Fault Analysis Automation on Software Targets

ASCEHS Workshop, IIT Kharagpur

Jakub Breier
3 July 2018

Physical Analysis and Cryptographic Engineering
Nanyang Technological University, Singapore
1. DFA Automation on Assembly Implementations

2. Automated Evaluation of Software Encoding Countermeasures

3. Conclusion
Why Automation?

- All the current symmetric block ciphers have been shown vulnerable against fault attacks (especially DFA).
- The question is not whether the algorithm is secure or not, but which part of it is insecure.
- Automated methods can provide an answer fast and with minimal need of human intervention.
DFA Automation on Assembly Implementations
Motivation

- In practice, the attack always has to be mounted on a real-world device.
- Different implementations of the same encryption algorithm do not necessarily share the same vulnerabilities.
- There might be an exploitable spot in the implementation that is not visible from the cipher design.
- 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.
Assumptions

- Known-ciphertext model and a single fault adversary.
- The implementation is available to the attacker and he can add annotations to the assembly code for the purpose of distinguishing different rounds, round keys, ciphertext words, etc.
- For the analysis in this work, we have chosen Atmel AVR instruction set. However, for analyzing different instruction sets, only the parsing subsystem of the analyzer has to be redefined.
- The implementation is unrolled, no direct/indirect jumps.
Assembly Program

Definition

- **program**: an ordered sequence of assembly instructions
  \[ \mathcal{F} = (f_0, f_1, \ldots, f_{N_{\mathcal{F}}-1}). \]
- \( N_{\mathcal{F}} \): the number of instructions for the program.
- For each instruction \( f \in \mathbb{F} \), we associate \( f \) with a 4-tuple \((f^{seq}, f^{mn}, f^{io}, f^{do})\)
  - \( f^{seq} \): sequence number
  - \( f^{mn} \): mnemonic of \( f \).
  - \( f^{io} \): 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.
Definition
For a pair of nodes $v$ and $u$ such that $u$ is a child of $v$, the $Gdistance$ between $v$ and $u$, denoted by $Gdistance(v,u)$ is defined to be the cardinality of the following set:

$$\{ e : e \text{ belongs to a directed path from } v \text{ to } u \text{ and } e \text{ is non-linear} \}.$$
### Assembly Code $F_{ex}$ for a Sample Cipher

<table>
<thead>
<tr>
<th>#</th>
<th>Instruction</th>
<th>#</th>
<th>Instruction</th>
<th>#</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LD r0 X+</td>
<td>5</td>
<td>EOR r1 r3</td>
<td>10</td>
<td>LD r3 key2+</td>
</tr>
<tr>
<td>1</td>
<td>LD r1 X+</td>
<td>6</td>
<td>ANDI r0 0x0F</td>
<td>11</td>
<td>EOR r0 r2</td>
</tr>
<tr>
<td>2</td>
<td>LD r2 key1+</td>
<td>7</td>
<td>ANDI r1 0xF0</td>
<td>12</td>
<td>EOR r1 r3</td>
</tr>
<tr>
<td>3</td>
<td>LD r3 key1+</td>
<td>8</td>
<td>OR r0 r1</td>
<td>13</td>
<td>ST x+ r0</td>
</tr>
<tr>
<td>4</td>
<td>EOR r0 r2</td>
<td>9</td>
<td>LD r2 key2+</td>
<td>14</td>
<td>ST x+ r1</td>
</tr>
</tbody>
</table>

