NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

pace Physical Analysis and Cryptographic Engineering

Fault Analysis Automation on Software Targets

ASCEHS Workshop, IIT Kharagpur

Jakub Breier

3 July 2018

Physical Analysis and Cryptographic Engineering
Nanyang Technological University, Singapore

Outline

1. DFA Automation on Assembly Implementations
2. Automated Evaluation of Software Encoding Countermeasures
3. Conclusion

Why Automation?

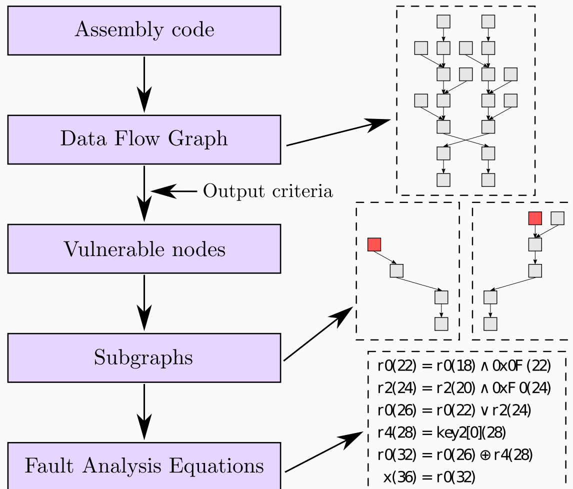
- All the current symmetric block ciphers have been shown vulnerable against fault attacks (especially DFA).
- The question is not whether the algorithm is secure or not, but which part of it is insecure.
- Automated methods can provide an answer fast and with minimal need of human intervention.

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.

DATAC – DFA Automation Tool for Assembly Code[1]



[1] J. Breier, X. Hou, Y. Liu: Fault Attacks Made Easy: Differential Fault Analysis Automation on Assembly Code, TCHES 2018.

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.

Gdistance

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 $\mathbb{F}_{ex}$ for a Sample Cipher

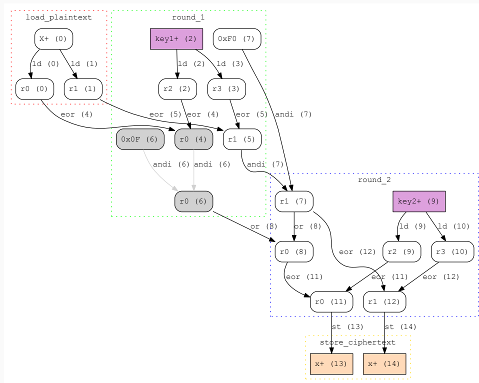
#	Instruction	#	Instruction	#	Instruction
	//round_1	5	EOR r1 r3	10	LD r3 key2+
0	LD r0 X+	6	ANDI r0 0x0F	11	EOR r0 r2
1	LD r1 X+	7	ANDI r1 0xF0	12	EOR r1 r3
2	LD r2 key1+	8	OR r0 r1		//store_ciphertext
3	LD r3 key1+		//round_2	13	ST x+ r0
4	EOR r0 r2	9	LD r2 key2+	14	ST x+ r1

Example

- $N_{\mathbb{F}_{ex}} = 15$.
- $f_6 = \texttt{ANDI r0 0x0F}$.
- f_6 is associated with the 4−tuple $(6, \texttt{ANDI}, \{\texttt{r0}, \texttt{0x0F}\}, \{\texttt{r0}\})$.

Data Flow Graph for $\mathbb{F}_{ex}$

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



Example

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

a	b	$c = a \,\&\, b$
0	0	0
0	1	0
1	0	0
1	1	1

- $f_1 =$`AND r1 r0`.
- ϑ_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.

Related to Key

- 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 $\text{Gdistance}(b, ch) = 0$. 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`.

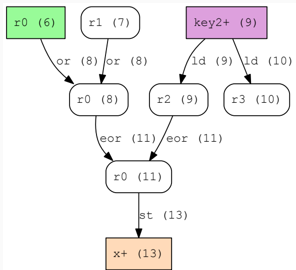
Output Criteria – Selection of Vulnerable Nodes

- `minAffectedCT`: minimal number of ciphertext nodes affected by the vulnerable node;
- `minDist`: minimal Gdistance between the node and a ciphertext node for at least `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 `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`.

