

Fault Model-Driven Formal Verification of Cryptographic Hardware Against Fault Attacks

Presenter: Daniel THIRION

Authors:

Daniel THIRION^{1,2}, George-Cristian SERCAIANU¹, Valentin EGLOFF²,
Jean-Marc DAVEAU¹, David HELY², Vincent BEROULLE², Philippe ROCHE¹

¹ STMicroelectronics, 850 Rue Jean Monnet 38926 Crolles Cedex, France { *firstname.lastname@st.com* }

² Univ. Grenoble Alpes, LCIS, 50 Rue Barthélemy de Laffemas, 26000 Valence, France {
firstname.lastname@lcis.grenoble-inp.fr }

Embedded Cryptography and Security

→ Growing connectivity of critical embedded systems

- Connectivity leads to **Cybersecurity** issues and **Security** requirements.
- Hardware cryptography is efficient but **hard to protect and verify**
- **How to verify against stringent standards?** Common Criteria ...
- **How to verify early in the design process?**



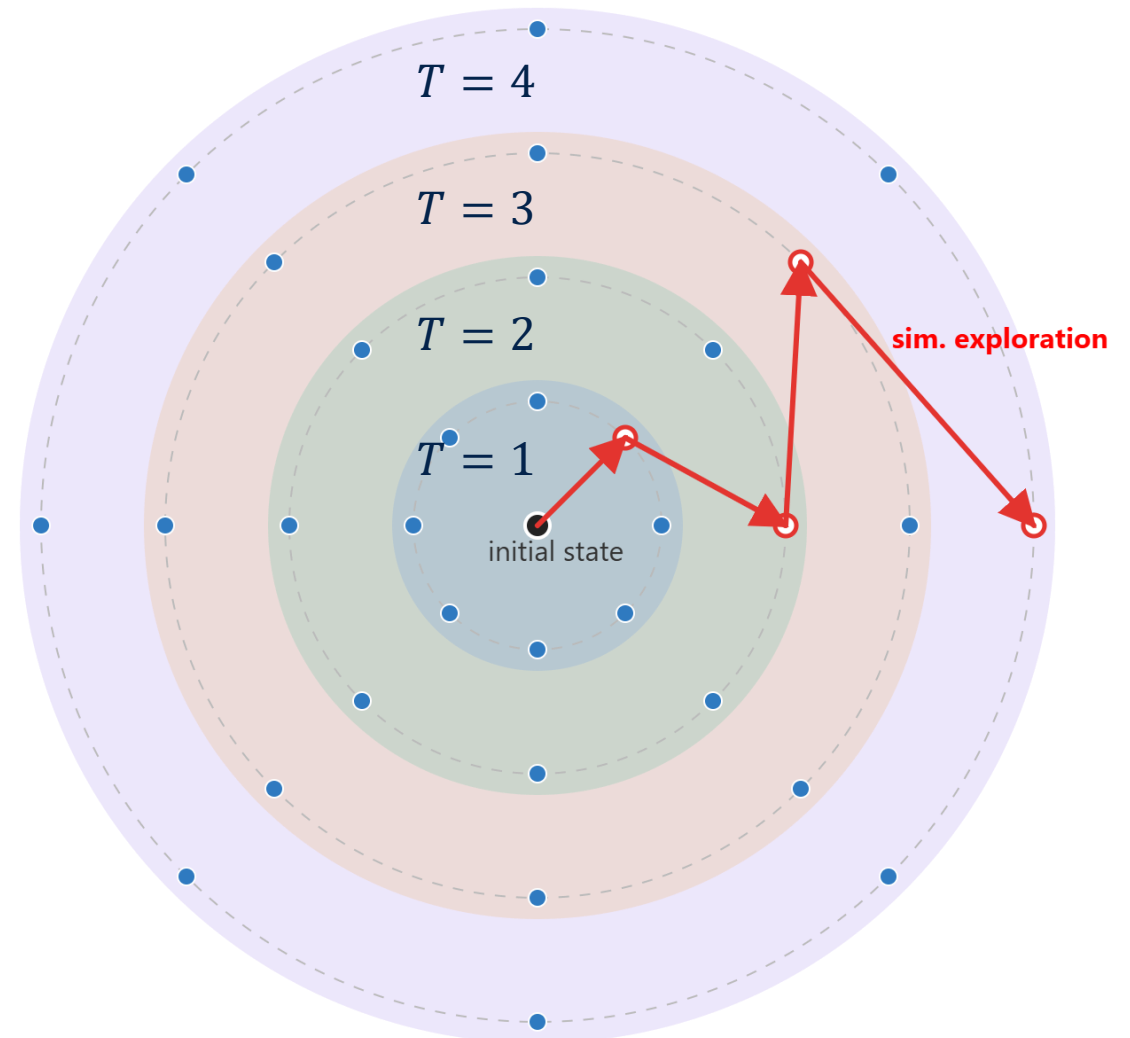
Formal Methods for Security Analysis

Early RTL/netlist analysis = **catch issues pre-silicon**

Fault Simulation: impossible to be exhaustive at higher multiplicities (large state space)

Answer: **Formal Proof (Model Checking)**

- **Exhaustive exploration** vs simulation
- **Strong guarantees** through mathematical proofs for given constraints



Presentation Outline

1 Introduction

2 Background

Fault attacks, verification frameworks and formal methods

3 Proposed Methodology

Core idea and methodology steps

4 Setup and Results

Tooling, harness, designs under test and results

5 Conclusion and Future Works

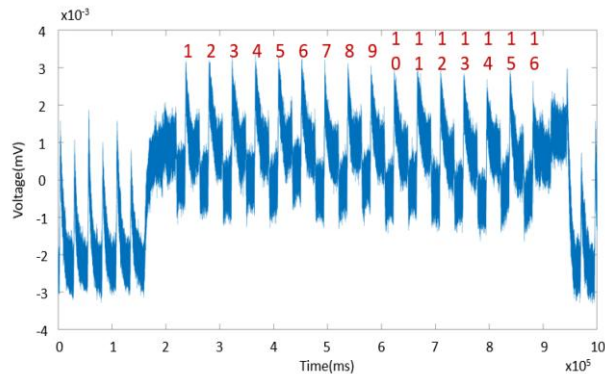
Background

- 1) Attacks on block ciphers
- 2) Existing formal verification frameworks
- 3) Previous Works
- 4) Formal methods for Fault Injection