### Example

- $N_{F_{ex}} = 15$.
- $f_6 = \text{ANDI } r0 \ 0x0F$.
- $f_6$ is associated with the 4–tuple $(6, \text{ANDI}, \{r0, 0x0F\}, \{r0\})$. 
<table>
<thead>
<tr>
<th>#</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LD r0 X+</td>
</tr>
<tr>
<td>1</td>
<td>LD r1 X+</td>
</tr>
<tr>
<td>2</td>
<td>LD r2 key1+</td>
</tr>
<tr>
<td>3</td>
<td>LD r3 key1+</td>
</tr>
<tr>
<td>4</td>
<td>EOR r0 r2</td>
</tr>
<tr>
<td>5</td>
<td>EOR r1 r3</td>
</tr>
<tr>
<td>6</td>
<td>ANDI r0 0x0F</td>
</tr>
<tr>
<td>7</td>
<td>ANDI r1 0xF0</td>
</tr>
<tr>
<td>8</td>
<td>OR r0 r1</td>
</tr>
<tr>
<td>9</td>
<td>LD r2 key2+</td>
</tr>
<tr>
<td>10</td>
<td>LD r3 key2+</td>
</tr>
<tr>
<td>11</td>
<td>EOR r0 r2</td>
</tr>
<tr>
<td>12</td>
<td>EOR r1 r3</td>
</tr>
<tr>
<td>13</td>
<td>ST x+ r0</td>
</tr>
<tr>
<td>14</td>
<td>ST x+ r1</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>#</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LD r0 key0+</td>
</tr>
<tr>
<td>1</td>
<td>AND r1 r0</td>
</tr>
<tr>
<td>2</td>
<td>ST x+ r1</td>
</tr>
</tbody>
</table>

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

• $f_1 = \text{AND } r1 \ r0$.
• $\vartheta_1$: a fault is injected at $f_1$ such that some bits in $r1$ are flipped before the execution of AND.
• Suppose the first bit of $r1$ is flipped.
• Observe the first bit of $x+$ to get the first bit of $r0$, which is the first bit of the key.

• $c = a \ & \ b$.
• Inject fault in $b$ by flipping it.
• Observe the change in $c$ to get the value of $a$. 
• A node $a$ is related to a key word node $b$ if $b$ is not a parent of $a$ and at least one of the children, say $ch$, of $a$ is a child of $b$ with $G\text{distance}(b, ch) = 0$. The distance condition specifies that we are only looking at nodes that are linearly related to the key.

• $a$ is related to a round key key if it is related to at least one key word node of key.
Output Criteria – Selection of Vulnerable Nodes

- **minAffectedCT**: minimal number of ciphertext nodes affected by the vulnerable node;
- **minDist**: minimal Gdistance between the node and a ciphertext node for at least \( \text{minAffectedCT} \) nodes;
- **maxDist**: maximum distance between the node and all the ciphertext nodes;
- **maxKey**: the number of the round keys, counting from the last round key, that are related to node \( a \) is at most \( \text{maxKey} \);
- **minKeyWords**: there exists at least one round key such that the number of its corresponding key word nodes related to \( a \) is at least \( \text{minKeyWords} \).
Subgraph Construction

- Given $G_{\mathcal{F},\text{full}} = (V, E)$ node $a \in V$ subgraph $G_a = (V_a, E_a)$ is a pair such that $V_a \subseteq V$ and $E_a \subseteq E$.
- $\text{KNGchild}$: the set of key word nodes that are related to $a$.
- $\text{depth}$: user input.
- $V_a = \left( \bigcup_{i=0}^{\text{depth}} U_i \right) \cup \left( \bigcup_{j=1}^{4} V_j \right)$
- $U_0 = \{ b : a, b \in f^{io} \}$
- $U_i = \{ b : b \in f^{io} \text{ s.t. } v \in f^{do} \text{ for some } v \in U_{i-1} \}$
- $V_1 = \{ b : b \text{ is a child of } a \}$
- $V_2 = \{ k : k \text{ is a round key that is related to } a \}$
- $V_3 = \{ b : b \text{ is a key word node for a key } k \in V_2 \}$
- $V_4 = \{ b : b \text{ is a child of a node } v \in \text{KNGchild} \text{ and } b \text{ is a parent of a child of } a \}$. 
Subgraphs for node “r0 (6)” with depth (a) 0 and (b) 1, output
criteria (minAffectedCT, minDist, maxDist, maxKey, minKeyWords) = (1, 1, 1, 1, 1)
## Construction of equations from assembly instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD r2 r3</td>
<td>carry</td>
</tr>
<tr>
<td>ADC r2 r3</td>
<td>carry</td>
</tr>
<tr>
<td>EOR r2 r3</td>
<td></td>
</tr>
<tr>
<td>AND r2 r3</td>
<td></td>
</tr>
<tr>
<td>OR r2 r3</td>
<td></td>
</tr>
<tr>
<td>MUL r2 r3</td>
<td></td>
</tr>
<tr>
<td>LD/MOV/ST r2 r3</td>
<td></td>
</tr>
<tr>
<td>ROL r2</td>
<td>carry</td>
</tr>
<tr>
<td>LSL r2</td>
<td>carry</td>
</tr>
<tr>
<td>LPM r2 Z</td>
<td></td>
</tr>
</tbody>
</table>
“r0 (6)” = “r0 (4)” \and “0xF0 (6)” \hspace{1em} (1)
“r1 (7)” = “r1 (5)” \and “0xF0 (7)” \hspace{1em} (2)
“r0 (8)” = “r0 (6)” \or “r1 (7)” \hspace{1em} (3)
“r2 (9)” = key2[0] \hspace{1em} (4)
“r0 (11)” = “r0 (8)” \xor “r2 (9)” \hspace{1em} (5)
“x+ (13)” = “r0 (11)” \hspace{1em} (6)

