# Fully Automated Differential Fault Analysis on Software Implementations of Block Ciphers

Xiaolu Hou[1], Jakub Breier[2], Fuyuan Zhang[2] and Yang Liu[2]

[1] Acronis, Singapore
[2] School of Computer Science and Engineeering, Nanyang Technological University, Singapore
ho0001lu@e.ntu.edu.sg,jbreier@jbreier.com,fuyuanzhang@163.com,yangliu@ntu.edu.sg

**Abstract.** 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 SPECK and PRIDE, exploiting implementation specific vulnerabilities, and previously published DFA on SIMON and 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. Moreover, we present a method to check the countermeasure-protected implementations in a way that helps implementers to decide how many rounds should be protected.
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, fault attacks, automation, assembly

## 1 Introduction

Lightweight cryptography is one of the areas that became crucial with the emergence of Internet of Things. There are numerous algorithms providing sufficient security properties, while keeping the footprint minimal [BP17]. Some of them work better in hardware, such as SIMON [BTCS+15] and SKINNY [BJK+16], while others aim at software, such as SPECK [BTCS+15] and ChaCha [Ber08]. 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 [KJJ99], this is done by observing physical characteristics of a device (electromagnetic emanation,

Licensed under Creative Commons License CC-BY 4.0.
IACR Transactions on CHES Vol. 0, No.0, pp.1—29, DOI:XXXXXXXX

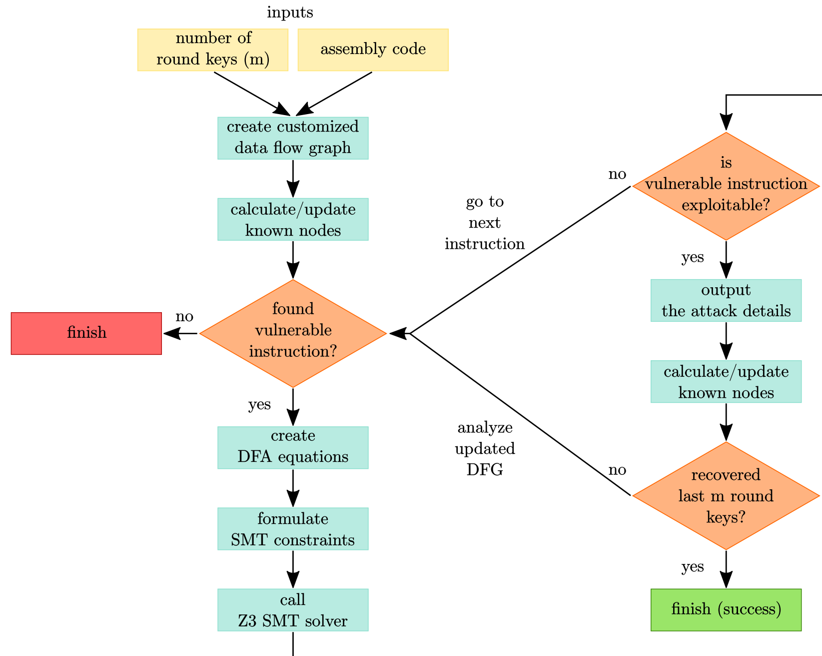


Figure 1: Overview of TADA.

timing, etc.) and correlating this information with the values processed in the algorithm. In case of FA [BS97], 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) [BS97] 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 [BS91], one can find many attacks that are on the cipher design 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 [BHL18]. 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 estimates imprecise. For example, in [HZFW15], the authors claim they can break SPECK cipher with only $5 \sim 8$ 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 $4\times$ bigger in the case of SPECK32/64 and $16\times$ 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 [BECN+06].

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. To solve this problem, automated approaches are starting to emerge, gaining more popularity in the fault analysis

community [BHB19].

**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 [SMD18] or are not completely automated [BHL18], thus falling short in finding an attack without further manual analysis.

Previously developed tools either search for a possible DFA by enumerating different inputs and then trying to find the key by solving equations for these [SMD18], or provide an estimate of vulnerabilities based on the structure of the cipher [KRH17, RRHB19]. Similar enumeration method was used in the Algebraic Fault Analysis tools [ZZG+13, ZGZ+16] where a subset of all the possible solutions is checked by the SAT solver. For example, when using CryptoMiniSAT, the number of solutions should be below $2^{18}$, otherwise the analysis time becomes impractical [ZGZ+16]. Therefore, while the tools are able to find a particular key, they fall short of 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 [BTCS+15], AES is the current NIST standard [DR02], and PRIDE [ADK+14] is a lightweight cipher optimized for 8-bit microcontrollers. For 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 SIMON and AES, we were able to find equivalent attacks that were presented in the literature on the cipher level. Thanks to TADA, we could identify

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 |

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.

To show the utility of our approach, we have designed an algorithm that follows up on results obtained by TADA, to optimize the implementation of DFA countermeasures. It analyzes the cipher implementations and outputs the earliest rounds which can be attacked using the vulnerabilities found by TADA. This information can be used to determine the minimum number of rounds that should be protected. We would like to emphasize that bit flip attack model represents the strongest adversarial model when considering DFA, and therefore, it is crucial to test implementations against such model.

**Organization.** The rest of the paper is structured as follows. Section 2 introduces related work. Section 3 provides preliminaries on DFA and SMT solvers. Section 4 presents the design and describes the usage of TADA. Section 5 shows experimental evaluations on SIMON, SPECK, AES and PRIDE. Section 6 shows how to utilize information found by TADA to implement efficient countermeasures. Finally, Section 7 concludes this work.