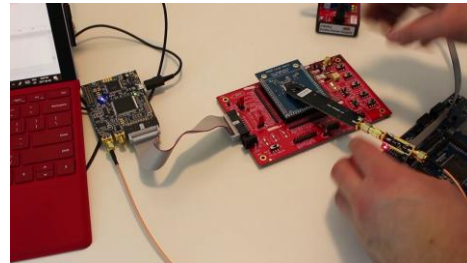
Attacks on block ciphers

Passive

Side-channel attacks



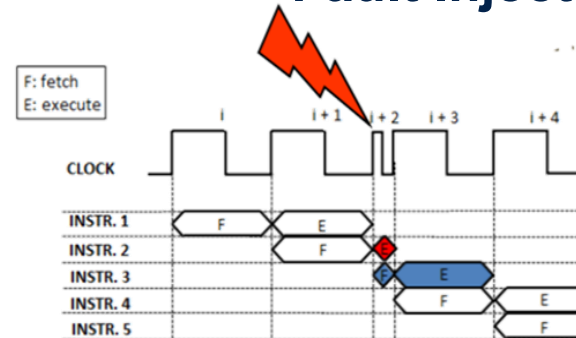
Power analysis



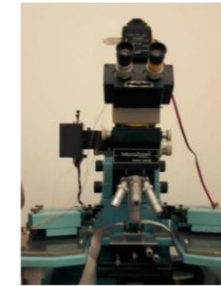
EM analysis

Active

Fault Injection attacks



Clock Glitch



Laser or EM injection

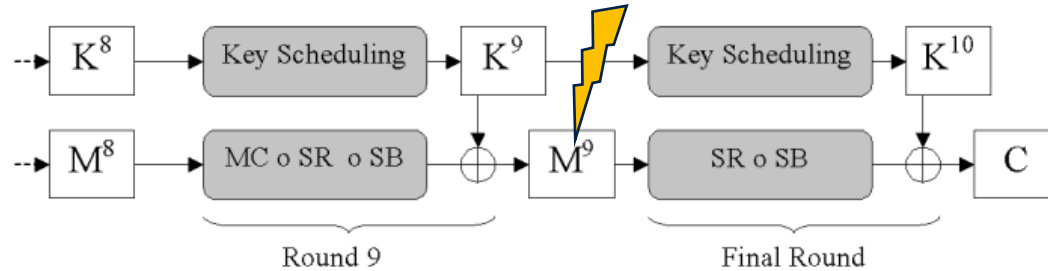
➔ Multiple faults at once

→ Can lead to cryptanalysis (leaked key)

→ Focus on FI attacks

Differential Fault Analysis

Example: Round 10 attack, Giraud C.



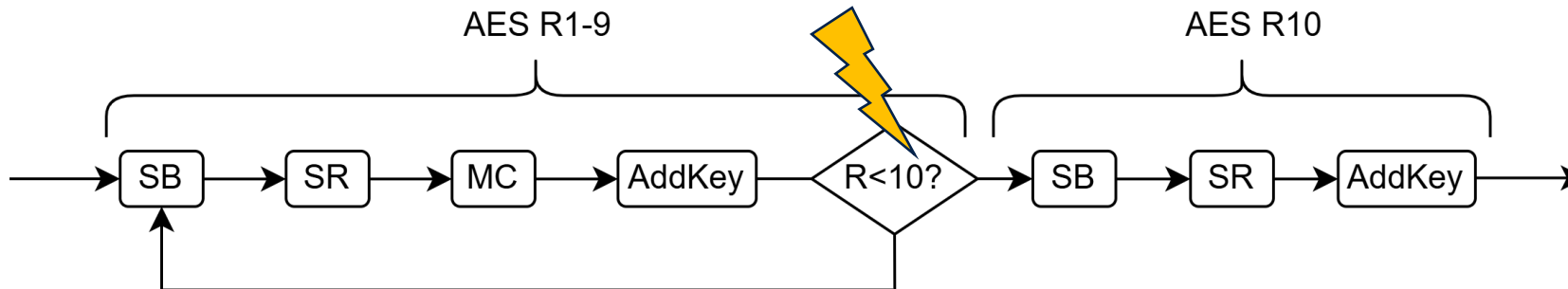
Fault Attacks on block ciphers

100+
ciphertexts

- **Key can be guessed,**
with very little ciphertext
counts
- Or state-space heavily
reduced for brute-forcing

Round-Skipping Attack

Example: Skip Rounds 1-9, Choukri H. and Tunstall M.



Other attacks: SIFA, SEA, ...

Existing fault verification frameworks

- **Equivalence Checking (FIRMER/CLEAVE)**

[Tan 2023, 2025]

- **Method:** Compare faulty design vs. "golden" model.
- **Flaw:** Too verbose. Flags harmless deviations. Huge post-processing overhead.

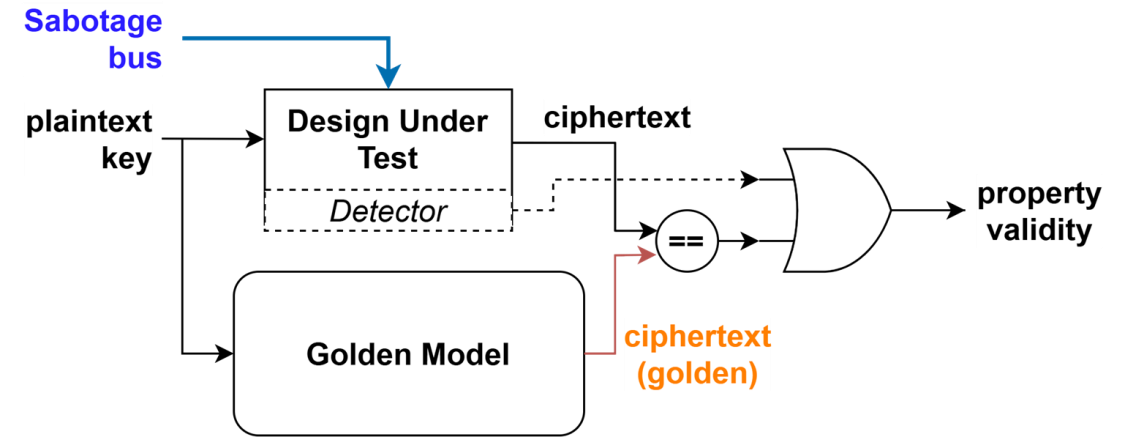
- **Property-Based Verification (SoFI)**
[Wang 2022]

- **Method:** Assert high-level security properties.
- **Flaw:** Not portable. Complex, design-specific setup. Hard to maintain.

