# Fault Attacks Made Easy: Differential Fault Analysis Automation on Assembly Code

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

[1] Physical Analysis and Cryptographic Engineering, Temasek Laboratories
[2] School of Compter Science and Engineering

Nanyang Technological University, Singapore
jbreier@ntu.edu.sg, ho0001lu@e.ntu.edu.sg, yangliu@ntu.edu.sg

**Abstract.** Over the past decades, fault injection attacks have been extensively studied due to their capability to efficiently break cryptographic implementations. Fault injection attack models are normally determined by analyzing the cipher structure and finding exploitable spots in non-linear and permutation layers. However, this level of abstraction is often too high to distinguish vulnerable parts of software implementations, due to specific operations and optimizations. On the other hand, manually analyzing the assembly code requires non-negligible amount of time and expertise.

In this paper, we propose an automated approach for analyzing cipher implementations in assembly. We represent the whole assembly program as a data flow graph so that the vulnerable spots can be found efficiently. Fault propagation is analyzed in a subgraph constructed from each vulnerable spot, allowing equations for Differential Fault Analysis (DFA) to be automatically generated.

We have created a tool that implements our approach: *DATAC – DFA Automation Tool for Assembly Code*. We have successfully used this tool for attacking PRESENT-80, being able to find implementation-specific vulnerabilities that can be exploited in order to recover the last round key with 16 faults. Our results show that DATAC is useful in finding attack spots that are not visible from the cipher structure, but can be easily exploited when dealing with real-world implementations.

**Keywords:** automated fault attack, software implementations, assembly code, differential fault analysis

## 1 Introduction

When it comes to attacking cryptographic algorithms, fault injection attacks are among the most serious threats, being capable of revealing the secret information by just one single disturbance in the execution [TM09, DRA16, JKP12]. Differential Fault Analysis (DFA) [BS97] has become the most commonly used fault analysis method for attacking symmetric block ciphers. It is the first method of choice when it comes to testing fault resilience of new cryptographic algorithms because of its simplicity and power to recover the secret key by a low number of faulty encryptions.

In practice, the attack always has to be mounted on a real-world device, in an implementation that is either hardware- or software-based. When we focus on software, there are many different ways to attack such implementations – one can corrupt the instruction opcodes resulting to instruction change, skip the instructions completely, flip the bits in processed constant values or register addresses, or change the values in the registers and


IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925, Vol. 2018, No. 2, pp. 96–122
DOI:10.13154/tches.v2018.i2.96-122

memories directly [BECN+06, BBKN12]. These attacks can be achieved by various fault injection methods, such as clock/voltage glitch, laser fault injection, or electromagnetic fault injection [BECN+06, BBB+12].

As a consequence, different implementations of the same encryption algorithm do not necessarily share the same vulnerabilities. Some attacks that work in theory might either not be possible, or be hard to execute in practice (e.g. precisely setting particular bits of the cipher state to some value [DP14]). On the other hand, there might be an exploitable spot in the implementation that is not visible from the specification of the encryption algorithm and can only be found by analyzing the assembly code. Up to now, the known DFA on PRESENT all aim before the execution of Sbox layer of the last round. In Section 5 we will show a DFA attack on one PRESENT implementation which aims at the end of player of the last round. The vulnerability comes from the implementation technique and is not intrinsic to the cipher.

Even though there are works on fault analysis of a cipher from the cipher design level, there is no work aiming at DFA on the assembly code level. Reason is that analyzing one particular implementation by human is time consuming and it cannot be generalized to all the implementations. Therefore, when attacking a single cipher, researchers rather aim at fault models that work universally. However, since the real attack is always done on a concrete implementation [CML+11, BJC15], it is important to have automated tools to find the vulnerabilities of specific implementations. Our work aims to contribute to the automation of DFA on assembly code.

Since DFA makes use of the data dependency between the intermediate values of the algorithm and the secret key, we represent an assembly code as a customized data flow graph in Static Single Assignment (SSA) form to record the operations (edges) and memory structures (nodes) holding the data, as well as their relations. Hence, assuming a fault is injected in one node (equivalently, the fault is injected in the instruction where this node is an output operand), we can construct the corresponding subgraphs that represent propagation of faults from this node to the ciphertext, while also showing its relation to the round key(s). Ultimately, this allows us to automatically generate differential fault analysis equations, by solving which we can mount a successful fault injection attack.

Different DFA attacks make use of different properties of the cryptosystems. The simplest attacks aim at the last round, requiring higher number of faults. More sophisticated attacks utilize properties of the cipher permutation layer, enabling them to aim at second or third last round, while lowering the number of faults. Hence, we allow the users to specify what kind of vulnerable spots to look for, which is realized as a user input variable called *output criteria*.

Our methodology was implemented in a tool named *DATAC – DFA Automation Tool for Assembly Code*, that takes an assembly code as input, and outputs subgraphs and equations for each vulnerable node according to user specified output criteria.

**Our Contribution.** In this work we design and implement an automated tool DATAC which automatically finds the vulnerable instructions with respect to DFA in an assembly implementation of cryptosystem and outputs the related DFA equations for further analysis.

DATAC does not require any specific inputs of the cryptosystem and works independently of plaintext and secret key, providing a general evaluation of the underlying implementation. To the best of our knowledge, there is no tool working on assembly level that could automate the whole process to such extent. We emphasize that DFA vulnerabilities specific to software implementations are not visible from cipher design level.

We provide a case study on PRESENT-80 cipher implementation for 8-bit AVR microcontroller that shows capabilities of DATAC by finding a new DFA attack which is implementation specific, being able to recover the last round key by 16 fault injections.

To show the usage of output criteria, we provide an analysis of SPECK 64/128 and SIMON 64/128 lightweight ciphers, as well as current industry standard, AES-128. Time

required for analysis of AES is less than a second, by using a standard laptop computer. The results show that the whole process is computationally feasible.

We would like to point out that our tool is modular and enables an easy extension to the instruction set, making it easy to evaluate implementations for different devices.

**Paper Organization.** The rest of the paper is structured as follows. Section 2 provides preliminaries for this paper. Section 3 specifies our approach, by detailing each step of the evaluation used in DATAC. Section 4 provides implementation details of DATAC. Section 5 explains an implementation specific DFA attack on PRESENT-80 found with DATAC. Section 6 provides a discussion and finally, Section 7 concludes this work and provides a motivation for future works.

# 2 Preliminaries

In this section, we first provide an overview of DFA in Section 2.1. Next, we detail several related works in Section 2.2, focusing on automated fault analysis on design level and on evaluating implementations. In Section 2.3, we explain necessary basics of intermediate representations that are related to our work. We continue with stating assumptions and scope for DATAC usage in Section 2.4. Finally Section 2.5 closes this part with formal definitions and notations that are used later in the paper.

## 2.1 Differential Fault Analysis

Differential Fault Analysis (DFA) is among the most popular fault analysis methods for analyzing symmetric block ciphers. The concept of DFA was introduced by Biham and Shamir in [BS97] in '97. The authors show that it is possible to break Data Encryption Standard by analyzing 50-200 faulty ciphertexts with a bit-flip fault model.

DFA attack consists of injecting a fault into the intermediate state of the cipher, normally during one of the last rounds. The fault then propagates, resulting into a faulty ciphertext. Difference between the original and the faulty ciphertext is then analyzed, giving the attacker information about the secret key. DFA exploits the properties of the non-linear operation that is normally used in the cryptosystem. The whole concept is similar to a classical differential cryptanalysis [BS91] of a reduced-round cipher.

Most of the block cryptosystems were shown to be vulnerable against DFA, since there are no efficient cipher designs that could prevent this analysis up to date. For further reading, one can find DFA on AES [TM09], DES [Riv09], PRESENT [BEG13], SIMON & SPECK [TBM14], etc.

## 2.2 Related Work

Agosta et al.[ABPS14] utilized an intermediate representation in order to check for single bit-flip vulnerabilities in the code to point out the exploitable parts. However, the approach aims at detecting cipher design vulnerabilities instead of low-level implementation specific vulnerabilities. That is also why instruction skip model is not considered, although it being one of the most powerful and easy to implement attacks.

Khanna et al. [KRH17] proposed XFC – a method that checks exploitable fault characteristics of block ciphers considering the DFA attack method. Their approach takes a cipher specification as an input and then indicates the fault propagation through the cipher. The main drawback of this work is its focus on a high-level cipher representation, and therefore, being unable to check the security of a particular cipher implementation.

Goubet et al. [GHEDK16] developed a framework that generates a set of equations for an SMT solver from assembly code. Then it uses this representation for evaluating robustness of countermeasures against fault injection attacks. The evaluation is based on

comparison of two code snippets: one that represents a code without any protection, and a hardened code. These snippets are then represented as a finite automata, unfolded, and analyzed. The main drawback of this work is its focus on code snippets instead of real implementations of cryptosystems and the fact that analyzing 10 lines of code requires 10.7 s. Therefore it is not feasible to analyze the full cipher in a reasonable time.

Dureuil et al. [DPdC+16] proposed an approach using fault model inference – they first determine fault models that can be achieved on a target hardware, together with probability of occurrence of these models. Based on this information, they compute a "vulnerability rate" that gives an estimate of the software robustness. The focus of this paper is to estimate time required in order to successfully inject the required fault model.

Gay et al. [GBH+16] took several hardware implementations of AES and provided their algebraic representations and translation into formulas in Conjunctive Normal Form (CNF). These can be directly used by a SAT solver to mount an algebraic fault analysis.

A different approach to fault analysis automation was presented by Endo et al. [EHH+14]. Their work does not focus on finding a DFA vulnerability. However, it automates the way to find a pre-defined vulnerable spot in a black-box implementation with a countermeasure by checking whether the cipher output matches the requirement for the attack.

Our approach analyzes the assembly code directly, by building a customized data flow graph, allowing users to tailor the requirements for vulnerabilities according to desired fault models. Thanks to this, we can identify the points of interest efficiently and design DFA equations automatically, so that only the solving part is left to the user. DATAC is also scalable and can be used for analyzing a whole cipher implementation efficiently.

## 2.3 Intermediate Representation

Compiler construction normally depends on program analysis, where statements from an abstract high-level language are translated into a binary form that is understandable by the underlying processor. This process is not done in a single step but there are several subprocesses involved, in order to optimize the resulting program. One of these steps involves creation of an intermediate form that can be represented as a directed graph. There are various intermediate forms that can represent a program. Here we detail three intermediate representations which are most related to our work.

