Fully Automated Differential Fault Analysis on Software Implementations of Cryptographic Algorithms

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

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

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

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

1 INTRODUCTION

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

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

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

ABSTRACT

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

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

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

KEYWORDS

differential fault analysis, cryptographic fault attacks, automation, assembly
When it comes to the attacks on assembly level, there are not many works in this field, since each implementation is unique and a specific attack on one implementation cannot be generalized to other implementations. However, these implementations can often contain DFA related vulnerabilities that are not visible on the first sight and cannot be identified by simply observing the cipher design [15]. Also, one has to take into account that the number of faults required by DFA on cipher design might be different than when attacking the implementation, making the high-level DFA estimates imprecise. For example, in [28], the authors claim they can break SPECK cipher with only 5 ~ 8 faults, but that is only if the whole cipher state is considered as one large variable. If we have an 8-bit implementation, this number would be 4× bigger in the case of SPECK32/64 and 16× bigger in the case of SPECK128/256.

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

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

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a ∧ b</th>
<th>a ⊕ b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

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

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

We design and implement **TADA – Tool for Automated DFA on Assembly**. An overview of TADA is shown in Figure 1. TADA reads an assembly code from a text file and creates a customized Data Flow Graph (DFG) that records the relations between the variables and identifies the non-linear operations used in the algorithm. It calculates the nodes that can be directly identified from the known data (ciphertext, constants). Then it finds instructions vulnerable to DFA by analyzing the graph and outputs subgraphs and DFA
equations for each of these instructions. DFA equations are passed to SMT solver to analyze. In case the instruction can be attacked by bit flip(s), the attack method is recorded and the graph is updated to capture the result of this attack. Then TADA continues to find next vulnerable instruction. TADA stops either when the correct number of round keys is recovered as required by the user or when no more vulnerable instructions can be found.

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

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

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

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

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

## 2 BACKGROUND

### 2.1 Symmetric Block Cipher

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

### 2.2 Differential Fault Analysis

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

As an example, let us consider the operation that takes \( a \) and \( b \) as inputs and outputs the result \( c = a \oplus b \), where \( a, b \in \{0, 1\} \) and \( \oplus \) is the bitwise XOR 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 \( d = a \& b \), where \( \& \) is 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 that whenever \( b \) is flipped, \( d \) will also be flipped.

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

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

DFA exploits non-linear functions of the cipher. It often works with just a single faulty and correct ciphertext pair from the encryption [47]. The attacker makes use of two sets of equations: one set that describes the correct execution of the encryption; one set corresponds to the faulted encryption process. For a simple example, let us consider the program that takes two binary inputs \( a, b \in \{0, 1\} \), calculates \( c = a \& b \) and outputs \( c \). The equation corresponding to the correct execution is \( c = a \& b \). If a fault is injected in \( b \) such that \( b \) is flipped, the equation corresponding to the faulted execution would be \( c' = a \& b' \), where \( b' = b \oplus \delta \) and \( \delta = 1 \). By calculating the difference of the output \( \Delta = c \oplus c' \), the attacker can get the value in \( a : a = 0 \) if \( \Delta = 0 \); \( a = 1 \) if \( \Delta = 1 \). The two sets of equations corresponding to correct and faulted executions are referred to as DFA equations. The change in \( b \), denoted by \( \delta \), 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, then it depends on user decision whether the
attack is carried out further on earlier rounds.

2.3 Satisfiability Modulo Theories
Satisfiability Modulo Theories (SMT) [18] is concerned with de-
ciding 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 veri-
fication, many problems can be naturally reduced to SMT problems
and SMT solvers have been used as back-end engines in many tools
for software testing, analysis and verification. The SMT solver we
use in TADA is Z3 [19].

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

\[
(c = a \& b) \land (c' = a \& b') \land (b' = b \oplus \delta) \land (\Delta = c \oplus c').
\]

To find 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. \( \forall V \setminus \{ \delta \} : ( (\psi \land \Delta = 1) \Rightarrow a = 1 ) \land ( (\psi \land \Delta = 0) \Rightarrow a = 0 ) \)
2. \( \forall V \setminus \{ \delta \} : ( (\psi \land \Delta = 1) \Rightarrow a = 0 ) \land ( (\psi \land \Delta = 0) \Rightarrow a = 1 ) \)

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 ask 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.