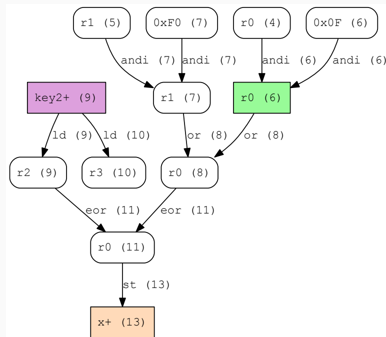
Subgraph Construction

- Given $G_{\mathbb{F},full} = (V, E)$ node $a \in V$ subgraph $G_a = (V_a, E_a)$ is a pair such that $V_a \subseteq V$ and $E_a \subseteq E$.
- *KNGchild*: the set of key word nodes that are related to a.
- `depth`: user input.
- $V_a = \left(\bigcup_{i=0}^{\texttt{depth}} U_i\right) \bigcup \left(\bigcup_{j=1}^{4} V_j\right)$
- $U_0 = \{b : a, b \in f^{io}\}$
- $U_i = \{b : b \in f^{io} \text{ s.t. } v \in f^{do} \text{ for some } v \in U_{i-1}\}$
- $V_1 = \{b : b \text{ is a child of } a\}$
- $V_2 = \{k : k \text{ is a round key that is related to } a\}$
- $V_3 = \{b : b \text{ is a key word node for a key } k \in V_2\}$
- $V_4 = \{b : b$ is a child of a node $v \in$ `KNGchild` and b is a parent of a child of $a\}$.

Subgraph Example



(a)



(b)

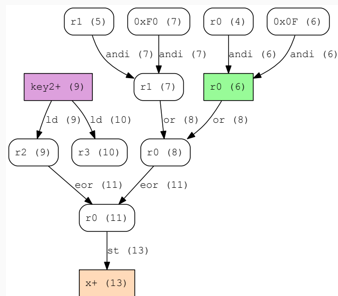
Subgraphs for node "r0 (6)" with depth (a) 0 and (b) 1, output criteria (minAffectedCT, minDist, maxDist, maxKey, minKeyWords)$= (1, 1, 1, 1, 1)$

Construction of equations from assembly instructions

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)

Example

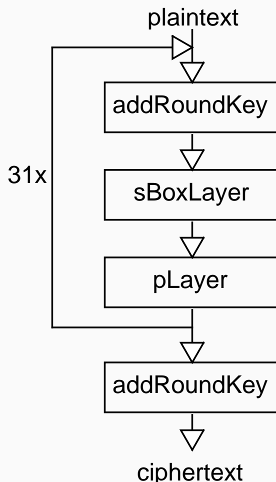
$$\text{“r0 (6)”} = \text{“r0 (4)”} \wedge \text{“0x0F (6)”} \quad (1)$$
$$\text{“r1 (7)”} = \text{“r1 (5)”} \wedge \text{“0xF0 (7)”} \quad (2)$$
$$\text{“r0 (8)”} = \text{“r0 (6)”} \vee \text{“r1 (7)”} \quad (3)$$
$$\text{“r2 (9)”} = \texttt{key2[0]} \quad (4)$$
$$\text{“r0 (11)”} = \text{“r0 (8)”}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



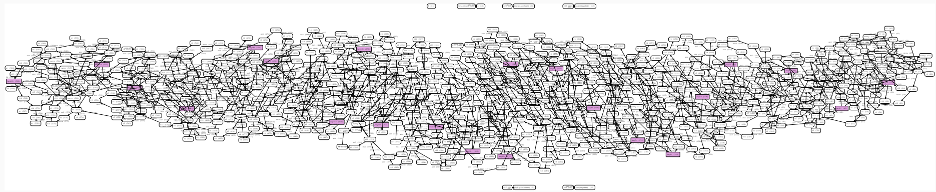
- (1): “r0 (6)” $= 0000 b_4 b_5 b_6 b_7,\ b_j \in \{0,1\}\ (j = 4,5,6,7)$.
- (3): if we skip instruction 8, the result of (1) will be used in instruction 11 (5) instead of the result of (3)
- (4) and (6): the instruction skip attack on instruction 8 would result in the first four bits of `key2[0]` to 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