## 2 Related work

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

**Assembly Analysis.** To the best of our knowledge, the only work on automation of DFA in assembly implementation is [BHL18], 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. [KRH17] 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. [ABPS14] 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. [DPP+16] 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 [ZZG+13, ZGZ+16] 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 developed tool needs either one or several pairs of ciphertext(s) and plaintext(s). To reduce the search space to feasible number, these methods normally fix the major portion of the input data and enumerate the rest. Another work utilizing SAT solver was automation of DFA on the circuit level [GBH+16]. 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.

**Automated Analysis of Public Key Cryptosystems.** Barthe et al. [BDF+14] identify implementation-independent fault conditions and use program synthesis to discover faulted implementations. Using their tool they found attacks on RSA and ECDSA.

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.

# 3 Background

In the following, we explain the necessary background on DFA and SMT solvers that will put the rest of our work into context.

## 3.1 Differential Fault Analysis

When performing DFA on symmetric block cipher, 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.

The above simple example shows why DFA normally exploits nonlinear operations and we will also be focusing on nonlinear operations in our analysis. DFA often works with just a single faulty and correct ciphertext pair from the encryption [TM09]. 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 $\delta$, which is equal to 1 in this case, is called the *fault mask*. The output difference $\Delta$ 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.

### 3.2 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) [DMB11] 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 [dMB08].

We would like to note that the analysis by the SMT solver involves an exhaustive search on the possible fault mask values and the output differences. Therefore, while it is feasible to perform such analysis on fault masks that correspond to bit-flips, when extending it to all the possible fault mask values, it becomes impractical. To check the random fault model, the analysis would have to be done in a different manner, shifting the result from provably exploitable to an estimation based on certain characteristics of the cipher (for example, based on properties of round operations). Such method can be found for example in [KRH17].

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

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

Finding a fault attack for $c = a \,\&\, b$ amounts to finding a mapping between the value of $\Delta$ 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 $\psi$:

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

Notice that $\delta$ is the only free variable in both formulas. We explain the first formula briefly. Since $\delta$ is the only free variable in formula 1), checking the satisfiability of formula 1) amounts to asking whether we can find a value for the fault mask $\delta$ 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.

## 4 TADA Methodology

In this section we present the methodology that was used when implementing TADA. Section 4.1 describes the fault models we consider. Section 4.2 details attacks on target instructions. Requirements on assembly code are stated in Section 4.3. Section 4.4 provides the design overview of TADA and details the automated analysis steps. Finally, Section 4.5 explains the analysis carried out by the SMT solver module.

## 4.1 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 name of the operation that $f$ uses. We say an instruction $f$ is *linear* if its mnemonic corresponds to a linear operation.

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 input operands/output operands of this instruction or it deletes this instruction from the sequence (instruction skip) [MDH+13]. In this work we do not focus on instruction skip faults.

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 by laser fault injection, where single bits in registers manufactured with 90 nm technology were targeted with 100% success rate [CLMFT14]. When it comes to flipping bits in DRAM, the Rowhammer attack can be used [SD15]. As it was shown, the DRAM memory bank can be first localized with a timing side-channel and later the exact row indices can be determined to achieve a precise bit flip [BM16]. Timing adjustment for targeting single instructions was previously shown in [BJC15].

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

## 4.2 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 [TBM14], bitwise OR [BHL18], addition [TBM14, JB15, BGV17], and table lookup [Riv09a, JLSH13, TM09], 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 [JT12].

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 3.1 (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:

[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.

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

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

We inject a fault $\delta$ 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 $\Delta$:

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

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

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

**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 $\delta$ 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 $\Delta$: 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 $\Delta$ & $\texttt{out}$ is either $\texttt{var}_0$ or $\texttt{var}_1$, and $\Delta$ & $\texttt{out} = \texttt{var}_0 \Longrightarrow a = 0$; $\Delta$ & $\texttt{out} = \texttt{var}_1 \Longrightarrow a = 1$.

**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 $\delta$ 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 $\Delta$ & $\texttt{out} = \texttt{var}_0$ or $\texttt{var}_1$ and $\Delta$ & $\texttt{out} = \texttt{var}_0 \Longrightarrow a = 0$; $\Delta$ & $\texttt{out} = \texttt{var}_1 \Longrightarrow a = 1$. Note that the choices of value for variables $\texttt{out}$, $\texttt{var}_0, \texttt{var}_1$ 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 possible 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 [Sha49].

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 Sbox(0) = 1, Sbox(1) = $a \ldots$, 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) [BS91], where the $(\Delta, \delta)-$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 $\delta$ 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 |

| $\Delta\backslash\delta$ | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| 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 $(1, b), (2, 1), (4, 2), (8, f)$ 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 $\Delta$.

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.

## 4.3 TADA Usage

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

The developed methodology requires identification of important variables, more specifically the round keys and ciphertext variables, to be able to correctly focus on key recovery.

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 |

TADA implements this in a way that for analyzed assembly code, it requires naming conventions. 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. 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 [Koc96, MHA+15]. Jump to a sub-routine can be skipped entirely, resulting to a trivial analysis [KPB+17]. Similarly, round counters used in loops can be attacked to reduce the number of rounds [DMM+13].

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).

Software countermeasures can be based for example on coding theory [BH16, BCC+14], instruction redundancy [PYGS16, LCFS17], or infection [GST12]. We note that TADA is capable of analyzing single bit flip vulnerabilities and therefore, it can check whether the countermeasure was implemented sufficiently.