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

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

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

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

As most DFA attacks, we assume known ciphertext attack with-
out the knowledge of the plaintext.

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

In our work, we focus on these operations and moreover, we
present a novel attack on multiplication with a constant. To the
best of our knowledge, this is the first attack on multiplication used
in a cryptographic implementation. Multiplications are not used in
symmetric block cipher designs, however they can be efficient for
performing logical bit shifts. For example, in the implementation
of SPECK that we analyzed, shift by 3 bits to the left was done by
multiplication with a constant value of 0x88.

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

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

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c = a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\(^{1}\)Bit sets/resets, although being more precise than bit flips, are used for other methods,
such as ineffective fault analysis and are out of scope of DFA.
We inject a fault $\delta$ in $b$ and we have the following equations:

<table>
<thead>
<tr>
<th>DFA equations</th>
<th>Fault mask</th>
<th>Output mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = a \oplus b$</td>
<td>$c' = a \oplus b'$</td>
<td>$b' = b \oplus \delta$</td>
</tr>
<tr>
<td>$\Delta = c \oplus c'$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

$$\Delta = 1 \implies a = 0; \quad \Delta = 0 \implies a = 1.$$  (1)

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

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

**Addition.** For addition, we have

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$c = a + b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

We inject a fault $\delta$ in $b$ and we have the following equations:

<table>
<thead>
<tr>
<th>DFA equations</th>
<th>Fault mask</th>
<th>Output mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = a + b \oplus \delta$</td>
<td>$c' = a + b' \oplus \delta$</td>
<td>$b' = b \oplus \delta$</td>
</tr>
<tr>
<td>$\Delta = c \oplus c'$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 out $= 11$, $\text{var}_0 = 01$, $\text{var}_1 = 11$, then $\Delta \& \text{out} = \text{either} \text{var}_0$ or $\text{var}_1$, and $\Delta \& \text{out} = \text{var}_0 \implies a = 0; \quad \Delta \& \text{out} = \text{var}_1 \implies a = 1$.

**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 $a, b, carry \in \{0, 1\}$, calculates $c = a + b + carry$ and outputs $c$ in binary format. We have:

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>carry</th>
<th>$c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>010</td>
</tr>
</tbody>
</table>

We inject a fault $\delta$ in $b$ and we have the following equations:

<table>
<thead>
<tr>
<th>DFA equations</th>
<th>Fault mask</th>
<th>Output mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = a \times b \oplus \delta$</td>
<td>$c' = a \times b' \oplus \delta$</td>
<td>$b' = b \oplus \delta$</td>
</tr>
<tr>
<td>$\Delta = c \oplus c'$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For $\text{carry}= 0, \Delta = 001 \implies a = 0$ and $\Delta = 011 \implies a = 1$. For $\text{carry}= 1, \Delta = 011 \implies a = 0$ and $\Delta = 001 \implies a = 1$. Let out $= 011$, $\text{var}_0 = 001$, $\text{var}_1 = 011$ if $\text{carry}= 0$ and let out $= 011$, $\text{var}_0 = 011$, $\text{var}_1 = 001$ if $\text{carry}= 1$, then $\Delta \& \text{out} = \text{var}_0$ or $\text{var}_1$ and $\Delta \& \text{out} = \text{var}_0 \implies a = 0; \quad \Delta \& \text{out} = \text{var}_1 \implies a = 1$.

Note that the choices of value for variables out, $\text{var}_0, \text{var}_1$ are not unique, taking out $= 111$, $\text{var}_0 = 001$, $\text{var}_1 = 011$ for $\text{carry}= 0$ and let out $= 010, \text{var}_0 = 010, \text{var}_1 = 000$ for $\text{carry}= 1$ also gives the same result. Furthermore, we can see that for DFA, it is necessary to consider both values of the carry bit.

