# Automated Fault Analysis of Assembly Code
## With a Case Study on PRESENT Implementation

Jakub Breier[1] and Xiaolu Hou[2]

[1]Physical Analysis and Cryptographic Engineering
Temasek Laboratories at Nanyang Technological University, Singapore
[2]School of Computer Science and Engineering
Nanyang Technological University, Singapore
`jbreier@ntu.edu.sg,ho0001lu@e.ntu.edu.sg`

**Abstract.** Fault injection attack models are normally determined by analyzing the cipher structure and finding exploitable spots in non-linear and diffusion 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 graph, allowing us to find vulnerable spots efficiently. Fault propagation is analyzed in a subgraph constructed from each vulnerable spot, allowing us to automatically generate equations for differential fault analysis.
We have created a tool that implements our approach: *ATLAS – Automated TooL for Assembly analysiS*. 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 secret key with 16 faults. Our results show that ATLAS 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 [21, 11, 18].

However, in the end, 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 memories directly [3, 4].

The problem is that different implementations of the same encryption algorithm do not necessarily share the same vulnerabilities, and therefore, some attacks that work in theory might either not be possible, or be hard to execute in practice. 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.

**Our Contribution.** In our work, we focus on automatic analysis of assembly code that implements a cryptographic algorithm w.r.t. fault injection attack. We develop a methodology to represent an assembly code as an oriented graph – a *memory/operation flow* (MOF) graph, that preserves the operations (edges) and memory structures (nodes) holding the data. By analyzing the nodes of this graph, we are able to find vulnerable spots in the cipher implementation. From these, we construct subgraphs that represent propagation of faults from vulnerable nodes to the ciphertext, while affecting some parts of round key(s). Ultimately, this allows us to automatically generate differential fault attack equations, by solving which we can mount a successful fault injection attack. Our methodology was implemented in a tool named *ATLAS - Automated TooL for Assembly analysiS*, that takes an assembly code as input, and outputs subgraphs and equations for each vulnerable node according to selected criteria. We provide a case study on PRESENT-80 cipher implementation for 8-bit AVR microcontroller that shows capabilities of ATLAS by finding an implementation specific fault attack, being able to recover the secret key by 16 fault injections. Moreover, we provide an analysis of SPECK 64/128 and SIMON 64/128 lightweight ciphers. We would like to point out that our tool is modular and enables an easy extension to the instruction set.

The rest of the paper is structured as follows. Sec. 2 provides an overview of related works. Sec. 3 formalizes fault attacks in software and provides notation that is used in the rest of the paper. Sec. 4 specifies our approach, by detailing each step of the evaluation implemented by ATLAS. Sec. 5 explains the usage of ATLAS on PRESENT-80. Sec. 6 provides a discussion and finally, Sec. 7 concludes this work and provides a motivation for future works.

## 2 Related Work

In this section we will detail several works focusing on analyzing implementations w.r.t. fault attacks.

Given-Wilson et al. [14] made a tool to detect vulnerabilities in the process of compilation, to show that there might be new exposures introduced during the transformation of the code from C to assembly.

Agosta et al.[2] used a compiler approach as well. They utilized the LLVM compiler in order to check single bit-flip vulnerabilities in the code to point out the exploitable parts. However, together with the previous approach, the intermediate representation is used for the analysis, and it therefore excludes the assembly implementations written directly by the programmer. Such implementations are usually more optimized, depending on the requirements (speed/memory/security) and therefore, often used for critical applications.

Khanna et al. [19] proposed XFC – a method that checks exploitable fault characteristics of block ciphers considering the *differential fault analysis* (DFA) method. Their approach takes a cipher specification as an input and then uses colors to indicate 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 cipher implementation.

Goubet et al. [15] 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 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. [12] 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.

Our approach analyzes the assembly code directly, by building a customized graph data structure, allowing us 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.

## 3 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 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. Similarly,* $f^{do}$ *is the set of destination operands of* $f$*, which can be registers or pointers to memory addresses.*

We have used an instruction set of 8-bit AVR microcontroller for examples in this paper.

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

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

| # | Instruction | # | Instruction | # | Instruction |
|---|---|---|---|---|---|
| | `//round 1` | 6 | `ANDI r0 0x0F` | 11 | `EOR r0 r2` |
| 0 | `LD r0 x+` | 7 | `ANDI r1 0xF0` | 12 | `EOR r1 r3` |
| 1 | `LD r1 x+` | 8 | `OR r0 r1` | | `//store ciphertext` |
| 2 | `LD r2 key1+` | | `//round 2` | 13 | `ST x+ r0` |
| 4 | `EOR r0 r4` | 9 | `LD r2 key2+` | 14 | `ST x+ r1` |
| 5 | `EOR r1 r5` | 10 | `LD r3 key2+` | | |

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 are interested in the 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 uses a known ciphertext attack with the knowledge of the fault model for the differential fault analysis.

## 4 Automated Assembly Code Analysis

The main idea of this work is to analyze the assembly code and find vulnerable spots w.r.t. differential fault injection attack. This can be a non-trivial task since it is non-deterministic – the same instruction can be vulnerable in one part of the code, but secure in the other part, depending on the context.