## 4.4 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 of the tool.

**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 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

**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 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.** 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)`.

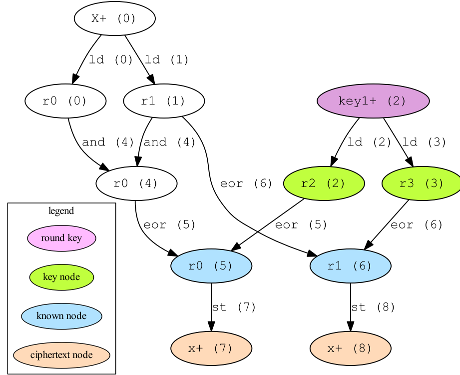


Figure 2: Data flow graph generated by TADA for example cipher implementation $\mathcal{F}_{ex}$ from Table 2.

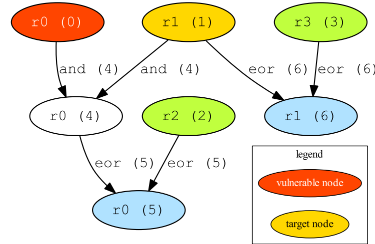


Figure 3: Subgraph generated by TADA for target node `r1(1)` and vulnerable node `r0(0)` from vulnerable instruction 4 of example cipher $\mathcal{F}_{ex}$ in Table 2.

**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 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 4.2. In AVR assembly,

**Algorithm 1:** Check if an instruction is vulnerable.

```
   Input  : f: an instruction of a program F, DFG: customized data flow graph
            corresponding to F, key: target round key
   Output: boolean: is f vulnerable?
1  if Mnemonic of f ∈ {AND, OR, ADD, ADC, LPM, MUL} then
2      if Mnemonic of f =MUL then
3          boolean vul = false;
4          for a: input nodes of f do
5              if a is a constant then
6                  vul=true;
7          if vul=false then
8              return false;
9      for a: output nodes of f do
10         for x: known nodes affected by a do
11             if distance(a, x) > 0 then
12                 return false;
13             for b: nodes of f do
14                 if b is linearly related to key then
15                     return true;
16 return false;
```

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 4.2.
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 Section 3.1, 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[3]. That is why we put our focus on non-linear instructions only.

**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 DFG that includes the vulnerable instruction and the nodes affected by it. The subgraph stops at the known nodes.

[3]While the linear layer might help to spread the fault and make the analysis more efficient, it is not the goal of our work – such analysis would be similar to injecting multiple faults into several nodes.

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) $r1(6) = r1(1) \oplus r3(3)$ | | |

**Example 4.** For $\mathcal{F}_{ex}$ in 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, 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 $\delta$. Input to SMT solver module also indicates which variables involved in the DFA equations represent known nodes.

**Example 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. The details of this module are presented in Section 4.5. After each successful attack on a target node, TADA updates the known nodes in the graph.

**Example 6.** The attack on vulnerable instruction 4 with target node `r1(1)` and vulnerable node `r0(0)` as described in Example 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 7).

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.

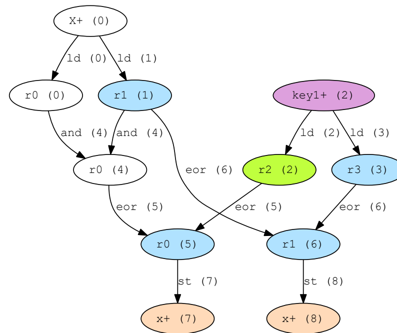


Figure 4: Updated graph generated by TADA after a successful attack on target node `r1(1)` from vulnerable instruction 4 of example cipher $\mathcal{F}_{ex}$ in Table 2.

**Algorithm 2:** The algorithm for attacking bitwise AND, bitwise OR.

**Input** : $\psi$: DFA equations and the equation for fault mask; $b$: vulnerable node; $a$: target node; $S$: the list of knowns nodes in the DFA equations; $\psi_\delta$: constraint for fault mask $\delta$ as in Equation (2).

**Output :** boolean: `true` if $a$ can be obtained by attacking $b$.

```
1  for x: S do
2      φ := ψ ∧ (Δ = x ⊕ x′);
3      counter = 0;
4      for k = 0, 1, . . . , 7 do
5          ψ1 := φ ⇒ ((Δ & out = val0) ∨ (Δ & out = val1));
6          ψ2 := (φ ∧ Δ & out = val0) ⇒ a[k] = 0;
7          ψ3 := (φ ∧ Δ & out = val1) ⇒ a[k] = 1;
8          Φ := ψδ ∧ ψ1 ∧ ψ2 ∧ ψ3;
9          V := all variables involved in Φ;
10         if (∀v ∈ V \ {out, δ, val0, val1} : Φ) is satisfiable then
11             save to file (k, δ, x, out, val0, val1);
12             counter++;
13     if counter = 8 then
14         return true;
15 return false;
```

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).

## 4.5 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 5). In this section, we detail how other constraints and the satisfiability problems are designed for each of the operations described in Section 4.2.