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

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

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

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

<table>
<thead>
<tr>
<th>DFA equations</th>
<th>Fault mask</th>
<th>Output mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = a \times 2$</td>
<td>$c' = a \times \text{const}$</td>
<td>$\text{const} = 2 \oplus \delta$</td>
</tr>
<tr>
<td>$\Delta = c \oplus c'$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$a$</th>
<th>$c = a \times 2$</th>
<th>$a'$</th>
<th>$c' = a' \times 0$</th>
<th>$a' = a \times 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>01</td>
<td>0010</td>
<td>000</td>
<td>0001</td>
<td>0111</td>
</tr>
<tr>
<td>10</td>
<td>0100</td>
<td>100</td>
<td>0000</td>
<td>1010</td>
</tr>
<tr>
<td>11</td>
<td>1100</td>
<td>1100</td>
<td>1001</td>
<td>1101</td>
</tr>
</tbody>
</table>

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.
Table 2: Assembly implementation $f_{ex}$ for example cipher.

<table>
<thead>
<tr>
<th>#</th>
<th>Instruction</th>
<th>#</th>
<th>Instruction</th>
<th>#</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LD r0 X+</td>
<td>3</td>
<td>LD r3 key1+</td>
<td>6</td>
<td>EOR r0 r3</td>
</tr>
<tr>
<td>1</td>
<td>LD r1 X+</td>
<td>4</td>
<td>AND r0 r1</td>
<td>7</td>
<td>ST x= r8</td>
</tr>
<tr>
<td>2</td>
<td>LD r2 key1+</td>
<td>5</td>
<td>EOR r0 r2</td>
<td>8</td>
<td>ST x= r1</td>
</tr>
</tbody>
</table>

**3.3 TADA Usage**

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

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

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

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

**3.4 TADA Design**

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

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

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

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

**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.

**Vulnerable instruction.** The goal of an attack on cryptographic implementations is normally the recovery of the master key. For some ciphers, the recovery of the last round key is sufficient (e.g. AES). For other ciphers, the attacker needs more than one round key to get the master key. For example, for SPECK, SIMON, and PRIDE, the last round key and the second last round key are both needed to get the master key. In view of this, we allow a user input, Number of target round keys, which indicates how many round keys
Algorithm 1: Check if an instruction is vulnerable.

Input: f: an instruction of a program 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;

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

1. The instruction is one of the operations as described in Section 3.2. In AVR assembly, these include operations with mnemonics AND, OR, ADD, ADC, LPM, MUL, which are respectively bitwise AND, bitwise OR, addition, addition with carry, table lookup and multiplication. For multiplication, we further check if one of the input nodes is a constant.

2. For each output node of the instruction, the distance from it and each of its affected known nodes is = 0. Thus, there is only one non-linear instruction between the input nodes of this instruction and the known nodes, which is the instruction under analysis. This ensures that there is only one non-linear equation to solve. Furthermore, this enables us to derive the SMT constraints based on the generic attacking method described in Section 3.2.

3. At least one of its nodes is linearly related to a key node that stores the value of the target round key, which is the round key under analysis during the execution.

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

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

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

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

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

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

Example 3.5. Equations (in the human readable form) generated by TADA for subgraph in Figure 3 are given in Table 3 (color scheme corresponds to Figures 2, 3). The real format of the equations is in the form of SMT solver module input. The list of known nodes, which is \{r0(5), r1(1)\} 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 3.5. After each successful attack on a target node, TADA updates the known nodes in the graph.

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

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

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

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

### Algorithm 2: The analysis of one vulnerable instruction

**Input**: \( f \) a vulnerable instruction.

**Output**: a boolean variable exploit.

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

### 3.5 SMT Solver Module

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

Depending on the mnemonics of the operation, TADA executes different algorithms. Since we are considering 8-bit architecture and bit flip fault model, in case the mnemonic is not \( \text{LPM} \), TADA generates the following constraint for the fault mask:

\[
(\delta = 1) \lor (\delta = 2) \lor (\delta = 4) \lor (\delta = 8) \lor (\delta = 16) \lor (\delta = 32) \\
\lor (\delta = 64) \lor (\delta = 128)
\]

(2)

