

Exploiting RISC-V for Security Research

Stefan Mangard, Robert Schilling, Mario Werner

Graz University of Technology

January 23rd, 2019

Secure Systems Group



- About 10 PhD Students and postdocs
- Part of IAIK with about 60 researchers on IT security
- Research focus
 - **Secure Implementation of Cryptography**
 - Side channel attacks (HW/SW)
 - Formal methods for side channel countermeasures (power /fault)
 - **Secure Processor and System Architectures**
 - Side Channels (HW/SW)
 - HW and compiler extensions for security

RISC-V Activites

- Research mostly done based on
 - Cores from ETH
 - LLVM
- Currently, 3 PhD students, 1 master student

Attack Settings

- Software Attacks
 - Classical software attacks
 - Software side channel attacks
- Side Channel Attacks
 - Power and EM analysis
- Fault Attacks
 - Attacker is able to flip bits in the system (laser, glitches, rowhammer, ...)

Attack Settings

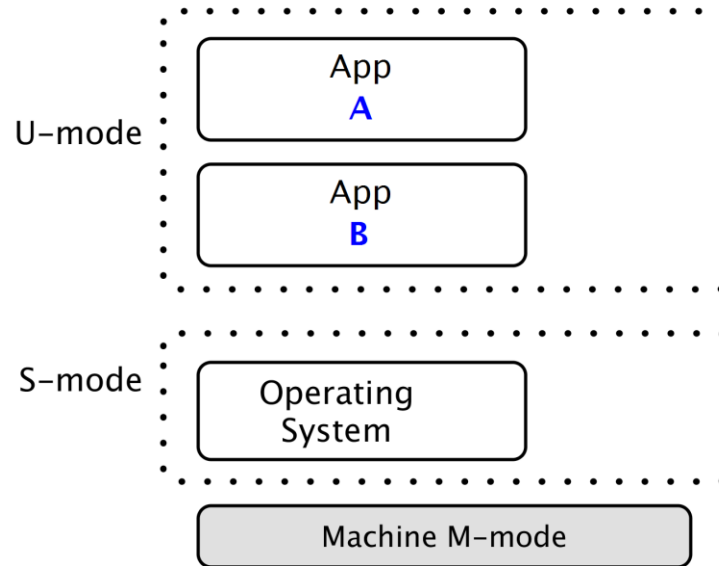
- Software Attacks
 - Timber V → Enclaves based on tagged memory
- Side Channel Attacks
 - Masked ALU → CPU protected against power analysis
- Fault Attacks
 - Control flow integrity
 - Branch decision integrity
 - Protected memory accesses

TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V [1]