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 2. For each known node $x$, TADA generates an equation for output mask $\Delta$ (line 2). Then it tries to attack each bit of target node $a$ (line 4). Line 5 specifies that for some variables $\texttt{out}$, $\texttt{var}_0$ and $\texttt{var}_1$, the output mask $\Delta$ has one of the two patterns: $\Delta \ \& \ \texttt{out} = \texttt{var}_0$ or $\Delta \ \& \ \texttt{out} = \texttt{var}_1$. Line 6 specifies if $\Delta$ is of the first pattern then the $k$th bit of the target node is 0. Line 7 says if $\Delta$ is of the second pattern, the $k$th bit of the target node is 1. Line 10 tests if there exist valuations to variables $\texttt{out}$, $\delta$, $\texttt{var}_0$ 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 $k$th bit of the target node $a$ can be obtained by attacking the vulnerable node using fault mask $\delta$ and observing the output mask $\Delta$ of the known node $x$: if $\Delta$ & $\texttt{out} = \texttt{var}_0$ then $a[k] = 0$ and if $\Delta$ & $\texttt{out} = \texttt{var}_1$ then $a[k] = 1$.

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.

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 8.

*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 3. 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 $(0, \delta, x, y, \texttt{out}, \texttt{var}_0, \texttt{var}_1)$ is saved to a file. This corresponds to: the 0th bit of the target node $a$ can be obtained by injecting fault mask $\delta$ in vulnerable node and observing the output mask $\Delta = y||x \oplus y'||x'$. If $\Delta$ & $\texttt{out} = \texttt{var}_0$, $a[0] = 0$ and if $\Delta$ & $\texttt{out} = \texttt{var}_1$, $a[0] = 1$. Similar to the discussion of the attack on addition with carry in Section 4.2, 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 $(k, 0, \delta, x, y, \texttt{out}, \texttt{var}_0, \texttt{var}_1)$ is saved to a file, which means: when carry is 0, the $k$th bit of the target node $a$ can be obtained by injecting fault mask $\delta$ in vulnerable node and observing the output mask $\Delta$. If $\Delta$ & $\texttt{out} = \texttt{var}_0$, $a[k] = 0$ and if $\Delta$ & $\texttt{out} = \texttt{var}_1$, $a[k] = 1$. 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 $k$th 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 – 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 2. 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 Algorithm 5 from Appendix B is used.

**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 2 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 $k$th bit of target node $a$ can be obtained by injecting fault mask $\delta$ in vulnerable node $b$ and observing the output mask $\Delta = y||x \oplus y'||x'$. If $\Delta$ & $\texttt{out} = \texttt{var}_0$, $a[k] = 0$ and if $\Delta$ & $\texttt{out} = \texttt{var}_1$, $a[k] = 1$.

**Algorithm 3:** The algorithm for attacking `ADD`.

**Input :** $\psi$: DFA equations and the equation for fault mask; $b$: vulnerable node; $a$: target node; $S$: the list of knowns nodes in the DFA equations; $\psi_\delta$: constraint for fault mask $\delta$ as in Equation (2).

**Output :** boolean: `true` if $a$ can be obtained by attacking $b$.

```
1  for x, y ∈ S, x ≠ y do
2      counter= 0;
3      φ := ψ ∧ (Δ = y||x ⊕ y'||x');
4      ψ1 := φ ⇒ ((Δ & out = val0) ∨ (Δ & out = val1));
5      ψ2 := (φ ∧ Δ & out = val0) ⇒ a[0] = 0;
6      ψ3 := (φ ∧ Δ & out = val1) ⇒ a[0] = 1;
7      Φ := ψδ ∧ ψ1 ∧ ψ2 ∧ ψ3;
8      V := all variables involved in Φ;
9      if (∀v ∈ V\{out, δ, val0, val1} : Φ) is satisfiable then
10         save to file (0, δ, x, y, out, val0, val1);
11         counter++;
12     else
13         continue;
14     for k = 1, 2, ..., 7 do
15         ψc0 := (a[k − 1, 0] + b[k − 1, 0])[k] = 0   // the carry from first k bits = 0;
16         φ := ψc0 ∧ ψ ∧ (Δ = y||x ⊕ y'||x');
17         ψ1 := (φ ⇒ ((Δ & out = val0) ∨ (Δ & out = val1)));
18         ψ2 := ((φ ∧ Δ & out = val0) ⇒ a[k] = 0);
19         ψ3 := ((φ ∧ Δ & out = val1) ⇒ a[k] = 1);
20         Φc0 := ψδ ∧ ψ1 ∧ ψ2 ∧ ψ3;
21         V := all variables involved in Φc0;
22         if ∀v ∈ V\{out, δ, val0, val1} : Φc0 is satisfiable then
23             let output0 = (k, 0, δ, x, y, out, val0, val1);
24         else
25             break;
26         ψc1 := (a[k − 1, 0] + b[k − 1, 0])[k] = 1   // the carry from first k bits = 1;
27         φ := ψc1 ∧ ψ ∧ (Δ = y||x ⊕ y'||x');
28         ψ1 := (φ ⇒ ((Δ & out = val0) ∨ (Δ & out = var1)));
29         ψ2 := ((φ ∧ Δ & out = val0) ⇒ a[k] = 0);
30         ψ3 := ((φ ∧ Δ & out = val1) ⇒ a[k] = 1);
31         Φc1 := ψδ ∧ ψ1 ∧ ψ2 ∧ ψ3;
32         V := all variables involved in Φc1;
33         if ∀v ∈ V\{out, δ, val0, val1} : Φc1 is satisfiable then
34             let output1 = (k, 1, δ, x, y, out, val0, val1);
35             save to file: output0 and output1;
36             counter++;
37         else
38             break;
39     if counter= 8 then
40         return true;
41 return false;
```

## 5 Evaluation

In this section, we will present evaluations of four ciphers using TADA: SIMON [BTCS+15], SPECK [BTCS+15], AES [DR02] and PRIDE [ADK+14]. Results are summarized in Table 4. More details are presented in Appendix D. The analysis was done on a standard laptop computer with Intel Haswell family CORE i7 and 8 GB RAM. For SPECK and

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 | [TBM14] | **new** | [Gir05] | **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 |

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 and SIMON, previously published DFA [Gir05, TBM14] were 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. 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(a).