(1): “r0 (6)” = 0000b_{4}b_{5}b_{6}b_{7}, \quad b_{j} \in \{0, 1\} \quad (j = 4, 5, 6, 7).
(3): if we skip instruction 8, the result of (1) will be used in instruction 11 (5) instead of the result of (3)
(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.
Recall – PRESENT Cipher

- Block length: 64 bits
- Key length: 128 bits or 80 bits
- `addRoundKey`: xor with the round key
- `sBoxLayer`: 4-bit SBox
- `pLayer`: bitwise permutation
- PRESENT-80
Data Flow Graph of PRESENT – Full Version
New Attack found on PRESENT-80

- We chose a speed-optimized assembly implementation for 8-bit AVR publicly available on GitHub\(^2\).
- Output criteria: \((\text{minAffectedCT}, \text{minDist}, \text{maxDist}, \text{maxKey}, \text{minKeyWords})=(1,1,1,1,1)\).
- Recover the last round key by 16 fault injections.
- Implementation specific.
- Existing DFA on PRESENT exploit Sbox look up which requires the analysis of the whole Sbox table.
- Our new attack exploits OR operation which only requires the analysis of a simple truth table.

\(^2\)https://github.com/kostaspap88/PRESENT_speed_implementation
New Attack found on PRESENT-80
New Attack found on PRESENT-80
Scalability of DATAc tested on AES with different number of rounds

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

Data collected on laptop computer with mobile Intel Haswell family CORE i7 processor, 8 GB RAM
• Proposed a methodology capable of finding spots vulnerable to DFA in software implementations of cryptographic algorithms.

• Created DATAC, which takes an assembly implementation and a user-specified output criteria as an input.

• DATAC outputs subgraphs for vulnerable nodes in the code, together with equations that can be directly used for DFA on the cipher implementation.

• New attacks on PRESENT-80, exploiting implementation-specific weaknesses.

• DATAC is scalable and can analyze current algorithms efficiently.
Automated Evaluation of Software Encoding Countermeasures
Motivation

- Encoding countermeasures are a subclass of redundancy-based countermeasures.
- If operations are calculated using table look-up, they can offer a solid protection against fault attacks\(^3\).
- It is necessary to check the robustness of the selected code and also of the implementation that is used.
- Automated method presented in this part deals with the evaluation of the implementation.

\(^3\)J. Breier, D. Jap, S. Bhasin. The other side of the coin: Analyzing software encoding schemes against fault injection attacks, HOST’16.
A binary code, denoted by $C$, is a subset of the $n$–dimensional vector space over $\mathbb{F}_2^n$, where $n$ is called the length of the code $C$.

