Software Countermeasures against Fault Attacks

Karine Heydemann
Fault injection attacks

Fault injection means
- Since 20 years via physical means: laser beam, electromagnetic pulse, clock or voltage glitch [El Bar et al., 2006]
- Recently via software means: row hammer, clock skrew

Impact
- Global: clock or voltage glitch [Yuce et al. 2017]
- Local: laser ou electromagnetic pulse [Dehbaoui et al. 2012]

Observed effects in storage elements
- Bit(s) set or reset, bit flip(s)
- Transient ou permanent (stuck-at)
Protections against fault injection attacks

- **Hardware-based countermeasures** [El Bar et al., 2006]
  - Light sensor, glitch detectors [Zussa et al., 2014]
  - Redundancy [Karaklajic et al., 2013]
  - Error correcting codes (registers, memory)

- **Too expensive for small devices and no full guaranty**

- **Software-based countermeasures** [Verbauhede, 2011] [Rauzy et al., 2015]
  - Redundancy at function level
  - Algorithm-specific protection (e.g. RSA)
  - Ad-hoc protections designed by expert engineers

- In practice combination of both in secure elements
SW protections against fault injection attacks

- **Manually added**
  - Tedious, error-prone
  - Highly expensive
  - Expertise needed

- **Need automation and capitalization**
  - Cost reduction, availability for non-experts
  - Adaptable to a specific product
  - Trade-off between security and performance

- **Need generic protections**
  - Not dedicated to a class of algorithms (crypto)
  - Against fault injection effects at software level...
Fault attacks at software level

Fault injection means
- Fault location / targeted part of the HW
- Running code

[In Algorithm]
```c
int verify(S, P) {
    int r;
    if (S == P)
        r = 1;
    else
        r = 0;
    return r;
}
```

Fault observation depends on
- Fault injection means
- HW target
- Fault location / targeted part of the HW
- Running code

[Yuce et al., JHSS 2018]
Fault attacks at software level

Fault Exploitation

Fault Observation

Fault Propagation

Fault Manifestation

Fault Injection

Fault exploitation

- Macro view of fault attacks
  - Cryptographic key retrieving [Dehbaoui et al., 2013]
  - Bypassing secure boot [Timmers et al., FDTC 2016]
  - Taking over a device [Timmers et al., FDTC 2017]
  - Privilege escalation [Vasselle et al. FDTC 2017]

- Useful from an attacker point of view

Journée Attaque par Injection de Fautes 29/05/2018

[Dehbaoui et al., JHSS 2018]
Software fault characterization

- Characterization of possible fault observations
- Necessary to design software countermeasures

Fault model

- **Simplified or abstracted representation** of a physical fault effects affecting an embedded software
- **At a given code level**: binary, assembly code, IR, source code
Fault models at software level

Common instruction-level fault models
- Instruction skip
- Instruction replacement
- Test inversion
- Jump insertion
- Computation or register corruption
- Data memory corruption

Application of Attack Potential to Smartcards
- Common Criteria, version 2.9. May 2013

[Yuce et al., JHSS 2018]
Fault models at software level

Fault modeling

- At source code level
  - Control-flow disruption
  - Variable corruption
  - Combination

- At assembly level
  - Instruction(s) skip
  - Instruction(s) replacement
  - Corruption of loaded data
  - Register(s) corruption(s)

[Berthomé et al., 2010]
[Berthomé et al., 2013]

[Kelly et al., 2017]
[Yuce et al., 2017]
[Timmers et al., 2016]
[Dureuil et al., 2015]
[Rivière et al., 2015]
[Moro et al., 2013]
[Balash et al., 2011]
[Verbauwhede et al., 2011]
[El Bar et al., 2006]

Physical effects
Software protection against fault attacks

**Code hardening**

- **At which code level?**
  - **Source**
    - Code review, portability, independent from tools
    - Fault models, compilation optimization
  - **Compilation**
    - Adaptability and/or control over code optimization
    - No existing compilation tool
  - **Assembly**
    - Realistic fault models, low level information available
    - Target specific, potential lack of source code information
  - **Binary**
    - Attacked code, global view, availability of library codes
    - Even more lack of semantic information