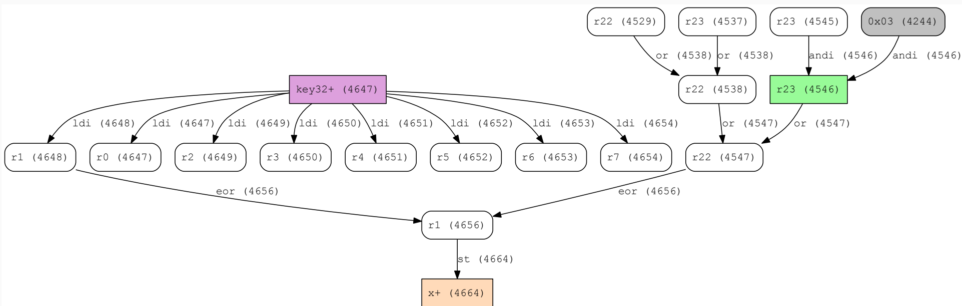


New Attack found on PRESENT-80

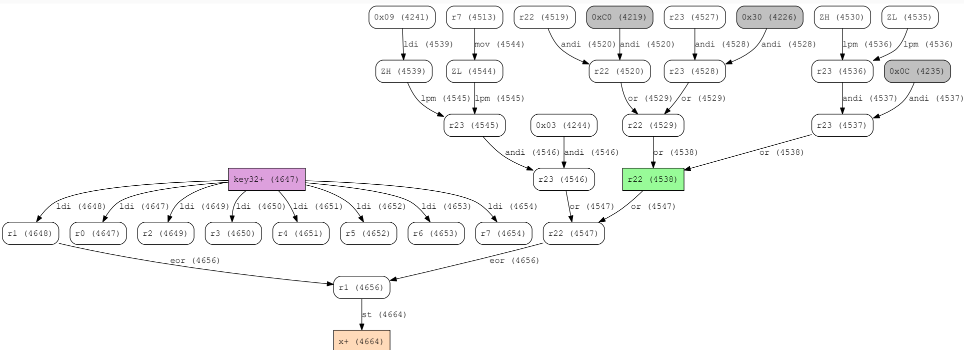
- We chose a speed-optimized assembly implementation for 8-bit AVR publicly available on GitHub[2].
- output criteria: (`minAffectedCT`, `minDist`, `maxDist`, `maxKey`, `minKeyWords`)$= (1, 1, 1, 1, 1)$.
- Recover the last round key by 16 fault injections.
- Implementation specific.
- Existing DFA on 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

# 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

Data collected on laptop computer with mobile Intel Haswell family CORE i7 processor, 8 GB RAM

Conclusion

- 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.

Binary Codes

A *binary code*, denoted by $\mathcal{C}$, is a subset of the $n-$dimensional vector space over $\mathbb{F}_2$-$\mathbb{F}_2^n$, where n is called the *length* of the code $\mathcal{C}$.

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

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

$$\mathrm{dis}\,(\mathcal{C}) = \min\{\mathrm{dis}\left(\boldsymbol{c}, \boldsymbol{c}'\right) : \boldsymbol{c}, \boldsymbol{c}' \in \mathcal{C}, \boldsymbol{c} \neq \boldsymbol{c}'\}.$$

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 δ. This value δ is the *maximum distance* of the anticode[4].

Definition

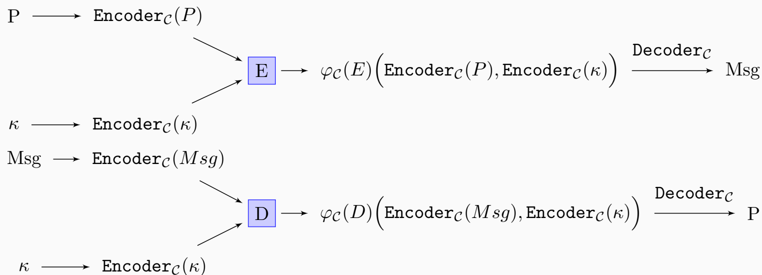
Let $\mathcal{C}$ be an $(n, M, d)-$binary code, if furthermore

$$\max_{\boldsymbol{c},\boldsymbol{c}'\in\mathcal{C}} \operatorname{dis}\left(\boldsymbol{c}, \boldsymbol{c}'\right) = \delta,$$

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

[4] P. G. Farrell. Linear binary anticodes, Electronic Letters, vol. 6 no. 13, 1970.