Take two codewords $c, c' \in C$, the Hamming distance between $c$ and $c'$, denoted by $\text{dis}(c, c')$, is defined to be the number of places at which $c$ and $c'$ differ.

Furthermore, for a binary code $C$, the (minimum) distance of $C$, denoted by $\text{dis}(C)$, is

$$\text{dis}(C) = \min\{\text{dis}(c, c') : c, c' \in C, c \neq c'\}.$$
From Codes to Anticodes

Definition
A binary anticode is an array of binary digits with $n$ rows and $M$ columns, constructed so that the maximum Hamming distance between any pair of rows is less than or equal to a certain value $\delta$. This value $\delta$ is the *maximum distance* of the anticode $^4$.

Definition
Let $C$ be an $(n, M, d)$—binary code, if furthermore

$$\max_{c, c' \in C} \text{dis}(c, c') = \delta,$$

where $d \leq \delta \leq n$, then $C$ is called an $(n, M, d, \delta)$—binary anticode. Furthermore, $d$ (resp. $\delta$) is called the *minimum distance* (resp. *maximum distance*) of $C$.

Overview of a Generalized Encoding Scheme

- Inputs are encoded based on the scheme.
- Operations are done on encoded data.
- Final output is decoded.
Types of Faults

- **Safe fault** – detected by the scheme, unable to be exploited by the DFA.
- **Exploitable fault** – undetected fault, resulting to an output that can be exploited by the DFA.

**Definition (Safe and exploitable faults)**

For a fixed plaintext $P \in \mathcal{P}$ and a key $\kappa \in \mathcal{K}$, a fault $\varrho_1$ on $\mathcal{F}_1$ is **safe** if $\varrho_1(\mathcal{F}_1)(\kappa, P) = \bot$ or $E(\kappa, P)$ and it is **exploitable** otherwise.

Similarly, a fault $\varrho_2$ on $\mathcal{F}_2$ is **safe** if $\varrho_2(\mathcal{F}_2)(\kappa, P) = \bot$ or $D(\kappa, P)$ and it is **exploitable** otherwise.
Algorithm 1: Anticode Generation Algorithm.

Input: $n$ : length of the anticode, $M$ : number of codewords, $d$ : minimum distance of the anticode, $\delta$ : maximum distance of the anticode, and $\varepsilon$ : probability of exploitable faults.

Output: An $(n, M, d, \delta)$--binary anticode $C$.

1. do
2.     boolean codeExists := false;
3.     for Every set $S$ of $M$ words which does not include $\bot$ do
4.         if $S$ is an $(n, M, d, \delta)$--binary anticode then
5.             if $1 - p_{m,S} < \varepsilon \forall 1 \leq m \leq n$ then
6.                 codeExists := true;
7.                 $C := S$;
8.                 break for;
9.         $\varepsilon := \varepsilon - \text{const}$;
10.     while codeExists;
11. return $C$. 

Algorithm 2: Fault analysis algorithm.

**Input**: $P$: plaintext, $\kappa$: secret key, $E(P, \kappa)$: ciphertext corresponding to encrypting $P$ with $\kappa$, $C$: $(n, M, d, \delta)$–binary anticode, $\mathcal{F}$: sequence of assembly instructions

**Output**: Distribution of safe and exploitable faults:
- Int[] SafeBitFlip: SafeBitFlip[$m$] = $\{\varsigma \in \mathcal{G}_{(\mathcal{F}, \text{fm}, m)} \text{is safe}\}$
- Int[] ExploitableBitFlip: SafeBitFlip[$m$] = $\{\varsigma \in \mathcal{G}_{(\mathcal{F}, \text{fm}, m)} \text{is exploitable}\}$
- Int SafeSkip: SafeBitFlip = $\{\varsigma \in \mathcal{G}_{(\mathcal{F}, \text{sk})} \text{is safe}\}$
- Int ExploitableSkip: SafeBitFlip = $\{\varsigma \in \mathcal{G}_{(\mathcal{F}, \text{sk})} \text{is exploitable}\}$

