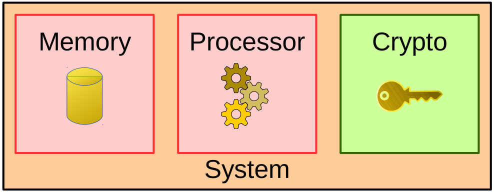
Protecting the Control Flow of Embedded Processors against Fault Attacks

Mario Werner[1], Erich Wenger[2], and Stefan Mangard[1],
[1] **Graz University of Technology**
[2] **Infineon Technologies AG, Munich**

5th November 2015, Bochum

Context and Motivation

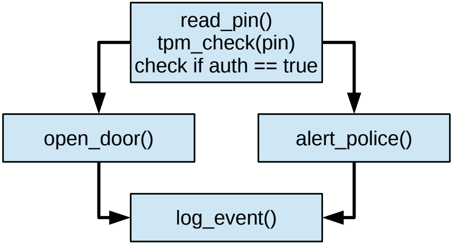
- Embedded systems are everywhere
- Assets in malicious environment



- Various assets
- Protecting cryptographic primitives is insufficient

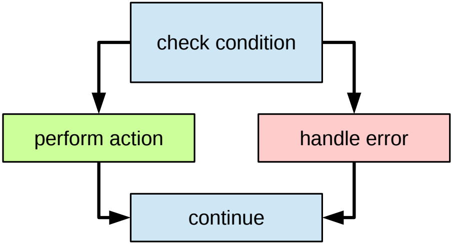
Do we really care about the Processor?

```
unsigned pin = read_pin();
bool auth = tpm_check(pin);
if( auth ) {
        open_door();
} else {
        alert_police();
}
log_event();
```

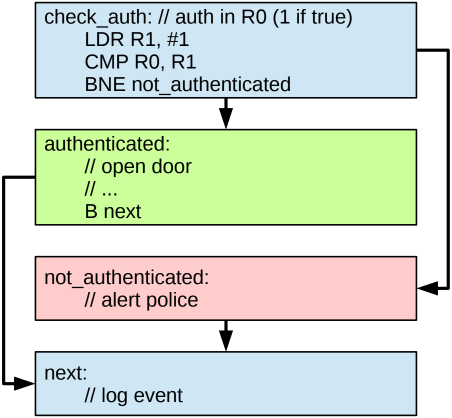


Do we really care about the Processor?

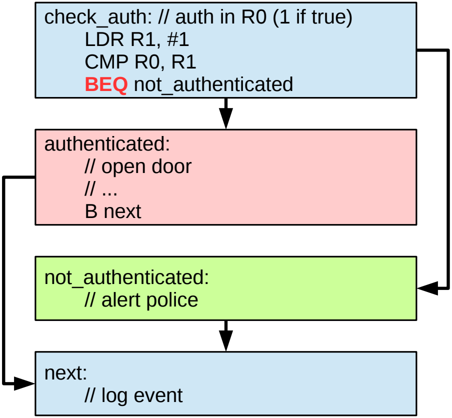
```
unsigned pin = read_pin();
bool auth = tpm_check(pin);
if( auth ) {
	open_door();
} else {
	alert_police();
}
log_event();
```



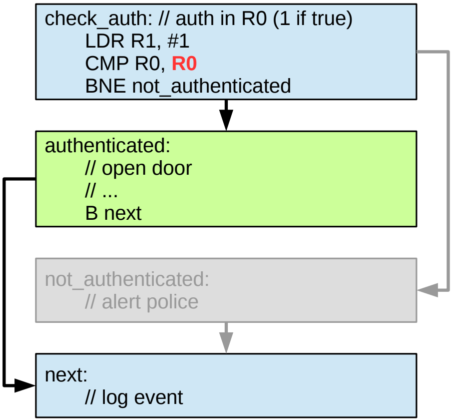
Do we really care about the Processor?