**Data Dependency Graph.** Data dependency graph [ASKL81, CGK80] represents dependencies between (arithmetic assignment) statements that exist within program loops. For assembly code, the data dependency graph is also called instruction dependency graph, where each node corresponds to one instruction and each edge represents a dependency [DJTK10].

**Data Flow Graph.** A data flow graph is a representation of data flow in a program execution. The nodes correspond to operations. Input data of an operation are represented as inward edges of the corresponding node and output data of an operation are represented as outward edges of the node [Nie98].

**Static Single Assignment.** Static Single Assignment (SSA) form requires that each variable is assigned exactly once and every variable is defined before it is used. As an example, we can take an assignment of the form $x = exp$. Then, the left-hand side variable $x$ is replaced by using a new variable, e.g. $x_1$. After this point, any reference to $x$, before $x$ is assigned again, is replaced with $x_1$ [Bar13].

**DATAC.** In our work, we are dealing with the dependency between different memory units that can be affected by performing operations on these units. Thus, DATAC construct a graph where each node corresponds to a variable and each edge corresponds to an instruction. There is an edge $f$ from a variable $a$ to a variable $b$ if and only if $a$ is one input operand of instruction $f$ and $b$ is one output operand of $f$.

Furthermore, we consider an unrolled implementation. Hence, we can adopt the SSA form to facilitate DFA. Therefore, a graph constructed by DATAC is a customized data

flow graph in SSA form.

## 2.4 Assumptions and Scope

Obviously, there are many aspects that have to be taken into account when analyzing a cipher implementation. It would be a daunting task to make a general-purpose analyzer that could work on unrestricted space of programs and fault analysis methods. Therefore, in the following we specify the scope for the tool usage and the rationale behind the design and implementation.

- As most of the DFA proposals, our method assumes known-ciphertext model and a single fault adversary (i.e., injecting one fault per encryption).
- We assume the implementation is available to the user and he can add annotations to the assembly code for the purpose of distinguishing different rounds, round keys, ciphertext words, etc.
- Majority of DFA proposals assume either bit-flip or random byte fault models. Additionally, some works utilize a powerful instruction skip model that can change the program flow in a way that some operations are avoided completely, resulting into a trivial cryptanalysis [FXKH17]. Therefore, we assume these three fault models in our analysis.
- The analysis automates the process of DFA that focuses on finding the vulnerable spot in the program and creating the difference equations.
- The set of vulnerable nodes that are outputs from the analysis, is dependent on parameters that are set by the user. These are referred to as the *output criteria*. While we give some suggestions on tuning these so that the user can get readily exploitable outputs, in the future there might be more efficient DFA methods available that would require different parameters. By defining these as an input variable, it makes it easy in the future to adjust the tool to these new attacks without rebuilding the analysis subsystem.
- While the program outputs the difference equations, the final step of the analysis has to be done manually. One of the reasons for this is that our tool does not operate on concrete values, only on dependencies between the variables.
- For the analysis in this paper, we have chosen Atmel AVR instruction set [1]. However, for analyzing different instruction sets, only the parsing subsystem of the analyzer has to be redefined (`AsmFileReader` class stated in the class diagram in Figure 8).

We would like to point out that the analysis is done on unrolled implementations. The main reason for this is that fault attack can always exploit a jump/branch operation in a way to render the computation vulnerable to DFA [YGS+16]. For example, if a round counter is tampered with, attacker can easily change the number of rounds to 1, making it trivial to recover the key. Or, if cipher operations are implemented as macros, one can skip the jump to such macro [KPB+17].

## 2.5 Formalization of Fault Attack

**Definition 1.** We define a *program* to be an ordered sequence of assembly instructions $\mathcal{F} = (f_0, f_1, \ldots, f_{N_\mathcal{F}-1})$. $N_\mathcal{F}$ is called the *number of instructions* for the program. For each instruction $f \in \mathcal{F}$, we associate $f$ with a 4-tuple $(f^{seq}, f^{mn}, f^{io}, f^{do})$, where $f^{seq}$ is

[1] http://www.atmel.com/images/Atmel-0856-AVR-Instruction-Set-Manual.pdf

Table 1: Assembly code $\mathcal{F}_{ex}$ for a sample cipher.

| # | Instruction | # | Instruction | # | Instruction |
|---|---|---|---|---|---|
| | //round_1 | 5 | EOR r1 r3 | 10 | LD r3 key2+ |
| 0 | LD r0 X+ | 6 | ANDI r0 0x0F | 11 | EOR r0 r2 |
| 1 | LD r1 X+ | 7 | ANDI r1 0xF0 | 12 | EOR r1 r3 |
| 2 | LD r2 key1+ | 8 | OR r0 r1 | | //store_ciphertext |
| 3 | LD r3 key1+ | | //round_2 | 13 | ST x+ r0 |
| 4 | EOR r0 r2 | 9 | LD r2 key2+ | 14 | ST x+ r1 |

the sequence number and $f^{mn}$ is the mnemonic of $f$. $f^{io}$ is the set of input operands of $f$, which can be registers, constant values or pointers to memory addresses. $f^{do}$ is the set of destination (output) operands of $f$, which can be registers or pointers to memory addresses.

**Example 1.** The assembly implementation $\mathcal{F}_{ex}$ of a simple sample cipher in Table 1 has $N_{\mathcal{F}_{ex}} = 15$ instructions. Instruction $f_6 =$ `ANDI r0 0x0F` has input operands `r0` and `0x0F`, destination operand `r0`. Thus $f_6$ is associated with the 4−tuple $(6, \texttt{ANDI}, \{\texttt{r0}, \texttt{0x0F}\}, \{\texttt{r0}\})$.

We note that for an instruction $f =$ `ADD r0 r1`, the output operands of $f$ are actually `r0` and `carry`, where `carry` is a flag, usually represented by a bit in the status register of a microcontroller. The `carry` itself does not appear in the assembly code directly, however, we consider it in our analysis as a standalone operand.

Fault attack is an intentional change of the original data value into a different value. This change can either happen in a register/memory, on the data path, or directly in ALU. In general, there are two main fault models to be considered – program flow disturbances and data flow disturbances. The first one is achieved by disturbing the instruction execution process that can result in changing or skipping the instruction currently being executed. The second one is achieved either by directly changing the data values in storage units, or by changing the data on the data paths or inside ALU.

Formally, we define a fault injection in a program $\mathcal{F} = \{f_0, f_1, \ldots, f_{N_\mathcal{F}-1}\}$ to be a function $\vartheta_i : \mathcal{F} \mapsto \mathcal{F}'$, where $0 \leq i < N_\mathcal{F}$ and $\mathcal{F}'$ is a program obtained from $\mathcal{F}$ with the instruction $f_i$ being tampered. Thus $\vartheta_i$ represents a fault injection on the instruction with sequence number $i$ in $\mathcal{F}$. There are different possible fault models, we focus on following:

- **Instruction skip**: $\vartheta_i(\mathcal{F}) = \mathcal{F} \backslash f_i$, i.e. instruction $i$ is skipped.
- **Bit flip**: $\vartheta_i(\mathcal{F}) = \{f_0, f_1, \ldots, f_i, f'_{i+1}, f'_{i+2}, \ldots, f'_{N_\mathcal{F}-1}, f'_{N_\mathcal{F}}\}$ such that $f'_{j+1} = f_j$ for $i < j < N_\mathcal{F}$ and $f'_{i+1} =$ `r xor` $\Delta$, where $r \in f_i^{do} \cup f_i^{io}$ is either a destination operand or an input operand of instruction $f_i$ and $\Delta$ is a pre-defined value which is called a *fault mask*. In the case $f_i^{do} = \emptyset$, $f'_{i+1} =$ `NOP`.
- **Random byte fault**: $\vartheta_i(\mathcal{F}) = \{f_0, f_1, \ldots, f_i, f'_{i+1}, f'_{i+2}, \ldots, f'_{N_\mathcal{F}-1}, f'_{N_\mathcal{F}}\}$ such that $f'_{j+1} = f_j$ for $i < j < N_\mathcal{F}$ and $f'_{i+1} =$ `r xor` $\Delta$, where $r \in f_i^{do} \cup f_i^{io}$ and $\Delta$ is a random value. In the case $f_i^{do} = \emptyset$, $f'_{i+1} =$`NOP`.

In the rest of this paper we assume that the attacker has the knowledge of the fault model for the differential fault analysis.

# 3 Automated Assembly Code Analysis

In Section 3.1, we first provide an overview of the internal working of DATAC. Then, we explain the methodology used in DATAC in details, following the same logical flow as the actual analysis. Section 3.2 defines the specifics of our customized data flow graph. Output criteria parameters are elaborated in Section 3.3. Subgraph and equation constructions are illustrated in Sections 3.4 and 3.5, respectively.

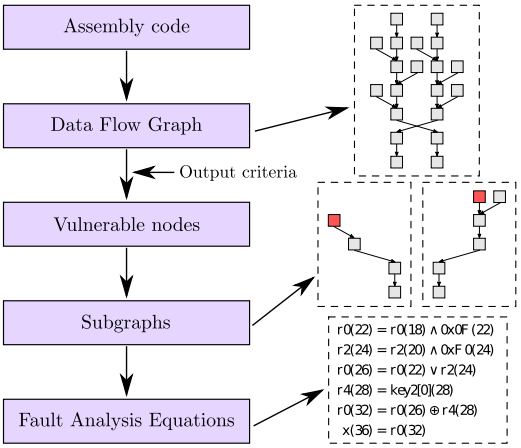



Figure 1: Our evaluation method for analyzing assembly code w.r.t. fault injection vulnerabilities.

## 3.1 Overview

The main goal of DATAC is to analyze the assembly code and find vulnerable spots w.r.t. DFA attack. This can be a challenging task, since the same instruction can be vulnerable in one part of the code, but secure in the other part, depending on the context.

For our purposes, we have to capture the following details when transforming the assembly code:

– Memory units holding the data (registers, RAM, flash, etc.) as well as direct operands (constants) – these will be represented as nodes.
– Transitions between the nodes.
– Operations (instructions) in the program – represented as edges.
– Properties of operations (linear/non-linear).
– Ability to distinguish important nodes, such as round keys and ciphertext.