**Bitwise AND, bitwise OR.** The algorithm for attacking bitwise AND, and bitwise OR, is outlined in Algorithm 3. For each known node \( x \), TADA generates an equation for output mask \( \Delta \) (line 2). Then it tries to attack each bit of target node \( a \) (line 4). Line 5 specifies that for some variables out, \( \text{var}_0 \) and \( \text{var}_1 \), the output mask \( \Delta \) has one of the two patterns: \( \Delta \land \text{out} = \text{var}_0 \lor \Delta \land \text{out} = \text{var}_1 \). 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 out, \( \delta \), \( \text{var}_0 \) and \( \text{var}_1 \) such that the above mentioned constraints are all satisfiable. In case it is satisfiable, the 6-tuple \( (k, \delta, x, \text{out}, \text{var}_0, \text{var}_1) \) 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 \( \Delta \land \text{out} = \text{var}_0 \) then \( a[k] = 0 \) and if \( \Delta \land \text{out} = \text{var}_1 \) then \( a[k] = 1 \).
Algorithm 3: 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 known 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 $b$.

```plaintext
for $x$: 5 do
  $\phi := \psi \land (\Delta = x \oplus x')$;
  counter = 0;
  for $k = 0, 1, \ldots , 7$ do
    $\psi_1 := \phi \Rightarrow ((\Delta \land \text{out} = \text{val}_0) \lor (\Delta \land \text{out} = \text{val}_1))$;
    $\psi_2 := (\phi \land \Delta \land \text{out} = \text{val}_0) \Rightarrow a[k] = 0$;
    $\psi_3 := (\phi \land \Delta \land \text{out} = \text{val}_1) \Rightarrow a[k] = 1$;
    $\Phi := \psi_\delta \land \psi_1 \land \psi_2 \land \psi_3$;
    $V := \text{all variables involved in } \Phi$;
    if $(\forall V \setminus \{\text{out}, \delta, \text{val}_0, \text{val}_1\} : \Phi)$ is satisfiable
    save to file $(x, \delta, x, \text{out}, \text{val}_0, \text{val}_1)$;
    counter++;
  if counter = 8 then
    return true;
return false;
```

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

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

**Remark 3.** The files output by TADA indicate that for target node $r\{1\}$ and vulnerable node $r\{0\}$, when the known node $r\{k\} (k)$ 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 $r\{k\}$. Thus, for each pair of target and vulnerable node, we need to iterate through all the known nodes that are involved in the DFA equations. Only when none of the known nodes can help us to find an attack, we consider the attack fails.

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

Algorithm 4: The algorithm for attacking ADD.

**Input**: $\psi$: DFA equations and the equation for fault mask $b$; $\psi_\delta$: constraint for fault mask $\delta$ as in Equation (2).

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

```plaintext
for $x, y \in S, x \neq y$ do
  counter= 0;
  $\phi := \psi \land (\Delta = y||x \oplus y'||x')$;
  $\psi_1 := \phi \Rightarrow ((\Delta \land \text{out} = \text{val}_0) \lor (\Delta \land \text{out} = \text{val}_1))$;
  $\psi_2 := (\phi \land \Delta \land \text{out} = \text{val}_0) \Rightarrow a[0] = 0$;
  $\psi_3 := (\phi \land \Delta \land \text{out} = \text{val}_1) \Rightarrow a[0] = 1$;
  $\Phi := \psi_\delta \land \psi_1 \land \psi_2 \land \psi_3$;
  $V := \text{all variables involved in } \Phi$;
  if $(\forall V \setminus \{\text{out}, \delta, \text{val}_0, \text{val}_1\} : \Phi)$ is satisfiable
  save to file $(0, \delta, x, y, \text{out}, \text{val}_0, \text{val}_1)$;
  counter++;
  //the carry from the first k bits = 0;
  $\psi_0 := (a[k - 1, 0] \land b[k - 1, 0]) \land \delta = 0$;
  $\phi := \psi_0 \land \psi \land (\Delta = y||x \oplus y'||x')$;
  $\psi_1 := \phi \Rightarrow ((\Delta \land \text{out} = \text{val}_0) \lor (\Delta \land \text{out} = \text{val}_1))$;
  $\psi_2 := (\phi \land \Delta \land \text{out} = \text{val}_0) \Rightarrow a[k] = 0$;
  $\psi_3 := (\phi \land \Delta \land \text{out} = \text{val}_1) \Rightarrow a[k] = 1$;
  $\Phi_0 := \psi_\delta \land \psi_1 \land \psi_2 \land \psi_3$;
  $V := \text{all variables involved in } \Phi_0$;
  if $(\forall V \setminus \{\text{out}, \delta, \text{val}_0, \text{val}_1\} : \Phi_0)$ is satisfiable
  let output0 = $(k, 0, \delta, x, y, \text{out}, \text{val}_0, \text{val}_1)$;
  else
  break;
  //the carry from the first k bits = 1;
  $\psi_1 := (a[k - 1, 0] \land b[k - 1, 0]) \land \delta = 1$;
  $\phi := \psi_1 \land \psi \land (\Delta = y||x \oplus y'||x')$;
  $\psi_1 := \phi \Rightarrow ((\Delta \land \text{out} = \text{val}_0) \lor (\Delta \land \text{out} = \text{val}_1))$;
  $\psi_2 := (\phi \land \Delta \land \text{out} = \text{val}_0) \Rightarrow a[k] = 0$;
  $\psi_3 := (\phi \land \Delta \land \text{out} = \text{val}_1) \Rightarrow a[k] = 1$;
  $\Phi_1 := \psi_\delta \land \psi_1 \land \psi_2 \land \psi_3$;
  $V := \text{all variables involved in } \Phi_1$;
  if $(\forall V \setminus \{\text{out}, \delta, \text{val}_0, \text{val}_1\} : \Phi_1)$ is satisfiable
  let output1 = $(k, 1, \delta, x, y, \text{out}, \text{val}_0, \text{val}_1)$;
  else
  break;
  if counter= 8 then
    return true;
return false;
```
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 \( \text{ADD} \) contains an extra step in SMT solver module such that it only returns true for the attack when both inputs of addition can be retrieved or when one can be retrieved and the other is known.