For SIMON implementation[4], 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 the same attack as state of the art DFA on SIMON [TBM14].

**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 5(b).

For SPECK implementation[4], 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 [TBM14]. 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 10.

[4] https://github.com/openluopworld/simon_speck_on_avr/tree/master/AVR

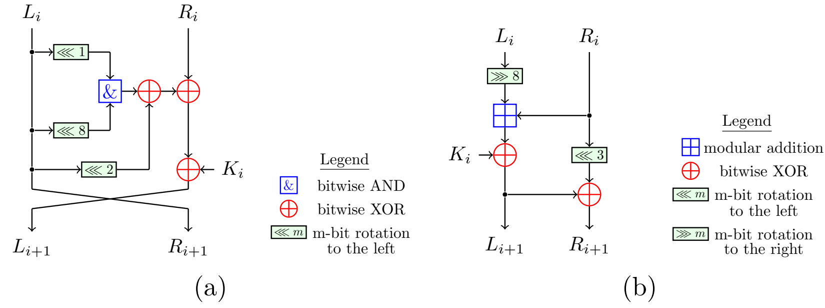


Figure 5: Schematic of one round of (a) SIMON and (b) SPECK block cipher.

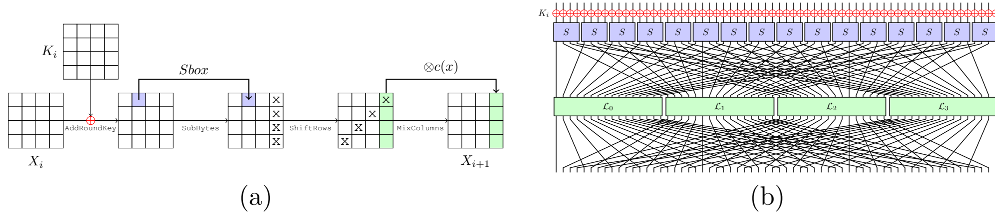


Figure 6: Schematic of one round of (a) AES and (b) PRIDE block cipher.

**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 6(a) (picture was drawn with a usage of library from [Jea16]).

For AES implementation taken from Ecrypt II public repository[5], TADA found the same attack as in [Gir05]. The attack takes advantage 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[6], the Sbox is implemented in algebraic form,

[5] https://perso.uclouvain.be/fstandae/source_codes/lightweight_ciphers/source/AES.asm
[6] https://github.com/FreeDisciplina/BlockCiphersOnAVR/tree/master/PRIDE_64_128_AVR

Table 5: Attack on SPECK found by TADA - Each row corresponds to a vulnerable instruction with sequence number "No." and operation with "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.

| 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]` |

therefore, unlike in AES implementation, no table lookup is necessary. Details on this implementation are shown in Appendix C. Schematic of PRIDE is depicted in Figure 6(b).

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

# 6 Countermeasure implementation based on TADA

An important decision when implementing a countermeasure is how many rounds should be protected. As some techniques, such as round redundancy, are relatively expensive ($> 2\times$ time/space overhead), it is vital to know the minimal protection threshold. Especially since there have been proposals targeting middle rounds of AES [PY06, DFL11] and DES [Riv09b]. To show the practicality of the attack information outputted by TADA, we will explain how to determine the number of rounds that have to be protected to make the found attacks ineffective.

In general, it can be assumed that the best differential cryptanalysis to date gives a good overview of the possibilities of DFA on internal rounds, since DFA is essentially a differential cryptanalysis on a reduced-round cipher. Therefore, in [PY06], the authors suggested a combination of these two techniques to attack round $R - 5$. Since in our approach we consider DFA in its pure form, the question reduces to how to achieve the fault found by TADA by injecting faults in earlier rounds without causing collisions[7]. To determine the number of rounds that need to be protected against DFA for a cryptographic implementation, we have designed an addition to TADA that implements Algorithm 4. Here we utilize the attacks found in Section 5 and refer to the nodes that were proven to reveal the key information when faulted as *exploitable*. Since the DFG is already constructed at the beginning step of the analysis with TADA , the algorithm checks the relationships between the nodes to identify the possible collisions. For each exploitable node $b$ and its parent $p$, Algorithm 4 checks if $p$ affects $b$ with a collision by analyzing pairs of nodes $x, y$ that are both parents of $b$ and children of $p$: a collision appears when there is no parent-child relationship between $x, y$; in other words, they lie in different paths from $p$ to $b$. The output is a set of nodes $P$ that affect the exploitable node $b$ without a collision, effectively determining the earliest round that can be attacked by the given DFA attack.

Since TADA allows assembly code annotations, with the usage of Algorithm 4 it is advised to use the "`round`" annotation so that it can be easily determined in which round is the node $p \in P$ located. After running the algorithm on exploitable nodes from Section 5, we obtained the information about the earliest possible rounds to be attacked, stated in Table 6. This information can be used for deciding on countermeasure implementation, and afterwards, for verification whether the protection was implemented correctly, thus extending the use cases of TADA greatly. Moreover, such information also helps to find more efficient attacker model, reducing the number of faults. To illustrate the amount of resources that can be saved by taking the information from Table 6 into account, it is $\geq$60% of resources in case of AES-128, 80% in case of PRIDE, $\geq$81% in case of SPECK, and $\geq$90% in case of SIMON.