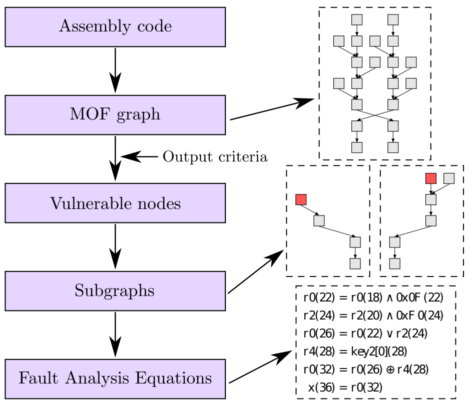


Fig. 1: Our evaluation method for analyzing assembly code w.r.t. fault injection vulnerabilities.

There are various intermediate forms that can represent a program, ranging from the simplest, such as control-flow and data-flow graphs, to more advanced static single assignment (SSA) and control dependence graphs [9]. These forms are normally used in compilers for optimizing the program. For example, Java HotSpot VM [20] and LLVM [1] use SSA as the intermediate representation. 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) responsible for changes and transitions – represented as edges.
– Properties of operations (linear/non-linear).
– Ability to distinguish important nodes, such as round keys and ciphertext.

Since none of the intermediate forms mentioned above contains details that are necessary for fault vulnerability assessment, we have decided to design a new representation that fits our purposes – *Memory/Operation Flow graph* (MOF).

Our evaluation method is depicted in Fig 1. First, an assembly code is fetched as the input. Based on its structure, an MOF graph is created. Based on the output criteria, vulnerable nodes are identified – places that will be later tested for the fault injection in order to determine the propagation pattern. This pattern specifies which data values will be affected by the fault and is captured by the subgraph – one subgraph is created for each vulnerable node. Together with it, fault analysis equations are generated – based on them, a fault attack can be executed. Each step is explained in a greater detail below.

Our approach was implemented in Java programming language and named *ATLAS - Automated TooL for Assembly analysiS*. ATLAS 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. The class diagram of ATLAS is provided in Appendix B.

In the rest of this section, we provide details on how our approach works.

## 4.1 From Assembly to MOF Graph

Given a program $\mathcal{F} = (f_0, f_1, \ldots, f_{N_\mathcal{F}-1})$, a *memory/operation 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 "`x` $(i)$" such that $x$ is a destination operand of instruction $i$. $B$ consists of labeled nodes with labels "`y` $(i)$" such that $y$ is an input operand of instruction $i$ and $y$ is not a destination operand of any instruction.
– $A = \{$"`x` $(i)$" $: x \in f_i^{do}$ for some $0 \le i < N_\mathcal{F}\}$;
– $B = \{$"`y` $(i)$" $: y \in f_i^{io}$ for some $0 \le i < N_\mathcal{F}$ and $y \notin f_i^{do}$ for any $0 \le i < N_\mathcal{F}\}$.

Then we draw an edge from node $a = `y\ (i)''$ to node $b =$ "$x\ (j)''$ if and only if the following conditions are satisfied:
– $i \le j$,
– $x$ is a destination operand of instruction $j$,
– $y$ is an input operand of instruction $j$,
– $y$ 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 $i \le j$, $x \in f_j^{do}$, $y \in f_j^{io}$ and "`y` $(k)$" $\notin V$ $\forall i + 1 \le k < j$. 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 $f$ and $b$ as an *output node* of $f$. 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 MOF graph $G_{\mathcal{F}_{ex},full}$ corresponding to the assembly program $\mathcal{F}_{ex}$ in Tab. 1 is shown in Fig. 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 previous instructions. 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 Fig. 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.

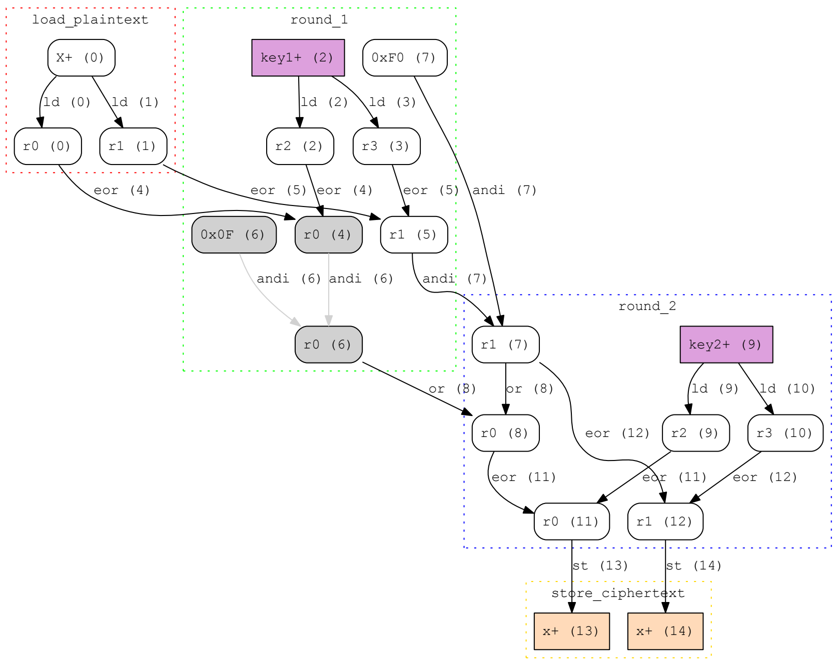


Fig. 2: MOF graph $G_{\mathcal{F}_{ex},full}$ corresponding to the assembly program $\mathcal{F}_{ex}$ in Tab. 1 constructed by ATLAS.