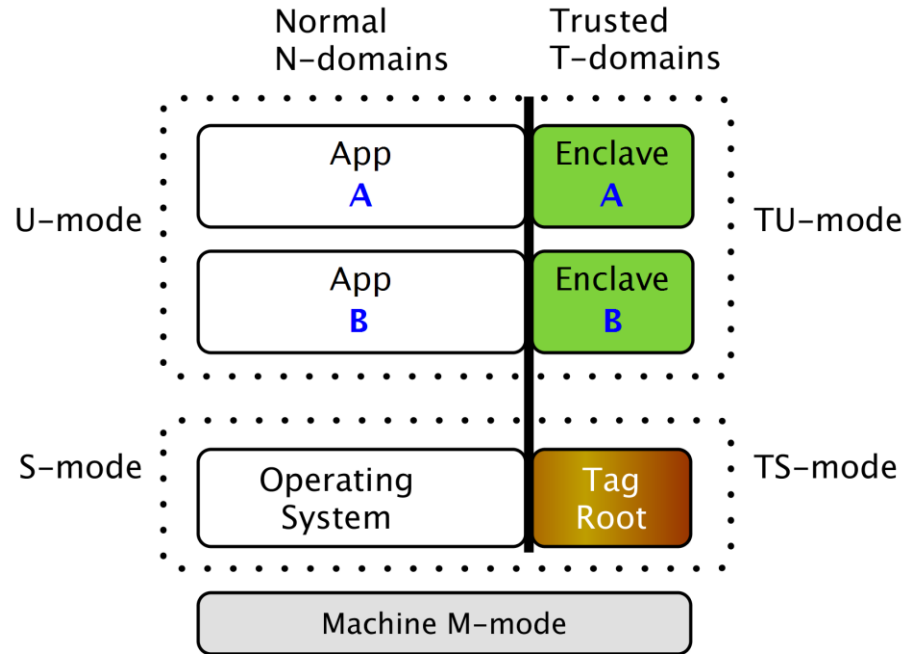
TIMBER-V Overview

- Provides lightweight trusted execution environments for embedded MPU-based systems
 - Tagged memory for fine grained in process isolation
 - MPU-based isolation between processes and the OS
- Low memory fragmentation (heap and stack interleaving)
- Shared memory between enclaves
- Remote attestation and sealing capabilities
- Prototype implementation with FreeRTOS and Spike