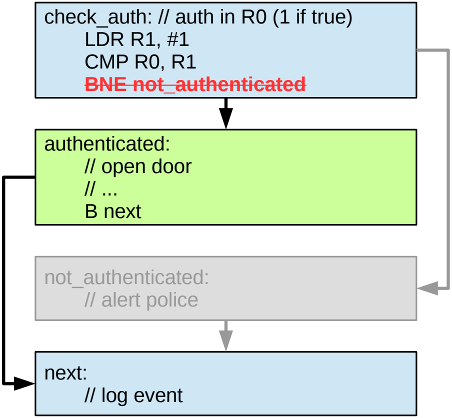
Do we really care about the Processor?



Do we really care about the Processor?



Do we really care about the Processor?



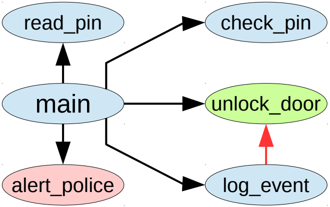
Goal and Results

- Goal: Enforce control-flow integrity
- Results:
  - Analysis and evaluation of signature functions
  - Detect a faulty instruction with 99.9 % within 3 cycles (arbitrary fault)
  - Resistant against at least 7 precise bit flips injected across two instructions
  - HDL implementation for a Cortex-M3 clone
  - LLVM based toolchain
  - 6.4 % hardware overhead
  - 2 % to 71 % runtime overhead

Control-Flow Integrity

Execute code as programmed

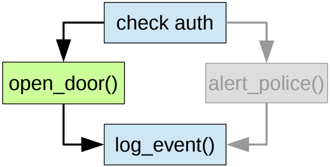
- Perform only intended function calls
- Traverse control flow graph along programmed edges
- Execute basic blocks from start to end
- Preserve instructions and their order



Control-Flow Integrity

Execute code as programmed

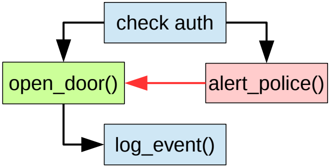
- Perform only intended function calls
- Traverse control flow graph along programmed edges
- Execute basic blocks from start to end
- Preserve instructions and their order



Control-Flow Integrity

Execute code as programmed

- Perform only intended function calls
- Traverse control flow graph along programmed edges
- Execute basic blocks from start to end
- Preserve instructions and their order



Control-Flow Integrity

Execute code as programmed

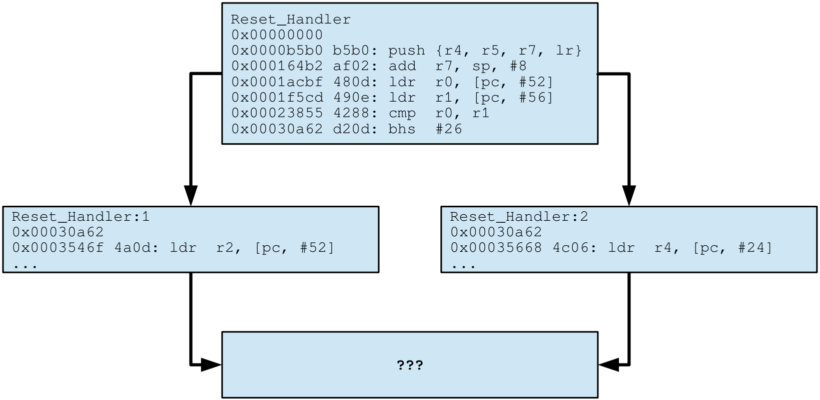
- Perform only intended function calls
- Traverse control flow graph along programmed edges
- Execute basic blocks from start to end
- Preserve instructions and their order

```
check_auth: // auth in R0 (1 if true)
        LDR R1, #1
        CMP R0, R0
        BNE not_authenticated
```

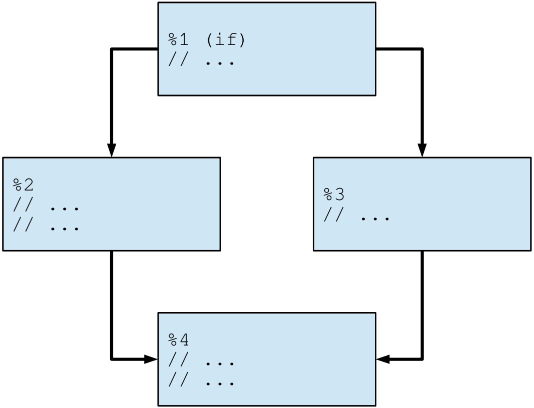
Concept

- Instruction stream integrity through derived signatures [MM88]
- Generalized path signature analysis (GPSA) [WS90]
- Optimize against fault attacks
- Implemented as hybrid scheme
- Dedicated assertions
- Continuous checks

