Jan Burchard*, Maël Gay, Ange-Salomé Messeng Ekossono**, Jan Horáček**, Bernd Becker*, Tobias Schubert*, Martin Kreuzer**, Ilia Polian****

*University of Freiburg, **University of Passau

AutoFault:

TOWARDS AUTOMATIC CONSTRUCTION OF ALGEBRAIC FAULT ATTACKS

* Partially supported by DFG project „Algebraic Fault Attacks“.

Motivation

- Novel applications (IoT, cyberphysical, industry 4.0) process sensitive data → crypto blocks in systems designed by people who are not crypto experts.
- Lightweight ciphers: marginal security is the price for very low area footprint and power consumption.
- Goal: automatic construction of fault-injection attacks for a given cipher with as little user input as possible.
- Quickly find vulnerabilities in a new crypto implementation (or in a tweak of an existing implementation).
- Automatically analyze existing countermeasures.

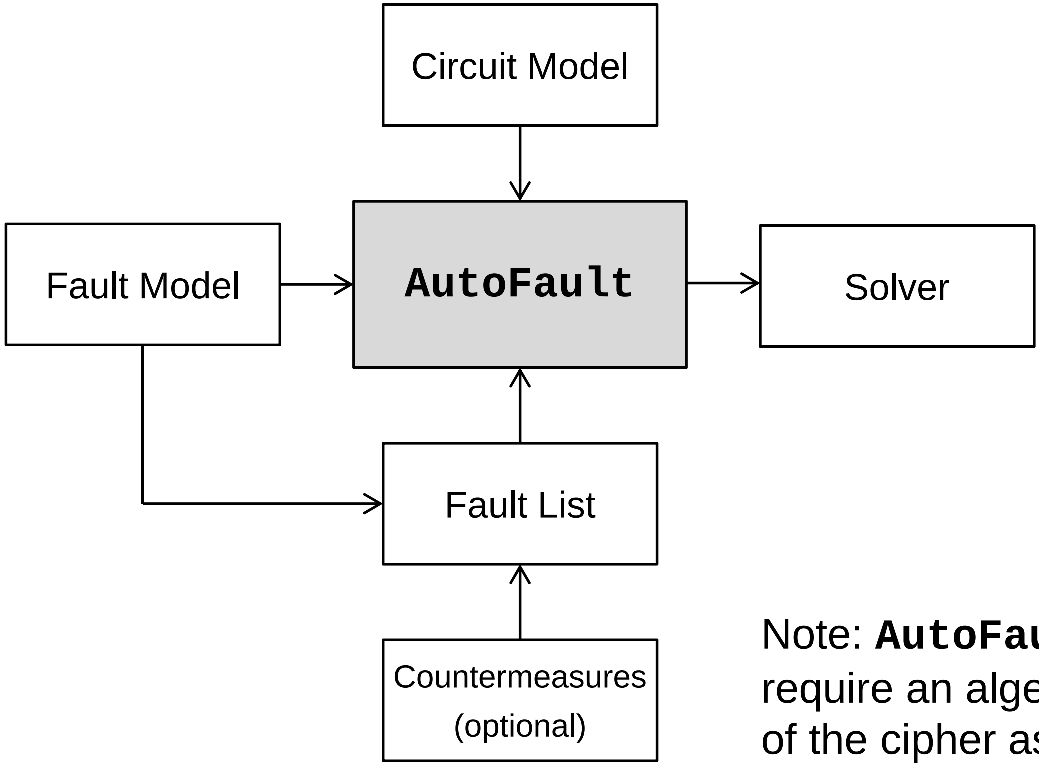
AutoFault

- Framework to construct an algebraic fault attack.
  - Less fault-injections than statistical attacks but higher precision of each fault-injection.
- Input 1: Hardware description of the cipher.
  - Current prototype: register-transfer, in principle: gate-level.
- Input 2: Fault list according to a fault model.
  - Information / guesses about possible fault types & locations.
- Output: Algebraic formula handed to a SAT solver.
- **Difference to earlier approaches: No inputs by skilled cryptoanalysts** (like “fault equations”)**!**

Outline

- Framework **`AutoFault:`** Motivation & potential use
- Realization
  - Attack construction
  - SAT solving
- Results
  - Small-scale AES
  - LED-64 (an actual lightweight cipher)

AutoFault Diagram



Note: **AutoFault** does not require an algebraic description of the cipher as input!

