DATE 18
DESIGN, AUTOMATION & TEST IN EUROPE
19 – 23 March, 2018 · ICC · Dresden · Germany
The European Event for Electronic System Design & Test

Securing Conditional Branches in the Presence of Fault Attacks

Robert Schilling[1,2], Mario Werner[1], Stefan Mangard[1]

[1]Graz University of Technology, [2]Know-Center GmbH

March 22, 2018

Overview

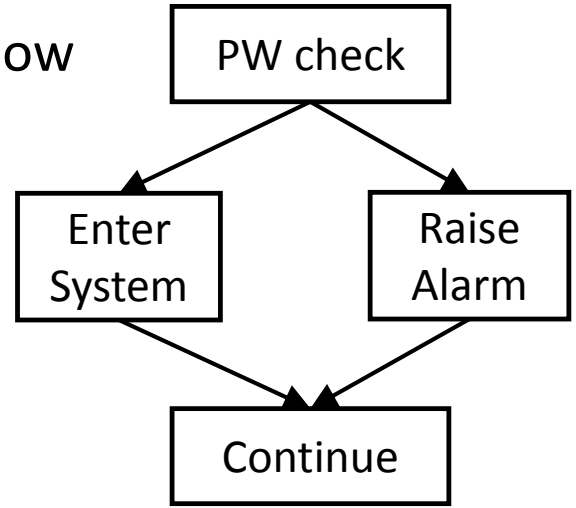
- Introduction to control-flow integrity and data protection
- Generic approach to protect conditional branches without hardware extensions
- Protected comparison algorithms based on AN-codes
- Prototype compiler based on LLVM
- Evaluation

Motivation

- Fault attacks can modify the code and data
- Control-flow integrity (CFI) restricts the control-flow to valid execution traces
- Data encoding to protect arithmetic

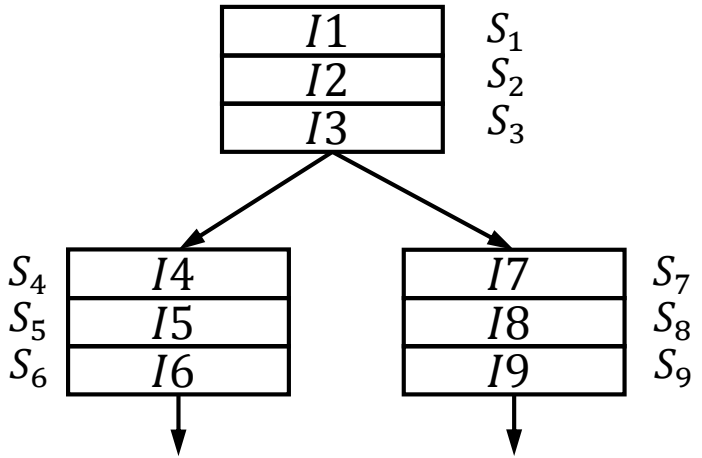
- **No protection for conditional branches**

- Conditional branches are critical instructions
  - Password checks, signature verification depend on conditional branches
  - Preferred target for fault attacks



Introduction to Control-Flow Integrity (CFI)

- Different CFI granularities → Instruction granularity
- Program counter dependent state S_i
  - Depends on the previous state
  - Depends on currently executed instruction



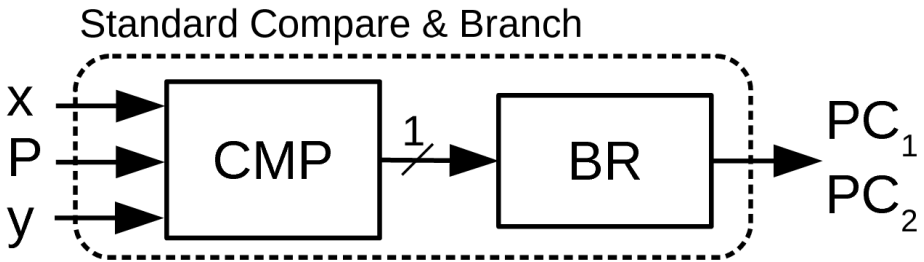
Conditional branches are not protected by means of CFI