→ Multiple needs

Journée Attaque par Injection de Fautes 29/05/2018
Outline

- Principle of software countermeasures
  - Data integrity
  - Code integrity
  - Control-flow integrity

- Compiler-assisted code hardening
  - Protection against instruction skip
  - Loop hardening scheme
Countermeasures for data integrity

Fault model
- Data corruption: register corruption, load corruption

Redundancy-based protections
- Duplication of instructions involved in a computation
- Comparison of results of duplicated computations
- Detection of
  - Register corruption (r1 or r2)
  - Load corruption
- Need available registers

A. Barenghi et al. *Countermeasures against fault attacks on software implemented AES*. 5th Workshop on Embedded Systems Security (WESS’10)
Countermeasures for data integrity

Fault model

- Data or data-related computation corruption: register corruption, load and memory corruption

Redundancy-based protections

- Data duplication in addition to instruction duplication
- Detection of
  - Memory corruption
  - Load corruption
  - Register corruption
- High overhead: performance and memory footprint

Countermeasures for code integrity

Fault model
- Instruction corruption

Redundancy-based protections
- Instruction duplication with detection
- Detection of
  - One instruction skip
  - Some instruction replacements

A. Barenghi et al. Countermeasures against fault attacks on software implemented AES. 5th Workshop on Embedded Systems Security (WESS’10)
Countermeasures for code integrity

Fault model
- Instruction skip

Redundancy-based protections
- Instruction duplication without detection
  - Tolerance to one instruction skip
  - Only for idempotent instructions
  - Transformation of non-idempotent instructions

Countermeasures for code integrity

Fault model
- Instruction skip

Redundancy-based protections
- Instruction duplication without detection
- Tolerance to one instruction skip
- Only for idempotent instructions
- Transformation of non-idempotent instructions

No software-only protection for full code integrity (i.e. against all kinds of instruction replacement or disruption)

Control flow integrity

Fault model
- Jump insertion

Different levels of control-flow integrity
- Intra basic block integrity of straight-line code
- Intra procedural integrity of control flow transfers inside a function (control flow graph)
- Inter procedural integrity of function calls and returns
Intra basic block control flow integrity

Counter-based protections [Akkar et al., 2003]

- Dedicated counters incremented between instructions
- Check of their values at some specific points
- At the end of each BB: only detects some intra BB jumps

 foo1:

... call foo2
...
... call foo3
...

 foo2:

\[
\begin{align*}
\text{cnt} &:= \text{val} \\
\text{inst1} &:= \text{cnt++} \\
... &:= \text{cnt++} \\
\text{instN} &:= \text{cnt++} \\
\text{cmp} & \text{ cnt, end_val} \\
\text{b.eq} & \text{ next} \\
J &< \text{error} \\
\end{align*}
\]
Control flow integrity

Counter-based protections

- Dedicated counters incremented between instructions
- Check of their values at some specific points
  - At the end of each BB: only detects some intra BB jumps
  - At the beginning of target blocks
    - Need for extra code

```plaintext
foo1:
...
call foo2
...
call foo3
...
foo2:

cnt := val1
inst1
cnt++
...
cnt++
instN
cnt++
bne next

foo1:
cmp cnt, XXXXX
b.ne <error>
```
Control flow integrity

Counter-based protections [Lalande et al., 2014]

- Dedicated counters incremented between instructions
- Check of their values at some specific points
  - At the end of each BB: only detects some intra BB jumps
  - At the beginning of target blocks
    - Need for extra code
    - Overlap of counters initialization and check

```plaintext
foo1:
...

call foo2
...

call foo3
...

foo2:
cnt := val
inst1

cnt++
...

cnt++
instN
cnt++
bne next

cnt++
inst++
...
cnt++
instN

cnt++
j next

cnt2 := val
cmp cnt, XXXXX
cnt2++
bne <error>
...
```
Countermeasures for control flow integrity

**Signature-based protections** [Oh et al. 2002] [Goloubeva et al., 2005]
- Unique identifier / signature assigned to every basic block (and function)
- Use to check every single control flow transfer
- Global signature computation limits the number of checks
- Ensure the CFG integrity
- Need branch condition integrity / data integrity

