{out}, \texttt{var}_0, \texttt{var}_1)$ is saved to a file. This tuple translates to: the kth bit of the target node a can be obtained by attacking the vulnerable node using fault mask δ and observing the output mask Δ of the known node x: if Δ & `out` = $\texttt{var}_0$ then $a[k] = 0$ and if Δ & `out` = $\texttt{var}_1$ then $a[k] = 1$.

Algorithm 3: The algorithm for attacking bitwise AND, bitwise OR.

Input : ψ: DFA equations and the equation for fault mask; b: vulnerable node; a: target node; S: the list of knowns nodes in the DFA equations; ψ_δ: constraint for fault mask δ as in Equation (2).
Output : boolean: `true` if a can be obtained by attacking b.

1 **for** x: S **do**
2 $\phi := \psi \wedge (\Delta = x \oplus x')$;
3 counter = 0;
4 **for** k = 0, 1, . . . , 7 **do**
5 $\psi_1 := \phi \Rightarrow ((\Delta$ & out = $\mathtt{val}_0) \vee (\Delta$ & out = $\mathtt{val}_1))$;
6 $\psi_2 := (\phi \wedge \Delta$ & out = $\mathtt{val}_0) \Rightarrow a[k] = 0$;
7 $\psi_3 := (\phi \wedge \Delta$ & out = $\mathtt{val}_1) \Rightarrow a[k] = 1$;
8 $\Phi := \psi_\delta \wedge \psi_1 \wedge \psi_2 \wedge \psi_3$;
9 V := all variables involved in Φ;
10 **if** *($\forall V \backslash \{$out, δ, $\mathtt{val}_0$, $\mathtt{val}_1\} : \Phi$) is satisfiable* **then**
11 save to file (k, δ, x, out, $\mathtt{val}_0$, $\mathtt{val}_1$);
12 counter++;
13 **if** *counter = 8* **then**
14 **return** `true`;
15 **return** `false`;

If, for one known node x, 8 bits of the target node can be all retrieved, then it returns `true` (line 13 − 14). Otherwise it goes to the next known node.

As indicated in Algorithm 2, the attack details will be retrieved from the files and output only when the attack is successful. The output from TADA for example cipher $\mathcal{F}_{ex}$ in Table 2 is summarized in Appendix A Table 6.

Remark 3. *The files output by TADA indicate that for target node* `r1(1)` *and vulnerable node* `r0(0)`*, when the known node* `r1(6)` *is considered, there is no attack (this can also be observed from Figure 2). This is because the injected fault does not affect value in* `r1(6)`*. Thus, for each pair of target and vulnerable node, we need to iterate through all the known nodes that are involved in the DFA equations. Only when none of the known nodes can help us to find an attack, we consider the attack fails.*

Addition (with carry). The algorithm for attacking addition is outlined in Algorithm 4. The sum of two 8−bit variables is stored in a variable of 8 bits and a carry bit. Thus, instead of considering the output mask of only one known node, we consider each pair of known nodes (line 1). Here || indicates concatenation. For example $10||10 = 1010$. We first attack the 0th bit of target node, which does not involve carry bit value. If this bit can be retrieved, the tuple $(0, \delta, x, y, \mathtt{out}, \mathtt{var}_0, \mathtt{var}_1)$ is saved to a file. This corresponds to: the 0th bit of the target node a can be obtained by injecting fault mask δ in vulnerable node and observing the output mask $\Delta = y||x \oplus y'||x'$. If Δ & out = $\mathtt{var}_0$, $a[0] = 0$ and if Δ & out = $\mathtt{var}_1$, $a[0] = 1$. Similar to the discussion of the attack on addition with carry in Section 3.2, for higher bits, we need to consider two cases separately: the carry bit from the previous bits is 0 or 1 (line 16, 28). Here $a[k-1, 0]$ denotes the integer that is the same as the first $k-1$ bits of a. For example $0110[2, 0] = 110$. If attack for carry bit = 0 is successful, the tuple $(k, 0, \delta, x, y, \mathtt{out}, \mathtt{var}_0, \mathtt{var}_1)$ is saved to

Algorithm 4: The algorithm for attacking `ADD`.

Input : ψ: DFA equations and the equation for fault mask; b: vulnerable node; a: target node; S: the list of knowns nodes in the DFA equations; ψ_δ: constraint for fault mask δ as in Equation (2).
Output : boolean: `true` if a can be obtained by attacking b.