Our evaluation method is depicted in Figure 1. First, an assembly code is fetched as the input. Based on its structure, a data flow graph is created. Depending on the output criteria, vulnerable nodes are identified. Then, a subgraph is created for each vulnerable node – it specifies a propagation pattern of the fault from the vulnerable node to the ciphertext. Together with it, fault analysis equations are generated – based on them, a fault attack can be executed. In the rest of this section, we provide details on how our approach works.

## 3.2 From Assembly to Data Flow Graph

Given a program $\mathcal{F} = (f_0, f_1, \ldots, f_{N_\mathcal{F}-1})$, a data flow graph is a directed graph $G_{\mathcal{F},full} = (V, E)$, where the set of nodes $V = A \cup B$ is the union of two sets of labeled nodes. $A$ consists of labeled nodes with labels "$\mathtt{x}$ $(i)$" such that $x$ is a destination operand (also called output operand) of instruction $i$. $B$ consists of labeled nodes with labels "$\mathtt{y}$ $(i)$" such that $y$ is an input operand of instruction $i$ and $y$ is not a destination operand of any instruction.

– $A = \{\text{“}\mathtt{x}\ (i)\text{”} : x \in f_i^{do} \text{ for some } 0 \leq i < N_\mathcal{F}\}$;
– $B = \{\text{“}\mathtt{y}\ (i)\text{”} : y \in f_i^{io} \text{ for some } 0 \leq i < N_\mathcal{F} \text{ and } y \notin f_j^{do} \text{ for any } 0 \leq j < N_\mathcal{F}\}$.

We draw an edge from node $a =$ "$y$ $(i)$" to node $b =$ "$x$ $(j)$" if and only if the following conditions are satisfied:

– $i \leq j$,
– $x$ is a destination operand of instruction $j$,
– $y$ is an input operand of instruction $j$,

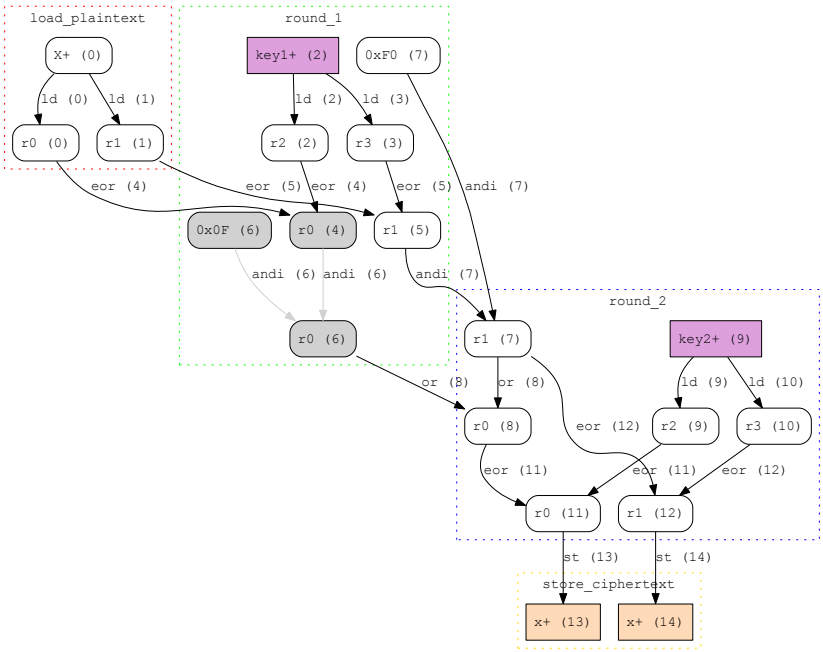



Figure 2: Data flow graph $G_{\mathcal{F}_{ex},full}$ corresponding to the assembly program $\mathcal{F}_{ex}$ in Table 1 constructed by DATAC.

– $y$ is not an output operand for any instruction between instruction $i$ and instruction $j$, which means the value in $y$ is not changed between instructions $i$ and $j$.

Formally, an edge $(a, b) \in E$ for $a =$ "`y` $(i)$", $b =$ "`x` $(j)$" $\in V$ iff $i \leq j$, $x \in f_j^{do}$, $y \in f_j^{io}$ and "`y` $(k)$" $\notin V$ $\forall i < k < j$. Furthermore, we label such an edge with "$f_j^{mn}$ $(j)$" and we say that this edge is *associated with* instruction $f_j$. We also refer to $a$ as an *input node* of $f_j$ and $b$ as an *output node* of $f_j$. Following the terminologies from graph theory, $a$ is called the *tail* of the edge $(a, b)$ and $b$ is called the *head* of $(a, b)$.

**Example 2.** The data flow graph $G_{\mathcal{F}_{ex},full}$ corresponding to the assembly program $\mathcal{F}_{ex}$ in Table 1 is shown in Figure 2. Instruction $f_6$ has input operands `r0` and `0x0F`, where `r0` is the output operand of $f_4$ and `0x0F` is not an output operand of any instruction. The output operand of $f_6$ is `r0`. Hence $f_6$ has two input nodes: "`r0` (4)", "`0x0F` (6)" and one output node "`r0` (6)" . Furthermore, $f_6$ is related to two edges in the graph, both labeled "`ANDI` (6)". Both edges have head "`r0` (6)", one with tail "`r0` (4)" and one with tail "`0x0F` (6)" (see the nodes and edges highlighted in gray).

Since we are dealing with implementations of ciphers, we highlight the round keys as well as the ciphertext in the graphs. As shown in Figure 2, the node that corresponds to round key in round $i$ will be denoted by "`keyi+` $(j)$", where $j$ is the sequence number of the first instruction that loads key values to registers in this round. Furthermore, the output nodes of those key loading instructions are called *key word nodes*. Depending on which word is loaded first, they are more specifically called *the first key word node*, *the second key word node*, etc[2].

**Example 3.** In Figure 2, "`r2` (2)" is the first key word node of round key for round one. "`r3` (10)" is the second key word node of round key for round two.

The nodes representing output operands that give us different words of the ciphertext are labeled "`x+` $(j)$", where "`x+` $(j)$" is an output node of instruction $f_j$, i.e. $j$ is the

[2]We note that this is only a naming convention to make the analysis consistent. In special cases, where the actual order of the key words is different from their loading sequence, user has to rearrange the key words after the analysis.

sequence number of the instruction that stores this word. We refer to them as *the words of the ciphertext*. For example, in Figure 2, "`x+` (13)" and "`x+` (14)" are the words of the ciphertext.

## 3.3 Output Criteria

For a directed graph $G$, a *directed path* from node $v$ to node $u$ is a sequence of edges $e_1, e_2, \ldots, e_k$ such that $e_1 = (v, x_1), e_2 = (x_1, x_2), e_3 = (x_2, x_3), \ldots, e_{k-1} = (x_{k-2}, x_{k-1})$, $e_k = (x_{k-1}, u)$. For any pair of nodes $v$ and $u$, if there exists a directed path from $v$ to $u$, we say $u$ is a *Gchild* of $v$ and $v$ is a *Gparent* of $u$. For any edge $e$ which appears in the sequence, we say $e$ *belongs* to this directed path from $v$ to $u$.

Now let us look at the following two simple scenarios in Tables 2 and 3.

Table 2: Assembly code snippet 1.

| # | Instruction |
|---|---|
| 0 | `LD r0 key0+` |
| 1 | `EOR r1 r0` |
| 2 | `ST x+ r1` |

Table 3: Assembly code snippet 2.

| # | Instruction |
|---|---|
| 0 | `LD r0 key0+` |
| 1 | `AND r1 r0` |
| 2 | `ST x+ r1` |

1. In Table 2, let us assume a fault is injected at $f_1$ such that some bits in `r1` are flipped before the execution of `EOR`. Then the exact same bits will be changed in ciphertext `x+`. Knowing how `r1` is changed and values of ciphertext with and without fault injection will not give us any information about `r0`.

2. In Table 3, we assume a fault is injected in $f_1$ such that some bits in register `r1` are flipped before the execution of `AND`. For example, suppose the first bit of `r1` is changed. Then we look at the first bit of the ciphertext `x+`. If the first bit of the ciphertext is also changed, we know that the first bit of the key is `1`, otherwise, it is `0`.

In view of the above, we say an instruction $f$ is *linear* if the following conditions are satisfied: Let $n$ be the register size in bits. For any pair $a, b$ ($a \in f^{io}, b \in f^{do}$), and for any $\Delta \in \mathbb{F}_2^n$, if $a$ is changed to $a' = a \oplus \Delta$, then $b$ will be changed to $b' = b \oplus \Delta$. Thus the linearity of an instruction $f$ is determined by its mnemonics $f^{mn}$. For example, the following commonly used mnemonics correspond to linear instructions: `EOR, LD, MV, ST`. And we say an edge $e$ is *non-linear* if the instruction associated with $e$ is non-linear.

For a pair of nodes $v$ and $u$ such that $u$ is a Gchild of $v$, the *Gdistance* between $v$ and $u$, denoted by Gdistance(v,u) is defined to be the cardinality of the following set:

$$\{e : e \text{ belongs to a directed path from } v \text{ to } u \text{ and } e \text{ is non-linear}\}.$$

**Example 4.** In Figure 2, "`x+` (13)" is a Gchild of "`r0` (6)" with distance 1. "`r0` (8)" is a Gchild of "`key1+` (2)" with distance 4.

For each node $a =$ "$x$ $(i)$", we define *CTGchild* of $a$ to be the set of ciphertext words which are Gchildren of $a$. Thus if a fault is injected in node $a$, the fault will be propagated to the ciphertext words that are in the set CTGchild of $a$.

To analyze the fault propagation that is useful, we need to focus on the nodes that are related to the key. We say a node $a$ is *related to a key word node* $b$ of a round key if $b$ is not a Gparent of $a$ and at least one of the Gchildren, say $ch$, of $a$ is a Gchild of $b$ with Gdistance$(b, ch) = 0$. The distance condition specifies that we are only looking at nodes that are linearly related to the key. The parent condition excludes nodes that would be related to several key words in a way that would combine these words linearly and therefore, making it impossible to recover the secret information. And we say $a$ is *related to a round key* `key` if it is related to at least one key word node of `key`.