In the following, we will focus on AES, where an optimal single fault DFA attack was already found [TM09], making the comparison easy. Table 7 shows the number of vulnerable nodes found by Algorithm 4 in earlier rounds. The result matches the expectation, where in total 16 nodes were identified in round 8 that spread into the full state at the input of the last Sbox. This is aligned with the optimal attack in [TM09]. However, it is also important to stress that in round 7, there are 64 nodes allow a fault attack, each of them spreading into 4 bytes in the last Sbox input. It means that rounds $7, 8, 9, 10$ have to

[7] By collision we understand reaching the exploitable node (found by TADA) by propagating through two different paths in the DFG.

**Algorithm 4:** The algorithm for finding vulnerable nodes in earlier rounds.

```
Input  : DFG: customized data flow graph corresponding to F; a: target node; b:
         exploitable node found by TADA.
Output : P : set of parent nodes of b which are vulnerable.
1  for p: parents of b do
2      if a ≠ b and p is a parent of b then
3          continue;
4      isCollision := false;
5      for x: parents of b and children of p do
6          for y: parents of b and children of p; x ≠ y do
7              if x is not a parent of y and y is not a parent of x then
8                  isCollision := true;
9                  break;
10     if not isCollision then
11         P.add(p);
12 return P;
```

Table 6: Results of running Algorithm 4 on exploitable nodes found in Section 5. $R$ denotes the number of rounds for each cipher.

| Cipher implementation | SIMON | SPECK | AES | PRIDE |
|---|---|---|---|---|
| Earliest round attacked | $R-2$ | $R-3$ | $R-3$ | $R-3$ |

be countermeasure-protected, saving over 60% resources when considering the minimal countermeasure (the last round does not contain the `MixColumns` operation and therefore, is significantly smaller). The results for the three remaining cipher implementations are provided in Appendix E.

# 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 SPECK, and PRIDE that were not published before. We also provided evaluation on AES and SIMON, where TADA was able to find existing DFA attack published in literature.

In the future, 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. Another line of interesting future work would be to extend the analysis to different ISAs. This would allow to compare the same code written in C, compiled for different architectures, providing insights on how the vulnerabilities differentiate among various ISAs.

This research was supported (in part) by following projects: NRF2016NCR-NCR002-026 (Smart Binary-Level Vulnerability Assessment For Cyber-Attack Prevention); NRF2018NCR-NSOE003-0001 (National Satellite Of Excellence In Trustworthy Software Systems); NRF2014NCR-NCR001-030 (Securify: A Compositional Approach Of Building Security Verified System).

Table 7: Results of analysing AES-128 implementation using Algorithm 4.

| Round | 7 | 8 | | | 9 | | | 10 |
|---|---|---|---|---|---|---|---|---|
| # of vulnerable nodes | 64 | 64 | 48 | 16 | 64 | 48 | 16 | 16 |
| Affects # exploitable nodes | 4 | 4 | 8 | 16 | 1 | 2 | 4 | 1 |

## References

[ABPS14] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. Differential fault analysis for block ciphers: An automated conservative analysis. In *Proceedings of the 7th International Conference on Security of Information and Networks*, SIN '14, New York, USA, 2014. ACM.

[ADK+14] Martin R Albrecht, Benedikt Driessen, Elif Bilge Kavun, Gregor Leander, Christof Paar, and Tolga Yalçın. Block ciphers–focus on the linear layer (feat. pride). In *International Cryptology Conference*, pages 57–76. Springer, 2014.

[BCC+14] Julien Bringer, Claude Carlet, HervÃľ Chabanne, Sylvain Guilley, and Houssem Maghrebi. 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.

[BDF+14] Gilles Barthe, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Jean-Christophe Zapalowicz. Synthesis of fault attacks on cryptographic implementations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1016–1027. ACM, 2014.

[BECN+06] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[Ber08] Daniel J Bernstein. Chacha, a variant of salsa20. In *Workshop Record of SASC*, volume 8, 2008.

[BGV17] Arthur Beckers, Benedikt Gierlichs, and Ingrid Verbauwhede. Fault analysis of the chacha and salsa families of stream ciphers. *Lecture Notes in Computer Science*, 2017.

[BH16] Jakub Breier and Xiaolu Hou. 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.

[BHB19] Jakub Breier, Xiaolu Hou, and Shivam Bhasin, editors. *Automated Methods in Cryptographic Fault Analysis*. Springer, 1st edition, Mar 2019.

[BHL18] Jakub Breier, Xiaolu Hou, and Yang Liu. Fault attacks made easy: Differential fault analysis automation on assembly code. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):96–122, 2018.

[BJC15] Jakub Breier, Dirmanto Jap, and Chien-Ning Chen. Laser profiling for the back-side fault attacks: With a practical laser skip instruction attack on aes. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, CPSS '15, pages 99–103, New York, NY, USA, 2015. ACM.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. 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.

[BM16] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious case of rowhammer: flipping secret exponent bits using timing analysis. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 602–624. Springer, 2016.

[BP17] Alex Biryukov and Leo Perrin. State of the art in lightweight symmetric cryptography. Cryptology ePrint Archive, Report 2017/511, 2017.