AutoFault Potential Uses

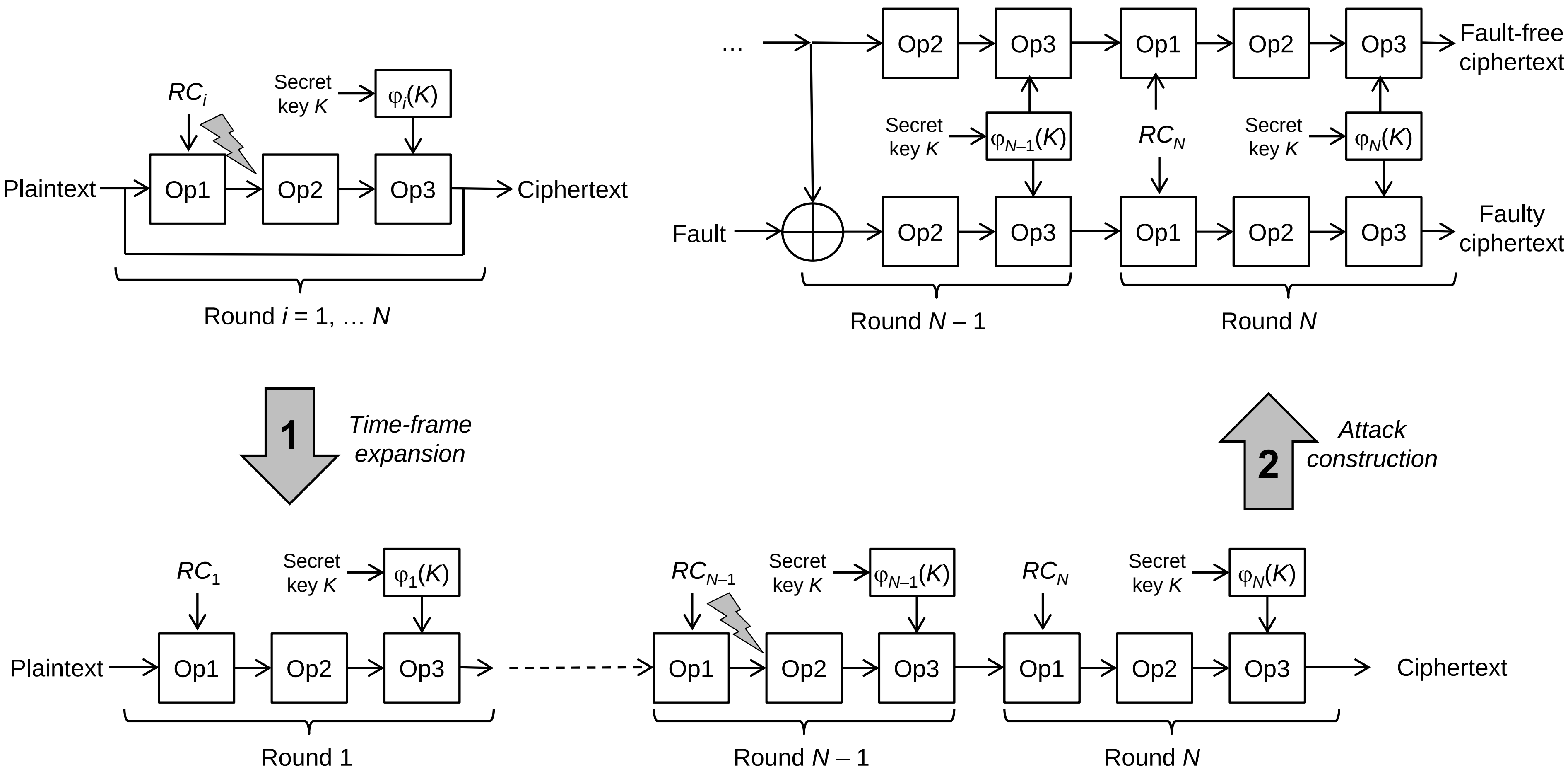
- Quickly analyze a tweak/„optimization“ of a known algorithm (e.g., less rounds, smaller space).
- Quickly analyze an implementation for, e.g., locations where faults lead to successful attacks.
- Quickly analyze locations not covered by low-level countermeasures (shields, sensors).