*Example 3.* In Fig. 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 sequence number of the instruction that stores this word. We refer to them as *the words of the ciphertext*. For example, in Fig. 2, "`x+` (13)" and "`x+` (14)" are the words of the ciphertext.

## 4.2 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 $e_1 = (v, x_1), e_2 = (x_2, x_3), e_3 = (x_3, x_4), \ldots, e_{k-1} = (x_{k-1}, x_k), e_k = (x_k, u)$. 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 Tabs. 2 and 3.

1. In Tab. 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

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` |

changed in `x+`. Knowing how `r1` is changed and values of ciphertext with and without fault injection won't give us any information about `r0`.

2. In Tab. 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. 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 *non-linear (with respect to a bit-wise operation)* if $f^{mn} \in \{$`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 Fig. 2, "`x+` (13)" is a Gchild of "`r0` (6)" with distance 1. "`r0` (8)" is a Gchild of "`key+` (2)" with distance 4.[1]

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 and therefore, relates to a key, we have to focus on nodes affected by the fault, which are at zero Gdistance from the key word nodes. 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 Gchildrean, say $ch$, of $a$ is a Gchild of $b$ with Gdistance$(b, ch) = 0$. And we say $a$ is *related to a round key* `key` if it is related to at least one key word node of `key`.

To decide if a node $a$ is vulnerable for DFA, we first need to output a set of nodes which are suitable for DFA. For a given node $a$ which is to be examined, the possible parameters that can be specified includes:

- `minAffectedCT`: $|\text{CTGchild}| \geq$ `minAffectedCT`, i.e. the number of nodes in CTGchild is bigger or equal than `minAffectedCT`;
- `minDist`: $|\{ch : ch \in \text{CTGchild, and Gdistance}(a, ch) \geq$ `minDist`$\}| \geq$ `minAffectedCT`, i.e. the number of nodes in CTGchild with Gdistance at least `minDist` from $a$ is at least `minAffectedCT`;

[1] We would like to point out that there are two directed paths from "`key1+` (2)" to "`r0` (8)", showing that in general MOF graphs are not directed trees.

- `maxDist`: Gdistance$(a, ch) \leq$ `maxDist` $\forall ch$ $\in$Gchild, i.e. the Gdistance between any Gchild and $a$ should be at most `maxDist`;
- `maxKey`: the number of round keys 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`.

The selection of values for each of the above parameters is referred to as an *output criteria.* ATLAS takes an MOF 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.

In general, `minAffectedCT` tells us 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 the differential fault attack, the `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 our calculations. If there are too many non-linear instructions, the fault propagation may be too scattered, resulting into too many possibilities to consider. For a similar reason, the value of `maxKey` should not be too big, otherwise some nodes in the output will be associated with too many non-linear instructions. For the obvious reason, `minKeyWords` should be set to be at least 1.

We note that the values of aforementioned parameters 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. Accordingly, `minKeyWords` should be set to a small value. Or, if the user would like to have all the ciphertext words being affected, i.e. setting `minAffectedCT`=to the number of ciphertext words, the other conditions should be loosened. For example, for the MOF graph $G_{\mathcal{F}_{ex},full}$ in Fig. 2, with an output criteria (`minAffectedCT, minDist, maxDist, maxKey, minKeyWords`) $= (2, 1, 1, 1, 1)$ we cannot get any output from ATLAS.

We suggest to use relatively loose output criteria as a preliminary test to see what are the possibilities, then tighten the criteria to find possible vulnerable nodes.

For the MOF graph in Fig. 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.

### 4.3 Subgraph Construction

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

Given an MOF 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 $1 \leq i \leq \texttt{depth}$, $U_i = \{b : b$ 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 $k \in B\}$

$-\ V_4 = \{b : b$ is a child of a node $v \in \texttt{KNGchild}$ and $b$ is a parent of a child of $a\}$.

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

In Fig. 3 (a) and (b) we present the subgraphs constructed from node "`r0` (6)" of the MOF graph $G_{\mathcal{F}_{ex},full}$ (Fig. 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)"$\}$

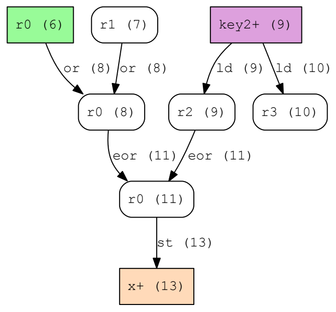
From the two figures, we can see that with `depth`$= 1$, we do get extra useful information: the two edges with label "`andi` (7)" show that the first four bits of "`r0` (6)" are 0.