[BS91] Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. In *Advances in Cryptology-CRYPTO*, volume 90, pages 2–21. Springer, 1991.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology - CRYPTO '97*, pages 513–525. Springer, 1997.

[BTCS+15] R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith, and L. Wingers. The simon and speck lightweight block ciphers. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[CLMFT14] Franck Courbon, Philippe Loubet-Moundi, Jacques JA Fournier, and Assia Tria. Adjusting laser injections for fully controlled faults. In *International workshop on constructive side-channel analysis and secure design*, pages 229–242. Springer, 2014.

[DFL11] Patrick Derbez, Pierre-Alain Fouque, and Delphine Leresteux. Meet-in-the-middle and impossible differential fault analysis on aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 274–291. Springer, 2011.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. 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*, pages 337–340, 2008.

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[DMM+13] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre, and Assia Tria. Electromagnetic glitch on the aes round counter. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design*, pages 17–31, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[DPP+16] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. Fissc: A fault injection and simulation secure collection. In *Computer Safety, Reliability, and Security: 35th International Conference, SAFECOMP 2016, Trondheim, Norway*, pages 3–11. Springer, 2016.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[GBH+16] M. Gay, J. Burchard, J. Horacek, A.S.M. Ekossono, T. Schubert, B. Becker, I. Polian, and M Kreuzer. Small scale aes toolbox: Algebraic and propositional formulas, circuit implementations and fault equations. FCTRU, 2016. http://hdl.handle.net/2117/99210.

[Gir05] Christophe Giraud. Dfa on aes. In *Proceedings of the 4th International Conference on Advanced Encryption Standard*, AES'04, pages 27–41. Springer, 2005.

[GST12] Benedikt Gierlichs, Jörn-Marc Schmidt, and Michael Tunstall. Infective computation and dummy rounds: Fault protection for block ciphers without check-before-output. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, pages 305–321, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[HZFW15] Yuming Huo, Fan Zhang, Xiutao Feng, and Li-Ping Wang. Improved differential fault attack on the block cipher speck. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2015 Workshop on*, pages 28–34. IEEE, 2015.

[JB15] Dirmanto Jap and Jakub Breier. Differential fault attack on lea. In Ismail Khalil, Erich Neuhold, A Min Tjoa, Li Da Xu, and Ilsun You, editors, *Information and Communication Technology*, pages 265–274, Cham, 2015. Springer International Publishing.

[Jea16] Jérémy Jean. TikZ for Cryptographers. https://www.iacr.org/authors/tikz/, 2016.

[JLSH13] Kitae Jeong, Yuseop Lee, Jaechul Sung, and Seokhie Hong. Improved differential fault analysis on present-80/128. *International Journal of Computer Mathematics*, 90(12):2553–2563, 2013.

[JT12] Marc Joye and Michael Tunstall. *Fault Analysis in Cryptography*. Springer Publishing Company, Incorporated, 2012.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology — CRYPTO' 99: 19th Annual International Cryptology Conference, California, USA*, pages 388–397. Springer, 1999.

[Koc96] Paul C. Kocher. 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, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[KPB+17] SV Dilip Kumar, Sikhar Patranabis, Jakub Breier, Debdeep Mukhopadhyay, Shivam Bhasin, Anupam Chattopadhyay, and Anubhab Baksi. 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*, 2017.

[KRH17] Punit Khanna, Chester Rebeiro, and Aritra Hazra. XFC: A Framework for eXploitable Fault Characterization in Block Ciphers. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 8:1–8:6. ACM, 2017.

[LCFS17] Benjamin Lac, Anne Canteaut, Jacques J.A. Fournier, and Renaud Sirdey. Thwarting fault attacks using the internal redundancy countermeasure (irc). Cryptology ePrint Archive, Report 2017/910, 2017. http://eprint.iacr.org/2017/910.

[MDH+13] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88, Aug 2013.

[MHA+15] Baolei Mao, Wei Hu, Alric Althoff, Janarbek Matai, Jason Oberg, Dejun Mu, Timothy Sherwood, and Ryan Kastner. Quantifying timing-based information flow in cryptographic hardware. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, pages 552–559, Piscataway, NJ, USA, 2015. IEEE Press.

[PY06] Raphael C-W Phan and Sung-Ming Yen. Amplifying side-channel attacks with techniques from block cipher cryptanalysis. In *International Conference on Smart Card Research and Advanced Applications*, pages 135–150. Springer, 2006.

[PYGS16] Conor Patrick, Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. Lightweight fault attack resistance in software using intra-instruction redundancy. Cryptology ePrint Archive, Report 2016/850, 2016. http://eprint.iacr.org/2016/850.

[Riv09a] Matthieu Rivain. Differential fault analysis on des middle rounds. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009: 11th International Workshop Lausanne, Switzerland, September 6-9, 2009 Proceedings*, pages 457–469, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Riv09b] Matthieu Rivain. Differential fault analysis on des middle rounds. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 457–469. Springer, 2009.

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

[SD15] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015.

[Sha49] C. E. Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715, Oct 1949.

[SMD18] Sayandeep Saha, Debdeep Mukhopadhyay, and Pallab Dasgupta. Expfault: An automated framework for exploitable fault characterization in block ciphers. Cryptology ePrint Archive, Report 2018/295, 2018. https://eprint.iacr.org/2018/295.

[TBM14] H. Tupsamudre, S. Bisht, and D. Mukhopadhyay. Differential fault analysis on the families of simon and speck ciphers. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 40–48, Sept 2014.