For a given node $a$ which is to be examined, several possible parameters have to be specified. We refer to them as to *output criteria*, and they include the following:

- `minAffectedCT`: |CTGchild| $\geq$ `minAffectedCT`, i.e. the number of nodes in CT-Gchild is bigger or equal than `minAffectedCT`;

- `minDist`: $|\{ch : ch \in \text{CTGchild, and Gdistance}(a, ch) \geq \texttt{minDist}\}| \geq \texttt{minAffectedCT}$, i.e. the number of nodes in CTGchild with Gdistance at least `minDist` from $a$ is at least `minAffectedCT`;
- `maxDist`: $\text{Gdistance}(a, ch) \leq \texttt{maxDist}$ $\forall ch \in$Gchild, i.e. the Gdistance between any Gchild and $a$ should be at most `maxDist`;
- `maxKey`: the number of the round keys, counting from the last round key, that are related to node $a$ is at most `maxKey`;
- `minKeyWords`: there exists at least one round key such that the number of its corresponding key word nodes related to $a$ is at least `minKeyWords`.

DATAC takes a data flow graph $G$ and an output criteria as input, then iterates through all the nodes in $G$ and outputs the nodes of $G$ that satisfy the output criteria. Recall, we assume that the information available to the attacker is the fault model, the correct and faulty ciphertext.

Below, we explain that the output criteria defined are necessary for DFA and are independent of the analyzed cipher. On the other hand, the choice of their values is dependent on the analyzed implementation.

#### Why are the Output Criteria Mandatory

For DFA attack, `minAffectedCT` specifies how many words of the ciphertext are faulted after the fault injection. This value should be at least 1 so that the ciphertext values can be used. `minDist` reflects on how many non-linear operations are involved between the faulted node and the ciphertext. For DFA, `minDist` should be at least 1 so that there are non-linear operations involved and hence some information can be drawn. `maxDist` is an upper bound on how many non-linear operations are involved in the calculations. `maxDist` and `minDist` together restrict the number of non-linear operations to be solved for DFA. `maxKey` restricts which round key(s) are to be attacked. In most DFA attacks, the attacker focuses on the last round key (`maxKey`$= 1$) or the second last round key (`maxKey`$= 2$). `minKeyWords` is able to exclude nodes which are related to only small number of key words.

*Remark* 1. The choice of output criteria is essential to DFA, which is the scope of this paper. In case a different fault analysis method is considered, the output criteria should be changed accordingly.

#### Choosing the Output Criteria Values

We note that the values of output criteria are closely related to each other and are highly dependent on the actual assembly program being analyzed. For example, if the program makes use of a high number of non-linear instructions right before storing the ciphertext, `maxDist` should be set higher so that there are actually key words related to the faulted node. On the other hand, `maxKey` should not be too big, otherwise some nodes in the output will be associated with too many round keys, making the analysis harder. Accordingly, `minKeyWords` should be set to a small value, but for obvious reasons, should be at least 1. Or, if the user would like to have all the ciphertext words being affected, i.e. setting `minAffectedCT`$=$ the number of ciphertext words, the other conditions should be loosened. For example, for the data flow graph $G_{\mathcal{F}_{ex},full}$ in Figure 2, with an output criteria (`minAffectedCT, minDist, maxDist, maxKey, minKeyWords`) $= (2, 1, 1, 1, 1)$ we cannot get any output from DATAC.

For the data flow graph in Figure 2, with an output criteria (`minAffectedCT, minDist, maxDist, maxKey, minKeyWords`)$= (1, 1, 1, 1, 1)$, we get two nodes "`r0` (6)" and "`r1` (7)". For illustration purpose, we will focus on node "`r0` (6)" in the following.

## 3.4 Subgraph Construction

For a full cipher assembly implementation, the corresponding data flow graph involves plenty of nodes and edges. It is not easy to see the fault propagation properties from the full data flow graph. Thus we would like to construct a subgraph which shows the fault propagation clearly.

Given a data flow graph $G_{\mathcal{F},full}$ for an assembly program $\mathcal{F}$ and node $a$ in $G_{\mathcal{F},full}$, we construct a graph $G_a$ which is a subgraph of $G_{\mathcal{F},full} = (V, E)$, i.e. $G_a = (V_a, E_a)$ is a pair such that $V_a \subseteq V$ and $E_a \subseteq E$.

Sometimes, knowing how the faulted node relates to previous instructions will also help with the fault analysis. Keeping this in mind, we define a parameter called `depth` for the construction of the graph $G_a$.

We define *KNGchild* to be the set of key word nodes that are related to $a$. Then $V_a = \left(\bigcup_{i=0}^{\texttt{depth}} U_i\right) \bigcup \left(\bigcup_{j=1}^{4} V_j\right)$, where
- $U_0 = \{b : b$ is an input node of an instruction $f$ for which $a$ is an input node$\}$
- For $1 \leq i \leq$ `depth`, $U_i = \{b : b$ is an input node for an instruction $f$ such that $v$ is an output node of $f$ for some $v \in U_{i-1}\}$
- $V_1 = \{b : b$ is a child of $a\}$
- $V_2 = \{k : k$ is a round key that is related to $a\}$
- $V_3 = \{b : b$ is a key word node for a key $k \in V_2\}$
- $V_4 = \{b : b$ is a child of a node $v \in$ `KNGchild` and $b$ is a parent of a child of $a\}$.

Let $V' = (V_a \backslash (V_2 \cup V_3)) \cup$ `KNGchild`. Then $E_a = E_1 \cup E_2$, where $E_1 = \{e :$ both the head the tail of $e$ are in $V'\}$ and $E_2 = \{(k, b) : k \in V_2, b \in V_3\}$.

**Example 5.** In Figure 3 (a) and (b) we present the subgraphs constructed from node "`r0` (6)" of the data flow graph $G_{\mathcal{F}_{ex},full}$ (Figure 2) with depths equal to 0 and 1 respectively. For this case, we have
- $U_0 = \{$"`r0` (6)", "`r1` (7)"$\}$
- $U_1 = \{$"`r1` (5)", "`0xF0` (7)", "`r0` (4)", "`0x0F` (6)"$\}$
- $V_1 = \{$"`r0` (8)", "`r0` (11)", "`x+` (13)"$\}$
- $V_2 = \{$"`key2+` (9)"$\}$
- $V_3 = \{$"`r2` (9)", "`r3` (10)"$\}$
- $V_4 = \{$"`r0` (11)"$\}$

From Figure 3, we can see that with `depth` = 1, we do get extra useful information: the two edges with label "`andi` (6)" show that the first four bits of "`r0` (6)" are 0.

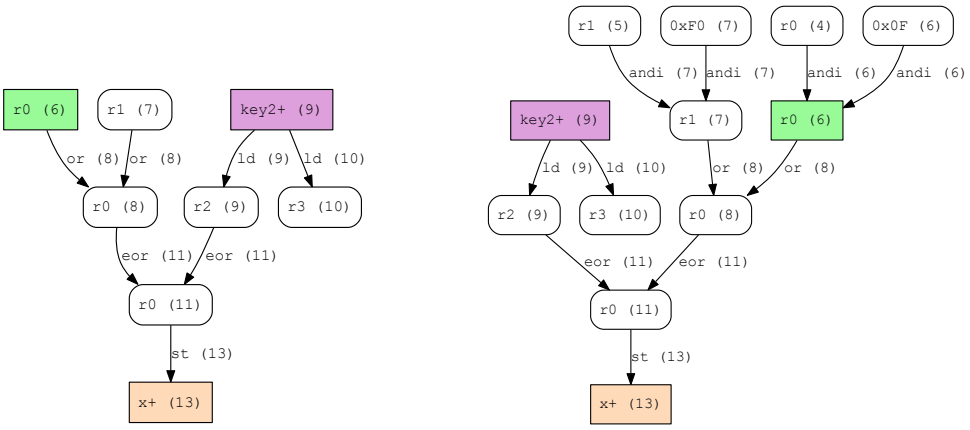
## 3.5 Equation Construction

Having the subgraph, constructed from a potentially vulnerable node, we would like to construct equations out of the subgraph to connect different input/output nodes, which can be easily analyzed by algebraic methods.

Given any subgraph $G_a = (V_a, E_a) \subseteq G_{\mathcal{F},full}$, where $G_{\mathcal{F},full}$ is the data flow graph of an assembly program $\mathcal{F}$, we take all the instructions in $\mathcal{F}$ that are related to at least one edge $e \in E_a$. Next, we order these instructions according to their sequence numbers. The equations are then constructed according to the input/output nodes and the edges associated with the corresponding instructions.

In Table 4 we show some representations of equations for different mnemonics. Here, the symbol "|" represents concatenation. For example, take $f =$ `MUL r2 r3`, it calculates the product of values in registers `r2` and `r3`, then the high byte of the product is stored in `r1` and the low byte of the product is stored in `r0`. Hence the product in the equation is represented as a concatenation of `r1` and `r0`: `r1 | r0`.

In case the equation is related to an instruction that loads a round key, DATAC is designed to indicate which key word node is involved in the equation (see Remark 2).




(a) (b)

Figure 3: Subgraph constructed from node "r0 (6)" of data flow graph in Figure 2 with depth (a) 0 and (b) 1.

Table 4: Construction of equations from assembly instructions.

| Instruction | Equation |
|---|---|
| ADD r2 r3 | carry \| r2 = r2 + r3 |
| ADC r2 r3 | carry \| r2 = r2 + r3 + carry |
| EOR r2 r3 | r2 = r2 $\oplus$ r3 |
| AND r2 r3 | r2 = r2 $\wedge$ r3 |
| OR r2 r3 | r2 = r2 $\vee$ r3 |
| MUL r2 r3 | r1 \| r0 = r2 $\times$ r3 |
| LD/MOV/ST r2 r3 | r2 = r3 |
| ROL r2 | carry \| r2 = r2 \| carry |
| LSL r2 | carry \| r2 = r2 \| 0 |
| LPM r2 Z | r2 = TableLookUp(ZH \| ZL) |

Now let us look at the assembly program $\mathcal{F}_{ex}$ for our sample cipher from Table 1. $\mathcal{F}_{ex}$ and output criteria $(1, 1, 1, 1, 1)$ (see Section 3.3) were given as input to DATAC. The data flow graph $G_{\mathcal{F}_{ex},full}$ for this sample cipher is in Figure 2. DATAC outputs two vulnerable nodes: "r0 (6)" and "r1 (7)". The subgraphs with depths 0 and 1, constructed from "r0 (6)", are shown in Figure 3. As we pointed out in Section 3.4, the subgraph with depth 1 gives some additional useful information, compared to the one with depth 0.