### 4.4 Equation Construction

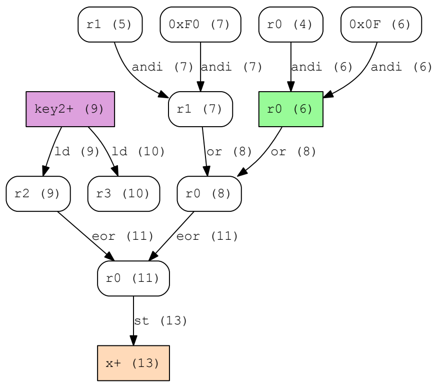
Having the subgraph, constructed from a potentially vulnerable node, we would like to construct equations out of the subgraph to connect different input/output nodes, which can be easily analyzed by algebraic methods.

Given any subgraph $G_a = (V_a, E_a) \subseteq G_{\mathcal{F},full}$, where $G_{\mathcal{F},full}$ is the MOF 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 Tab. 4 we show some representations of equations for different mnemonics. Here, the symbol "|" represents concatenation. For example, take $f =$



(a)



(b)

Fig. 3: Subgraph constructed from node "`r0` (6)" of MOF graph in Fig. 2 with depth (a) 0 and (b) 1.

Table 4: Construction of equations from assembly instructions.

| Instruction | Equation |
|---|---|
| `ADD r2 r3` | `carry` $\mid$ `r2` $=$ `r2` $+$ `r3` |
| `ADC r2 r3` | `carry` $\mid$ `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` $\mid$ `r0` $=$ `r2` $\times$ `r3` |
| `LD/MOV/ST r2 r3` | `r2` $=$ `r3` |
| `ROL r2` | `carry` $\mid$ `r2` $=$ `r2` $\mid$ `carry` |
| `LSL r2` | `carry` $\mid$ `r2` $=$ `r2` $\mid$ `0` |
| `LPM r2 Z` | `r2` $=$ `TableLookUp(ZH` $\mid$ `ZL)` |

`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`.

In case the equation is related to an instruction that loads a round key, ATLAS is designed to indicate which key word node is involved in the equation (see Remark 1).

Now let us look at the assembly program $\mathcal{F}_{ex}$ for our sample cipher from Tab. 1. Following the definition in Sec. 4.1, the MOF graph $G_{\mathcal{F}_{ex},full}$ for this sample cipher was constructed (see Fig. 2). Then, we applied the output criteria described in Sec. 4.2 to $G_{\mathcal{F}_{ex},full}$ and obtained two vulnerable nodes "`r0` (6)" and "`r1` (7)". The subgraphs with depths 0 and 1, constructed from "`r0` (6)", are shown in Fig. 3. As we pointed out in Sec. 4.3, the subgraph with depth 1 gives some additional useful information, compared to the one with depth 0.

The equations obtained by using ATLAS from the subgraph with depth 1, constructed from “`r0` (6)” (Fig. 3 (b)), are as follows:

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

Eq. (1) shows “`r0` (6)” $= \mathtt{0000}b_4b_5b_6b_7$ for some $b_j \in \{0,1\}$ $(j = 4,5,6,7)$. Equation (3) shows that if we skip instruction 8, the result of Eq. (1) will be used instead of the result of Eq. (3) in instruction 11, which corresponds to Eq. (5). Together with the information from Eqs. (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.

*Remark 1.* The index [0] in the right hand of Eq. (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.

## 5 Case Study

In this section, we will describe a fault attack on PRESENT that was automatically generated by ATLAS. 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. [13, 8, 10, 16, 17]), 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 [7]. 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 Fig. 4. Because of its lightweight character, PRESENT-80 is usually used, therefore we focus on this variant in this section.

As a target, we chose a speed-optimized assembly implementation for 8-bit AVR from Verstegen and Papagiannopoulos, publicly available on GitHub[2].

[2] `https://github.com/kostaspap88/PRESENT_speed_implementation`

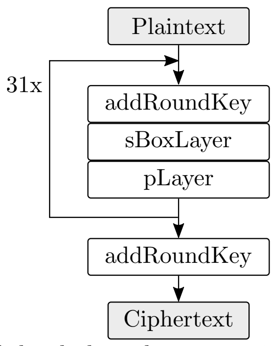


Fig. 4: High-level algorithmic overview of PRESENT cipher.

| # | Instruction |
|---|---|
| 0 | `LDI ZH 0x06` |
| 1 | `MOV ZL r0` |
| 2 | `LPM r21 Z` |
| 3 | `ANDI r21 0xC0` |
| 4 | `LDI ZH 0x07` |
| 5 | `MOV ZL r1` |
| 6 | `LPM r2 Z` |
| 7 | `ANDI r23 0x30` |
| 8 | `OR r21 r23` |

Table 5: Assembly code of a table look-up for PRESENT implementation.