A Primer to AN-Codes

- Arithmetic codes defined by: $x_c = A \cdot x$
- All code words are multiples of the encoding constant A
  - AN-code congruence: $0 \equiv x_c \ mod \ A$
- Support different arithmetic operations
  - +, -, *, /
- Closed under addition/subtraction
  - Adding two AN-code words results in another valid AN-code word
  - $z_c = x_c + y_c = A \cdot x + A \cdot y = A \cdot (x + y)$

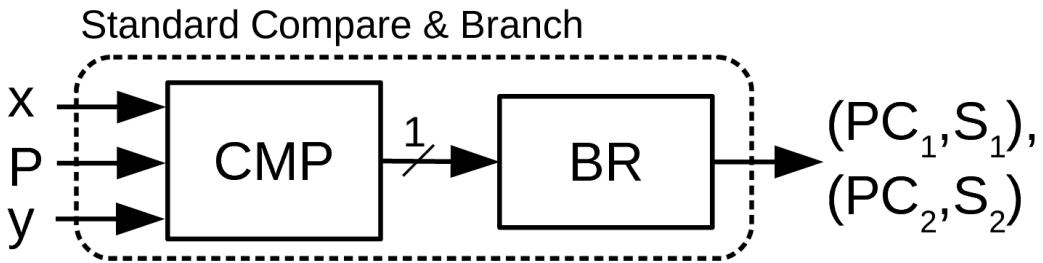
What is a Conditional Branch

1. First operation: Comparison
   - Takes two inputs x, y and comparison predicate P
   - Returns 1-bit signal if the comparison is true or false
2. Second operation: Branch
   - Determines how to update the program counter (PC_1,PC_2)



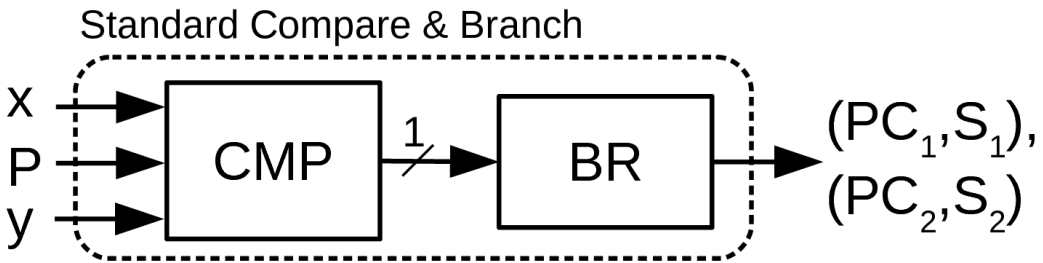
Conditional Branch with CFI

- CFI introduces a program counter dependent state S
- State is different if branch is taken or not
- Decision if the branch is taken still relies on a 1-bit signal



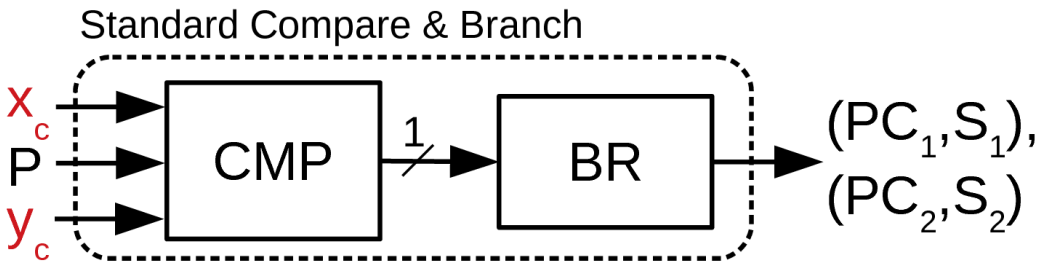
Generic Protected Conditional Branches

- Multiple attack vectors to bypass conditional branches
  1. Faulting the operands
  2. Faulting the comparison
  3. Faulting the branch