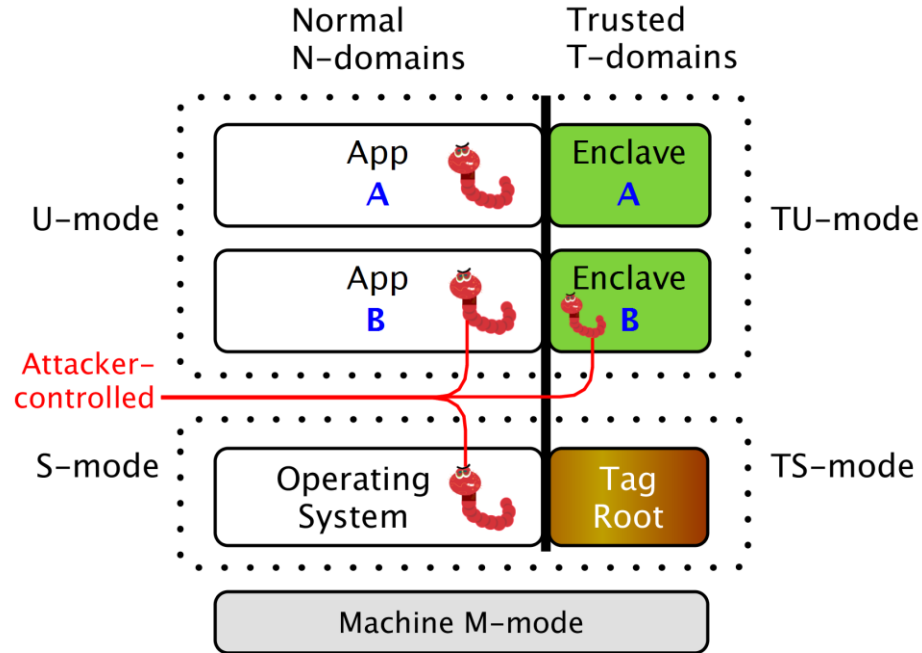
Security Domains



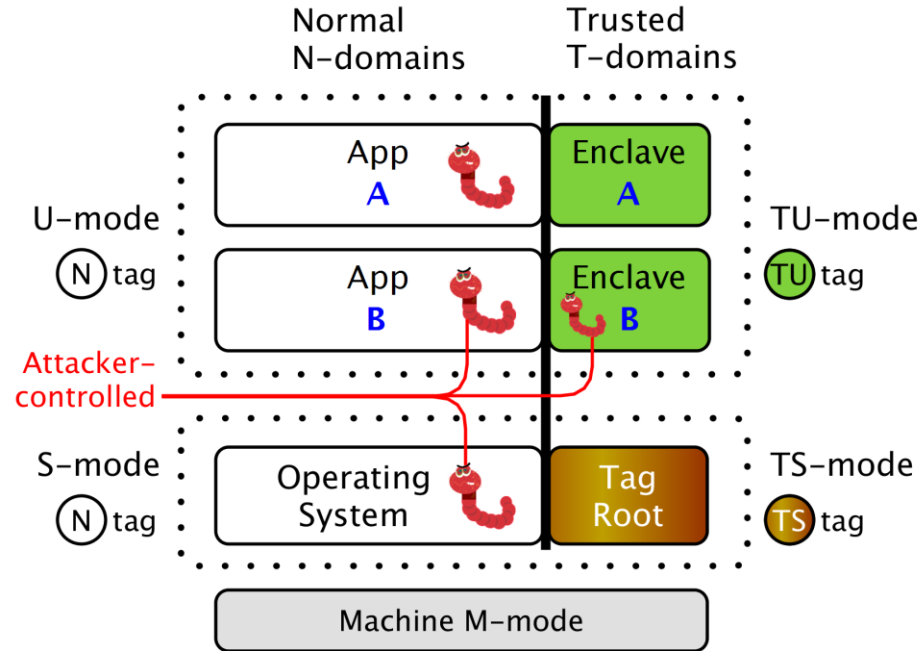
Security Domains



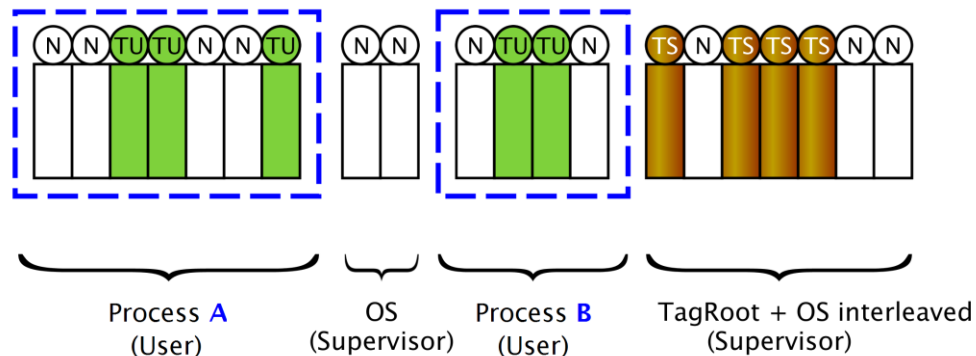
Security Domains



Security Domains

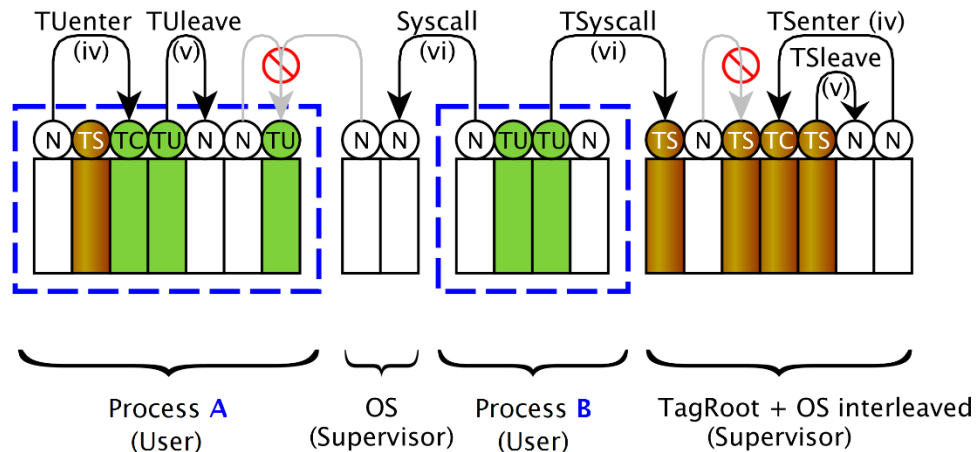


MPU + Tagged Memory Isolation



- MPU isolates between processes
- Tags isolate within a process

Trust Domain Transitions



- TC tag marks a call gates between trusted and normal domain
- Calls for horizontal transitions, syscalls for vertical transitions
- No diagonal security domain transitions