1 **for** $x, y \in S, x \neq y$ **do**
2 counter= 0;
3 $\phi := \psi \wedge (\Delta = y||x \oplus y'||x')$;
4 $\psi_1 := \phi \Rightarrow ((\Delta$ & out = $\mathtt{val}_0) \vee (\Delta$ & out = $\mathtt{val}_1))$;
5 $\psi_2 := (\phi \wedge \Delta$ & out = $\mathtt{val}_0) \Rightarrow a[0] = 0$;
6 $\psi_3 := (\phi \wedge \Delta$ & out = $\mathtt{val}_1) \Rightarrow a[0] = 1$;
7 $\Phi := \psi_\delta \wedge \psi_1 \wedge \psi_2 \wedge \psi_3$;
8 V := all variables involved in Φ;
9 **if** *($\forall V \backslash \{$out, δ, $\mathtt{val}_0$, $\mathtt{val}_1\} : \Phi$) is satisfiable* **then**
10 save to file (0, δ, x, y, out, $\mathtt{val}_0$, $\mathtt{val}_1$);
11 counter++;
12 **else**
13 continue;
14 **for** k = 1, 2, . . . , 7 **do**
15 //the carry from the first k bits = 0:;
16 $\psi_{c0} := (a[k-1, 0] + b[k-1, 0])[k] = 0$;
17 $\phi := \psi_{c0} \wedge \psi \wedge (\Delta = y||x \oplus y'||x')$;
18 $\psi_1 := (\phi \Rightarrow ((\Delta$ & out = $\mathtt{val}_0) \vee (\Delta$ & out = $\mathtt{val}_1)))$;
19 $\psi_2 := ((\phi \wedge \Delta$ & out = $\mathtt{val}_0) \Rightarrow a[k] = 0)$;
20 $\psi_3 := ((\phi \wedge \Delta$ & out = $\mathtt{val}_1) \Rightarrow a[k] = 1)$;
21 $\Phi_{c0} := \psi_\delta \wedge \psi_1 \wedge \psi_2 \wedge \psi_3$;
22 V := all variables involved in Φ_{c0};
23 **if** $\forall V \backslash \{$out, δ, $\mathtt{val}_0$, $\mathtt{val}_1\} : \Phi_{c0}$ *is satisfiable* **then**
24 let $output_0$ = (k, 0, δ, x, y, out, $\mathtt{val}_0$, $\mathtt{val}_1$);
25 **else**
26 break;
27 //the carry from the first k bits = 1:;
28 $\psi_{c1} := (a[k-1, 0] + b[k-1, 0])[k] = 1$;
29 $\phi := \psi_{c1} \wedge \psi \wedge (\Delta = y||x \oplus y'||x')$;
30 $\psi_1 := (\phi \Rightarrow ((\Delta$ & out = $\mathtt{val}_0) \vee (\Delta$ & out = $\mathtt{var}_1)))$;
31 $\psi_2 := ((\phi \wedge \Delta$ & out = $\mathtt{val}_0) \Rightarrow a[k] = 0)$;
32 $\psi_3 := ((\phi \wedge \Delta$ & out = $\mathtt{val}_1) \Rightarrow a[k] = 1)$;
33 $\Phi_{c1} := \psi_\delta \wedge \psi_1 \wedge \psi_2 \wedge \psi_3$;
34 V := all variables involved in Φ_{c1};
35 **if** $\forall V \backslash \{$out, δ, $\mathtt{val}_0$, $\mathtt{val}_1\} : \Phi_{c1}$ *is satisfiable* **then**
36 let $output_1$ = (k, 1, δ, x, y, out, $\mathtt{val}_0$, $\mathtt{val}_1$);
37 save to file: $output_0$ and $output_1$;
38 counter++;
39 **else**
40 break;
41 **if** *counter*= 8 **then**
42 **return** `true`;
43 **return** `false`;

a file, which means: when carry is 0, the kth bit of the target node a can be obtained by injecting fault mask δ in vulnerable node and observing the output mask Δ. If Δ & out = $\mathtt{var}_0$, $a[k] = 0$ and if Δ & out = $\mathtt{var}_1$, $a[k] = 1$. Similarly, when the attack for carry bit

= 1 is successful, the tuple $(k, 1, \delta, x, y, \texttt{out}, \texttt{var}_0, \texttt{var}_1)$ is saved to a file.

Note that the attack on the kth bit assumes the knowledge of the first $k-1$ bits of both of the operands. Thus, if one bit cannot be attacked, the algorithm goes to next known node directly (line 13, 26, 40). Moreover, the algorithm for attacking ADD contains an extra step in SMT solver module such that it only returns true for the attack when both inputs of addition can be retrieved or when one can be retrieved and the other is known.

The attack for ADC can be obtained by minor modifications of Algorithm 3. For example, the analysis of the 0th bit needs to consider two cases: the carry bit is 1 and the carry bit is 0. Furthermore, for attacking the addition with carry, it is necessary to require that the node representing carry is a known node.

Table lookup. If the vulnerable instruction corresponds to Sbox table lookup, the attack follows Algorithm 5. The algorithm aims to construct the 8 pairs $(\delta_1, \Delta_1), (\delta_2, \Delta_2), \ldots, (\delta_8, \Delta_8)$, where $\delta_i (1 \leq i \leq 8)$ are 8 different fault masks that satisfy the constraint given in Equation (3), and $\Delta_j (1 \leq j \leq 8)$ denote the corresponding output masks. First we identify the variables that change when input mask δ changes and store them in `list` (lines $1-4$). Next we make 8 copies of ϕ (line 9). They are identical to ϕ except for the variables in `list`, which are replaced by 8 different variables in each of the 8 copies. For example, δ is replaced by $\delta_1, \delta_2, \ldots, \delta_8$ in $\phi_1, \phi_2, \ldots, \phi_8$ respectively. If the attack on the kth bit is successful, (k, x) is saved to file (line $16-18$).

$$(\delta_1 = 1) \wedge (\delta_2 = 2) \wedge (\delta_3 = 4) \wedge (\delta_4 = 8) \wedge (\delta_5 = 16) \\ \wedge (\delta_6 = 32) \wedge (\delta_7 = 64) \wedge (\delta_8 = 128). \quad (3)$$

This pair (k, x) means that by flipping the bits of vulnerable node b and observing the change in the known node x, the eight pairs of input and output masks can uniquely identify the kth bit of a. If all 8 bits of the target node a can be obtained by attacking vulnerable node b, the algorithm returns `true` (line $19-20$).