The equations obtained by using DATAC from the subgraph with depth 1, constructed from "r0 (6)" (Figure 3 (b)), are as follows:

$$\text{"r0 (6)"} = \text{"r0 (4)"} \wedge \text{"0x0F (6)"} \tag{1}$$
$$\text{"r1 (7)"} = \text{"r1 (5)"} \wedge \text{"0xF0 (7)"} \tag{2}$$
$$\text{"r0 (8)"} = \text{"r0 (6)"} \vee \text{"r1 (7)"} \tag{3}$$
$$\text{"r2 (9)"} = \text{key2[0]} \tag{4}$$
$$\text{"r0 (11)"} = \text{"r0 (8)"} \oplus \text{"r2 (9)"} \tag{5}$$
$$\text{"x+ (13)"} = \text{"r0 (11)"}. \tag{6}$$

Equation (1) shows "r0 (6)" = $0000 b_4 b_5 b_6 b_7$ for some $b_j \in \{0, 1\}$ $(j = 4, 5, 6, 7)$. Equation (3) shows that if we skip instruction 8, the result of Equation (1) will be used instead of the result of Equation (3) in instruction 11, which corresponds to Equation (5). Together with the information from Equations (4) and (6), the instruction skip attack on instruction 8 would result in the first four bits of key2[0] to appear as the first four bits of the faulted ciphertext.

Table 5: List of assembly code annotations and variable names required for DATAC .

| Action/variable name | DATAC format |
|---|---|
| Loading plaintext | `//load_plaintext` |
| Start of round $i$ | `//round_i` |
| Storing ciphertext | `//store_ciphertext` |
| Name of key variable pointer $i$ | `keyi` |
| Name of ciphertext variable pointer | `x` |

*Remark* 2. The index [0] in the right hand of Equation (4) indicates that the node "`r2` (9)" is the first key word node of `key2`, i.e. the value in "`r2` (9)" is the first byte of the second round key.

# 4 Implementation

DATAC was implemented in Java programming language, to support multi-platform environments and provide efficient running times. In this section, we provide implementation details of DATAC. Section 4.1 specifies the input format of the assembly code file. Section 4.2 outlines the execution steps of DATAC. Section 4.3 provides instructions on the usage of DATAC.

## 4.1 Assembly Code Properties

DATAC assumes the assembly code to be written in a text file, one instruction per line. Some annotations are also required, as well as naming conventions for important variables. These are stated in Table 5.

Another requirement for the code is that it should be unrolled. It would have been possible to add a loop and function dependency analysis into DATAC, however, it was shown before that loops and function calls can be easily disturbed by fault injection (e.g. in [MDH+13]) and therefore, it would make more sense to just skip the whole portion of the program instead of trying to analyze it with DFA.

## 4.2 Analysis Steps

In the following, we will outline the steps of the analysis that are being executed each time DATAC is run:

*Input:* assembly code text file, output criteria, depth.

Step 1: Read the assembly file and construct an array of instructions.

When DATAC reads an instruction, it first recognizes the mnemonics. Based on the type, it reads the operands that follow after the mnemonics. These are stored together with the sequence number.

Some operands are put explicitly, e.g. `LDI r0 0x01` has two operands – input operand is `0x01`, and the output operand is `r0`. In some cases, the operands are implicit, e.g. `MUL r5 r6` has four operands – input operands `r5`, `r6` are explicit and there are two implicit output operands: `r0` and `r1`. The same holds for instructions that use a carry bit, for example: `ADC`, `ROL` – in these cases, the carry is treated as input or/and output operand, since a fault in a carry can cause changes in later values.

Step 2: Construct the data flow graph of the program.

DATAC iterates through the array of instructions and analyzes both the operations and operands. Operands are represented as nodes and operations are represented as edges.

To allow tracing the behavior under fault, cross-dependencies between edges and nodes are also stored – each node has a set of input/output edges, and each edge has its head (output node) and tail (input node).

Some operands involve pointers to memory use pre-/post- increment or decrement operators (`+/-`). In this case, an array of consecutive memory values is used by the program, normally either for loading plaintext and key or for storing ciphertext. In the case of storing ciphertext, we treat different entries of the array separately, since it is important to know which word is affected by the fault. Since a fault injection in a memory cell can cause the same data disturbance as a fault on the data bus or register directly, we do not differentiate particular memory cells in case of loading the values. Therefore, such array would appear in the data flow graph as a single variable from which other variables are being loaded.

Step 3: For each node, record the following sets of nodes: Gparent, Gchild, CTGchild, related key words. For each Gchild of the node, calculate Gdistance between the node and this Gchild.

DATAC iterates through each edge. The head of an edge, say $v$, is a Gchild of the tail of this edge, say $u$. The Gchildren of $v$ are assigned as Gchildren of $u$. Furthermore, Gchildren of $v$ are also assigned as Gchildren of the Gparents of $u$.

Step 4: Select the nodes that satisfy the output criteria, based on the information from the previous step. These are the vulnerable nodes.

Step 5: Generate subgraph for each vulnerable node.

Step 6: Generate one set of equations for each subgraph.

*Output:* data flow graph, subgraphs and difference equations for each node that satisfies the output criteria.

## 4.3 Usage of DATAC

Together with the assembly code file, user has to specify the output criteria as an input. We suggest to use relatively tight values of output criteria as a preliminary test to see whether all the key words can be recovered, then loosen the criteria to find possible vulnerable nodes. We would like to point out that it is also possible to automate finding of the optimal values in order to get the desired number of vulnerable nodes. One can start with tight values and then loosen chosen parameters until this number is above some threshold.

Another user-specified parameter is the depth. One can get a good understanding about a usefulness of this parameter by observing Figures 6 and 7 from Section 5. While both vulnerable nodes are related to the same round key, their analysis requires usage of different depths in order to get information about the respective key words. We suggest user to start with higher values of this parameter (e.g. 5) and lower them after each iteration in case there are nodes in the subgraph that are not useful in the analysis.

DATAC is capable of analyzing any microcontroller instruction set, after specifying this set as a subclass of the `Mnemonics` class and specifying instruction properties (linearity, table look-up, etc.) in a subclass of the `MnemonicRecognizer` class.

Also, the relation of the vulnerable node to the key can be adjusted in case some other analysis instead of DFA is required. This can be done in the `analyzeFaultedNodes()` method of the `FaultAnalyzer` class. The class diagram of DATAC is provided in Appendix B.

# 5 Case Study

In this section, we will describe a DFA attack on PRESENT that was automatically generated by DATAC. Section 5.1 gives the specifications of PRESENT. Section 5.2 details the DFA attack we found using DATAC. This DFA attack is new and it requires 16 faults to recover the last round key.

We would like to point out that while all the fault attacks proposed on this cipher so far exploit the differential characteristics of the Sbox (e.g. [GYS, BH15, DSGSS, GLL+12, JLSH13]), our tool was able to find the vulnerable spots in the program that are implementation dependent, easily exploitable, and yet not trivial to find in the assembly code by a manual inspection.

## 5.1 PRESENT Cipher

For the case study, we have chosen a lightweight cipher PRESENT [BKL+07]. It is a symmetric block cipher, designed as a substitution-permutation network (SPN). Block length is 64 bits and key length can be either 128 bits or 80 bits (denoted as PRESENT-128 and PRESENT-80, respectively). A round function consists of three operations: `xor` of the state with the round key, followed by a substitution by 4-bit SBox, and finally, a bitwise permutation. After 31 rounds, there is one more *addRoundKey*, used for post-whitening. The encryption process is depicted in Figure 4. Because of its lightweight character, PRESENT-80 is usually used, therefore we focus on this variant in this section.

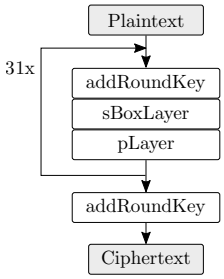



Figure 4: High-level algorithmic overview of PRESENT cipher.

Table 6: Assembly code of a table look-up for PRESENT implementation.

| # | Instruction |
|---|---|
| 0 | `LDI ZH 0x06` |
| 1 | `MOV ZL r0` |
| 2 | `LPM r21 Z` |
| 3 | `ANDI r21 0x0C` |
| 4 | `LDI ZH 0x07` |
| 5 | `MOV ZL r1` |
| 6 | `LPM r2 Z` |
| 7 | `ANDI r23 0x03` |
| 8 | `OR r21 r23` |

As a target, we chose a speed-optimized assembly implementation for 8-bit AVR from Verstegen and Papagiannopoulos, publicly available on GitHub[3]. We note that we did not use the key schedule for our analysis, since we were targeting the main encryption routine.

## 5.2 Fault Analysis

In order to get the vulnerable nodes from the cipher implementation, we have chosen our output criteria to be (`minAffectedCT, minDist, maxDist, maxKey, minKeyWords`)= $(1, 1, 1, 1, 1)$. With this output criteria, DATAC outputs 16 vulnerable nodes, out of the total 4,712 nodes. We will explain the fault attack procedure on one of these nodes – "`r23` (4546)". Subgraph for "`r23` (4546)" with depth 1 is stated in Figure 6 in Appendix A.