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 expoited 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 ϱ_1 on $\mathcal{F}_1$ is *safe* if $\varrho\Big(\mathcal{F}_1\Big)(\kappa, P) = \perp$ or $E(\kappa, P)$ and it is *exploitable* otherwise. Similarly, a fault ϱ_2 on $\mathcal{F}_2$ is *safe* if $\varrho\Big(\mathcal{F}_2\Big)(\kappa, P) = \perp$ or $D(\kappa, P)$ and it is *exploitable* otherwise.

Anticode Generation Algorithm

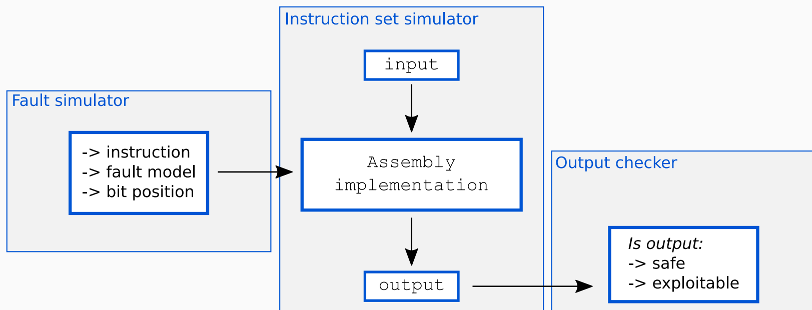
Algorithm 1: Anticode Generation Algorithm.

Input : n : length of the anticode, M : number of codewords, d : minimum distance of the anticode, δ : maximum distance of the anticode, and ε : probability of *exploitable* faults.

Output: An (n, M, d, δ)−binary anticode C.

```
1  do
2      boolean codeExists := false;
3      for Every set S of M words which does not include ⊥ do
4          if S is an (n, M, d, δ)−binary anticode then
5              if 1 − p_{m,S} < ε ∀1 ≤ m ≤ n then
6                  codeExists := true;
7                  C := S;
8                  break for;
9      ε := ε − const;
10 while codeExists;
11 return C.
```

Automated Analyzer



Software Implementation Evaluation Algorithm

Algorithm 2: Fault analysis algorithm.

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

Output: Distribution of *safe* and *exploitable* faults:

Int[] SafeBitFlip: SafeBitFlip[m]= $|\{\varsigma \in \mathcal{G}_{(\mathcal{F},\text{fm},m)}$is safe$\}|$

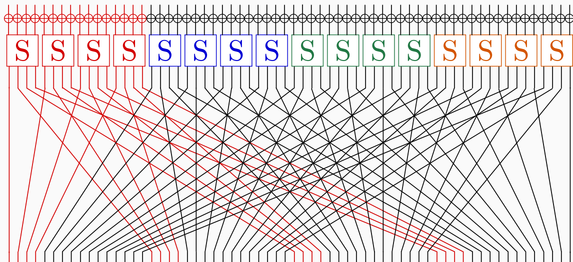
Int[] ExploitableBitFlip: SafeBitFlip[m]= $|\{\varsigma \in \mathcal{G}_{(\mathcal{F},\text{fm},m)}$is exploitable$\}|$

Int SafeSkip: SafeBitFlip= $|\{\varsigma \in \mathcal{G}_{(\mathcal{F},\text{sk})}$is safe$\}|$

Int ExploitableSkip:SafeBitFlip= $|\{\varsigma \in \mathcal{G}_{(\mathcal{F},\text{sk})}$is exploitable$\}|$