However, we did not use the key schedule for our analysis, since we were targeting the main encryption routine.

## 5.2 Fault Analysis

We have only used the last three rounds of the cipher, since the output criteria we selected for the attack would exclude all the earlier nodes in case the full cipher was used. This code was 499 instructions long. The running time for the assembly analysis was 36 ms: 8 ms reading the assembly source file, 8 ms constructing the MOF graph, 4 ms finding the vulnerable nodes and 16 ms outputting the subgraphs and fault difference equations for all the nodes that satisfy the output criteria (testing was done on a standard Intel Haswell family CORE i7 processor with 8 GB RAM).

In order to get the vulnerable nodes from the cipher implementation, we have chosen our output criteria to be (`minAffectedCT, minDist, maxDist, minKey, minKeyWords`)$= (1,1,1,1,1)$. With this output criteria, ATLAS outputs 16 vulnerable nodes, out of the total 512 nodes. We will explain the fault attack procedure on one of these nodes: "`r23` (374)". Subgraph for "`r23` (374)" with depth 1 is stated in Fig. 6 in Appendix A.

Equations generated for the subgraph with depth 1 from "`r23` (374)" are as follows:

$$\text{“}\texttt{r22}\text{ (366)”} = \text{“}\texttt{r22}\text{ (357)”} \vee \text{“}\texttt{r23}\text{ (365)”} \tag{7}$$

$$\text{“}\texttt{r23}\text{ (374)”} = \text{“}\texttt{r23}\text{ (373)”} \wedge \text{“}\texttt{0x03}\text{ (72)”} \tag{8}$$

$$\text{“}\texttt{r22}\text{ (375)”} = \text{“}\texttt{r22}\text{ (366)”} \vee \text{“}\texttt{r23}\text{ (374)”} \tag{9}$$

$$\text{“}\texttt{r1}\text{ (476)”} = \texttt{key4[1]} \tag{10}$$

$$\text{“}\texttt{r1}\text{ (484)”} = \text{“}\texttt{r1}\text{ (476)”} \oplus \text{“}\texttt{r22}\text{ (375)”} \tag{11}$$

$$\text{“}\texttt{x}\text{ (492)”} = \text{“}\texttt{r1}\text{ (484)”}. \tag{12}$$

Eq. (8) shows "`r23` (374)" $= \texttt{000000}b_6b_7$ for some $b_6, b_7 \in \{0,1\}$. Together with the other equations we get

$$\text{“}\texttt{x}\text{ (492)”} = \texttt{key4[1]} \oplus (\text{“}\texttt{r22}\text{ (366)”} \vee \texttt{000000}b_6b_7) \tag{13}$$

Consider a bit flip fault injection with fault mask $\Delta = \texttt{11111100}$ in "`r23` (374)" right before the execution of instruction 375, which corresponds to Eq. (9), then Eq. (13) becomes:

$$\text{“}\texttt{x'}\ (492)\text{”} = \texttt{key4[1]} \oplus (\text{“}\texttt{r22}\ (366)\text{”} \vee \texttt{111111}b_6b_7) \tag{14}$$

where "`x'` (492)" denotes the faulted output. Let $\delta = \delta_0\delta_1\delta_2\delta_3\delta_4\delta_5\delta_6\delta_7 =$ "`x'` (492)" $\oplus$ "`x` (492)" and let "`r22` (366)" $= a_0a_1a_2a_3a_4a_5a_6a_7$. Since both $\oplus$ and $\vee$ are bitwise operations, together with equations (13) and (14) we have

$$\begin{aligned}\delta_0\delta_1\delta_2\delta_3\delta_4\delta_5 &= (a_0a_1a_2a_3a_4a_5 \vee \texttt{000000}) \oplus (a_0a_1a_2a_3a_4a_5 \vee \texttt{111111}) \\ &= a_0a_1a_2a_3a_4a_5 \oplus 111111 \Longrightarrow a_0a_1a_2a_3a_4a_5 = \delta_0\delta_1\delta_2\delta_3\delta_4\delta_5 \oplus 1111111.\end{aligned}$$

Since the value of $\delta$ is known and the value of "`x` (492)" is also known, together with Eq. (13), we have

$$\text{the first 6 bits of } \texttt{key4[1]} = \text{first 6 bits of “}\texttt{x}\ (492)\text{”} \oplus \delta_0\delta_1\delta_2\delta_3\delta_4\delta_5 \oplus 1111111,$$

which gives the first 6 bits of the second byte of round key for round 4.

With a subgraph constructed from "`r22` (366)" with depth 3, a similar fault analysis helps us to recover the last 2 bits of `key4[1]`. The subgraph is stated in Fig. 7 in Appendix A. The same analysis can be carried out for the remaining 14 nodes to get all the bits of the round key for round 4.