[3] https://github.com/kostaspap88/PRESENT_speed_implementation

Equations generated for the subgraph with depth 1 from "r23 (4546)" are as follows:

$$\text{“r22 (4538)”} = \text{“r22 (4529)”} \vee \text{“r23 (4537)”} \tag{7}$$
$$\text{“r23 (4546)”} = \text{“r23 (4545)”} \wedge \text{“0x03 (4244)”} \tag{8}$$
$$\text{“r22 (4547)”} = \text{“r22 (4538)”} \vee \text{“r23 (4546)”} \tag{9}$$
$$\text{“r1 (4648)”} = \texttt{key32[1]} \tag{10}$$
$$\text{“r1 (4656)”} = \text{“r1 (4648)”} \oplus \text{“r22 (4547)”} \tag{11}$$
$$\text{“x+ (4664)”} = \text{“r1 (4656)”}. \tag{12}$$

Equation (8) shows "r23 (4545)" = $000000b_6b_7$ for some $b_6, b_7 \in \{0, 1\}$. Together with the other equations we get

$$\text{“x+ (4664)”} = \texttt{key32[1]} \oplus (\text{“r22 (4538)”} \vee 000000b_6b_7). \tag{13}$$

Consider a bit flip fault injection with fault mask $\Delta =$ 11111100 in "r23 (4546)" right before the execution of instruction 4547, which corresponds to Equation (9), then Equation (13) becomes:

$$\text{“x+ (4664)”} = \texttt{key32[1]} \oplus (\text{“r22 (4538)”} \vee 111111b_6b_7), \tag{14}$$

where "x+ (4664)"′ denotes the faulted output. Let $\delta = \delta_0\delta_1\delta_2\delta_3\delta_4\delta_5\delta_6\delta_7 =$ "x+ (4664)"′ $\oplus$ "x+ (4664)" and let "r22 (4538)" = $a_0a_1a_2a_3a_4a_5a_6a_7$. Since both $\oplus$ and $\vee$ are bitwise operations, together with Equations (13) and (14) we have

$$\begin{aligned}\delta_0\delta_1\delta_2\delta_3\delta_4\delta_5 &= (a_0a_1a_2a_3a_4a_5 \vee 000000) \oplus (a_0a_1a_2a_3a_4a_5 \vee 111111)\\ &= a_0a_1a_2a_3a_4a_5 \oplus 111111 \Longrightarrow a_0a_1a_2a_3a_4a_5 = \delta_0\delta_1\delta_2\delta_3\delta_4\delta_5 \oplus 111111.\end{aligned}$$

Since the value of $\delta$ is known and the value of "x+ (4664)" is also known, together with Equation (13), we have

$$\text{the first 6 bits of } \texttt{key32[1]} = \text{first 6 bits of “x+ (4664)”} \oplus \delta_0\delta_1\delta_2\delta_3\delta_4\delta_5 \oplus 111111,$$

which gives the first 6 bits of the last round key.

With a subgraph constructed from "r22 (4538)" a similar fault analysis helps us to recover the last 2 bits of key32[1]. This is the case where we have to utilize the depth parameter of DATAC to get enough information about the faulted node. The subgraph for the node "r22 (4538)" is stated in Figure 7 in Appendix A. As can be seen in this subgraph to see the constants, three layers above the vulnerable node have to be revealed.

The same analysis can be carried out for the remaining 14 nodes to get all the bits of the last round key.

To understand the found vulnerability, we examined the assembly code and provide an explanation below on why the cipher implementation contains the exploitable operations output by DATAC. This implementation combines the pLayer with the sBoxLayer in the form of 5 look-up tables. We will explain how this procedure works on a simple example. Table 6 contains the code for two table look-ups, which results into one nibble output. First, a table index is loaded into higher byte of register Z (instructions 0 and 4) – this decides which table will be used. Then, the intermediate state is loaded into lower byte of register Z – it contains two nibbles of data, therefore, we expect to get 2 bits of data back after the Sbox and the bit permutation are applied. To clear the remaining 6 bits, ANDI instruction is used (instructions 3 and 7). Finally, we combine the values of these two look-ups into a nibble with an OR instruction. The attack exploits the properties of this combined layer as well as merging of the bits of the intermediate results together into a single register.

# 6 Discussion

This part discusses various aspects of our work. In Section 6.1, we discuss the output criteria in details, by showing the DATAC analysis results of assembly implementations of SIMON, SPECK and AES ciphers and compare the results to PRESENT. The scalability of DATAC is discussed in Section 6.2, where we show analysis results on AES with various number of rounds. Possibilities of extending DATAC to other ciphers and devices are elaborated in Section 6.3. In Section 6.4, we discuss possible countermeasures against fault attacks on software implementations and their selection. Finally, in Section 6.5 we give a remark on security verification of cryptographic software and its relation to our work.

## 6.1 Output Criteria

To test our DATAC tool and give more insight into effect of output criteria, we analyzed two more lightweight ciphers: SIMON and SPECK [BTCS+15], together with AES [DR02] as a reference. For SIMON and SPECK, we used assembly implementations for 8-bit AVR from Luo Peng, available from GitHub[4] (The implementation is based on the proposal from the authors of SIMON and SPECK, specified in [BSS+14]). More specifically, we tested SIMON 64/128 and SPECK 64/128, both high-throughput implementations. AES implementation, written by Francesco Regazzoni [EGG+12] is available from Ecrypt II project website[5].

Results are shown in Figure 5, which plots numbers of vulnerable nodes for various output criteria values. When looking at SIMON and SPECK, because of the structure of these ciphers, where only half of the state is directly related to one round key, the number of vulnerable nodes is lower compared to PRESENT-80 and AES-128 in most of the cases.

Recall that `maxDist` specifies the maximum number of non-linear operations between the vulnerable node and the ciphertext. To attack nodes in earlier computations, the user needs to set `maxDist` to higher value. Thus, when `maxDist` is set to a bigger value, more nodes would satisfy the output criteria. This can be seen from Figure 5 (a) – when the value of `maxDist` parameter increases, we obtain more vulnerable nodes.

`maxKey` restricts the round keys to be attacked. To attack earlier round keys, the user needs bigger values of `maxKey`. Hence, when this value increases, the number of vulnerable nodes will increase, as shown in Figure 5 (b). In this case, we set the `maxDist` to a very high number because otherwise the restriction from this parameter would be too tight such that not that many rounds can be involved in the analysis.

Plot depicted in Figure 5 (c) varies the `minKeyWords` parameter. In this case, we set the `maxKey` to 1 to restrict the analysis to the last round key, hence only nodes related to last round key are considered. The graph shows that when the value of `minKeyWords` increases, the number of vulnerable nodes decreases rapidly. For `minKeyWords` value of 1, the number of vulnerable nodes for PRESENT is relatively high. It is because the PRESENT implementation combines `sBoxLayer` and `pLayer`, where multiple operations have to be executed to merge particular bits after this layer, resulting in more nodes in the graph.

In Figure 5 (d) we vary the value of `minAffectedCT` parameter, which is the minimum number of ciphertext words related to vulnerable node. PRESENT and AES have much higher number of nodes per round compared to SPECK and SIMON, therefore the initial number of vulnerable nodes is high for these two ciphers. However, it is uncommon for a node to relate to too many ciphertext nodes, therefore there is a significant drop when the `minAffectedCT` increases from 1.

The results show that DATAC is capable of finding the vulnerable spots automatically in different implementations, without additional knowledge of the internal cipher structure

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

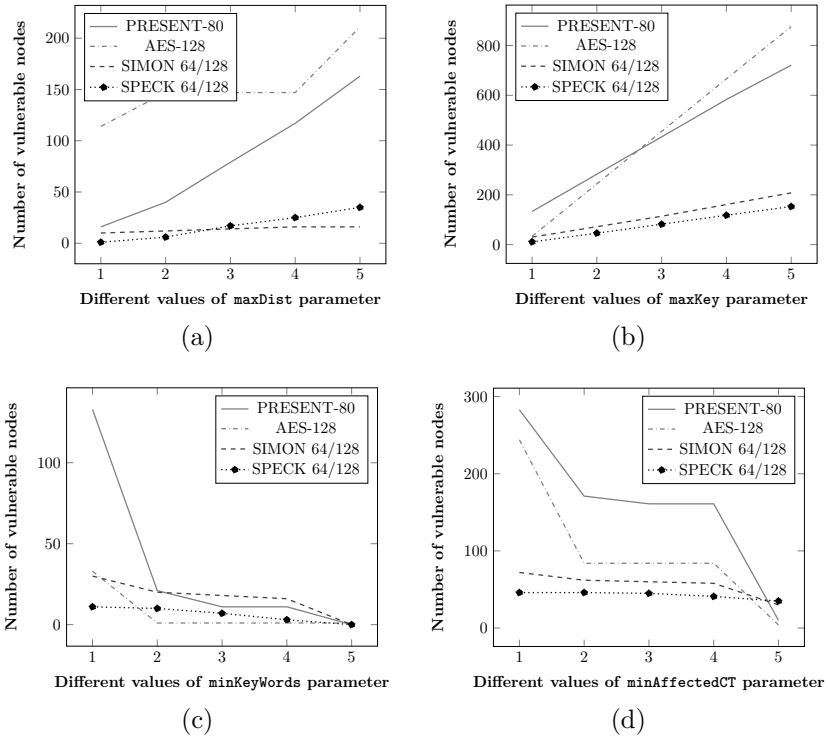



Figure 5: Comparison of different output criteria on different ciphers. Plot (a) varies the `maxDist` parameter, while the other parameters are (`minAffectedCT, minDist,maxKey, minKeyWords`) $= (1, 1, 2, 1)$. Plot (b) varies the `maxKey` parameter, while the other parameters are (`minAffectedCT, maxDist, minDist,minKeyWords`) $= (1, 200, 1, 1)$. Plot (c) varies the `minKeyWords` parameter, while the other parameters are (`minAffectedCT, maxDist, minDist,maxKey`) $= (1, 200, 1, 1)$. Plot (d) varies the `minAffectedCT` parameter, while the other parameters are (`maxDist, minDist, maxKey, minKeyWords`) $= (200, 1, 2, 1)$.

(only annotations from Section 4.1 are required). Also, the running times show that our approach is scalable.

### Vulnerable Nodes and Exploitability

It is not guaranteed that every vulnerable node is exploitable by DFA. For example, for the PRESENT implementation in the case study (Section 5), the maximum distance between a node used in the last round and its CTGchild is 5. Thus for an extreme example, if we set the output criteria to be (`minAffectedCT, minDist, maxDist, maxKey, minKeyWords`)$=$ $(0, 0, 5, 1, 0)$, we get 150 vulnerable nodes, which involve all the nodes that are related to the last round key. Or for a more reasonable example, if we set output criteria to be (`minAffectedCT, minDist, maxDist, maxKey, minKeyWords`)$= (1, 1, 5, 1, 1)$, we get 133 vulnerable nodes, which involve all the nodes that satisfy the following: affect at least 1 ciphertext node; have at least distance 1 from at least one ciphertext node; are related to the last round key. By an easy analysis, not all such nodes are exploitable. On the other hand, with output criteria (`minAffectedCT, minDist, maxDist, maxKey, minKeyWords`)$= (1, 1, 1, 1, 1)$, we get 16 nodes, which are all exploitable as explained in Section 5.

As stated in Section 4.3, we suggest the user to start with relatively tight values for output criteria, analyze the resulting vulnerable nodes. If no exploitable node can be found, then loosen the values of output criteria to get more vulnerable nodes.