Our Previous Works

Equivalence Checking

- Compare output against “perfect” result
- Easy setup (no knowledge of internals needed)
- **Results are not all exploitable security flaws**



→ Demonstrated on AES in “*Comparative Study of Safety and Security-Protected AES Designs*,” DDECS 2025

Problem Statement and Proposed Approach

- Need a method which:
 - Is **fast** (quicker proof) and **precise** (only highlight actual attacks)
 - Directly gives result for a specific attack
 - Doesn't require writing complex, per-design security properties
 - Keeps the black-box, 'no knowledge' approach
- Solution proposed: **Fault model-driven approach**
 - Search for injections leading to specific exploitable faulted ciphertexts rather than any faulty ciphertexts
 - “Antivirus” → create models only for known attacks

Proposed Methodology

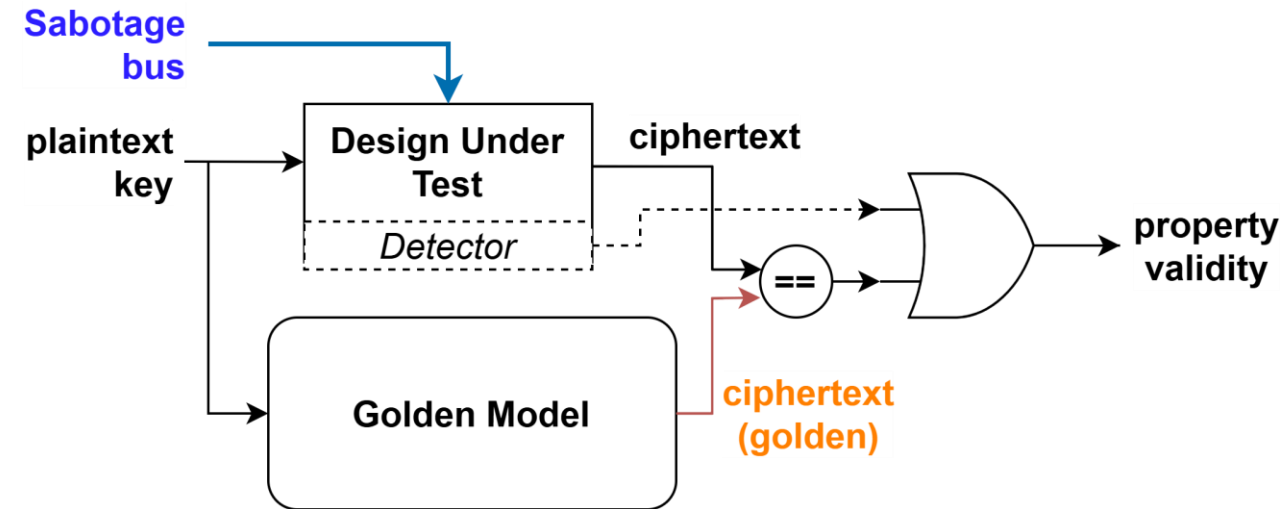
- 1) Attacker Model
- 2) Core Idea
- 3) Methodology Overview
- 4) Methodology Steps

Attacker model

- Simplified model: **Single/Multiple Event Upset** (“Bit Flips”)
 - Models laser attacks (very “strong” attacker)
 - Multiplicity (M) up to 3 (= 3 independent faults per execution)
 - Can be all at once (same cycle), or spread out (different cycles)

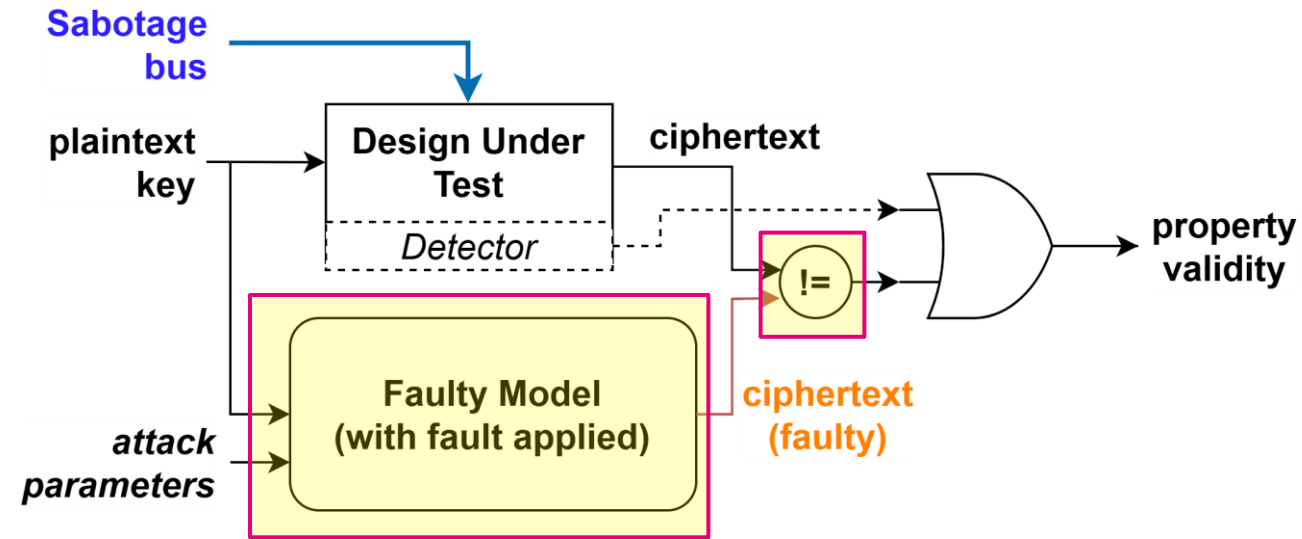
Core Idea

- Equivalence checking searches for any deviation from the Golden Model without detection