1. for Fault mask Int $x$: 1 to $2^n$
2. for Int $j$: 0 to $|\mathcal{F}|$
3. for Instruction $f$ in $\mathcal{F}$ do
4.     Execute instruction $f$;
5.     if $f = j$ and $f$ has a destination register then
6.         $r_f = r_f \oplus x$
7.     if output == $\perp$ or output == $E(P, \kappa)$ then
8.         SafeBitFlip[HammingWeight($x$)]++;  
9.     else
10.    ExploitableBitFlip[HammingWeight($x$)]++;  
11. for Int $j$: 0 to $|\mathcal{F}|$
12. for Instruction $f$ in $\mathcal{F}$ do
13.     if $f = j$ then
14.         continue;
15.     else
16.         Execute instruction $f$;
17.     if output == $\perp$ or output == $E(P, \kappa)$ then
18.         SafeSkip++;  
19.     else
20.         ExploitableSkip++;  
Case Study on PRESENT-80

- $sBoxLayer$ and $pLayer$ can be implemented as 4 identical blocks of 16 bits.
- For look-up tables, these two layers can be combined for more efficient implementation.
Overview of Combined Layer

Encoder_c(a_0, a_1, a_2, a_3)

Encoder_c(as_0, 0, 0, 0)
Encoder_c(as_1, 0, 0, 0)
Encoder_c(as_2, 0, 0, 0)
Encoder_c(as_3, 0, 0, 0)

Encoder_c(b_0, b_1, b_2, b_3)

Encoder_c(bs_0, 0, 0, 0)
Encoder_c(bs_1, 0, 0, 0)
Encoder_c(bs_2, 0, 0)
Encoder_c(bs_3, 0, 0)

Encoder_c(c_0, c_1, c_2, c_3)

Encoder_c(cs_0, 0, 0)
Encoder_c(cs_1, 0, 0, 0)
Encoder_c(cs_2, 0, 0, 0)
Encoder_c(cs_3, 0, 0, 0)

Encoder_c(d_0, d_1, d_2, d_3)

Encoder_c(ds_0, 0, 0, 0)
Encoder_c(ds_1, 0, 0, 0)
Encoder_c(ds_2, 0, 0)
Encoder_c(ds_3, 0, 0)

Encoder_c(as_0, 0)
Encoder_c(bs_0, 0, 0)
Encoder_c(cs_0, 0)
Encoder_c(ds_0, 0)

Encoder_c(as_0, ds_0, cs_0, ds_0)
Evaluation Results: $n = 8$

Figure 1: Simulated results for anticodes with $n = 8$, $d = 2, 3$. 
Evaluation Results: \( n = 10 \)

Figure 2: Simulated results for anticodes with \( n = 10, d = 2 \).
• Timing overheads are reasonable – as low as 26.1% compared to fastest non-bitsliced PRESENT implementation on AVR (9424 cycles vs. 8721).
• Memory overheads are high – optimized implementation needs one xor table and 8 shifting table, resulting to 81,920 bytes of memory.

<table>
<thead>
<tr>
<th>Operation Type</th>
<th>Code Length</th>
<th>Required Memory (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary (Sbox, shifts)</td>
<td>≤ 8</td>
<td>2,048</td>
</tr>
<tr>
<td></td>
<td>≤ 16</td>
<td>524,288</td>
</tr>
<tr>
<td>Binary (XOR, AND, modular addition)</td>
<td>≤ 8</td>
<td>65,536</td>
</tr>
<tr>
<td></td>
<td>≤ 16</td>
<td>33,554,432</td>
</tr>
</tbody>
</table>
Conclusion
Conclusion

- Two automated methods were presented – one for analyzing unprotected implementation to find an attack, the other for analyzing implementation protected by encoding to evaluate robustness.
- Automated methods are standard in software engineering, especially in vulnerability analysis.
- There are only a few works to automate either SCA or FIA.
- As shown in the previous slides, it is a promising area with practical impact.
Thank you!
Any questions?

Web: http://jbreier.com
E-mail: jbreier@ntu.edu.sg