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

**Table lookup.** If the vulnerable instruction corresponds to Shox table lookup, the attack follows Algorithm 5. The algorithm aims to construct the 8 pairs \( \left( \delta_1, \Delta_1 \right), \left( \delta_2, \Delta_2 \right), \ldots, \left( \delta_8, \Delta_8 \right) \), where \( \delta_j, 1 \leq j \leq 8 \) are 8 different fault masks that satisfy the constraint given in Equation (3), and \( \Delta_j, 1 \leq j \leq 8 \) denote the corresponding output masks. First we identify the variables that change when input mask \( \delta \) changes and store them in \( \text{list} \) (lines 1–4). Next we make 8 copies of \( \phi \) (line 9). They are identical to \( \phi \) except for the variables in \( \text{list} \), 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).

\[
\left( \delta_1 = 1 \right) \land \left( \delta_2 = 2 \right) \land \left( \delta_3 = 4 \right) \land \left( \delta_4 = 8 \right) \land \left( \delta_5 = 16 \right) \land \left( \delta_6 = 32 \right) \land \left( \delta_7 = 64 \right) \land \left( \delta_8 = 128 \right). \tag{3}
\]

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).

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

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

\[
\text{for } x, y \in S, x \neq y \text{ do}
\]

\[
\phi := \psi \land (\Lambda = y \mid x \oplus y' \mid x')
\]

Accordingly, the output to a file (line 11) is changed to \((k, \delta, x, y, \text{out}, \text{var}_0, \text{var}_1)\), 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 \( \Lambda = y \mid x \oplus y' \mid x' \). If \( \Lambda \) & \text{out} = \text{var}_0, a[k] = 0 \) and if \( \Lambda \) & \text{out} = \text{var}_1, a[k] = 1 \).

### Algorithm 5: The algorithm for attacking table lookup.

**Input:** \( \psi \): DFA equations and the equation for fault mask; \( b \): vulnerable node; \( a \): target node; \( S \): the list of known nodes in the DFA equations; \( \psi_b \): constraint for fault masks as in Equation (3).