Multiplication with a constant. When the vulnerable instruction is multiplication and one of the input operands is a constant, the algorithm for the attack is obtained from Algorithm 3 with the following changes:

Since the product of two 8−bit variables is stored in two 8−bit variables, we consider each pair of known nodes instead of only one known node. Lines $1-2$ are changed to

for $x, y \in S, x \neq y$ **do**
$\phi := \psi \wedge (\Delta = y||x \oplus y'||x')$

Accordingly, the output to a file (line 11) is changed to $(k, \delta, x, y, \texttt{out}, \texttt{var}_0, \texttt{var}_1)$, which indicates the kth bit of target node a can be obtained by injecting fault mask δ in vulnerable node b and observing the output mask $\Delta = y||x \oplus y'||x'$. If Δ & out = $\texttt{var}_0$, $a[k] = 0$ and if Δ & out = $\texttt{var}_1$, $a[k] = 1$.

4 EVALUATION

In this section, we will present evaluations of four ciphers using TADA: SIMON [6], SPECK [6], AES [17] and PRIDE [3]. Results are summarized in Table 4. More details are presented in Appendix C. The analysis was done on a standard laptop computer with Intel Haswell family CORE i7 and 8 GB RAM. For SIMON, SPECK and PRIDE, we were able to find implementation specific attacks that have not been presented yet, since it is not possible to identify such attack from the cipher-level. For AES, a previously published DFA [26] was found.

It is to be noted that while AES uses a full key length in each round, SIMON, SPECK and PRIDE only use half of it. Therefore, for a full key recovery, it is necessary to attack consecutive two rounds of these ciphers. In case of DFA, the first step is to recover the last round key, then peel-off this round, and continue with the attack on the penultimate round.

SIMON. SIMON is an ultra-lightweight block cipher, based on the balanced Feistel structure. It supports block sizes from 32 up to 128 bits, with key sizes ranging from 64 to 256 bits. Number of rounds depends on the key size, and ranges between 32 and 72. In each round, it uses three operations – bitwise AND, bitwise shift, and XOR. Schematic of SIMON is depicted in Figure 5.

Algorithm 5: The algorithm for attacking table lookup.

Input : ψ: DFA equations and the equation for fault mask; b: vulnerable node; a: target node; S: the list of knowns nodes in the DFA equations; ψ_δ: constraint for fault masks as in Equation (3).
Output: boolean: `true` if a can be obtained by attacking b.

```
1  list= {δ, Δ};
2  for c: variables in DFA do
3      if b affects c then
4          list.add(c);
5  for x in S do
6      counter = 0;
7      for k = 0, 1, . . . , 7 do
8          φ := ψ ∧ (Δ = x ⊕ x′);
9          make 8 copies of φ w.r.t. list;
10         let φ_i denote the ith copy of φ;
11         ψ := φ1 ∧ φ2 ∧ φ3 ∧ φ4 ∧ φ5 ∧ φ6 ∧ φ7 ∧ φ8;
12         V := all variables in ψ;
13         V′ := V\{δ1, ..., δ8, Δ1, ..., Δ8};
14         C0 := ∀V′ : ψ ⇒ a[k] = 0;
15         C1 := ∀V′ : ψ ⇒ a[k] = 1;
16         if (ψδ ∧ (∀{Δ1, ..., Δ8} : C0 ∨ C1)) is satisfiable then
17             counter++;
18             save to file (k, x);
19     if counter=8 then
20         return true;
21 return false;
```

Table 4: Evaluation by TADA on different implementations.

Cipher implementation	SIMON	SPECK	AES	PRIDE
# of lines of code (unrolled)	1,272	663	2,057	1590
# of nodes in DFG	1,595	843	2,060	1763
# of edges in DFG	2,709	1,562	3,209	2586
evaluation time (min)	17.2	9.8	298.7	4.6
fault attack found	new	new	[26]	new
# of known nodes before attack	66	32	69	16
# of known nodes after attack	162	117	149	196
# of round keys found	2	2	1	2

For SIMON implementation[3], TADA found 8 vulnerable bitwise AND instructions that are all exploitable. The last round key and the second last round key were recovered. This attack is implementation specific. The flaw in this implementation is that the key xor operation was implemented before the xor of left and right side of the intermediate values – opposite as the specification. Thus, presenting a DFA vulnerability which cannot be seen from the cipher design level.

SPECK. Similarly to SIMON, it is an ultra-lightweight block cipher. It offers the same block and key sizes, however the number of rounds ranges from 22 to 34. It follows ARX structure – each round consists of a modular addition, rotations, and XORs. Schematic of SPECK is depicted in Figure 6.

For SPECK implementation[3], TADA found 11 vulnerable instructions, among which 9 are exploitable. These 9 consist of 8 additions (with carry) and 1 multiplication. The other 2 vulnerable instructions are additions with carry which can only give 7 bits of the target nodes. Details are summarized in Table 5. Here "keyx[y]" denotes the $(y+1)$th byte of round key in round x.

The attack on multiplication is novel and implementation specific. This particular multiplication instruction (no. 595 in Table 5) multiplies a value with constant 8, which corresponds to 3–bit rotation to the left in the second last round of the cipher design. Attack details output by TADA suggests fault masks $0x10, 0x40$ for retrieving the 0th and 1st bit of target node and $0x08$ for retrieving the $2-7$th bits. If a fault mask $0x08$ is injected in the constant 8, the operation will be changed to multiplication by 0. We emphasize that such attack cannot be seen from a cipher design level, which only shows the rotations, but leaves it to implementer on how to realize them. Normally, rotation is a linear operation and therefore, cannot be attacked by DFA – the input and output difference would remain the same, it would only change the position, giving no information on the processed data. We note that multiplication by 8 is not the only way to implement 3–bit rotation. Only after the implementation analysis by TADA, one can observe the vulnerability caused by using the multiplication by a constant.