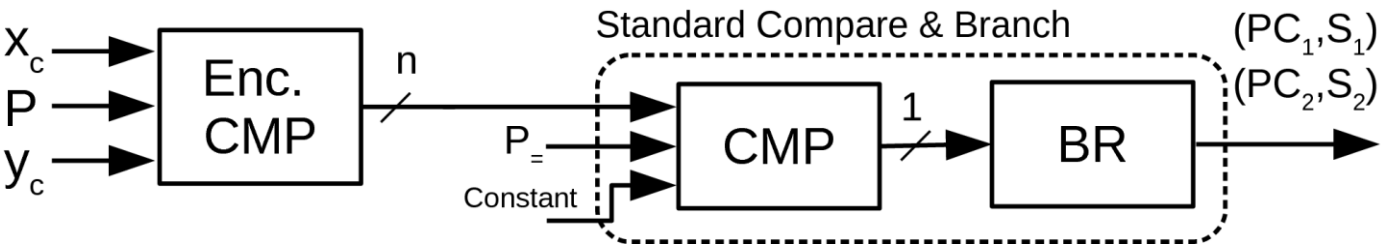
Generic Protected Conditional Branches

- Multiple attack vectors to bypass conditional branches
  1. Faulting the operands → Add redundancy to **x** and **y** (AN-codes)
  2. Faulting the comparison
  3. Faulting the branch



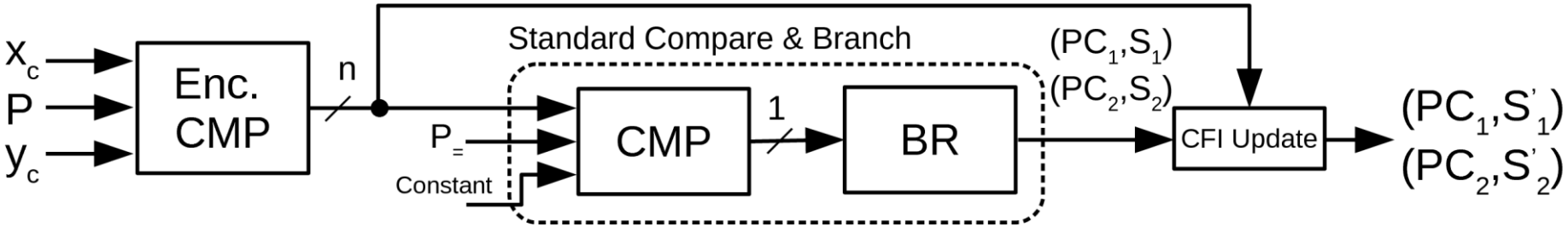
Generic Protected Conditional Branches

- Multiple attack vectors to bypass conditional branches
  1. Faulting the operands → Add redundancy to **x** and **y** (AN-codes)
  2. Faulting the comparison → **Encoded comparison** in software
  3. Faulting the branch



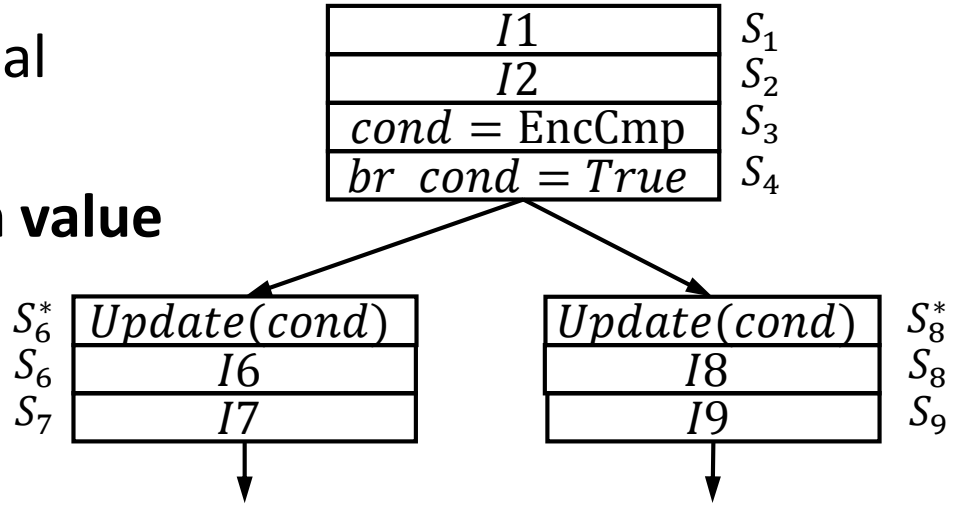
Generic Protected Conditional Branches

- Multiple attack vectors to bypass conditional branches
  1. Faulting the operands → Add redundancy to **x** and **y** (AN-codes)
  2. Faulting the comparison → **Encoded comparison** in software
  3. Faulting the branch → **Link** the redundant **condition value** with the CFI state