**Output:** boolean: true if \( a \) can be obtained by attacking \( b \).

1. \( \text{list} = (\delta, \Lambda); \)
2. for \( c \) variables in \( \text{DFA} \) do
3. if \( b \) affects \( c \) then
   4. \( \text{list}.\text{add}(c); \)
5. for \( x \) in \( S \) do
6. \( \text{counter} = 0; \)
7. for \( k = 0, 1, \ldots, 7 \) do
8. \( \phi := \psi \land (\Lambda = x \oplus x'); \)
9. make 8 copies of \( \phi \) w.r.t. \( \text{list} \);
10. let \( \phi_1 \) denote the \( \delta \)th copy of \( \phi \);
11. \( \psi := \phi_1 \land \phi_2 \land \phi_3 \land \phi_4 \land \phi_5 \land \phi_6 \land \phi_7 \land \phi_8; \)
12. \( V := \text{all variables in } \psi; \)
13. \( V' := V \setminus \{\delta_1, \ldots, \delta_8, \Delta_1, \ldots, \Delta_8\}; \)
14. \( C_0 := V \setminus \psi; \)
15. \( C_1 := \psi; \)
16. if \( (\psi_b \land (V \setminus \{\delta_1, \ldots, \Delta_8\} \land C_0 \lor C_1)) \) is satisfiable then
   17. \( \text{counter}++; \)
   18. save to file \((k, x)\);
19. if \( \text{counter} = 8 \) then
20. return true;
21. return false;

### Table 4: Evaluation by TADA on different implementations.

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

Table 4

have not been presented yet, since it is not possible to identify such attack from the cipher-level. For AES, a previously published DFA [26] was found.

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

**SIMON.** SIMON is an ultra-lightweight block cipher, based on the balanced Feistel structure. It supports block sizes from 32 up to 128 bits, with key sizes ranging from 64 to 256 bits. Number of rounds depends on the key size, and ranges between 32 and 72. In each round, it uses three operations — bitwise AND, bitwise shift, and XOR. Schematic of SIMON is depicted in Figure 5.
For SIMON implementation\textsuperscript{3}, TADA found 8 vulnerable bitwise AND instructions that are all exploitable. The last round key and the second last round key were recovered. This attack is implementation specific. The flaw in this implementation is that the key xor operation was implemented before the xor of left and right side of the intermediate values – opposite as the specification. Thus, presenting a DFA vulnerability which cannot be seen from the cipher design level.

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

For SPECK implementation\textsuperscript{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 “key\textsubscript{y}[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 2–7th bits. If a fault mask 0x08 is injected in the constant 8, the operation will be changed to multiplication by 0. We emphasize that such attack cannot be seen from a cipher design level, which only shows the rotations, but leaves it to implementer on how to realize them. Normally, rotation is a linear operation and therefore, cannot be attacked by DFA – the input and output difference would remain the same, it would only change the position, giving no information on the processed data. We note that multiplication by 8 is not the only way to implement 3-bit rotation. Only after the implementation analysis by TADA, one can observe the vulnerability caused by using the multiplication by a constant.

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

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

For AES implementation taken from Ecrypt II public repository\textsuperscript{5}, TADA found the same attack as in [26]. 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\textsuperscript{5}, the Sbox is implemented in algebraic form, therefore, unlike in AES implementation, no table lookup is necessary. Details on this implementation are shown in Appendix B. Schematic of PRIDE is depicted in Figure 8.

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

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

### 5 RELATED WORK

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

\textsuperscript{3}https://github.com/openluopworld/simon_speck_on_avr/tree/master/AVR

\textsuperscript{4}https://perso.uclouvain.be/ftstandae/source_codes/lightweight_ciphers/source/AES.asm

\textsuperscript{5}https://github.com/FreeDisciplina/BlockCiphersOnAVR/tree/master/PRIDE_64_128_AVR
Assembly Analysis. To the best of our knowledge, the only work on automation of DFA in assembly implementation is [15], where the authors automated the search for vulnerable instructions according to user input. However, whether the found instruction is really exploitable and how to exploit it has to be done manually. Therefore, the developed tool outputs larger number of vulnerable instructions while only a small subset might be actually exploitable. Moreover, the vulnerability criteria for finding these instructions have to be defined by the user.