ISA Integration and Tag Policies

Checked Loads	Checked Stores
lbct etag, dst, src	sbct etag, ntag, src, dst
lbuct etag, dst, src	shct etag, ntag, src, dst
lhct etag, dst, src	swct etag, ntag, src, dst
lhuct etag, dst, src	Load Test Tag
lwct etag, dst, src	lth etag, dst, src

Access permitted	N-tag	TC-tag	TU-tag	TS-tag
N-domains	rwX	--e	---	---
TU-mode	rw1	r-x	rwX	---
TS-mode	rw1	rwX	rw-	rwX
M-mode	rwX	rwX	rwX	rwX

- Tags are encoded into the immediate of the I/S instruction format (reduces the offset to 10 or 8 bit respectively)
- CSR registers for TS interrupt/trap handling were added

Evaluation

- Simple code transformation on the assembler level
- OS based on RV32G port of FreeRTOS (~12500 LoC)
- Custom TS mode software (i.e., Tag Root implementation)
 - ~370 ASM LoC
 - ~1700 C LoC (HMAC + SHA256 ~ 300 LoC)
- CoreMark and programs from BEEBS as workload
- Overhead estimation via Spike and different models
 - between 2.6% and 25.2% average performance cost

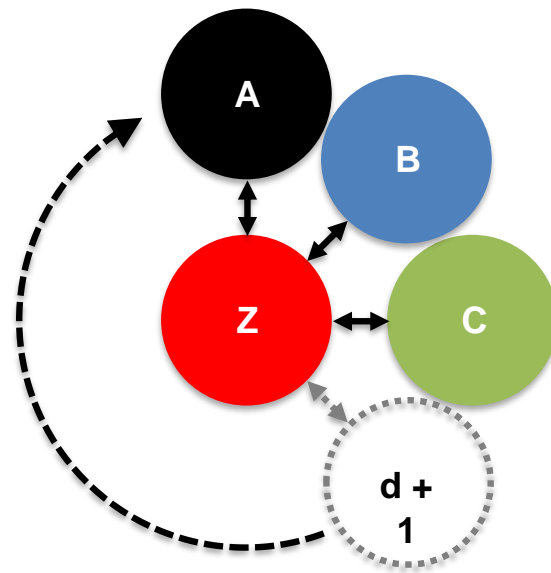
Side-channel Secure RISC-V [2]