- Fundamental question: How far can we get in a fully automated manner, without non-trivial cryptanalsis?

Fault Attack Construction

- If cipher circuit is sequential, map it to a combinational circuit via time-frame expansion (TFE).
  - *N* clock cycles (e.g., *N* rounds) = *N* copies glued together.
  - Fault in cycle *i* = fault in *i*-th copy in TFE circuit.
- Construct differential model.
  - Two copies of TFE circuit fed by idential input (plaintext), round constants, key; the only difference is at fault site.
  - Differential. model starts from first fault-affected location.
  - Output of the circuit (ciphertext) set to values observed (in case of an actual attack) or simulated (during analysis).

Construction for a Hypothetical Cipher



SAT Solving

- Differential circuit mapped into conjunctive normal form (CNF) using Tseitin transformation.
  - E.g., AND gate with inputs a, b and output c is mapped to $c \equiv (a \cdot b)$ or, in CNF, $(\neg c + a)(\neg c + b)(\neg a + \neg b + c)$.
- Represent fault by SAT clauses involving variables from fault-free and fault-affected circuits according to fault model (e.g., maximal number of faulty bits).
- Set circuit output variables to ciphertext bits and run SAT solver in incremental mode.
  - if the solution is not a correct key, generate a conflict clause and continue searching for a different solution.

Experimental Evaluation

- Considered ciphers: Small-Scale AES, LED-64.

Cipher	Block size	# Rounds	Formula size (# clauses)
AES 2-2-4	16 bit	10	3,086
AES 4-4-4	64 bit	10	13,420
LED-64	64 bit	32	15,544

- Considered faults: Exactly 1, exactly 2, ≤ 4 or ≤ 8 bits in one nibble or two neighboring nibbles.
  - ≤ 4 faults in one nibble = „nibble fault“ in earlier work.
  - ≤ 8 faults in 2 neighboring nibbles = „byte fault“.

Results for Small-Scale AES

Fault model	AES 2-2-4		AES 4-4-4	
	Mean solve time [s]	Avg. # key candidates	Mean solve time [s]	Avg. # key candidates
1 bit, 1st nibble	16.46	5.16	9,574.61	620.26
1 bit, 1st / 2nd nibble	16.39	7.61	7,173.82	324.18
2 bit, 1st nibble	16.32	11.93	26,357.30	170.40
2 bit, 1st / 2nd nibble	17.98	25.76	23,661.00	55.00

- Runtimes increase drastically for larger space, and attack on the full AES (4-4-8) does not terminate.
- Higher fault multiplicity tends to complicate the search but can provide better restrictions.

UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

Results for LED-64

Fault model	Mean solve time [s]	Avg. # key candidates
1 bit, 1^{st} nibble	254.78	3,508.33
1 bit, 1^{st} / 2^{nd} nibble	442.72	3,044.23
2 bit, 1^{st} nibble	384.96	6,395.38
2 bit, 1^{st} / 2^{nd} nibble	847.77	2,303.87
1 bit, any nibble	712.70	4,896.29
1 bit, after Sbox	127.66	6,858.65
$\leq$ 4 bit, 1^{st} nibble	365.78	7,051.83
$\leq$ 8 bit, 1^{st} / 2^{nd} nibble	762.79	1,163.36