Example: Protected Conditional Branch

1. Compute the encoded compare
2. Perform a standard conditional branch
3. **Link** the redundant **condition value** with the CFI state



Wrong branch and wrong condition lead to invalid CFI state

Protected Comparisons with AN-Codes

- Problem:
  - $condition \leftarrow EncodedCompare(P, x_c, y_c)\ with$
    $condition\ \in \{C_1, C_2\}\ and$
    $Hamming\ Distance\ \geq D$
- Find an algorithm for all comparison predicates: $<, \leq, >, \geq, =, \neq$
- How to compute $x_c < y_c$?

Protected < Comparison with AN-Codes

- Step 1: Subtract $x_c - y_c$
  - $x_c - y_c \begin{cases} \text{positve if } x_c \geq y_c \\ \text{negative if } x_c < y_c \end{cases}$
  - Sign bit determines the comparison → **No redundancy**
  - Returns a valid AN-code word because AN-codes are closed under subtractions
  - AN-code congruence true
  - How to map the sign bit to a redundant condition value?

Protected < Comparison with AN-Codes

- Step 2: Condition mapping
  - $x_c - y_c \begin{cases} \text{positve if } x_c \geq y_c \\ \text{negative if } x_c < y_c \end{cases}$
- Map the difference to redundant condition values
- **Trick:** Cast difference to unsigned

Protected < Comparison with AN-Codes

- Step 2: Condition mapping
  - $x_c - y_c \begin{cases} \text{positve if } x_c \geq y_c \\ \end{cases}$
- AN-code congruence still true
- $0 \equiv (x_c - y_c)_u \ mod\ A$
- No change for positive differences due to the cast

Protected < Comparison with AN-Codes

- Step 2: Condition mapping
  - $x_c - y_c \begin{cases} \\ \text{negative if } x_c < y_c \end{cases}$
- $(x_c - y_c)_u = 2^{32} + (x_c - y_c) = 2^{32} + A \cdot (x - y)$
- AN-code congruence **not** true anymore
- $(x_c - y_c)_u \; mod \; A = (2^{32} + \underbrace{A \cdot (x - y)}_{\text{cancels out}}) \; mod \; A$

$$= 2^{32} \; mod \; A$$

Protected < Comparison with AN-Codes

- Condition mapping
  - $(x_c - y_c)_u \ \ mod\ A \begin{cases} 0 & \text{if } x_c \geq y_c \\ 2^{32}\ mod\ A & \text{if } x_c < y_c \end{cases}$
- To avoid a zero condition value add a constant C

Protected < Comparison with AN-Codes

- Condition mapping
  - $(x_c - y_c + \mathrm{C})_u \mod A \begin{cases} \mathrm{C} & \text{if } x_c \geq y_c \\ C + 2^{32} \mod A & \text{if } x_c < y_c \end{cases}$
- To avoid a zero condition value add a constant C
- Final algorithm:

Algorithm 1: AN-encoded < comparison.

Data: $x_c, y_c \in$ AN-code, $0 < C < A$.
Result: $cond \in \{C_1, C_2\}$.
begin
  diff $\longleftarrow$ (unsigned) $x_c - y_c + C$
  cond $\longleftarrow$ diff % A
end