Motivation

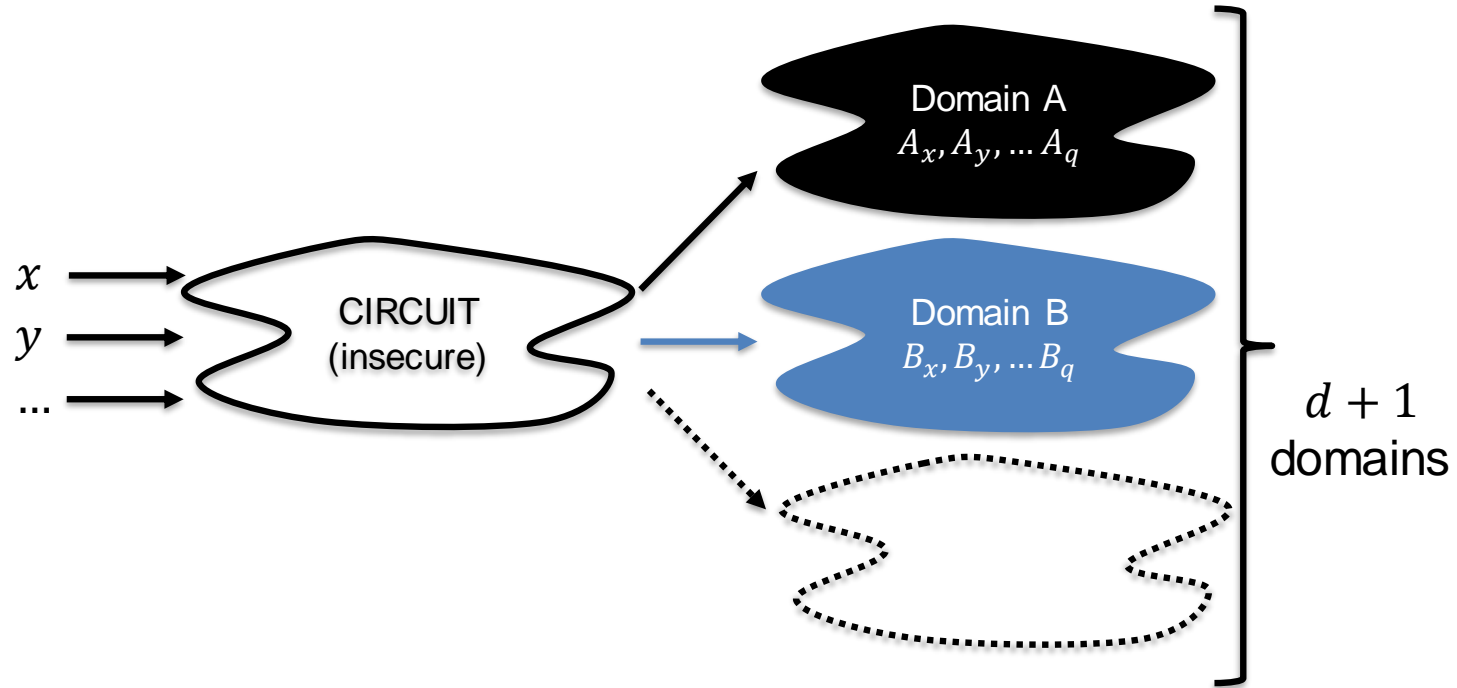
- SCA attacks pose a serious threat
- Research merely focuses on cryptographic algorithms
 - Dedicated hardware implementations
 - Software implementations
- No flexibility, long development time, ...
- **Solution:** Integrate SCA countermeasures directly into the processor

SCA Protected RISC-V Processor

- RISC-V processor
- +
- Domain-Oriented Masking
- =
- SCA protected RISC-V
 - Arbitrary protection level
 - Flexible and updateable
 - Transparent to software designers
 - Open source: https://github.com/hgrosz/vscale_dom

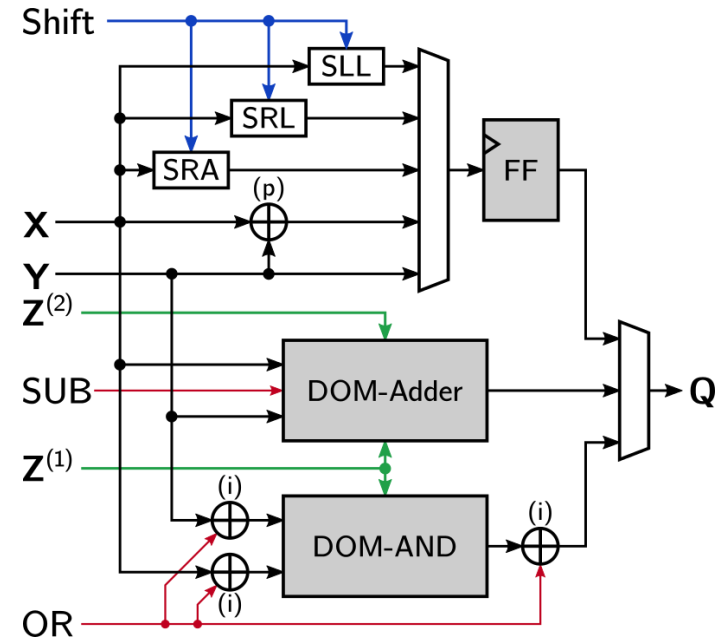


Domain-Oriented Masking (DOM)



