Designing and Implementing Robust Code against Fault Injection Attacks

CLAPs Project - IRT - Nonoelec

April 23rd, 2019

The CLAPs project [2018-20]

Partners

CEA-DACLE, CEA-DSYS, CESTI, INRIA Corse, Verimag PACSS

Objectives

Methods and tools for the design and deployment of secure IoT solutions

- code robustness analysis against fault injection
- automated counter-measures integration
- attack detection mechanisms
- physical-level security analysis

Case study: Firmware Update / Bootloader

Outline

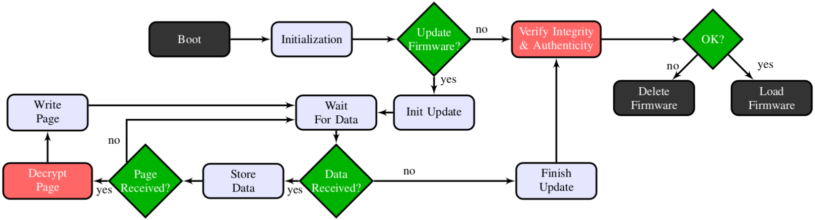
Security of Bootloader and Firmware Updater

Robustness analysis of a Firmware Updater

Using monitors to detect fault injection

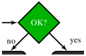
Some on-going work

Secured BL-FU Control Flow [Atmel2013][1]



Cryptographic functions: implemented in SW, dedicated HW IPs, or SW+specific processor instructions

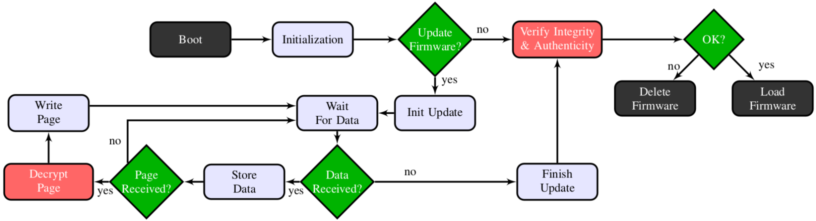
"System" components: implemented in SW, mostly HW-dependant and/or supported by dedicated HW (e.g. DMA for data movement)



Control logic: implemented in SW

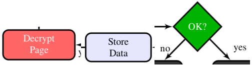
(1) At02333: Safe and secure bootloader implementation for sam3/4

Fault Injection Attacks applied to SBFU

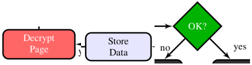


Fault models, at the Instruction Set Architecture (ISA) level:

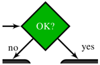
- Data alteration, down to the bit level.
  - ROM / RAM, processor registers
  - Bit flip, bit stuck-at
  - Typically: modification of loop counters, crypto data, opcode corruption.



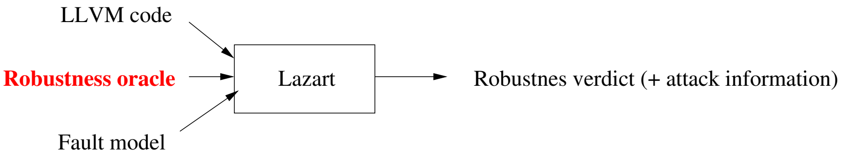
- Instruction skip, instruction modification
  - Typically: NOP execution, arbitrary jumps



- Modification of the control flow,
  - e.g., test inversion



(source-level) Robustness analysis of a FU with Lazart



Input

- the source code (LLVM) of the target application
- an "oracle", specifying the expected *security property*
- a (source-level) fault model $\rightsquigarrow$ **effects** of the φsycal attacks

Output

A robustness verdict

(+ attack statistics + non-robust executions + counter-measure metrics + . . .)