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

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

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

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

6 DISCUSSION

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

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

7 CONCLUSIONS

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

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

REFERENCES

target node | vulnerable node | $(k, \delta, x, out, val_0, val_1)$ | retrieved key byte
---|---|---|---
$r(1)$ | $r(8)$ | $(0, 1, r_7(5), 0, x_1, 0, 0, 0)$ | $key(1)$
$r(1)$ | $r(8)$ | $(1, 2, r_7(5), 0, x_2, 0, 0, 0)$ | $key(1)$
$r(1)$ | $r(8)$ | $(2, 4, r_7(5), 0, 0, 4, 0, 0)$ | $key(1)$
$r(1)$ | $r(8)$ | $(3, 8, r_7(5), 0, 0, 0, 0, 0)$ | $key(1)$
$r(1)$ | $r(8)$ | $(4, 16, r_7(5), 0, 0, 0, 0, 0)$ | $key(1)$
$r(1)$ | $r(8)$ | $(5, 32, r_7(5), 0, 0, 0, 0, 0)$ | $key(1)$
$r(1)$ | $r(8)$ | $(6, 64, r_7(5), 0, 0, 0, 0, 0)$ | $key(1)$
$r(1)$ | $r(8)$ | $(7, 128, r_7(5), 0, 0, 0, 0, 0)$ | $key(1)$

Table 6: Summary of TADA output for DFA attacks on $F_{ex}$ in Table 2 (values of $x, out, var_0, var_1$ are in hexadecimal format)

---

A ATTACK DETAILS FOR $F_{ex}$

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

---

B PRIDE SBOX IMPLEMENTATION

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

\[
A = c \oplus (a \& b) \\
B = d \oplus (b \& c) \\
C = a \oplus (A \& B) \\
D = b \oplus (B \& C),
\]

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

---

C ATTACK DETAILS

In this part, we provide the attack details on cipher implementations that were chosen for TADA evaluation in Section 4. Each of the tables provides the information on which instructions were identified as vulnerable and what was the attack flow leading to secret key retrieval.
Table 7 shows the attack on SIMON, Table 8 shows attack on SPECK (see Remark 4), additionally to one described in Section 4. Table 9 provides attack details on AES, and finally Table 10 details attack on PRIDE. Here “keyx[y]” refers to the \((y + 1)th\) byte of round key in round \(x\).

### Table 7: Attack on SIMON found by TADA

<table>
<thead>
<tr>
<th>No.</th>
<th># of known nodes</th>
<th>key nodes recovered</th>
</tr>
</thead>
<tbody>
<tr>
<td>1136</td>
<td>73</td>
<td>-</td>
</tr>
<tr>
<td>1137</td>
<td>83</td>
<td>-</td>
</tr>
<tr>
<td>1138</td>
<td>92</td>
<td>-</td>
</tr>
<tr>
<td>1139</td>
<td>106</td>
<td>-</td>
</tr>
<tr>
<td>1174</td>
<td>115</td>
<td>key32[3]</td>
</tr>
<tr>
<td>1175</td>
<td>129</td>
<td>key31[1], key32[0]</td>
</tr>
<tr>
<td>1176</td>
<td>142</td>
<td>key31[2], key32[1]</td>
</tr>
<tr>
<td>1177</td>
<td>162</td>
<td>key31[0], key31[3], key32[2]</td>
</tr>
</tbody>
</table>

### Table 8: Attack on SPECK found by TADA (considering obtaining 7 bits as successful attack)

<table>
<thead>
<tr>
<th>No.</th>
<th># of known nodes</th>
<th>key nodes recovered</th>
</tr>
</thead>
<tbody>
<tr>
<td>606</td>
<td>43</td>
<td>key22[0]</td>
</tr>
<tr>
<td>607</td>
<td>52</td>
<td>key22[1]</td>
</tr>
<tr>
<td>608</td>
<td>63</td>
<td>key22[2]</td>
</tr>
<tr>
<td>609</td>
<td>68</td>
<td>key22[3]</td>
</tr>
<tr>
<td>578</td>
<td>79</td>
<td>key21[0]</td>
</tr>
<tr>
<td>579</td>
<td>88</td>
<td>key21[1]</td>
</tr>
<tr>
<td>580</td>
<td>99</td>
<td>key21[2]</td>
</tr>
<tr>
<td>581</td>
<td>104</td>
<td>key21[3]</td>
</tr>
</tbody>
</table>

### Table 9: Attack on AES found by TADA

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

### Table 10: Attack on PRIDE found by TADA

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

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

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

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

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