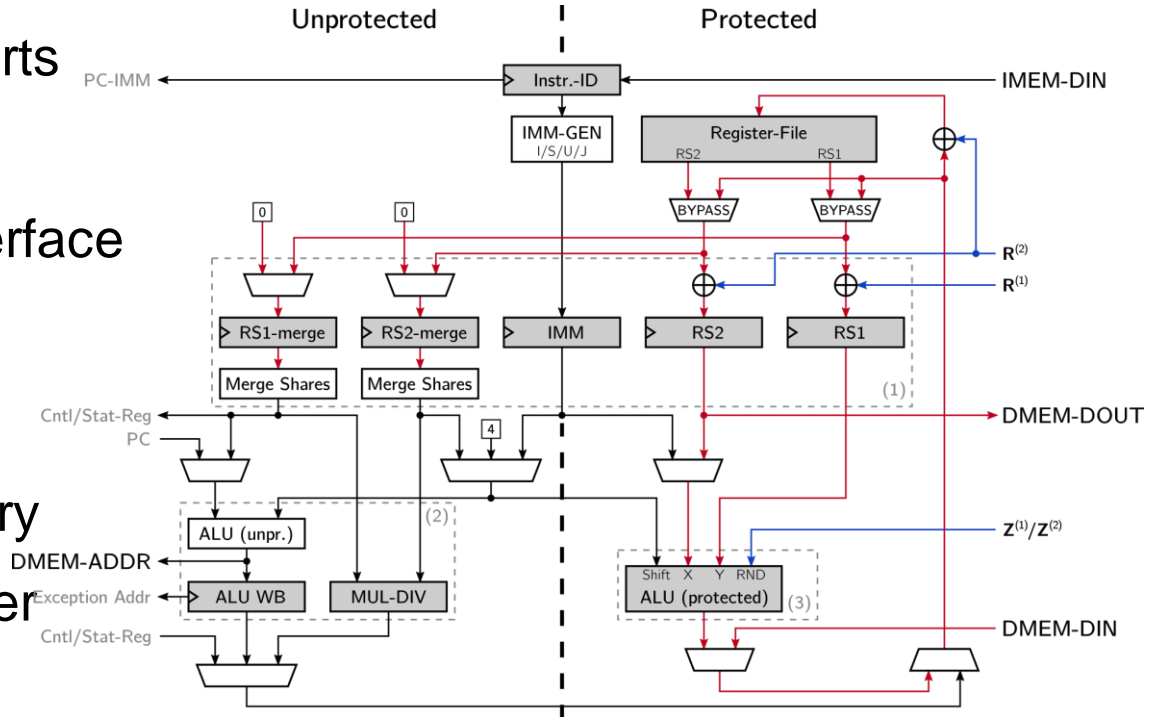
Protected ALU

- Linear functions
 - Shifts
 - XOR
- Nonlinear functions
 - AND (OR)
 - Adder
- Two fresh random Z's