To understand the found vulnerability, we examined the assembly code and provide an explanation below on why the cipher structure contains the exploitable operations output by ATLAS. 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. Tab. 5 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

To test our ATLAS tool, we analyzed two more lightweight ciphers: SIMON and SPECK [6]. For this purpose, we used an assembly implementation for 8-bit AVR from Luo Peng, available from GitHub[3]. More specifically, we tested SIMON 64/128 and SPECK 64/128, both high-throughput implementations. Results are

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

shown in Fig. 5, which plots numbers of vulnerable nodes for various parameters. Obviously, because of the structure of these ciphers, where only half of the state is directly related to the round key, the number of vulnerable nodes is lower compared to PRESENT. However, these results show that ATLAS is capable of finding the vulnerable spots automatically in different implementations, without additional knowledge of the internal cipher structure.

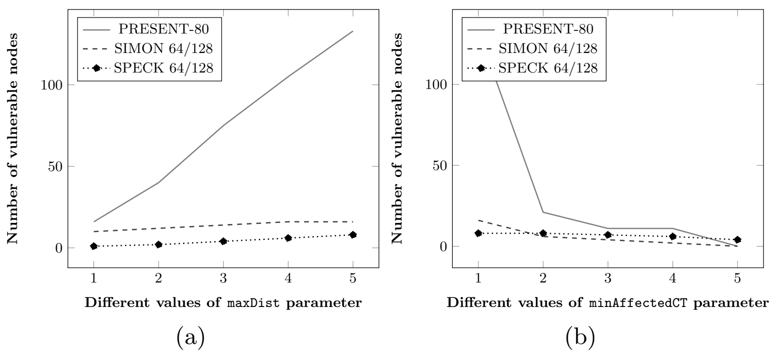


Fig. 5: Comparison of different output criteria on different ciphers. Plot (a) shows varying the `maxDist` parameter, while the other parameters are (`minAffectedCT, minDist, minKey, minKeyWords`) = $(1, 1, 1, 1)$. Plot (b) varies the `minAffectedCT` parameter, while the other parameters are (`maxDist, minDist, minKey, minKeyWords`) = $(5, 1, 1, 1)$.

## 7 Conclusion

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

We have presented a detailed overview of a fault attack on PRESENT-80, exploiting implementation weaknesses found by ATLAS. Our results show that by using our tool, 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 cipher implementations, SPECK 64/128 and SIMON 64/128.

For the future work, we would like to extend ATLAS 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. [5]) to find loopholes and propose additional countermeasures automatically.

## References

1. The LLVM compiler infrastructure project. http://llvm.org, accessed: 2017-04-13
2. Agosta, G., Barenghi, A., Pelosi, G., Scandale, M.: Differential fault analysis for block ciphers: An automated conservative analysis. In: Proceedings of the 7th International Conference on Security of Information and Networks. pp. 137:137–137:144. SIN '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2659651.2659709
3. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. Proceedings of the IEEE 94(2), 370–382 (Feb 2006)
4. Barenghi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. Proceedings of the IEEE 100(11), 3056–3076 (Nov 2012)
5. Barry, T., Couroussé, D., Robisson, B.: Compilation of a countermeasure against instruction-skip fault attacks. In: Proceedings of the Third Workshop on Cryptography and Security in Computing Systems. pp. 1–6. CS2 '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2858930.2858931
6. Beaulieu, R., Treatman-Clark, S., Shors, D., Weeks, B., Smith, J., Wingers, L.: The simon and speck lightweight block ciphers. In: 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6 (June 2015)
7. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems. pp. 450–466. CHES '07, Springer-Verlag, Berlin, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-74735-2_31
8. Breier, J., He, W.: Multiple fault attack on present with a hardware trojan implementation in fpga. In: 2015 International Workshop on Secure Internet of Things (SIoT). pp. 58–64 (Sept 2015)
9. Click, C., Paleczny, M.: A simple graph-based intermediate representation. In: Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations. pp. 35–49. IR '95, ACM, New York, NY, USA (1995), http://doi.acm.org/10.1145/202529.202534
10. De Santis, F., Guillen, O.M., Sakic, E., Sigl, G.: Ciphertext-only fault attacks on present. In: Eisenbarth, T., Öztürk, E. (eds.) Lightweight Cryptography for Security and Privacy: Third International Workshop, LightSec 2014, Istanbul, Turkey, September 1-2, 2014. pp. 85–108
11. Dey, P., Rohit, R.S., Adhikari, A.: Full key recovery of acorn with a single fault. J. Inf. Secur. Appl. 29(C), 57–64 (Aug 2016), http://dx.doi.org/10.1016/j.jisa.2016.03.003
12. Dureuil, L., Potet, M.L., de Choudens, P., Dumas, C., Clédière, J.: From code review to fault injection attacks: Filling the gap using fault model inference. In: Homma, N., Medwed, M. (eds.) Smart Card Research and Advanced Applications: 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers. pp. 107–124. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-31271-2_7
13. Ghalaty, N.F., Yuce, B., Schaumont, P.: 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. pp. 174–188