A basic Firmware Updater (inspired from [Atmel2013])

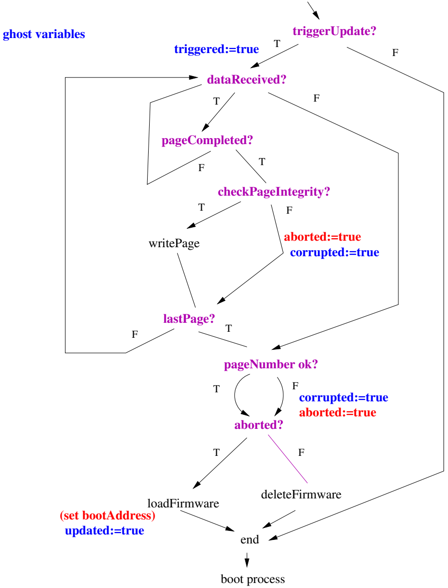
- firmware transmission *modeled* as a buffer copy:
  payload = sequence of page + actual page number
  - each page copied byte/byte
  - **integrity check** performed on each page

- verification performed upon transfer termination:
  - is the number of pages received correct ?
  - is the firmware integrity correct ?

- if yes, the copy is loaded as a new firmware
  $\rightarrow$ update successful, **boot address** is set
  otherwise the copy is deleted
  $\rightarrow$ no update

- **ghost variables** added to specify the security properties

Control-flow



Security Properties

P1: **do not** *load a* **corrupted** *firmware*

1. the attacker requests for an update of a corrupted firmware
2. and breaks the verification steps

$\Rightarrow$ execution resumes with an incorrect firmware ...

$$P_1 \equiv \neg(\mathit{triggered} \wedge \mathit{corrupted} \wedge \mathit{updated})$$

Security Properties

P1: **do not** *load a* **corrupted** *firmware*

1. the attacker requests for an update of a corrupted firmware
2. and breaks the verification steps

$\Rightarrow$ execution resumes with an incorrect firmware ...

$$P_1 \equiv \neg(\mathit{triggered} \wedge \mathit{corrupted} \wedge \mathit{updated})$$

P2: **do** *(correctly) load a* **correct** *firmware*

1. the user requests for an update (e.g., in case of security patch)
2. the attacker prevents the update to succeed

$\Rightarrow$ execution continues with an out-of-date firmware ...

$$P_2 \equiv \neg(\mathit{triggered} \wedge ((\overline{\mathit{corrupted}} \wedge \overline{\mathit{updated}}) \vee (\mathit{updated} \wedge \mathit{ldAddr} \neq \mathit{bootAddr})))$$

Some results

About 64 experiments, combining:

- 2 fault models:
  - test inversion
  - data mutation: `curPage`, `pageNumber`, `aborted`, `ldAddr`
- 2 counter-measure levels
  - no counter-measure ($\overline{CM}$)
  - duplicate twice each control-flow condition (CM)

Example of results obtained:

Fault model	**1 flt - $\overline{CM}$**	**2 flts - $\overline{CM}$**	**1 flt - CM**	**2 flts - CM**
P1 - test inversion	2	14	0	1
P1 - pageNumber	0	0	0	0
P1 - aborted	2	13	0	1
P2 - test inversion	1	5	0	1
P2 - loadAddress	1	0	0	1

So what . . . ?

Confirm some expected behaviors

. . . but also exhibit some *less* expected ones, e.g.:

- no attacks on `pageNumber` mutation
- property $P1$ could be refined . . .

$\rightarrow$ highlights critical execution paths w.r.t. **security properties**

So what . . . ?

Confirm some expected behaviors

. . . but also exhibit some *less* expected ones, e.g.:

- no attacks on `pageNumber` mutation
- property $P1$ could be refined . . .

$\rightarrow$ highlights critical execution paths w.r.t. **security properties**

Next steps

- refine the implementation (and/or use **existing** ones)
  - memory layout
  - firmware encryption

  $\rightarrow$ more detailed properties, dedicated specification language
- other counter-measures
  (e.g., invariant synthesis for CFI, runtime-monitors, etc.)
- counter-measure analysis . . .

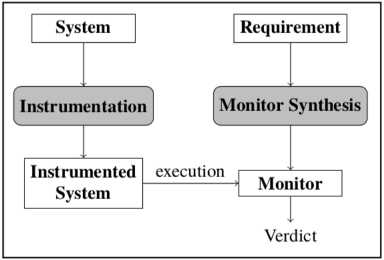
Runtime Verification, aka Monitoring

Monitoring is a verification method to analyse system executions (at runtime).