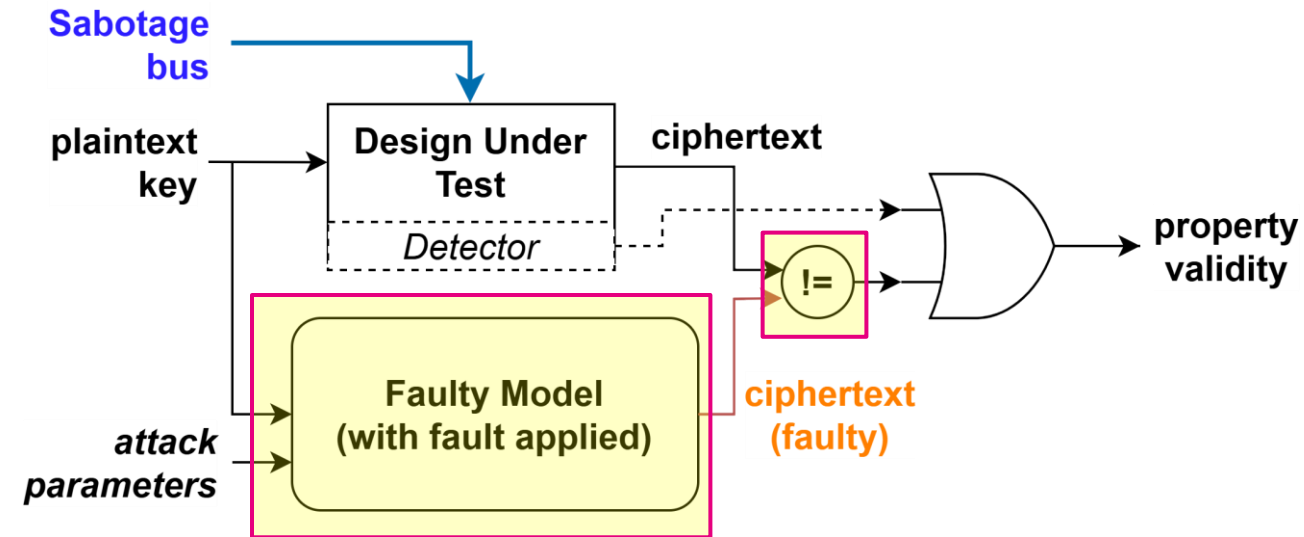
Core Idea

- Equivalence checking searches for any deviation from the Golden Model without detection
- Replace golden model with faulty model
→ Generates ciphertext(s) corresponding to known exploitable faults
- Search for sabotages in DUT, leading to outputs equaling faulty model, without detection



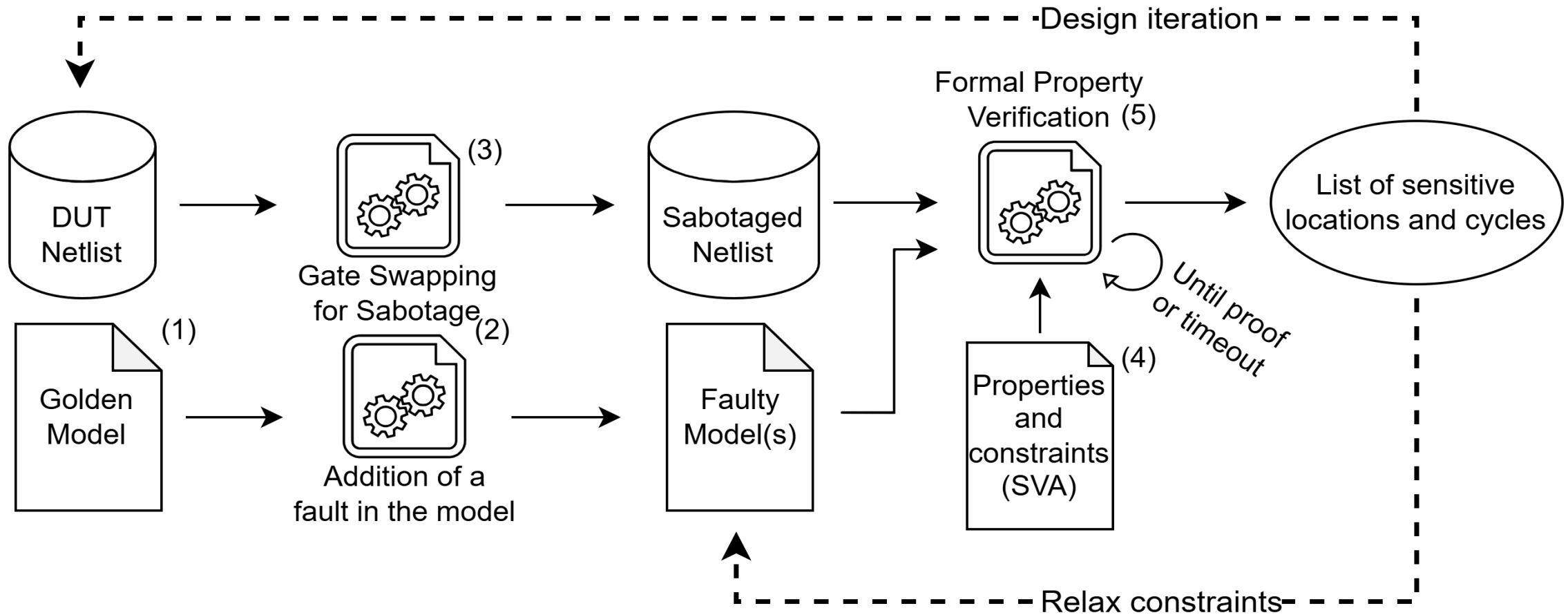
Core Idea

- Equivalence checking searches for any deviation from the Golden Model without detection
- Replace golden model with faulty model
 - Generates ciphertext(s) corresponding to known exploitable faults
- Search for sabotages in DUT, leading to outputs equaling faulty model, without detection

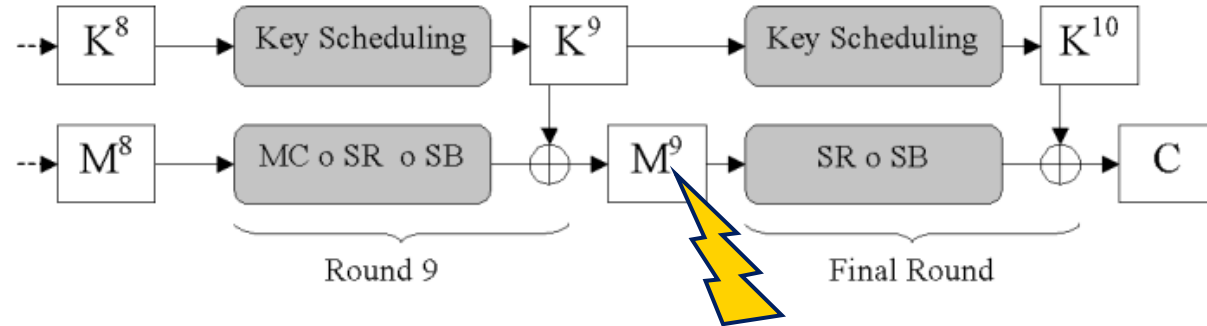


→ No knowledge of DUT required
→ Faulty model can be reused with other designs of same security primitive (AES...)