Remark 4. *If we consider the attack to be successful with only* 7 *bits recovered, the analysis time on SPECK is reduced to* 1.9 *minutes. In such case, TADA gave us the state-of-the-art attack published in [48]. It found* 8 *vulnerable addition (with carry) operations, recovered the last round key and the second last round key. But* 2 *bits of brute force is required. Details are outlined in Table 8.*

AES. AES is the current NIST standard for symmetric cryptography and therefore, widely used in real world applications. The block size of AES is 128 bits, while the key sizes can be chosen between 128, 192, and 256-bit variants. Number of rounds varies accordingly, and can be either 10, 12, or 14. It is based on substitution-permutation network structure (SPN) and consists of four operations per round: `AddRoundKey`, `SubBytes`, `MixColumns`, and `ShiftRows`. Schematic of AES is depicted in Figure 7 (picture was drawn with a usage of library from [30]).

For AES implementation taken from Ecrypt II public repository[4], TADA found the same attack as in [26]. The attack takes advantage

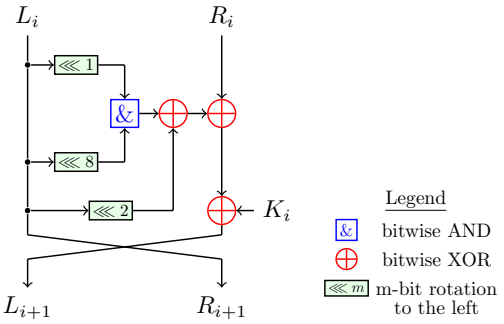

Figure 5: Schematic of one round of SIMON block cipher.

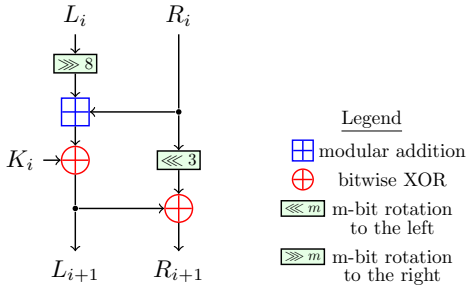

Figure 6: Schematic of one round of SPECK block cipher.

of `SubBytes` operation, implemented as a table lookup, which is the only non-linear part of the algorithm. With successful attacks on 16 table lookups, TADA recovered the last round key.

PRIDE. As another SPN representative, we have chosen lightweight cipher PRIDE. The block size is 64 bits, while the key size is 128 bits. Number of rounds is 20, where the first 19 rounds are identical and the last round ends with a substitution layer. In the implementation taken from public repository[5], the Sbox is implemented in algebraic form, therefore, unlike in AES implementation, no table lookup is necessary. Details on this implementation are shown in Appendix B. Schematic of PRIDE is depicted in Figure 8.

TADA found a new attack that exploits the bitwise `AND` operations, which are used for the implementation of Sbox (see Appendix B). 10 of such operations are analyzed and exploited, revealing the last two round keys.

We note that there are multiple countermeasure schemes proposed to thwart single fault injections in software [5, 37, 41, 42]. However, there is no full protected cipher implementation publicly available to date, only code snippets targeting single operations. Therefore, to mitigate the threat, we suggest the implementer to protect the cipher operations one by one while continuously checking the resulting code with TADA.

5 RELATED WORK

In this section we outline several works that present automated approaches to fault analysis with different focus.

[3]https://github.com/openluopworld/simon_speck_on_avr/tree/master/AVR
[4]https://perso.uclouvain.be/fstandae/source_codes/lightweight_ciphers/source/AES.asm
[5]https://github.com/FreeDisciplina/BlockCiphersOnAVR/tree/master/PRIDE_64_128_AVR

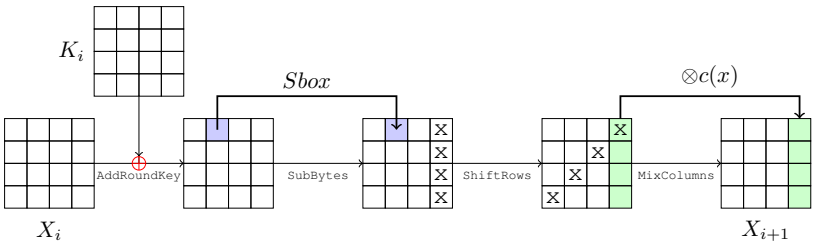

Figure 7: Schematic of one round of AES block cipher.

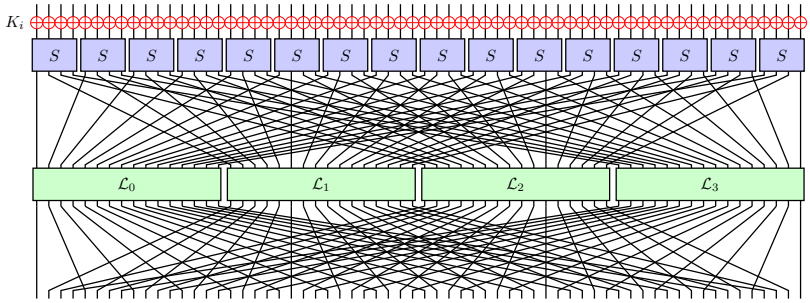

Figure 8: Schematic of one round of PRIDE block cipher.