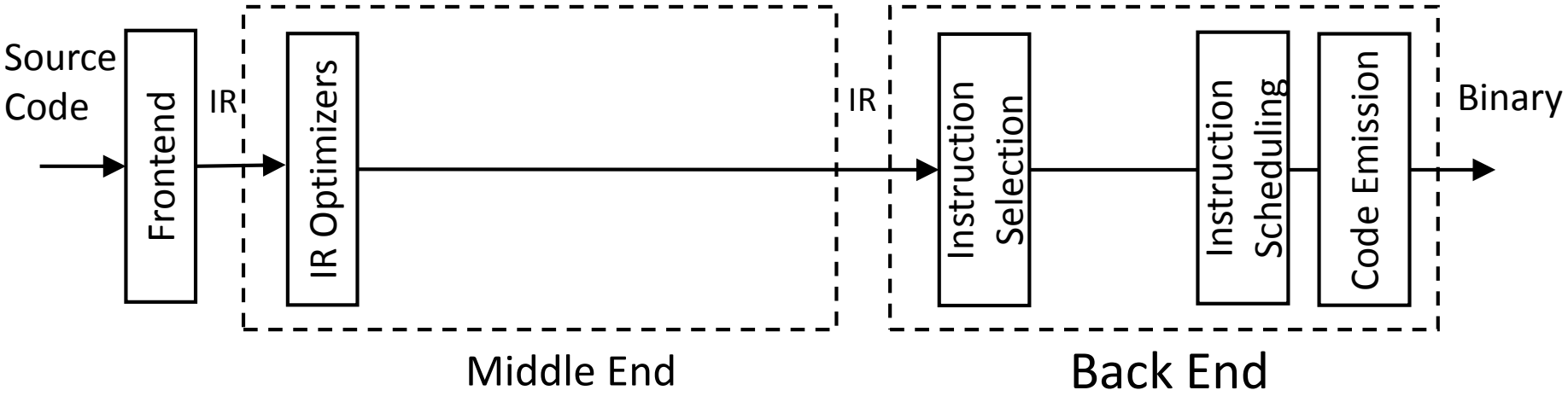
Protected Comparisons with AN-Codes

- Applicable to $\leq, >, \geq$ by
  - Swapping the operands of the first subtraction
  - Swapping the true and false constants
- $=/\neq$ equal comparison assembled using $\leq$ and $\geq$

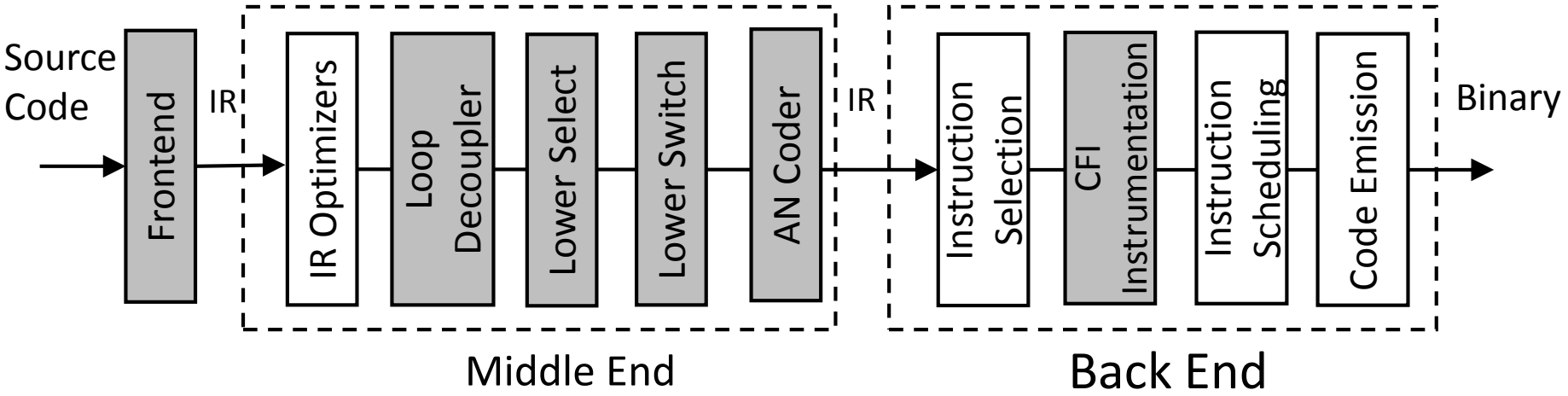
LLVM Compiler Prototype

- Annotate functions using attribute **`protect_branches`**
- Transformation operates on LLVM IR and is target independent
  1. Searches conditional branches
  2. Slice operands
  3. Transform all dependent operations into the AN-code domain
  4. Insert protected comparison algorithm
- Backend links comparison with CFI mechanism