## 6.2 Scalability of DATAC

Table 7 shows the time for analyzing implementations with different number of rounds of AES using DATAC. The results show that DATAC is capable of handling heavy algorithms in reasonable time, while using a laptop computer with average computing power (mobile Intel Haswell family CORE i7 processor with 8 GB RAM). For 50 rounds AES, DATAC needs less than 40 seconds, with memory consumption of 500 MB. Given the number of instructions in AES, this shows DATAC is capable of evaluating all current block ciphers. The same holds for countermeasures – 50 rounds of AES can be considered as a 5× instruction redundancy compared to standard AES-128, while the usual overheads in literature rarely go over 2× the original [KSV13].

We would like to point out that DATAC aims at DFA, where most of the attacks target the last three rounds of the cipher (e.g. [TM09, JLSH13, TBM14]). Therefore, for a practical usage of DATAC, it would be sufficient to analyze three rounds of a target block cipher implementation.

## 6.3 Extension to Other Ciphers and Devices

As already mentioned in the introduction, DATAC can be used for analysis of current state-of-the-art block ciphers. We are not aware of any restrictions on the cipher design or structure – since the data propagation is always captured in the assembly implementation, the customized DFG will identify all the necessary information needed for DFA. We have tested DATAC on various cipher designs, including SPN with MDS matrix (AES), SPN with bit permutation (PRESENT), Feistel with bit shifting (SIMON and SPECK). Round functions of these ciphers encompass different non-linear elements, such as 4-bit and 8-bit Sbox, modular addition, binary AND.

Our case studies were done on 8-bit AVR assembly code. When it comes to different device architectures, identification of vulnerable nodes, construction of subgraphs and generation of fault analysis equations do not need any change (see Figure 1). Only the parsing subsystem, responsible for analyzing the assembly code and converting it to DFG, has to be adjusted. More specifically, the `AsmFileReader` class (full class diagram is stated in Figure 8) has to be changed so that nodes are properly identified. For example, for AVR assembly code `AND r0, r1`, the input nodes are `r0` and `r1`, the output node is `r0`. For ARM assembly code `AND r0, r1, r2`, the input nodes are `r1` and `r2`, the output node is `r0`. We would like to point out that we aim at single fault injection. Thus we assume the attacker has an ability to perform a single fault even for pipelined architectures.

## 6.4 Countermeasures

After identifying the vulnerable nodes, a natural question for the implementer would be: How to avoid these vulnerabilities?

Over the time, several approaches to thwart fault attacks in software have emerged. They are often based on instruction duplication/triplication or parity checks [BPB+10, MHD+14]. However, as shown in [YGS+16], these instruction-level countermeasures can be easily broken by a simple clock-glitch. Especially, the instruction skip protection provided by duplication can be simply overcome by targeting two instructions at once. Therefore,

Table 7: Scalability of DATAC tested on AES with different number of rounds.

| # of rounds of AES | 1 | 10 | 30 | 50 |
|---|---|---|---|---|
| # of nodes | 281 | 2,060 | 6,300 | 10,540 |
| # of edges | 415 | 3,209 | 9,909 | 16,609 |
| # of instructions | 259 | 1,901 | 5,801 | 9,701 |
| Time (s) | 0.07 | 0.87 | 5.11 | 38.89 |
| Average time per round (s) | 0.07 | 0.09 | 0.17 | 0.78 |
| Memory (MB) | 3 | 19 | 170 | 500 |

one has to look at proposals offering more guarantees, such as coding theory based countermeasures [BH16, BCC+14] or redundancy methods that spread the data across several instructions [PYGS16, LCFS17]. Another approach is a technique called *infective countermeasure* that tries to distribute the fault evenly to the whole cipher state so that the attacker does not get any usable information about the secret key [PCM15].

Each of these countermeasures has some assumptions, e.g. fault model, number of faults, fault precision, or number of flipped bits. Since DATAC is capable of analyzing instruction skips and bit-level faults, it can evaluate countermeasures that claim protection against these. However, if breaking the protection technique requires several faults to be injected during a single encryption, such model falls out of the current scope of DATAC. In the future, we would like to enhance the tool capability to include these more sophisticated techniques.

### 6.5 Security Verification

We would like to emphasize that while our approach introduces new methods that can be used in the direction of security verification, it cannot be used to guarantee the security of the underlying implementation directly. To an extent, DATAC works in a similar way to *Sleuth* [BRNI13], proposed to verify software implementations against side-channel attacks. *Sleuth*, as a verification tool, provides an analysis of three specific countermeasures against a user-defined leakage model – Boolean masking, arithmetic masking, and random precharging. While it is able to point out the weaknesses of the implementation, it does not guarantee the implementation is secure against all the SCA techniques. In the same way, DATAC is able to detect DFA-vulnerable parts and provide the analyst with the information on weaknesses in the code. However, when no vulnerable nodes can be found, it does not mean the code cannot be broken by utilizing different fault analysis methods, especially since some of them might not be known at the time of analysis.

## 7 Conclusion

We have proposed a methodology capable of finding spots vulnerable to DFA in software implementations of cryptographic algorithms. Following our approach, we have created the DATAC tool, which takes an assembly implementation and a user-specified output criteria as an input. It outputs subgraphs for vulnerable nodes in the code, together with equations that can be directly used for DFA on the cipher implementation.

We have presented a detailed DFA attack on PRESENT-80, exploiting implementation weaknesses found by DATAC. Our results show that by using DATAC, it is possible to find fault injection vulnerabilities that are not visible from observing the cipher structure and are hard to find from an assembly code that is normally hundreds to thousands lines long. To further prove its capabilities, we tested another two lightweight cipher implementations, SPECK 64/128 and SIMON 64/128, together with current NIST symmetric key standard, AES-128. All the implementations that we analyzed are publicly available programs. The only adjustment was to add several annotations (and unroll the loops in some cases). The results show that DATAC is scalable and can analyze current algorithms efficiently.

For the future work, we would like to extend DATAC to be able to analyze the differential properties of non-linear operations in the cipher and solve the generated equations. There is also a potential to extend the analysis technique from DFA to algebraic fault analysis. Additionally, we would like to focus on protected implementations (e.g. [BCR16]) to find loopholes and propose additional countermeasures automatically.

## Acknowledgments

This research is supported (in part) by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate.

## 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, pages 137:137–137:144, New York, NY, USA, 2014. ACM.

[ASKL81] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Trans. Comput.*, 30(5):341–356, May 1981.

[Bar13] Clark Barrett. "decision procedures: An algorithmic point of view," by daniel kroening and ofer strichman, springer-verlag, 2008. *Journal of Automated Reasoning*, 51(4):453–456, 2013.

[BBB+12] Alessandro Barenghi, Guido M. Bertoni, Luca Breveglieri, Mauro Pelliccioli, and Gerardo Pelosi. *Injection Technologies for Fault Attacks on Microprocessors*, pages 275–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[BBKN12] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, Nov 2012.

[BCC+14] Julien Bringer, Claude Carlet, Hervé 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.

[BCR16] Thierno Barry, Damien Couroussé, and Bruno Robisson. Compilation of a countermeasure against instruction-skip fault attacks. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, CS2 '16, pages 1–6, New York, NY, USA, 2016. ACM.

[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, Feb 2006.

[BEG13] Nasour Bagheri, Reza Ebrahimpour, and Navid Ghaedi. New differential fault analysis on present. *EURASIP Journal on Advances in Signal Processing*, 2013(1):145, Sep 2013.

[BH15] J. Breier and W. He. Multiple fault attack on present with a hardware trojan implementation in fpga. In *2015 International Workshop on Secure Internet of Things (SIoT)*, pages 58–64, Sept 2015.

[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.

[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.

[BKL+07] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '07, pages 450–466, Berlin, Heidelberg, 2007. Springer-Verlag.

[BPB+10] Alessandro Barenghi, Gerardo Pelosi, Luca Breveglieri, Francesco Regazzoni, and Israel Koren. Low-cost software countermeasures against fault attacks: Implementation and performances trade offs, 2010.

[BRNI13] Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. Sleuth: Automated verification of software power analysis countermeasures. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013: 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, pages 293–310, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[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 Jr. Kaliski, BurtonS., editor, *Advances in Cryptology - CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer Berlin Heidelberg, 1997.

[BSS+14] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The simon and speck block ciphers on avr 8-bit microcontrollers. Cryptology ePrint Archive, Report 2014/947, 2014. http://eprint.iacr.org/2014/947.

[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.

[CGK80] A. E. Casavant, D. D. Gajski, and D. J. Kuck. Automatic design with dependence graphs. In *17th Design Automation Conference*, pages 506–515, June 1980.

[CML+11] G. Canivet, P. Maistri, R. Leveugle, J. Clédière, F. Valette, and M. Renaudin. Glitch and laser fault attacks onto a secure aes implementation on a sram-based fpga. *Journal of Cryptology*, 24(2):247–268, Apr 2011.

[DJTK10] Ralf Dreesen, Thorsten Jungeblut, Michael Thies, and Uwe Kastens. Dependence analysis of vliw code for non-interlocked pipelines. In *Proceedings of the 8th Workshop on Optimizations for DSP and Embedded Systems (ODES-8)*, 2010.

[DP14] Avijit Dutta and Goutam Paul. Deterministic hard fault attack on trivium. In Maki Yoshida and Koichi Mouri, editors, *Advances in Information and Computer Security: 9th International Workshop on Security, IWSEC 2014, Hirosaki, Japan, August 27-29, 2014. Proceedings*, pages 134–145, Cham, 2014. Springer International Publishing.

[DPdC+16] Louis Dureuil, Marie-Laure Potet, Philippe de Choudens, Cécile Dumas, and Jessy Clédière. From code review to fault injection attacks: Filling the gap using fault model inference. In Naofumi Homma and Marcel Medwed, editors, *Smart Card Research and Advanced Applications: 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, pages 107–124, Cham, 2016. Springer International Publishing.

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

[DRA16] Prakash Dey, Raghvendra Singh Rohit, and Avishek Adhikari. Full key recovery of acorn with a single fault. *J. Inf. Secur. Appl.*, 29(C):57–64, August 2016.

[DSGSS] Fabrizio De Santis, Oscar M. Guillen, Ermin Sakic, and Georg Sigl. Ciphertext-only fault attacks on present. In Thomas Eisenbarth and Erdinç Öztürk, editors, *Lightweight Cryptography for Security and Privacy: Third International Workshop, LightSec 2014, Istanbul, Turkey, September 1-2, 2014*, pages 85–108.