Methodology Overview



(1,2) Creation of the Faulty Models



Giraud C., “DFA on AES”
 (“Round 10” attack)

Golden model

```
// Initial round key addition
state = plaintext ^ key;
round_key = key;

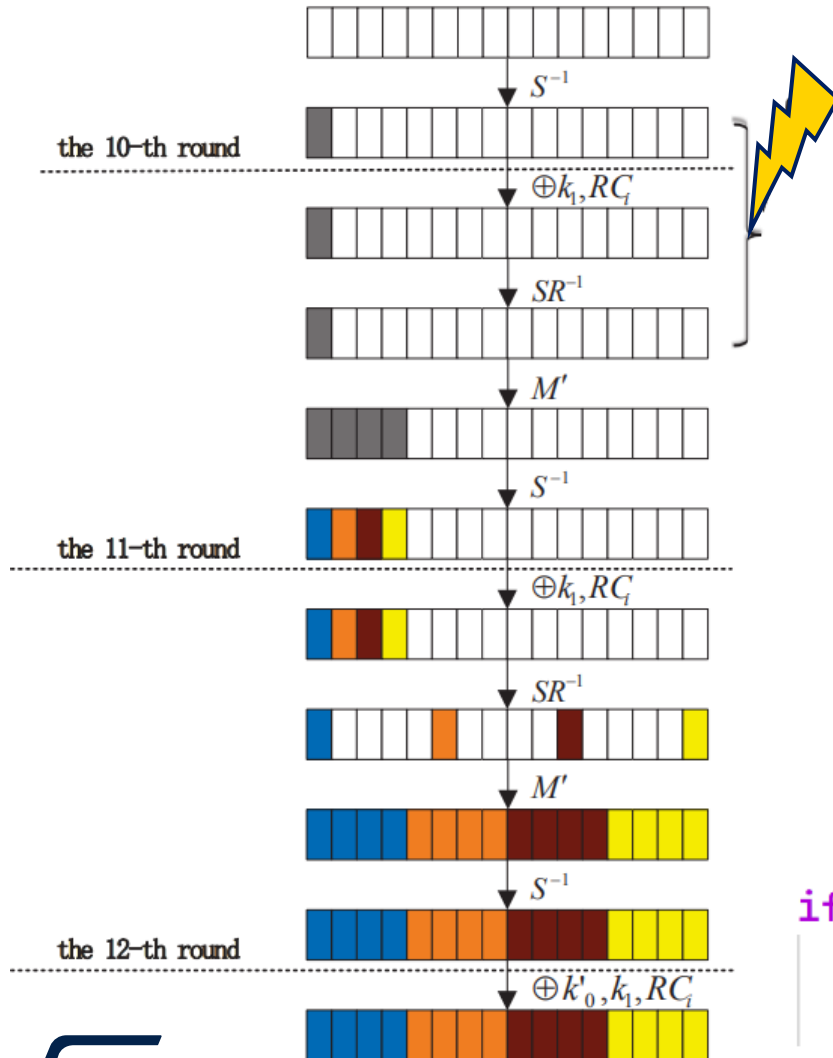
// Rounds 1-9
for (int round = 1; round < 10; round++) begin
    state = SubByte(state);
    state = ShiftRows(state);
    state = MixColumns(state);
    round_key = ComputeRoundKey(round_key, round);
    state = state ^ round_key;
end

// Round 10
state = ShiftRows(SubByte(state));
round_key = ComputeRoundKey(round_key, 10);
state = state ^ round_key;

return state;
```

Faulty model

```
// Round 10 with one-bit XOR mask injection
state = state ^ (128'b1 << index);
state = SubByte(state);
state = ShiftRows(state);
state = state ^ ComputeRoundKey(round_key, 10);
return state;
```



(1,2) Creation of the Faulty Models

Golden model

```
// --- 3. PRINCE Core ---
```

```
// Forward Rounds (5 rounds, using RC0 through RC5)
state = m(s_layer(state ^ k1 ^ rc(0))) ^ k1 ^ rc(1);
state = m(s_layer(state)) ^ k1 ^ rc(2);
state = m(s_layer(state)) ^ k1 ^ rc(3);
state = m(s_layer(state)) ^ k1 ^ rc(4);
state = m(s_layer(state)) ^ k1 ^ rc(5);
```

```
// Middle Round (S -> M' -> S_inv)
state = si_layer(mp(s_layer(state)));
```

```
// Inverse Rounds (5 rounds, using RC6 through RC11)
state = si_layer(mi(state ^ k1 ^ rc(6))) ^ k1 ^ rc(7);
state = si_layer(mi(state)) ^ k1 ^ rc(8);
state = si_layer(mi(state)) ^ k1 ^ rc(9);
state = si_layer(mi(state)) ^ k1 ^ rc(10);
state = si_layer(mi(state)) ^ k1 ^ rc(11);
```

```
// --- 4. Final Whitening ---
return state ^ k0 prime;
```

Faulty model

```
if (fault_location == 1'b0) begin
    state = si_layer(mi(state)) ^ k1 ^ rc(10) ^ fault_mask;
    state = si_layer(mi(state)) ^ k1 ^ rc(11);
```