No.	Mnemonics	# of known nodes	Key nodes recovered
606	ADD	43	key22[0]
607	ADC	52	key22[1]
608	ADC	63	key22[2]
578	ADD	72	-
579	ADC	81	key21[1]
580	ADC	90	key21[2]
595	MUL	99	key21[0],key22[3]
550	ADD	108	-
551	ADC	117	key21[3]

Table 5: Attack on SPECK found by TADA - Each row corresponds to a vulnerable instruction with sequence number "No." and operation "Mnemonics" such that after the attack on this instruction, the number of known nodes is given by "# of known nodes" and the key nodes that are retrieved are given by the last column.

Assembly Analysis. To the best of our knowledge, the only work on automation of DFA in assembly implementation is [15], where the authors automated the search for vulnerable instructions according to user input. However, whether the found instruction is really exploitable and how to exploit it has to be done manually. Therefore, the developed tool outputs larger number of vulnerable instructions while only a small subset might be actually exploitable. Moreover, the vulnerability criteria for finding these instructions have to be defined by the user.

Cipher level fault analysis. Khanna et al. [33] recently proposed XFC – a framework for exploitable fault characterization in block ciphers. It takes a cipher specification as input and analyzes it w.r.t. DFA by coloring the fault propagation throughout the cipher state. While the authors show that this approach works when analyzing a high-level representation of a cipher, it is not sufficient to discover vulnerabilities that are implementation specific. Agosta et al. [1] utilized an approach that works on intermediate representations in order to identify single bit-flip vulnerabilities in the code. While this approach takes the analysis one level lower, it still aims at detecting spots that can be exploited from the cipher level instead of finding implementation specific vulnerabilities.

Hardware level analysis. Dureuil et al. [22] presented a fault model inference approach that outputs vulnerability rate for a particular hardware. By observing the possible fault models and their occurrence probabilities, they could estimate a robustness of embedded software. The main aim of their approach was to approximate a time that is needed to successfully inject a required fault model.

SAT related. There are several automation works for algebraic fault attacks on cipher level [50, 51] utilizing SAT solver. The main idea is to describe the cipher algorithm as well as the fault attack in algebraic equations, then use SAT solver to solve for the key. But this also limits the attack to a particular key. The tool developed needs either one or several pairs of ciphertext and plaintext. Another work utilizing SAT solver was automation of DFA on circuit level [23]. They describe the circuit as well as the fault in conjunctive normal form and use SAT to solve for the key. Similarly to previous works, the developed tool aims to solve one key at a time.

Fault injections in dependable systems. Parallel to cryptographic fault attacks, there is an area of dependable systems that analyzes errors and their transitions in computer systems. The same terminology applies here, such as fault injection, fault model, fault resilience etc. These works (e.g. [24, 45, 49]) however do not focus on exploiting deliberately injected faults. They analyze the propagation in software and possible consequences, such as program failing to produce the output or corrupting the data. Similar to these works, Goubet et al. [27] introduced a framework for evaluating countermeasures against fault attacks. It is based on comparing two code snippets – with and without protection. However, it does not focus on cryptographic implementations, only on determining the robustness of countermeasures.

In comparison to these approaches, TADA works on an assembly level in a way that makes it possible to discover implementation-specific vulnerabilities. Furthermore, it does not require any ciphertext-plaintext pairs. It analyzes the implementation without knowledge of the data being processed.

6 DISCUSSION

Countermeasures. Software countermeasures can be based for example on coding theory [14, 16], instruction redundancy [37, 42], or infection [25]. As mentioned before, TADA is capable of analyzing single bit flip vulnerabilities, however, fault countermeasures are normally intended to protect against such models. To be able to evaluate other fault models or even multiple faults during one encryption, an extension to SMT solver module would have to be done, which is the aim of the future work. Nevertheless, it is

possible to check for the potential flaws in these countermeasure implementations by running the analysis with TADA.

Jumps and loops. As stated in previous sections, TADA conducts the analysis on an unrolled implementation. While for standard static code analysis, the conditional branches constitute a non-trivial problem, in our case we do not need to consider implementations with these or other types of jumps and branches. The reason is that these implementations are inherently vulnerable against physical attacks and the attacker can target them with much simpler methods than those considered in this work. For example, a conditional branch decides on a jump based on processed variables, and therefore leaks a timing information [35, 39]. Jump to a subroutine can be skipped entirely, resulting to a trivial analysis [36]. Similarly, round counters used in loops can be attacked to reduce the number of rounds [20].

7 CONCLUSIONS

In this work, we proposed a method for fully automated DFA attack on assembly implementations of symmetric key cryptographic algorithms. The automation of this approach was implemented in *TADA – Tool for Automated DFA on Assembly*. To show the practicality of TADA, we presented novel implementation-specific attacks on SIMON, SPECK, and PRIDE that were not published before. We also provided evaluation on AES, where TADA was able to find existing DFA attack published in literature.

In the future, we would like to focus on other fault analysis methods than DFA. Also, we would like to implement a multi-fault adversarial model, allowing injecting more than one fault during one encryption/decryption routine. Such model is necessary for defeating wide range of fault countermeasures based on redundancy.

REFERENCES

[1] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. 2014. Differential Fault Analysis for Block Ciphers: An Automated Conservative Analysis. In *Proceedings of the 7th International Conference on Security of Information and Networks (SIN '14)*. ACM, New York, USA, Article 137.

[2] Michel Agoyan, Jean-Max Dutertre, Amir-Pasha Mirbaha, David Naccache, Anne-Lise Ribotta, and Assia Tria. 2010. How to flip a bit?. In *On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International*. IEEE, 235–239.

[3] Martin R Albrecht, Benedikt Driessen, Elif Bilge Kavun, Gregor Leander, Christof Paar, and Tolga Yalçın. 2014. Block ciphers–focus on the linear layer (feat. PRIDE). In *International Cryptology Conference*. Springer, 57–76.