14. Given-Wilson, T., Jafri, N., Lanet, J.L., Legay, A.: An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT – Extended Version (Apr 2017), https://hal.inria.fr/hal-01400283, working paper or preprint
15. Goubet, L., Heydemann, K., Encrenaz, E., De Keulenaer, R.: Efficient design and evaluation of countermeasures against fault attacks using formal verification. In: Homma, N., Medwed, M. (eds.) Smart Card Research and Advanced Applications: 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers. pp. 177–192. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-31271-2_11
16. Gu, D., Li, J., Li, S., Ma, Z., Guo, Z., Liu, J.: Differential fault analysis on lightweight blockciphers with statistical cryptanalysis techniques. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2012 Workshop on. pp. 27–33 (Sept 2012)
17. Jeong, K., Lee, Y., Sung, J., Hong, S.: Improved differential fault analysis on present-80/128. International Journal of Computer Mathematics 90(12), 2553–2563 (2013)
18. Jovanovic, P., Kreuzer, M., Polian, I.: A fault attack on the led block cipher. In: Schindler, W., Huss, S.A. (eds.) Constructive Side-Channel Analysis and Secure Design: Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings. pp. 120–134. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), https://doi.org/10.1007/978-3-642-29912-4_10
19. Khanna, P., Rebeiro, C., Hazra, A.: XFC: A Framework for eXploitable Fault Characterization in Block Ciphers. In: Proceedings of the 54th Annual Design Automation Conference 2017. pp. 8:1–8:6. DAC '17, ACM, New York, NY, USA (2017), http://doi.acm.org.ezlibproxy1.ntu.edu.sg/10.1145/3061639.3062340
20. Paleczny, M., Vick, C., Click, C.: The java hotspottm server compiler. In: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1. pp. 1–1. JVM'01, USENIX Association, Berkeley, CA, USA (2001), http://dl.acm.org/citation.cfm?id=1267847.1267848
21. Tunstall, M., Mukhopadhyay, D.: 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

## A Subgraphs for Fault Analysis of PRESENT

In the following graphs, we highlight the nodes (in gray) which give information about the bits of the faulted nodes (in green).

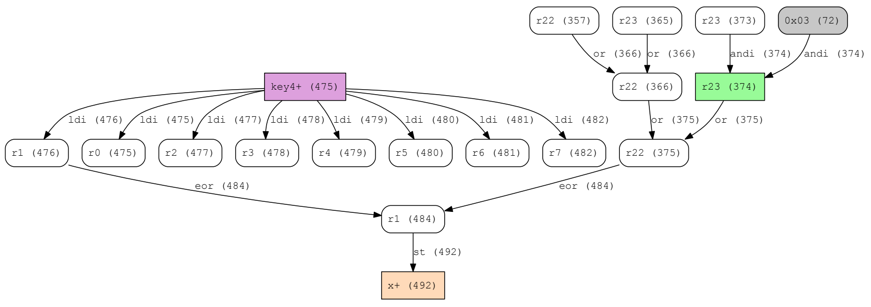


Fig. 6: Subgraph with depth 3 generated from the assembly implementation of PRESENT, corresponding to vulnerable node “**r23** (374)”.

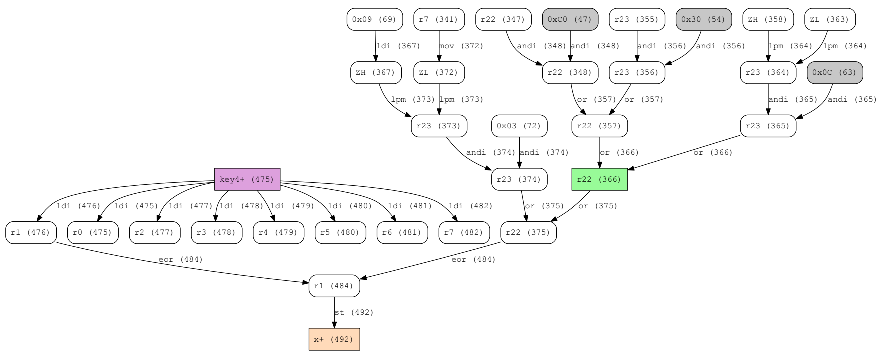


Fig. 7: Subgraph with depth 3 generated from the assembly implementation of PRESENT, corresponding to vulnerable node “**r22** (366)”.