(1,2) Creation of the Faulty Models

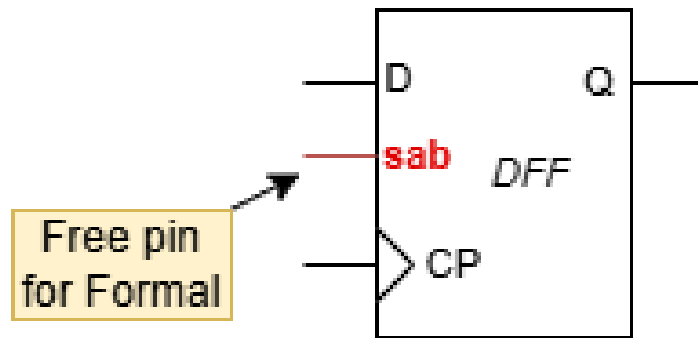
- Models are “parametrized” and can be constrained in formal (speed tradeoff).

```
function automatic logic [127:0] model_R10_start_fault (  
    input logic [127:0] plaintext,  
    input logic [127:0] key,  
    input logic [6:0] index  
)
```

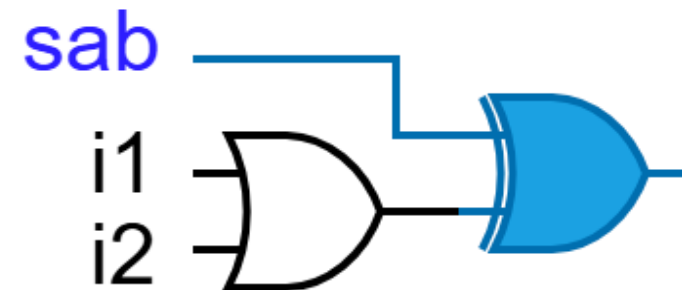
- Example:* “Round 10” DFA attack can be done on any of the bits of the vector (*index* parameter, $2^7 = 128$ possibilities).

(3) DUT Instrumentation

- Addition of **Saboteur inputs**: **flips FF output** on assertion
- Saboteur bus as **free inputs** for formal tool: constrained to multiplicity M



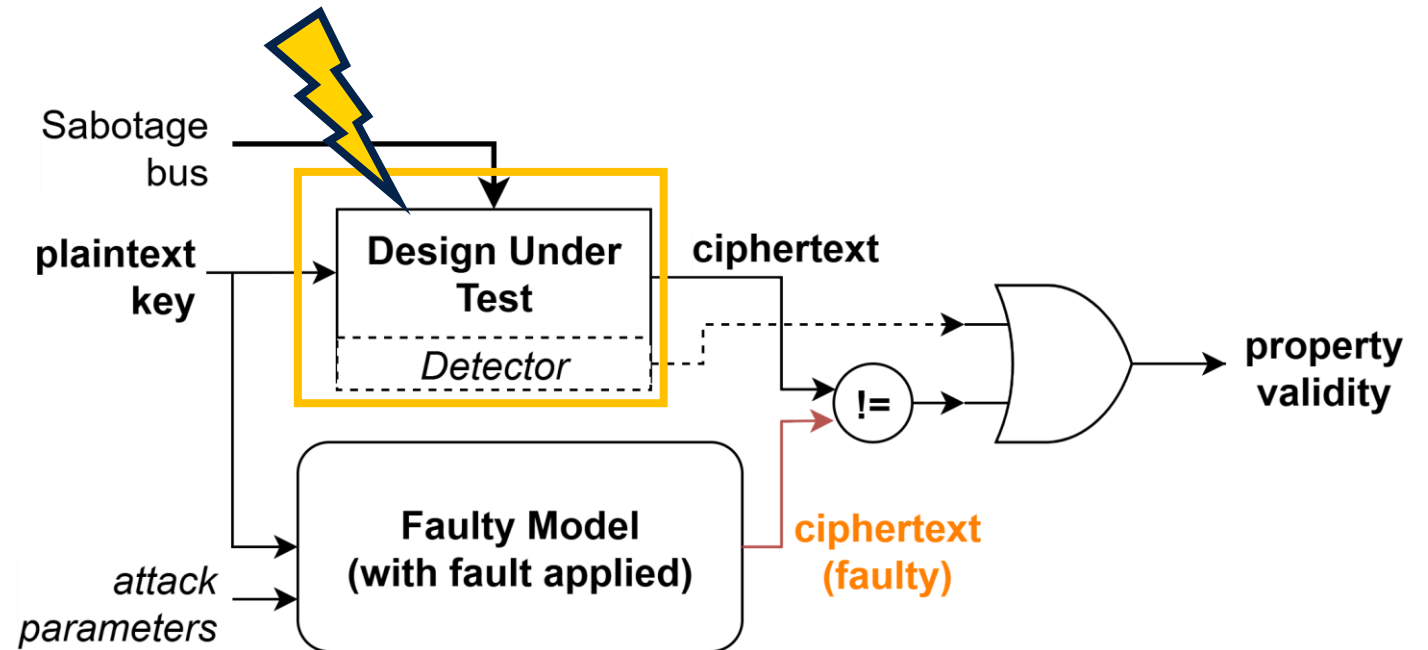
- Saboteur inputs on combinational gates: flips output for 1 cycle



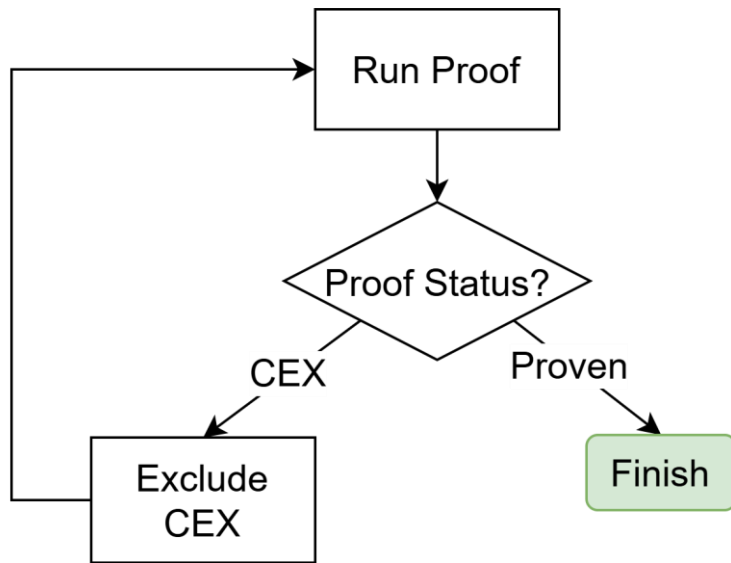
(4) Properties and constraints

- Sabotage bus constrained with SVA properties (multiplicity, cycles to inject...)
- Attack parameters constraints can be tuned for faster (but less precise) proof

```
(model_R10_start_fault(plaintext, key, fault_index)
| != dut_ciphertext) || error_detected
```