[4] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. 2006. The Sorcerer's Apprentice Guide to Fault Attacks. *Proc. IEEE* 94, 2 (2006), 370–382.

[5] Alessandro Barenghi, Gerardo Pelosi, Luca Breveglieri, Francesco Regazzoni, and Israel Koren. 2010. Low-Cost Software Countermeasures Against Fault Attacks: Implementation and Performances Trade Offs. (2010).

[6] R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith, and L. Wingers. 2015. The SIMON and SPECK lightweight block ciphers. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.

[7] Arthur Beckers, Benedikt Gierlichs, and Ingrid Verbauwhede. 2017. Fault Analysis of the ChaCha and Salsa Families of Stream Ciphers. *Lecture Notes in Computer Science* (2017).

[8] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. 2016. The SKINNY Family of Block Ciphers and its Low-Latency Variant MANTIS. Cryptology ePrint Archive, Report 2016/660. (2016). https://eprint.iacr.org/2016/660.

[9] Daniel J Bernstein. 2008. ChaCha, a variant of Salsa20. In *Workshop Record of SASC*, Vol. 8.

[10] Sarani Bhattacharya and Debdeep Mukhopadhyay. 2016. Curious case of rowhammer: flipping secret exponent bits using timing analysis. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 602–624.

[11] Eli Biham and Adi Shamir. 1991. Differential cryptanalysis of DES-like cryptosystems. In *Advances in Cryptology-CRYPTO*, Vol. 90. Springer, 2–21.

[12] Eli Biham and Adi Shamir. 1997. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology - CRYPTO '97*. Springer, 513–525.

[13] Alex Biryukov and Leo Perrin. 2017. State of the Art in Lightweight Symmetric Cryptography. Cryptology ePrint Archive, Report 2017/511. (2017).

[14] Jakub Breier and Xiaolu Hou. 2016. Feeding Two Cats with One Bowl: On Designing a Fault and Side-Channel Resistant Software Encoding Scheme (Extended Version). Cryptology ePrint Archive, Report 2016/931. (2016). http://eprint.iacr.org/2016/931.

[15] Jakub Breier and Xiaolu Hou. 2017. Automated Fault Analysis of Assembly Code (With a Case Study on PRESENT Implementation). Cryptology ePrint Archive, Report 2017/829. (2017). https://eprint.iacr.org/2017/829.

[16] Julien Bringer, Claude Carlet, HervÃľ Chabanne, Sylvain Guilley, and Houssem Maghrebi. 2014. Orthogonal Direct Sum Masking: A Smartcard Friendly Computation Paradigm in a Code, with Builtin Protection against Side-Channel and Fault Attacks. Cryptology ePrint Archive, Report 2014/665. (2014). http://eprint.iacr.org/2014/665.

[17] Joan Daemen and Vincent Rijmen. 2002. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[18] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (Sept. 2011), 69–77. https://doi.org/10.1145/1995376.1995394

[19] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2008), Budapest, Hungary*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[20] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre, and Assia Tria. 2013. Electromagnetic Glitch on the AES Round Counter. In *Constructive Side-Channel Analysis and Secure Design*, Emmanuel Prouff (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 17–31.

[21] Anh Nguyen Duc, Ronald Jabangwe, Pangkaj Paul, and Pekka Abrahamsson. 2017. Security Challenges in IoT Development: A Software Engineering Perspective. In *Proceedings of the XP2017 Scientific Workshops (XP '17)*. ACM, Article 11, 5 pages.

[22] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. 2016. FISSC: A Fault Injection and Simulation Secure Collection. In *Computer Safety, Reliability, and Security: 35th International Conference, SAFECOMP 2016, Trondheim, Norway*. Springer, 3–11.

[23] M. Gay, J. Burchard, J. Horacek, A.S.M. Ekossono, T. Schubert, B. Becker, I. Polian, and M Kreuzer. 2016. Small scale AES toolbox: Algebraic and propositional formulas, circuit implementations and fault equations. FCTRU. (2016). http://hdl.handle.net/2117/99210.

[24] Giorgis Georgakoudis, Ignacio Laguna, Dimitrios S Nikolopoulos, and Martin Schulz. 2017. REFINE: realistic fault injection via compiler-based instrumentation for accuracy, portability and speed. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 29.

[25] Benedikt Gierlichs, Jörn-Marc Schmidt, and Michael Tunstall. 2012. Infective Computation and Dummy Rounds: Fault Protection for Block Ciphers without Check-before-Output. In *Progress in Cryptology – LATINCRYPT 2012*, Alejandro Hevia and Gregory Neven (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 305–321.

[26] Christophe Giraud. 2005. DFA on AES. In *Proceedings of the 4th International Conference on Advanced Encryption Standard (AES'04)*. Springer, 27–41.

[27] Lucien Goubet, Karine Heydemann, Emmanuelle Encrenaz, and Ronald De Keulenaer. 2015. Efficient Design and Evaluation of Countermeasures against Fault Attacks Using Formal Verification. In *Smart Card Research and Advanced Applications: 14th International Conference, CARDIS 2015, Bochum, Germany*, Naofumi Homma and Marcel Medwed (Eds.). Springer, Cham, 177–192.

[28] Yuming Huo, Fan Zhang, Xiutao Feng, and Li-Ping Wang. 2015. Improved differential fault attack on the block cipher SPECK. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2015 Workshop on*. IEEE, 28–34.