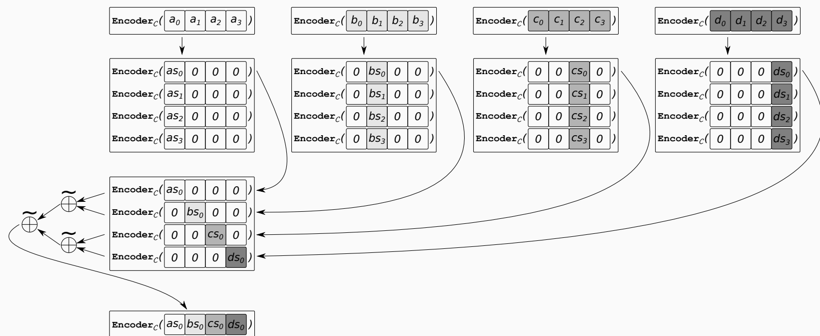
1 **for** *Fault mask Int x: 1 to* 2^n **do**
2   **for** *Int j: 0 to* $|\mathcal{F}|$ **do**
3   **for** *Instruction f in* $\mathcal{F}$ **do**
4   Execute instruction f;
5   **if** $f == j$ ***and*** *f has a destination register* **then**
6                 $r_f = r_f \oplus$ x;
7   **if** *output* $== \perp$ ***or*** *output* $== E(P, \kappa)$ **then**
8   SafeBitFlip[HammingWeight(x)]++;
9   **else**
10   ExploitableBitFlip[HammingWeight(x)]++;
11 **for** *Int j: 0 to* $|\mathcal{F}|$ **do**
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++;
21 **return** ExploitableBitFlip, SafeBitFlip, ExploitableSkip, SafeSkip.

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



Evaluation Results: $n = 8$

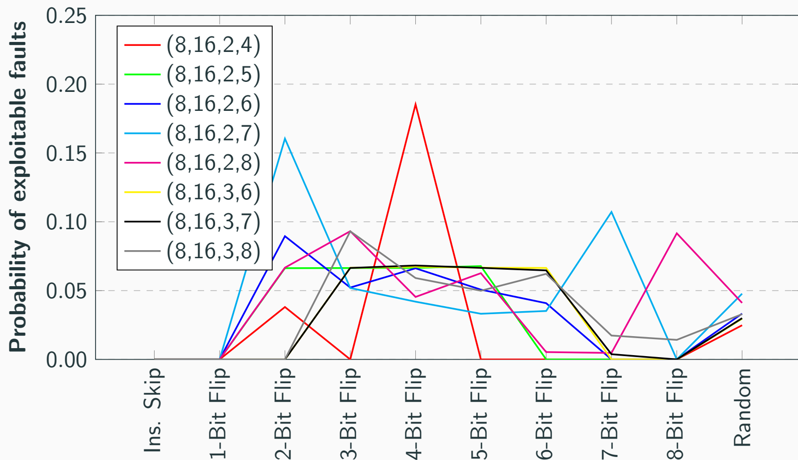


Figure 1: Simulated results for anticodes with n = 8, d = 2, 3.

Evaluation Results: $n = 10$

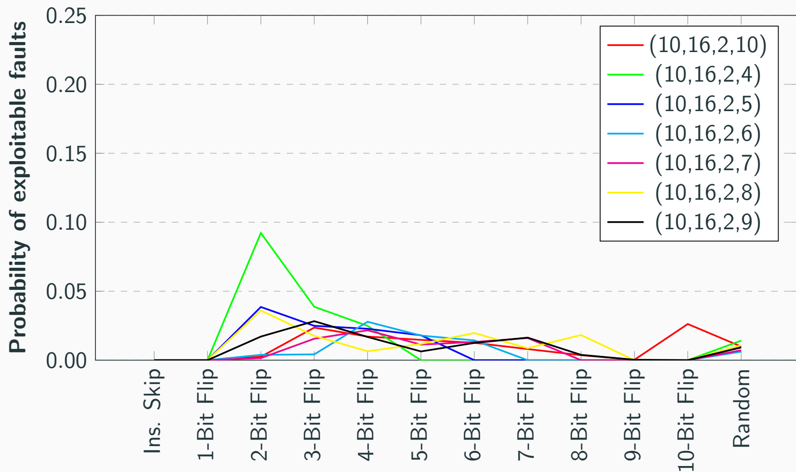


Figure 2: Simulated results for anticodes with n = 10, d = 2.

Encoded Implementation Overheads

- Timing overheads are reasonable – as low as 26.1% compared to fastest non-bitsliced PRESENT implementation on AVR (9 424 cycles vs. 8 721).
- Memory overheads are high – optimized implementation needs one `xor` table and 8 shifting table, resulting to 81,920 bytes of memory.

Operation Type	Code Length	Required Memory (B)
Unary (Sbox, shifts)	≤ 8	2,048
	≤ 16	524,288
Binary (XOR, AND, modular addition)	≤ 8	65,536
	≤ 16	33,554,432

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