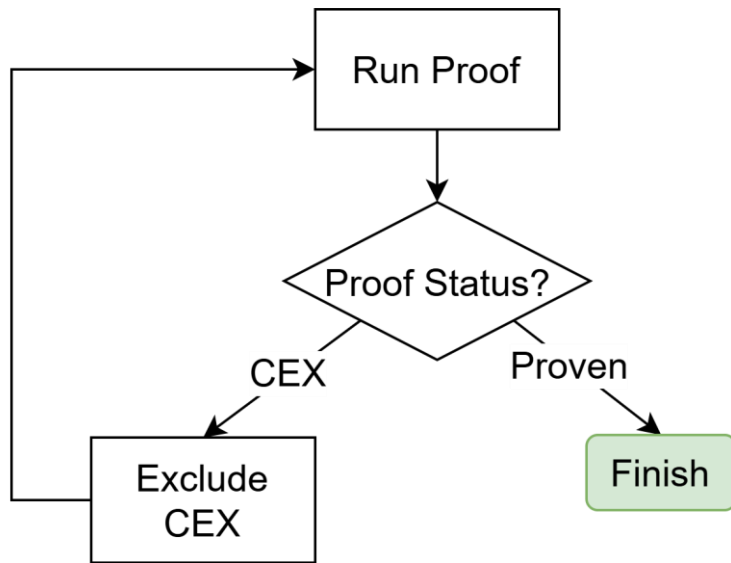
(5) Formal Property Verification



FPV search loop

- Loop to find **all possible violations**, or **until timeout**
- For each counter example (CEX), **exclude it** and restart.
- For $M + 1$, exclude all CEX from previous M (avoids repeating CEX + silent bit)

(6) Results Exploitation



FPV search loop

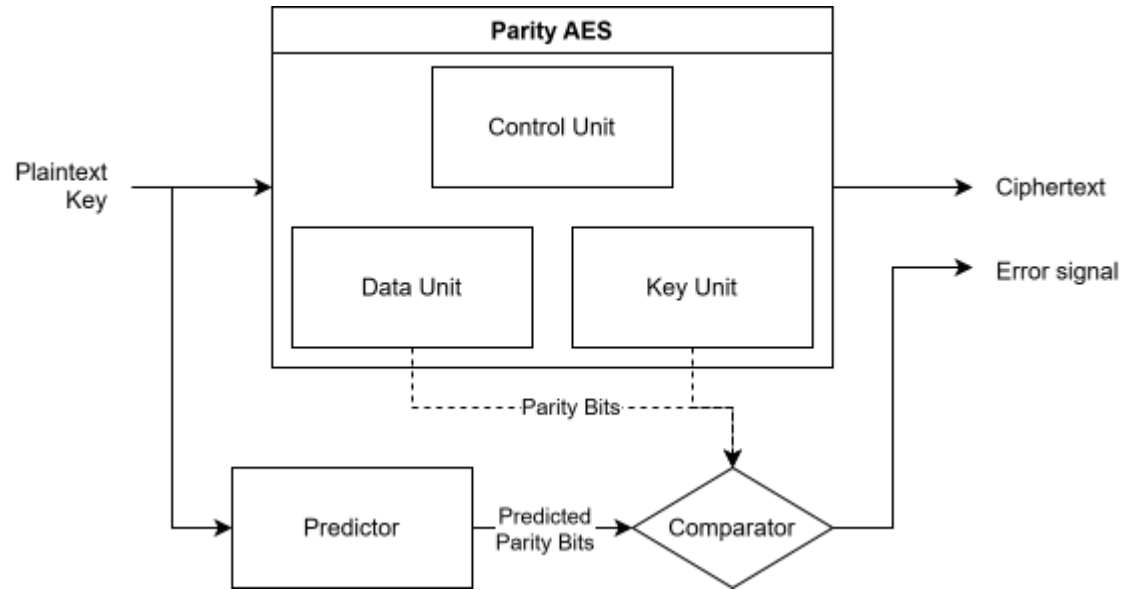
- **If run until exhaustion** (proven), results are all possible injection locations to enable modeled attack.
- **If not exhaustive**, CEX found can still be used to fix flaws, iterate design, find formal simplifications or tighten model constraints.

Experimental Setup & Results

- 1) Tooling
- 2) Designs under test
- 3) Models
- 4) Results

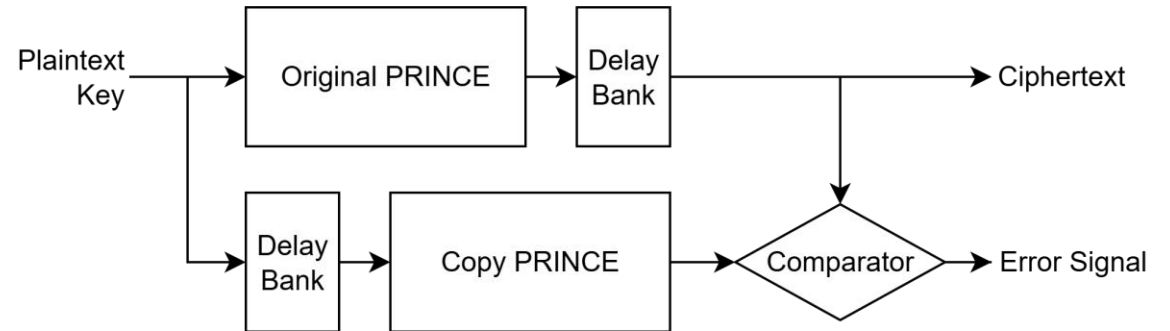
- Netlist synthesis: Synopsys Design Compiler
- Netlist parsing and instrumentation: *pyverilog* / *iverilog* (Python)
- Formal engine: Cadence JasperGold, TCL harness
 - Engines: {*Hts*, *Hps*}
 - CPU: 2 × EPYC 7713
 - Timeout: 7 days (168h)

Designs Under Test



- **AES Parity-Predictor** protected for Laser attacks
- Parity bits compared with predicted bits
- Injections into Data Unit, Key Unit, Predictor → only Flip Flops.