[29] Dirmanto Jap and Jakub Breier. 2015. Differential Fault Attack on LEA. In *Information and Communication Technology*, Ismail Khalil, Erich Neuhold, A Min Tjoa, Li Da Xu, and Ilsun You (Eds.). Springer International Publishing, Cham, 265–274.

[30] Jérémy Jean. 2016. TikZ for Cryptographers. https://www.iacr.org/authors/tikz/. (2016).

[31] Kitae Jeong, Yuseop Lee, Jaechul Sung, and Seokhie Hong. 2013. Improved differential fault analysis on PRESENT-80/128. *International Journal of Computer Mathematics* 90, 12 (2013), 2553–2563. arXiv:http://dx.doi.org/10.1080/00207160.2012.760732

[32] Marc Joye and Michael Tunstall. 2012. *Fault Analysis in Cryptography*. Springer Publishing Company, Incorporated.

[33] Punit Khanna, Chester Rebeiro, and Aritra Hazra. 2017. XFC: A Framework for eXploitable Fault Characterization in Block Ciphers. In *Proceedings of the 54th Annual Design Automation Conference 2017 (DAC '17)*. ACM, Article 8, 6 pages.

[34] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology — CRYPTO' 99: 19th Annual International Cryptology Conference, California, USA*. Springer, 388–397.

[35] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '96)*. Springer-Verlag, London, UK, UK, 104–113. http://dl.acm.org/citation.cfm?id=646761.706156

[36] SV Dilip Kumar, Sikhar Patranabis, Jakub Breier, Debdeep Mukhopadhyay, Shivam Bhasin, Anupam Chattopadhyay, and Anubhab Baksi. 2017. A practical fault attack on ARX-like ciphers with a case study on ChaCha20. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC, Taipei, Taiwan*.

[37] Benjamin Lac, Anne Canteaut, Jacques J.A. Fournier, and Renaud Sirdey. 2017. Thwarting Fault Attacks using the Internal Redundancy Countermeasure (IRC). Cryptology ePrint Archive, Report 2017/910. (2017). http://eprint.iacr.org/2017/910.

[38] Sam Lucero. 2016. IoT platforms: enabling the Internet of Things. IHS Technology – Whitepaper. (2016), 21 pages. https://cdn.ihs.com/www/pdf/enabling-IOT.pdf

[39] Baolei Mao, Wei Hu, Alric Althoff, Janarbek Matai, Jason Oberg, Dejun Mu, Timothy Sherwood, and Ryan Kastner. 2015. Quantifying Timing-Based Information Flow in Cryptographic Hardware. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '15)*. IEEE Press, Piscataway, NJ, USA, 552–559. http://dl.acm.org/citation.cfm?id=2840819.2840896

[40] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. 2013. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 77–88. https://doi.org/10.1109/FDTC.2013.9

[41] N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, and E. Encrenaz. 2014. Experimental evaluation of two software countermeasures against fault attacks. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 112–117. https://doi.org/10.1109/HST.2014.6855580

[42] Conor Patrick, Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. 2016. Lightweight Fault Attack Resistance in Software Using Intra-Instruction Redundancy. Cryptology ePrint Archive, Report 2016/850. (2016). http://eprint.iacr.org/2016/850.

[43] Matthieu Rivain. 2009. Differential Fault Analysis on DES Middle Rounds. In *Cryptographic Hardware and Embedded Systems - CHES 2009: 11th International Workshop Lausanne, Switzerland, September 6-9, 2009 Proceedings*, Christophe Clavier and Kris Gaj (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 457–469. https://doi.org/10.1007/978-3-642-04138-9_32

[44] Sayandeep Saha, Debdeep Mukhopadhyay, and Pallab Dasgupta. 2018. ExpFault: An Automated Framework for Exploitable Fault Characterization in Block Ciphers. Cryptology ePrint Archive, Report 2018/295. (2018). https://eprint.iacr.org/2018/295.

[45] Behrooz Sangchoolie, Karthik Pattabiraman, and Johan Karlsson. 2017. One Bit is (Not) Enough: An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE, 97–108.

[46] C. E. Shannon. 1949. Communication theory of secrecy systems. *The Bell System Technical Journal* 28, 4 (Oct 1949), 656–715. https://doi.org/10.1002/j.1538-7305.1949.tb00928.x

[47] Michael Tunstall and Debdeep Mukhopadhyay. 2009. Differential Fault Analysis of the Advanced Encryption Standard using a Single Fault. Cryptology ePrint Archive, Report 2009/575. (2009).

[48] H. Tupsamudre, S. Bisht, and D. Mukhopadhyay. 2014. Differential Fault Analysis on the Families of SIMON and SPECK Ciphers. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 40–48. https://doi.org/10.1109/FDTC.2014.14

[49] Erik van der Kouwe and Andrew S Tanenbaum. 2016. HSFI: accurate fault injection scalable to large code bases. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*. IEEE, 144–155.

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

[51] Fan Zhang, Xinjie Zhao, Shize Guo, Tao Wang, and Zhijie Shi. 2013. Improved algebraic fault analysis: A case study on piccolo and applications to other lightweight block ciphers. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 62–79.