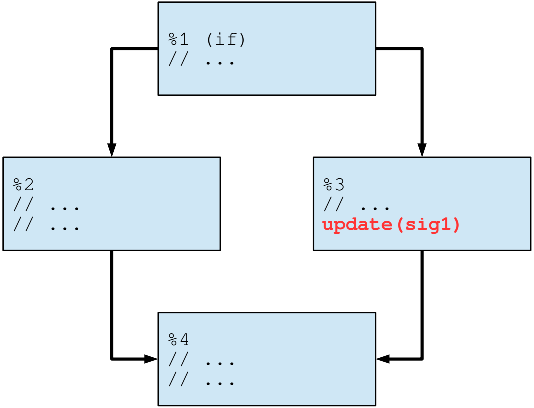
Derived Signatures [MM88]



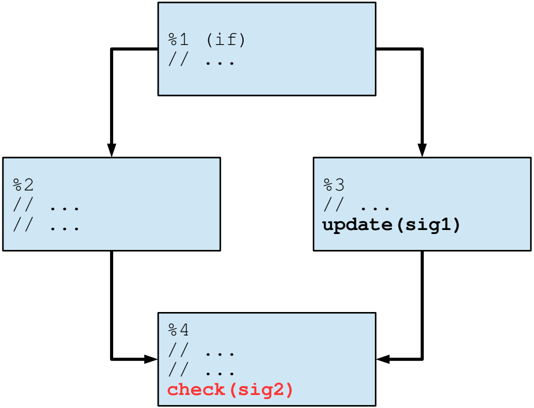
Generalized Path Signature Analysis [WS90]



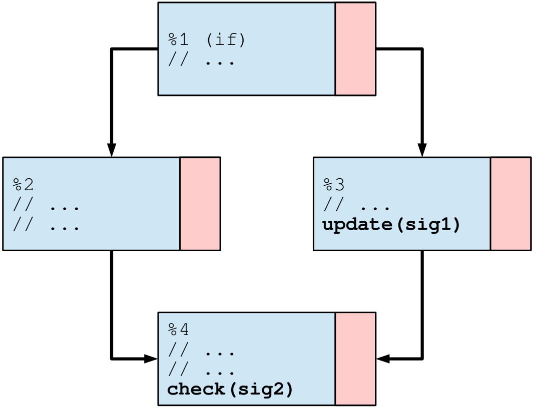
Generalized Path Signature Analysis [WS90]



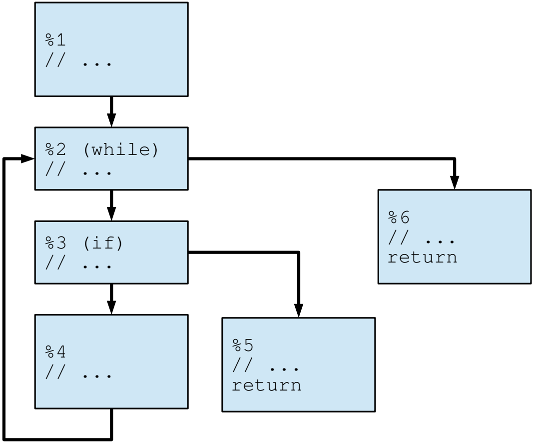
Generalized Path Signature Analysis [WS90]



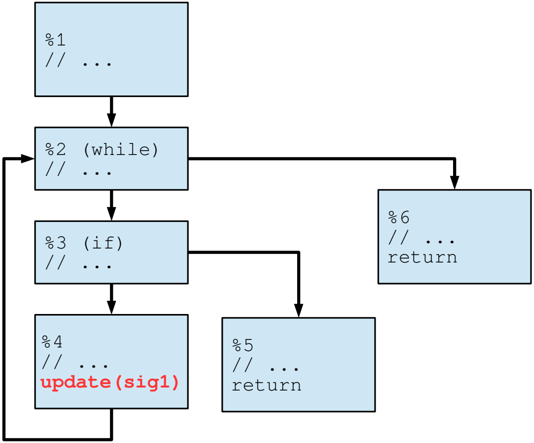
Generalized Path Signature Analysis [WS90]