[TM09] Michael Tunstall and Debdeep Mukhopadhyay. Differential fault analysis of the advanced encryption standard using a single fault. Cryptology ePrint Archive, Report 2009/575, 2009.

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

[ZZG+13] Fan Zhang, Xinjie Zhao, Shize Guo, Tao Wang, and Zhijie Shi. 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*, pages 62–79. Springer, 2013.

# A Attack details for $\mathcal{F}_{ex}$

Here we present more details output by TADA for $\mathcal{F}_{ex}$ (in Table 2). Table 8 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 7 shows the updated DFG after the attack on instruction 4. We can see that all the nodes are known now.

Table 8: Summary of TADA output for DFA attacks on $\mathcal{F}_{ex}$ in Table 2 (values of $x, \texttt{out}, \texttt{var}_0, \texttt{var}_1$ are in hexadecimal format)

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

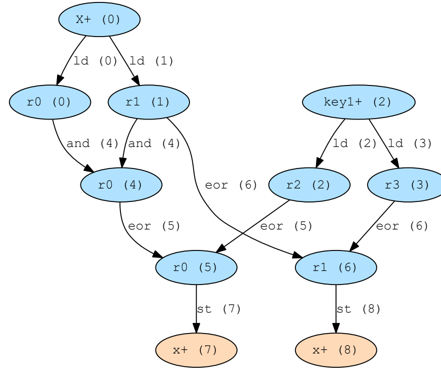


Figure 7: Updated graph generated by TADA after two successful attacks on vulnerable instruction 4 of example cipher $\mathcal{F}_{ex}$ in Table 2.

# B Algorithm for attacking table lookup

In this part, we detail the algorithm for attacking Sbox implemented as a table lookup.

**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 := φ_1 ∧ φ_2 ∧ φ_3 ∧ φ_4 ∧ φ_5 ∧ φ_6 ∧ φ_7 ∧ φ_8;
12         V := all variables in ψ;
13         V′ := V \ {δ_1, ..., δ_8, Δ_1, ..., Δ_8};
14         C_0 := ∀v ∈ V′ : ψ ⇒ a[k] = 0;
15         C_1 := ∀v ∈ V′ : ψ ⇒ a[k] = 1;
16         if (ψ_δ ∧ (∀{Δ_1, ..., Δ_8} : C_0 ∨ C_1)) is satisfiable then
17             counter++;
18             save to file (k, x);
19     if counter=8 then
20         return true;
21 return false;
```

The algorithm aims to construct the 8 pairs $(\delta_1, \Delta_1), (\delta_2, \Delta_2), \ldots, (\delta_8, \Delta_8)$, where $\delta_i (1 \le i \le 8)$ are 8 different fault masks that satisfy the constraint given in Equation (3), and $\Delta_j (1 \le j \le 8)$ denote the corresponding output masks. First we identify the variables that change when input mask $\delta$ changes and store them in `list` (lines $1-4$). Next we make 8 copies of $\phi$ (line 9). They are identical to $\phi$ except for the variables in `list`, which are replaced by 8 different variables in each of the 8 copies. For example, $\delta$ is replaced by $\delta_1, \delta_2, \ldots, \delta_8$ in $\phi_1, \phi_2, \ldots, \phi_8$ respectively. If the attack on the $k$th 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 $k$th 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 $19-20$).

## C PRIDE Sbox implementation

The equations for Sbox of PRIDE [ADK+14] are as follows:

$$\begin{aligned} A &= c \oplus (a \& b) & (4) \\ B &= d \oplus (b \& c) & (5) \\ C &= a \oplus (A \& B) & (6) \\ D &= b \oplus (B \& C), & (7) \end{aligned}$$

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

# D Attack details

In this part, we provide the attack details on cipher implementations that were chosen for TADA evaluation in Section 5. 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 9 shows the attack on SIMON, Table 10 shows attack on SPECK (see Remark 4), additionally to one described in Section 5. Table 11 provides attack details on AES, and finally Table 12 details attack on PRIDE. Here "`keyx[y]`" refers to the $(y+1)$th byte of round key in round $x$.

Table 9: 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 |
|---|---|---|
| 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 10: 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 | 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 11: 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 |
|---|---|---|
| 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 12: 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.

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

# E Further results from Algorithm 4

This part provides the remaining results obtained by running Algorithm 4 from Section 6. More specifically, Tables 13, 14, and 15 provide results for SPECK, SIMON, and PRIDE, respectively.

Table 13: Results for SPECK32/64 implementation.

| Round | 19 | | 20 | | 21 | | | 22 |
|---|---|---|---|---|---|---|---|---|
| # of vulnerable nodes | 9 | 2 | 12 | 7 | 9 | 9 | 1 | 13 |
| Affects # exploitable nodes | 1 | 2 | 1 | 2 | 1 | 2 | 3 | 1 |

Table 14: Results for SIMON32/64 implementation.

| Round | 30 | | | 31 | | | 32 | | |
|---|---|---|---|---|---|---|---|---|---|
| # of vulnerable nodes | 5 | 9 | 4 | 10 | 11 | 5 | 5 | 2 | 1 |
| Affects # exploitable nodes | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |

Table 15: Results for PRIDE implementation.

| Round | 17 | 18 | | | 19 | | | 20 |
|---|---|---|---|---|---|---|---|---|
| # of vulnerable nodes | 4 | 13 | 5 | 5 | 12 | 8 | 4 | 6 |
| Affects # exploitable nodes | 1 | 1 | 2 | 3 | 1 | 2 | 3 | 1 |