A ATTACK DETAILS FOR $\mathcal{F}_{ex}$

Here we present more details output by TADA for $\mathcal{F}_{ex}$ (in Table 2). Table 6 summaries the attack details on vulnerable instruction 4. After the analysis of instruction 4, both key bytes are recovered and the cipher is broken. Figure 9 shows the updated DFG after the attack on instruction 4. We can see that all the nodes are known now.

target node	vulnerable node	(k, δ, x, out, $\mathtt{val}_0$, $\mathtt{val}_1$)	retrieved key byte
r1(1)	r0(0)	(0, 1, r0(5), 0x01, 0x00, 0x01) (1, 2, r0(5), 0x02, 0x00, 0x02) (2, 4, r0(5), 0x04, 0x00, 0x04) (3, 8, r0(5), 0x28, 0x00, 0x08) (4, 16, r0(5), 0x10, 0x00, 0x10) (5, 32, r0(5), 0x20, 0x00, 0x20) (6, 64, r0(5), 0x40, 0x00, 0x40) (7, 128, r0(5), 0x80, 0x00, 0x80)	key1[1]
r0(0)	r1(1)	(0, 1, r0(5), 0x01, 0x00, 0x01) (1, 2, r0(5), 0x02, 0x00, 0x02) (2, 4, r0(5), 0x04, 0x00, 0x04) (3, 8, r0(5), 0x08, 0x00, 0x08) (4, 16, r0(5), 0x10, 0x00, 0x10) (5, 32, r0(5), 0x20, 0x00, 0x20) (6, 64, r0(5), 0x40, 0x00, 0x40) (7, 128, r0(5), 0x80, 0x00, 0x80)	key1[0]

Table 6: Summary of TADA output for DFA attacks on $\mathcal{F}_{ex}$ in Table 2 (values of x, out, $\mathtt{var}_0$, $\mathtt{var}_1$ are in hexadecimal format)

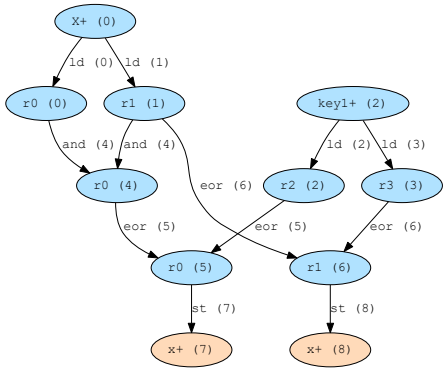

Figure 9: Updated graph generated by TADA after two successful attacks on vulnerable instruction 4 of example cipher $\mathcal{F}_{ex}$ in Table 2.

B PRIDE SBOX IMPLEMENTATION

The equations for Sbox of PRIDE [3] are as follows:

$$A = c \oplus (a\&b) \quad (4)$$

$$B = d \oplus (b\&c) \quad (5)$$

$$C = a \oplus (A\&B) \quad (6)$$

$$D = b \oplus (B\&C), \quad (7)$$

where the input is a 4–bit variable with bits a, b, c, d and the output is a 4–bit variable with bits A, B, C, D.

C ATTACK DETAILS

In this part, we provide the attack details on cipher implementations that were chosen for TADA evaluation in Section 4. Each of the tables provides the information on which instructions were identified as vulnerable and what was the attack flow leading to secret key retrieval.

Table 7 shows the attack on SIMON, Table 8 shows attack on SPECK (see Remark 4), additionally to one described in Section 4. Table 9 provides attack details on AES, and finally Table 10 details attack on PRIDE. Here "keyx[y]" refers to the $(y+1)$th byte of round key in round x.

No.	# of known nodes	key nodes recovered
1136	73	-
1137	83	-
1138	92	-
1139	106	-
1174	115	key32[3]
1175	129	key31[1], key32[0]
1176	142	key31[2], key32[1]
1177	162	key31[0], key31[3], key32[2]

Table 7: Attack on SIMON found by TADA - Each row corresponds to a vulnerable instruction with sequence number "No." such that after the attack on this instruction, the number of known nodes is given by "# of known nodes" and the key nodes that are retrieved are given by the third column.

No.	# of known nodes	key nodes recovered	brute force
606	43	key22[0]	-
607	52	key22[1]	-
608	63	key22[2]	-
609	68	key22[3]	1
578	79	key21[0]	-
579	88	key21[1]	-
580	99	key21[2]	-
581	104	key21[3]	1

Table 8: Attack on SPECK found by TADA (considering obtaining 7 bits as successful attack) - Each row corresponds to a vulnerable instruction with sequence number "No." such that after the attack on this instruction, the number of known nodes is given by "# of known nodes" and the key nodes that are retrieved are given by the third column, in case only 7 bits of the target node are obtained, the brute force complexity is indicated by 1.

No.	# of known nodes	key nodes recovered
1806	73	key11[0]
1808	77	key11[1]
1810	81	key11[2]
1812	85	key11[3]
1814	91	key11[7]
1816	96	key11[4]
1818	101	key11[5]
1820	106	key11[6]
1822	112	key11[10]
1824	118	key11[11]
1826	123	key11[8]
1828	128	key11[9]
1830	133	key11[13]
1832	138	key11[14]
1834	143	key11[15]
1836	149	key11[12]

Table 9: Attack on AES found by TADA - Each row corresponds to a vulnerable instruction with sequence number "No." such that after the attack on this instruction, the number of known nodes is given by "# of known nodes" and the key nodes that are retrieved are given by the third column.

No.	# of known nodes	key nodes recovered
1504	21	-
1506	27	-
1508	32	-
1516	54	-
1522	106	key20[0],key20[1], key20[2],key20[3], key20[4],key20[5], key20[6],key20[7]
1422	111	-
1424	117	-
1426	122	-
1434	144	-
1440	196	key19[0],key19[1], key19[2],key19[3], key19[4],key19[5], key19[6],key19[7]

Table 10: Attack on PRIDE found by TADA - Each row corresponds to a vulnerable instruction with sequence number "No." such that after the attack on this instruction, the number of known nodes is given by "# of known nodes" and the key nodes that are retrieved are given by the third column.