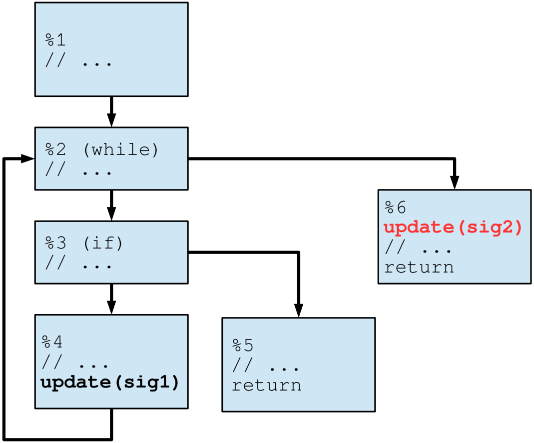
Generalized Path Signature Analysis [WS90]



Generalized Path Signature Analysis [WS90]



Generalized Path Signature Analysis [WS90]



Signature Functions against Fault Attacks

- Compression function
- Avoid collisions within one cycle
- Qualitative Requirements for GPSA:
  - Reliability: $S_{j+1} \oplus \Delta_{S_{j+1}} = f(S_j, I_j \oplus \Delta_{I_j})$
  - Error preservation: $S_{j+1} \oplus \Delta_{S_{j+1}} = f(S_j \oplus \Delta_{S_j}, I_j)$
  - Non associativity: $f(f(S_j, I_j), I_k) \neq f(f(S_j, I_k), I_j)$
  - Invertibility: $S_j = f^{-1}(S_{j+1}, I_j)$

$\rightarrow$ single faulty instructions detectable

Quantitative Evaluation

- MISRs and CRCs with various polynomials
- How hard is it to bypass the protection?
- Quality function: $q(j, t) = HW(\Delta_{l_j}) + HW(\Delta_{l_{j+t}})$
- Worst case behavior $\texttt{min}(q)$ matters

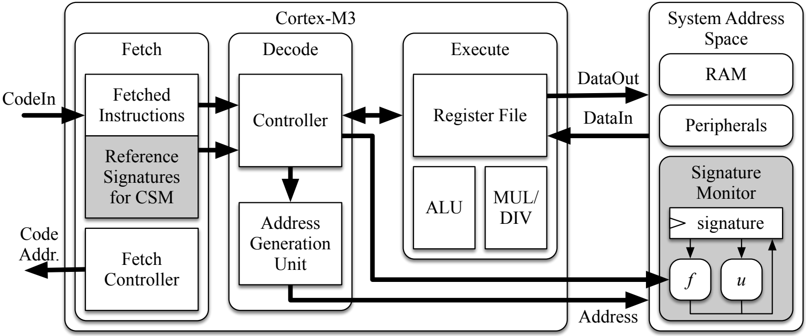
  $\rightarrow$ CRCs are better than MISRs against faults

  $\rightarrow$ $\texttt{min}(q) = 8$ for CRC-32C and CRC-32Q

Implementation

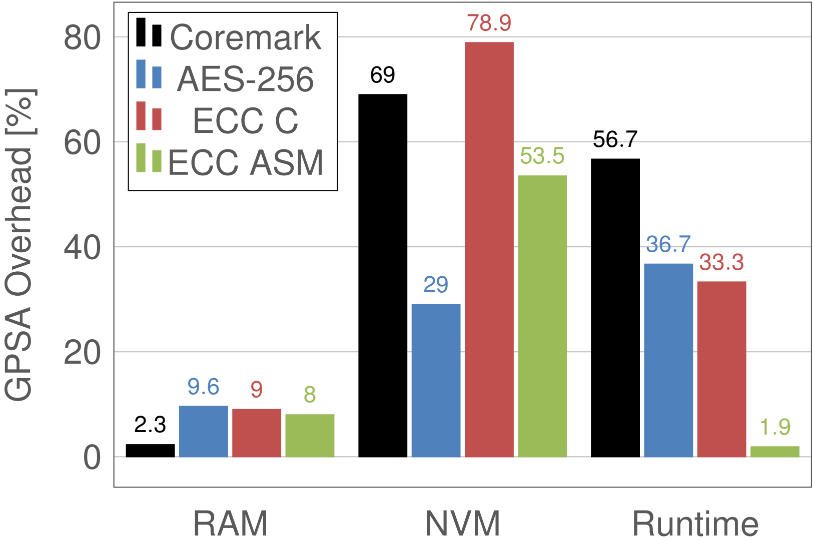
- Hardware:
  - Monitor for derived signatures
  - Extended fetch unit
- Software:
  - Compiler for
  - ... GPSA signature updates
  - ... assertions
  - Post-processing tool for
  - ... update and check constants
  - ... continuous signature monitoring (CSM)