LLVM Compiler Prototype

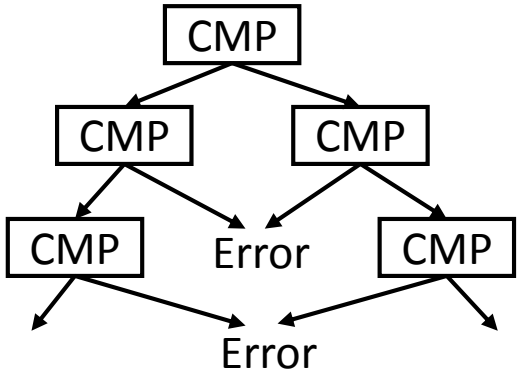


LLVM Compiler Prototype



Evaluation Setting

- ARMv7-M instruction set simulator
- Software-centered CFI scheme
  - State updates via store to the memory-mapped CFI unit
- AN-code with 6-bit Hamming distance
- Compare with duplication (5 times)
- Benchmarks: integer comparison, memcmp, bootloader



Evaluation

Benchmark	Metric	CFI abs	Duplication abs	+ / %	Prototype abs	+ / %
integer compare	Size / B Runtime / c	12 20	128 91	967 355	86 63	617 215
memcmp	Size / B Runtime / c	68 1689	272 10210	300 504	276 8905	306 427
bootloader[1]	Size / B Runtime / c	17252 51888k			17672 51888k	2.435 0.001

Evaluation

Benchmark	Metric	CFI abs	Duplication abs	Duplication + / %	Prototype abs	Prototype + / %
integer compare	Size / B Runtime / c	**12** **20**	128 91	967 355	86 63	617 215
memcmp	Size / B Runtime / c	68 1689	272 10210	300 504	276 8905	306 427
bootloader[1]	Size / B Runtime / c	17252 51888k			17672 51888k	2.435 0.001

Evaluation

Benchmark	Metric	CFI abs	Duplication abs	+ / %	Prototype abs	+ / %
integer compare	Size / B Runtime / c	12 20	**128** **91**	967 355	86 63	617 215
memcmp	Size / B Runtime / c	68 1689	272 10210	300 504	276 8905	306 427
bootloader[1]	Size / B Runtime / c	17252 51888k			17672 51888k	2.435 0.001

Evaluation

Benchmark	Metric	CFI abs	Duplication abs	+ / %	Prototype abs	+ / %
integer compare	Size / B Runtime / c	12 20	128 91	967 355	**86** **63**	617 215
memcmp	Size / B Runtime / c	68 1689	272 10210	300 504	276 8905	306 427
bootloader[1]	Size / B Runtime / c	17252 51888k			17672 51888k	2.435 0.001

Evaluation

Benchmark	Metric	CFI abs	Duplication abs	+ / %	Prototype abs	+ / %
integer compare	Size / B Runtime / c	12 20	128 91	967 355	86 63	617 215
memcmp	Size / B Runtime / c	68 1689	**272** **10210**	300 504	**276** **8905**	306 427
bootloader[1]	Size / B Runtime / c	17252 51888k			17672 51888k	2.435 0.001

Evaluation

Benchmark	Metric	CFI abs	Duplication abs	+ / %	Prototype abs	+ / %
integer compare	Size / B Runtime / c	12 20	128 91	967 355	86 63	617 215
memcmp	Size / B Runtime / c	68 1689	272 10210	300 504	276 8905	306 427
bootloader[1]	Size / B Runtime / c	17252 51888k			17672 51888k	2.435 0.001

[1]Only signature verification and all subsequent branches protected

Performance Improvements

- Better support for remainder operation
  - Remainder operation assembled using UDIV and MLS
  - Reduces code overhead up to 33% per comparison
- Better hardware support for CFI
  - No software-based CFI state manipulation
  - Combined instruction for compare, branch, and state update

Conclusion

- Close the gap between data protection and CFI by protecting conditional branches
- Generic approach: Link a redundant condition with the CFI state
- Exploit arithmetic properties of AN-codes to develop redundant comparison algorithms
- Prototype compiler based on LLVM

DATE 18
DESIGN, AUTOMATION & TEST IN EUROPE
19 – 23 March, 2018 · ICC · Dresden · Germany
The European Event for Electronic System Design & Test

Securing Conditional Branches in the Presence of Fault Attacks

Robert Schilling[1,2], Mario Werner[1], Stefan Mangard[1]

[1]Graz University of Technology, [2]Know-Center GmbH

March 22, 2018