**Combination** [SIED, 2003]
- Step counters inside basic blocks
- Signature for control flow transfers
- Signature computed with the branch condition value

---

```
foo1:
  ...
  ... call foo2
  ...
  ... call foo3
  ...

foo2:
  mov rs, id2
  check(rs, id2)
  mov rs, id3
  mov rs, id4
  check(rs, id3)
  check(rs, id4)
  mov rs, id5
  check(rs, id5)
  mov rs, id2
  check(rs, id6)
```

Journée Attaque par Injection de Fautes 29/05/2018
Outline

- **Principle of software countermeasures**
  - Data integrity
  - Code integrity
  - Control-flow integrity

- **Compiler-assisted code hardening**
  - Protection against instruction skip
  - Loop hardening scheme
Protection at compilation-time

- Protection scheme against instruction skip [Moro et al. 2014]
- Main principle: duplication of idempotent instructions
- Take advantage of compilation flow to
  - Force the generation of idempotent instructions
    - Avoidance of some instructions at the selection step
    - Modification of the register allocation
    - Additional transformation for remaining non-idempotent instructions (e.g. push and pop instruction that use and modify the stack pointer)
  - Add an instruction duplication pass
  - Let the scheduler optimize the resulting protected code
- Results in automatically protected code with better code size and performance

Compile-time loop hardening

Motivation

- Several attacks exploit a corruption of loop iteration count (early or deferred exit)
  - Buffer overflows [Nashimoto et al. 2017]
  - Cryptanalysis by round reduction [Dehbaoui et al. 2013, Espitau et al. 2016]
  - Authentication process [Dureuil et al., FISSC, 2016]
- Full duplication schemes are too expensive
- How to automatically protect a loop?
Loop hardening scheme

Fault model

- One instruction skip
- One general purpose register corruption
- During loop execution

Security objective

- The loop performs the right iteration count
- The loop exits from the right exit
- Otherwise an attack is detected
Loop hardening scheme

Protection principle

- For each loop exit, check its outcome

Realisation

- **Duplication** of all the instructions involved in the computation of an exit condition
- Addition of **verification basic blocks** on all the paths following from an exiting block
- Protection of the **internal control flow** that may impact an exit condition
Loop hardening scheme

For each exit of a loop

- Determination by a backward analysis of the instructions involved in an exit condition or in an condition of a branch that may influence an exit condition

- Instruction duplication
  - Creation of a second data flow leading to a duplicated value of the condition, independant from the original one

- Addition of verification blocks
  - Checks of the duplicated exit condition inside and outside of the loop to verify the exiting branch
  - Checks of the duplicated conditions of the internal branches on all possible following paths
  - Call to a fault detection handler
Loop hardening pass and a compilation flow

Automation and insertion in a compilation flow
- Implemented in a compiler (LLVM 3.9+) at the intermediate level
- Insertion after optimization passes that may alter the protection

Experimental results
- 99% of harmful simulated fault are detected
- Low overhead in performance and code size

Harmful post-securing transformations and optimizations
- All kind of redundancy elimination
- Instruction selection, register allocation, code placement optimization

- Compiler is not compliant with protection / security properties
- Need to analyze the generated code
- Need to deactivate, adapt, or add some passes to enforce the security property

Summary and conclusion

- Various types of protection
  - Large set of fault models / attacker capabilities

- Need of automatic code hardening and against a large set of (faults) attacks
  - Compiler-assisted code hardening
  - Framework enabling the analysis and the preservation of security properties
    - In the compilation flow
    - For a post-compilation robustness analysis

- Combination of protections
  - Interaction between protections? Stacking or smarter combination?
References


References


[Kumar et al., 2017] S. V. Dilip Kumar; Sikhar Patranabis; Jakub Breier; Debdeep Mukhopadhyay; Shivam Bhasin; Anupam Chattopadhyay; Anubhab Baksi. A Practical Fault Attack on ARX-Like Ciphers with a Case Study on ChaCha20. 2017 Workshop Fault Diagnosis and Tolerance in Cryptography Workshop (FDTC), 2017. IEEE.


References


References