Sabotaged gates (FFs)	Execution Latency (cycles)
1006	48



- **PRINCE**: Lightweight block cipher
- **Lockstep configuration**: spatial redundancy with time shift, compared
- Injection into Original and Copy → Flip Flops and Comb gates

Sabotaged gates (Comb. gates & FFs)	Execution Latency (cycles)
12820	7

AES

- Round 10 DFA (1-bit injection)
- Round 9 / 8 DFA (1-bit injection)
- Round Skipping 1~9 (keeps R1 and R10)

PRINCE

- Inverse pass, Round 10 DFA (4-bit nibble injection)

Results: AES

	Method	$M = 1$	$M = 2$	$M = 3$
Duration	Eq. Checking	0.13 h	168.00 h [†]	N/A*
	128-bit DFA	0.95 h	R10: 12.40 h R9: 48.00 h R8: 161.00 h	R10: 18.95 h R9: 17.45 h R8: 31.35 h
	Round-skip	0.09 h	0.27 h	14.88 h
Total sets found	Eq. Checking	0 [‡]	6436 [†]	N/A*
	128-bit DFA	0 [‡]	R10: 144 R9: 294 R8: 294	R10: 7 R9: 0 [‡] R8: 0 [‡]
	Round-skip	0 [‡]	4	14

→ $M = 1$ → Proven but slower

Results: AES

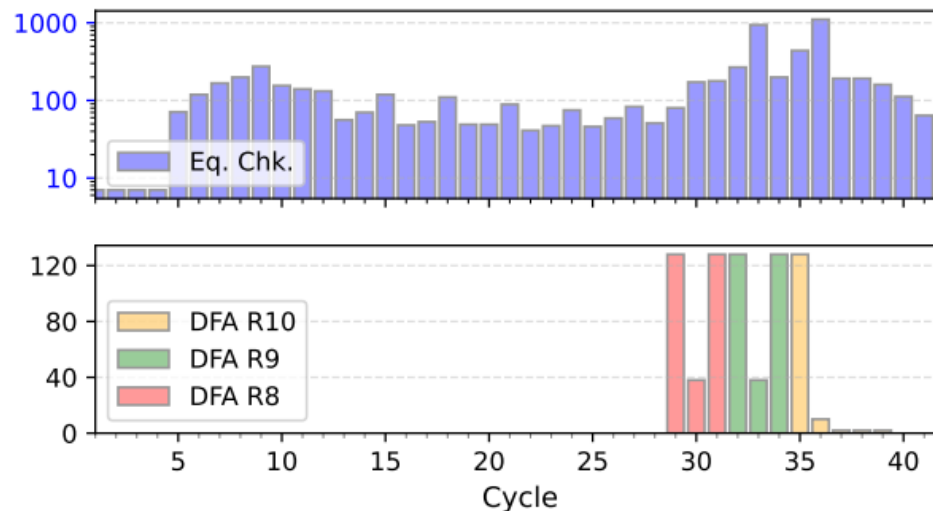
Method		$M = 1$	$M = 2$	$M = 3$
Duration	Eq. Checking	0.13 h	168.00 h [†]	N/A*
	128-bit DFA	0.95 h	R10: 12.40 h R9: 48.00 h R8: 161.00 h	R10: 18.95 h R9: 17.45 h R8: 31.35 h
	Round-skip	0.09 h	0.27 h	14.88 h
Total sets found	Eq. Checking	0 [‡]	6436 [†]	N/A*
	128-bit DFA	0 [‡]	R10: 144 R9: 294 R8: 294	R10: 7 R9: 0 [‡] R8: 0 [‡]
	Round-skip	0 [‡]	4	14

- ➔ $M = 1$ ➔ Proven but slower
- ➔ $M = 2$ ➔ Timeout for Eq. Checking
- ➔ $M = 3$ ➔ **Converges**

Results: AES

	Method	$M = 1$	$M = 2$	$M = 3$
Duration	Eq. Checking	0.13 h	168.00 h [†]	N/A*
	128-bit DFA	0.95 h	R10: 12.40 h R9: 48.00 h R8: 161.00 h	R10: 18.95 h R9: 17.45 h R8: 31.35 h
	Round-skip	0.09 h	0.27 h	14.88 h
Total sets found	Eq. Checking	0 [‡]	6436 [†]	N/A*
	128-bit DFA	0 [‡]	R10: 144 R9: 294 R8: 294	R10: 7 R9: 0 [‡] R8: 0 [‡]
	Round-skip	0 [‡]	4	14

- $M = 1$ → Proven but slower
- $M = 2$ → Timeout for Eq. Checking
- $M = 3$ → **Converges**
- **Lower number of CEX, directly linked to model**



- Found CEX are in expected cycles for DFA at R8, R9, R10

Results: PRINCE

➔ $M = 1 \rightarrow$ Proven

	Method	$M = 1$	$M = 2$	$M = 3$
Duration	Eq. Checking	0.02 h	168.00 h [†]	N/A*
	1-nibble DFA	0.02 h	4.98 h	2.88 h
Total sets found	Eq. Checking	0 [‡]	12842 [†]	N/A*
	1-nibble DFA	0 [‡]	25	2

Results: PRINCE

	Method	$M = 1$	$M = 2$	$M = 3$
Duration	Eq. Checking	0.02 h	168.00 h [†]	N/A*
	1-nibble DFA	0.02 h	4.98 h	2.88 h
Total sets found	Eq. Checking	0 [‡]	12842 [†]	N/A*
	1-nibble DFA	0 [‡]	25	2

- $M = 1$ → Proven
- $M = 2$ → Timeout for Eq. Checking
- $M = 3$ → **Converges**
- **Lower number of CEX**

Conclusion & Future Works

Conclusion

- Up to **92% speedup** and **44x CEX reduction**
- Convergence for higher multiplicities
- **Faulty models are easy to write** and reusable (not linked to design)
- **Direct mapping to exploits**, allows incremental and targeted analysis of attacks

Limitations

- Linked to specific, known attacks (“Antivirus”-like)
- Scalability issues with larger designs: requires over-constraining

Future Works

- Multithreading of proof with time slicing
- More realistic modeling: Clock glitching, EM...?
- More complex attacks: “Round 8 diagonal DFA attack”
- Attacks on other block ciphers (AES-256, PQC...), more complex designs (scalability)...

Our technology starts with You

© STMicroelectronics - All rights reserved.

ST logo is a trademark or a registered trademark of STMicroelectronics International NV or its affiliates in the EU and/or other countries.

For additional information about ST trademarks, please refer to www.st.com/trademarks.

All other product or service names are the property of their respective owners.