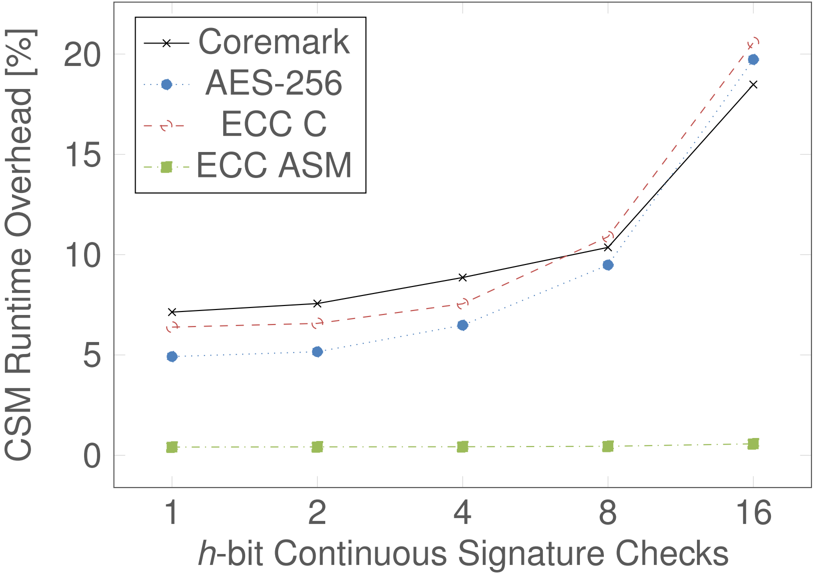
Hardware Modifications



Evaluation

- Hardware:
  - CPU Core: 37 kGE
  - Monitor: 1.5 kGE (4 %)
  - Monitor + Core with CSM: 39 kGE (6.4 %)
- Benchmarks: Modified vs stock LLVM
  - Coremark
  - AES-256
  - Elliptic Curve Cryptography in C and with ASM


Coremark
AES-256
ECC C
ECC ASM
GPSA Overhead [%]
80
60
40
20
0
2.3
9.6
9
8
69
29
78.9
53.5
56.7
36.7
33.3
1.9
RAM
NVM
Runtime


Coremark
AES-256
ECC C
ECC ASM
CSM Runtime Overhead [%]
0
5
10
15
20
1
2
4
8
16
h-bit Continuous Signature Checks

Conclusion

- Analysis and evaluation of signature functions
- Detect a faulty instruction with 99.9 % within 3 cycles (arbitrary fault)
- Resistant against at least 7 precise bit flips injected across two instructions
- HDL implementation for a Cortex-M3 clone
- LLVM based toolchain
- 6.4 % hardware overhead
- 2 % to 71 % runtime overhead

Protecting the Control Flow of Embedded Processors against Fault Attacks

Mario Werner[1], Erich Wenger[2], and Stefan Mangard[1],
[1] **Graz University of Technology**
[2] **Infineon Technologies AG, Munich**

5th November 2015, Bochum

References I

[MM88] A Mahmood and E.J. McCluskey, Concurrent error detection using watchdog processors-a survey, IEEE Transactions on Computers **37** (1988), no. 2, 160–174.

[WS90] Kent D. Wilken and John Paul Shen, Continuous signature monitoring: low-cost concurrent detection of processor control errors, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **9** (1990), no. 6, 629–641.