[EGG+12] Thomas Eisenbarth, Zheng Gong, Tim Güneysu, Stefan Heyse, Sebastiaan Indesteege, Stéphanie Kerckhof, François Koeune, Tomislav Nad, Thomas Plos, Francesco Regazzoni, François-Xavier Standaert, and Loic van Oldeneel tot Oldenzeel. Compact implementation and performance evaluation of block ciphers in attiny devices. In Aikaterini Mitrokotsa and Serge Vaudenay, editors, *Progress in Cryptology - AFRICACRYPT 2012: 5th International Conference on Cryptology in Africa, Ifrance, Morocco, July 10-12, 2012. Proceedings*, pages 172–187, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[EHH+14] Sho Endo, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji, and Takafumi Aoki. A multiple-fault injection attack by adaptive timing control under black-box conditions and a countermeasure. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design*, pages 214–228, Cham, 2014. Springer International Publishing.

[FXKH17] K. Fukushima, R. Xu, S. Kiyomoto, and N. Homma. Fault injection attack on salsa20 and chacha and a lightweight countermeasure. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 1032–1037, Aug 2017.

[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.

[GHEDK16] Lucien Goubet, Karine Heydemann, Emmanuelle Encrenaz, and Ronald De Keulenaer. Efficient design and evaluation of countermeasures against fault attacks using formal verification. In Naofumi Homma and Marcel Medwed, editors, *Smart Card Research and Advanced Applications: 14th International Conference, CARDIS 2015, Bochum, Germany, November*

*4-6, 2015. Revised Selected Papers*, pages 177–192, Cham, 2016. Springer International Publishing.

[GLL+12] Dawu Gu, Juanru Li, Sheng Li, Zhouqian Ma, Zheng Guo, and Junrong Liu. Differential fault analysis on lightweight blockciphers with statistical cryptanalysis techniques. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2012 Workshop on*, pages 27–33, Sept 2012.

[GYS] Nahid Farhady Ghalaty, Bilgiday Yuce, and Patrick Schaumont. Differential fault intensity analysis on present and led block ciphers. In *Constructive Side-Channel Analysis and Secure Design: 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015.*, pages 174–188.

[JKP12] Philipp Jovanovic, Martin Kreuzer, and Ilia Polian. A fault attack on the led block cipher. In Werner Schindler and Sorin A. Huss, editors, *Constructive Side-Channel Analysis and Secure Design: Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*, pages 120–134, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[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.

[KPB+17] S. V. 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 2017, Taipei, Taiwan, September 25, 2017*, pages 33–40, 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, New York, NY, USA, 2017. ACM.

[KSV13] Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware designer's guide to fault attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, 2013.

[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.

[MHD+14] N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, and E. Encrenaz. Experimental evaluation of two software countermeasures against fault attacks. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 112–117, May 2014.

[Nie98] Ralf Niemann. *Hardware/software co-design for data flow dominated embedded systems*. Springer Science & Business Media, 1998.

[PCM15] Sikhar Patranabis, Abhishek Chakraborty, and Debdeep Mukhopadhyay. Fault tolerant infective countermeasure for aes. In Rajat Subhra Chakraborty, Peter Schwabe, and Jon Solworth, editors, *Security, Privacy, and Applied Cryptography Engineering: 5th International Conference, SPACE 2015, Jaipur, India, October 3-7, 2015, Proceedings*, pages 190–209, Cham, 2015. Springer International Publishing.

[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.

[Riv09] 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.

[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. http://eprint.iacr.org/2009/575.

[YGS+16] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont. Software fault resistance is futile: Effective single-glitch attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 47–58, Aug 2016.

# A Subgraphs for Fault Analysis of PRESENT

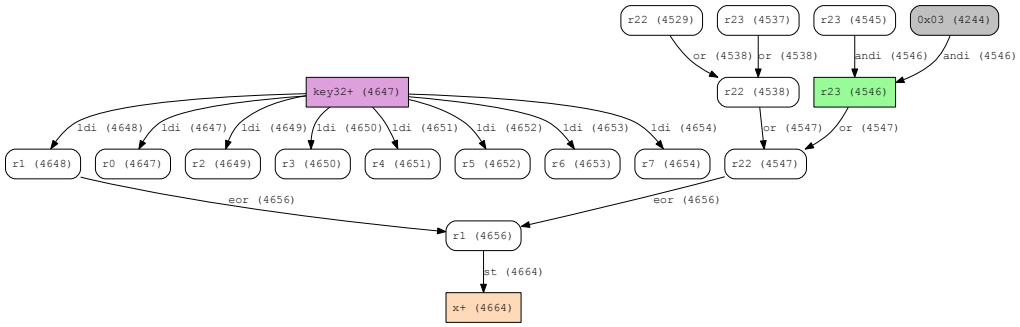



Figure 6: Subgraph with depth 1 generated by DATAC from the assembly implementation of PRESENT, corresponding to vulnerable node "`r23` (4546)".

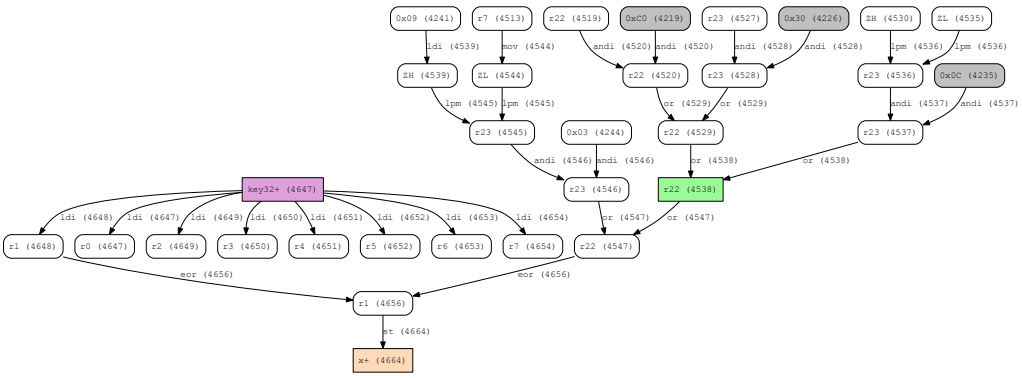



Figure 7: Subgraph with depth 3 generated by DATAC from the assembly implementation of PRESENT, corresponding to vulnerable node "`r22` (4538)".

# B Class Diagram of DATAC

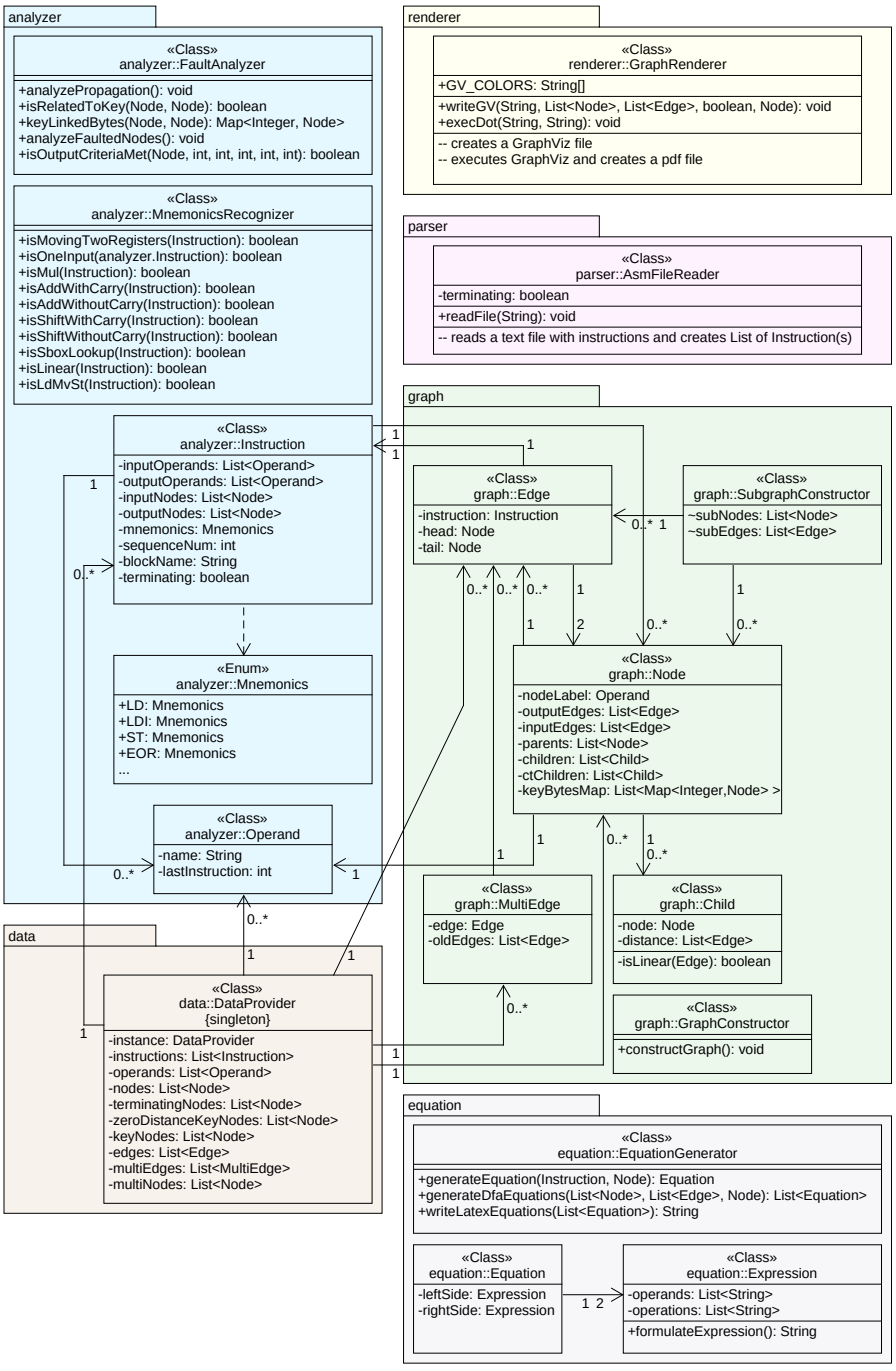



Figure 8: Class diagram of DATAC . Some details were omitted, such as getters/setters and helper methods. Colors represent different packages.