Corresponds to „nibble faults“ in [Jovanovic COSADE‘12]

Corresponds to „byte faults“ in [Zhao FDTC ‘13]

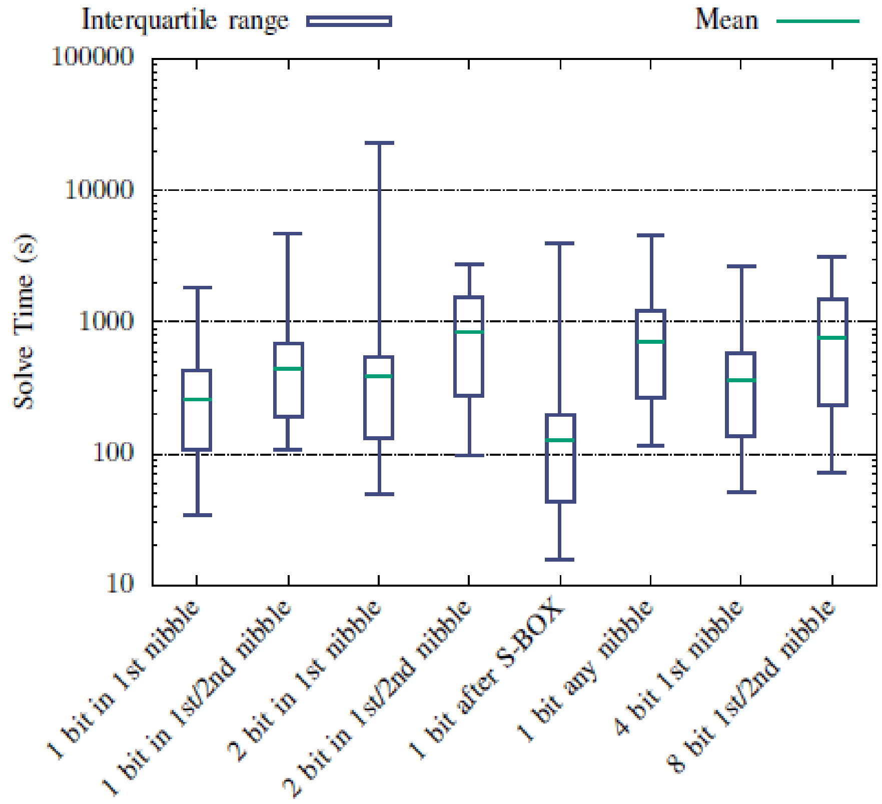
LED-64: Discussion

- First breaking of a state-of-the-art cipher with no manually derived cipher-specific cryptanalysis.
  - [Jovanovic IACR ePrint 2012]: fault tuples.
  - [Zhao FDTC 2013]: fault-dependent differentials.
- Number of key candidates (~7,000) inconsistent with [Jovanovic COSADE'12] (2^{19} – 2^{26}).
  - Fault tuples in [Jovanovic COSADE'12] may include candidates inconsistent with differential model.
  - Conflict clauses learned by SAT solver may eliminate parts of solution space with inconsistent solutions.

UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

LED-64: Run Time

- Typically, 10–15 minutes.
  - Better than [Jovanovic‘12] (several hours).
  - Worse than fastest configuration in [Zhao‘13] (45 seconds) with reverse cipher rounds and clauses from fault differentials.
- ~ 1 order of magnitude slowdown for a fully automatic attack without any cipher-specific tricks.



Conclusions

- Do the automatically constructed fault attacks work?
  – Yes, but only for lightweight ciphers.
- How much do we need to pay?
  – Approximately 1 order of magnitude in run time.
- What is the status of **`AutoFault`**?
  – Prototype implementation which reads a subset of VHDL and supports basic fault models.
- What is the next step in developing **`AutoFault`**?
  – Integrate advanced fault models (timing!)
- Are any fundamental questions still open?
  – Deriving cryptoanalytic conditions during solving.