## B Class Diagram of ATLAS

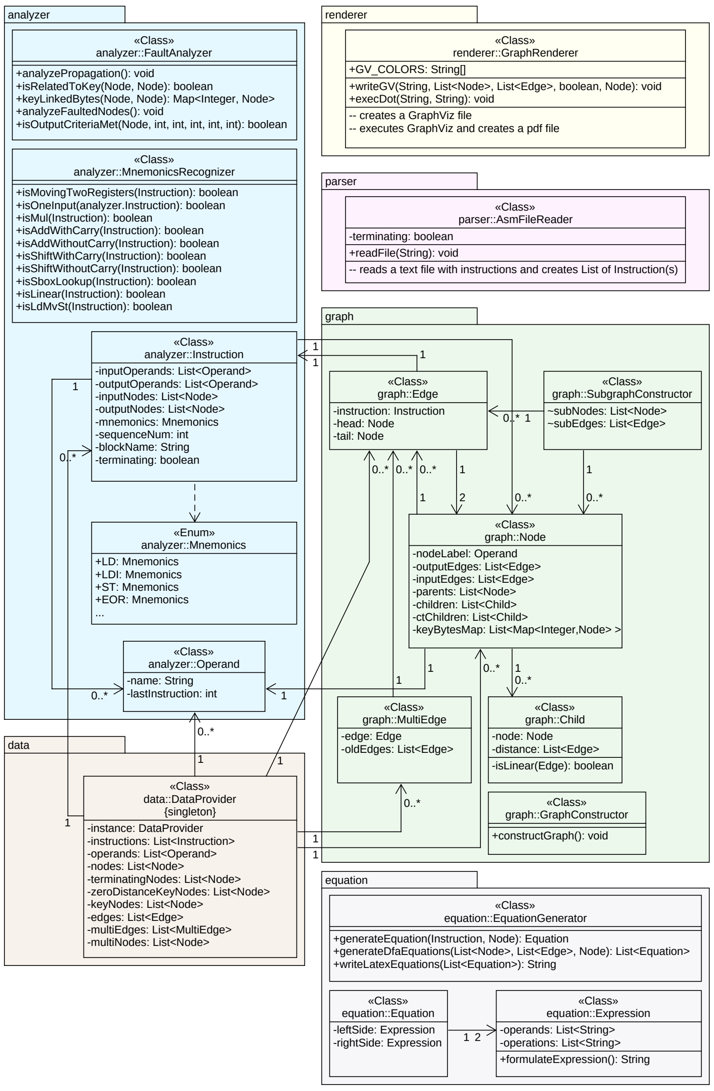

analyzer
«Class»
analyzer::FaultAnalyzer
+analyzePropagation(): void
+isRelatedToKey(Node, Node): boolean
+keyLinkedBytes(Node, Node): Map<Integer, Node>
+analyzeFaultedNodes(): void
+isOutputCriteriaMet(Node, int, int, int, int, int): boolean
«Class»
analyzer::MnemonicsRecognizer
+isMovingTwoRegisters(Instruction): boolean
+isOneInput(analyzer.Instruction): boolean
+isMul(Instruction): boolean
+isAddWithCarry(Instruction): boolean
+isAddWithoutCarry(Instruction): boolean
+isShiftWithCarry(Instruction): boolean
+isShiftWithoutCarry(Instruction): boolean
+isSboxLookup(Instruction): boolean
+isLinear(Instruction): boolean
+isLdMvSt(Instruction): boolean
«Class»
analyzer::Instruction
-inputOperands: List<Operand>
-outputOperands: List<Operand>
-inputNodes: List<Node>
-outputNodes: List<Node>
-mnemonics: Mnemonics
-sequenceNum: int
-blockName: String
-terminating: boolean
«Enum»
analyzer::Mnemonics
+LD: Mnemonics
+LDI: Mnemonics
+ST: Mnemonics
+EOR: Mnemonics
...
«Class»
analyzer::Operand
-name: String
-lastInstruction: int
renderer
«Class»
renderer::GraphRenderer
+GV_COLORS: String[]
+writeGV(String, List<Node>, List<Edge>, boolean, Node): void
+execDot(String, String): void
-- creates a GraphViz file
-- executes GraphViz and creates a pdf file
parser
«Class»
parser::AsmFileReader
-terminating: boolean
+readFile(String): void
-- reads a text file with instructions and creates List of Instruction(s)
graph
«Class»
graph::Edge
-instruction: Instruction
-head: Node
-tail: Node
«Class»
graph::SubgraphConstructor
~subNodes: List<Node>
~subEdges: List<Edge>
«Class»
graph::Node
-nodeLabel: Operand
-outputEdges: List<Edge>
-inputEdges: List<Edge>
-parents: List<Node>
-children: List<Child>
-ctChildren: List<Child>
-keyBytesMap: List<Map<Integer,Node> >
«Class»
graph::MultiEdge
-edge: Edge
-oldEdges: List<Edge>
«Class»
graph::Child
-node: Node
-distance: List<Edge>
-isLinear(Edge): boolean
«Class»
graph::GraphConstructor
+constructGraph(): void
data
«Class»
data::DataProvider
{singleton}
-instance: DataProvider
-instructions: List<Instruction>
-operands: List<Operand>
-nodes: List<Node>
-terminatingNodes: List<Node>
-zeroDistanceKeyNodes: List<Node>
-keyNodes: List<Node>
-edges: List<Edge>
-multiEdges: List<MultiEdge>
-multiNodes: List<Node>
equation
«Class»
equation::EquationGenerator
+generateEquation(Instruction, Node): Equation
+generateDfaEquations(List<Node>, List<Edge>, Node): List<Equation>
+writeLatexEquations(List<Equation>): String
«Class»
equation::Equation
-leftSide: Expression
-rightSide: Expression
«Class»
equation::Expression
-operands: List<String>
-operations: List<String>
+formulateExpression(): String
1
0..*
1 2

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