- system is instrumented to retrieve the relevant information
- monitor analyses this information

Monitor produces verdict:

- Pass when execution respects requirements
- Fail in case of a violation

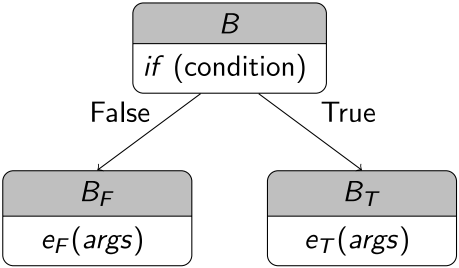


Advantages:

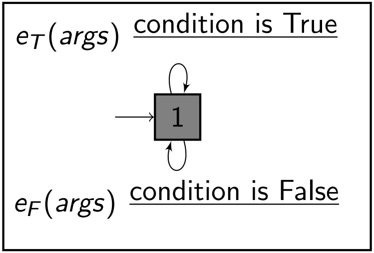
- Lightweight & "easy" to deploy
- Verify the actual execution
- Rigorous method which provides formal correctness guarantees
- Compatible with other verification solutions

Inria specifies and implements monitors for the test inversion and jump attacks.

Test Inversion Attack (simplified)



Monitor:

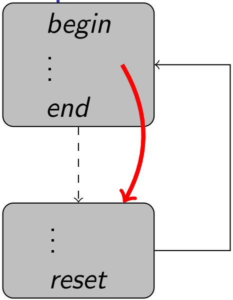


Attack when:

- B_F is executed and condition is True; or
- B_T is executed and condition is False.

The monitor reports a violation if and only if there is a test inversion attack (sound & complete).

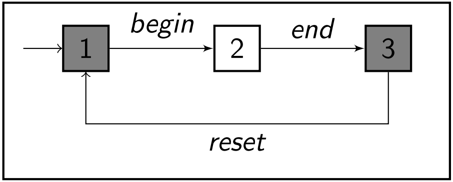
Jump Attacks – A (simplified) Example



Attack: interrupt execution of a basic block using a forward jump.

Requirement: *begin* is followed by *end* before a new occurrence of *reset*.

Monitor:



Good execution:

- *begin.end.reset.begin.end*
- State (3)

Bad executions:

- *begin.end.reset.begin*
- State (2)

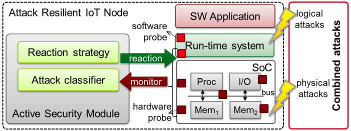
The monitor reports a violation if and only if there is a jump attack (sound & complete).

Some other on-going work

Attack detection in IoT

- existing counter-measures are from smart-card domains
- IoT node behavior rather “predictable”, less data dependent

$\rightarrow$ lightweight supervised ML

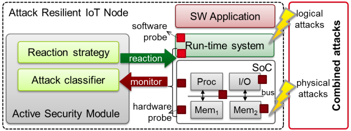


Some other on-going work

Attack detection in IoT

- existing counter-measures are from smart-card domains
- IoT node behavior rather "predictable", less data dependent

→ lightweight supervised ML



Cross-analysis of SW and Physical robustness results

3 evaluation levels addressed in the project:

- source/LLVM level (Lazart)
- binary code level (Celtic)
- physical attacks

→ how to combine them for development/certification purposes ?

Thanks for your attention . . .

Attack Detection in IoT

- Why: many counter-measures are from the smart-card domain
  - Expensive in cost and power consumption
  - Stack-up counter-measures for the increasing number of attacks
  - Products 2-3 years life-long $\longrightarrow$ IoT devices: more than 10 years
- What: active security = detection of attack and adequate reaction
  - Model the application behavior, and detect deviations $\longrightarrow$ supervised machine learning (no costly neural-networks)
  - IoT nodes are rather "predictable", unlike host IDS
- Differentiators $\longrightarrow$ low-cost/low-power detection of combined attacks
  - No need for training with attack data (in theory)
  - Potentially unknown attacks
  - Programmable solution