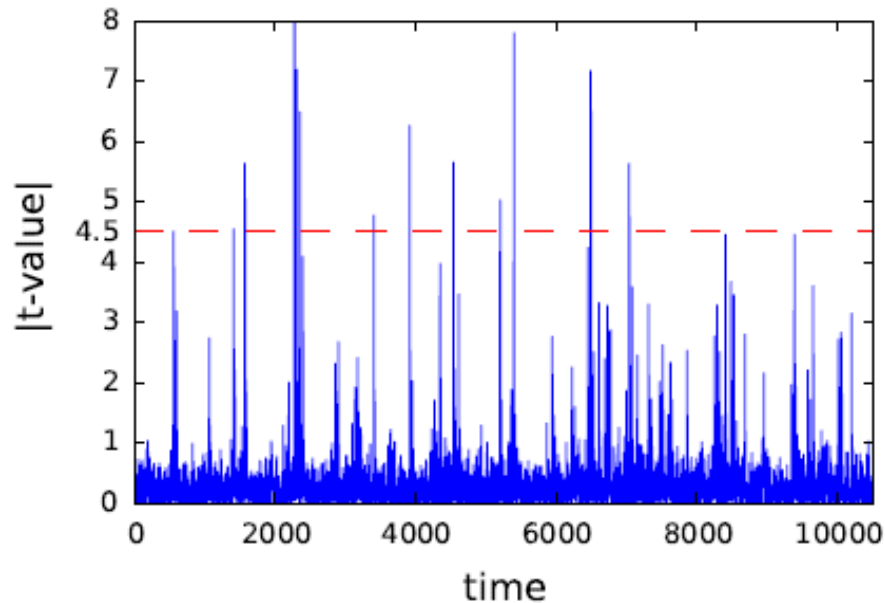
DOM Protected RISC-V Processor

- Protected (shared) parts
 - “I” instruction set
 - Data memory interface
 - Register file
- Unprotected parts
 - Instruction memory
 - Instruction decoder
 - Program counter



Evaluation

- Welsh's t-test for 2M traces and no protection (left



Conclusions

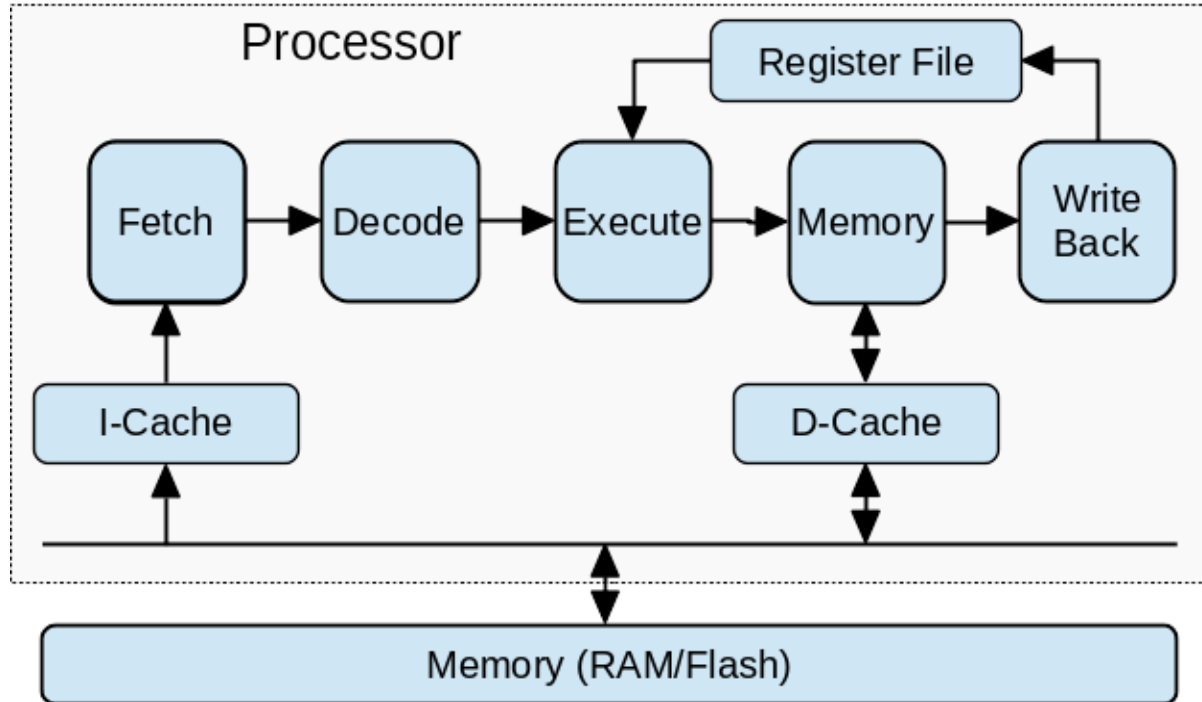
- SCA resistant RISC-V processor
- DOM for arbitrary protection level
- Advantages
 - More flexible
 - Transparent for software designers
 - Faster development of secure systems
 - Faster than software based masking

Sponge-Based Control-Flow Protection [3]

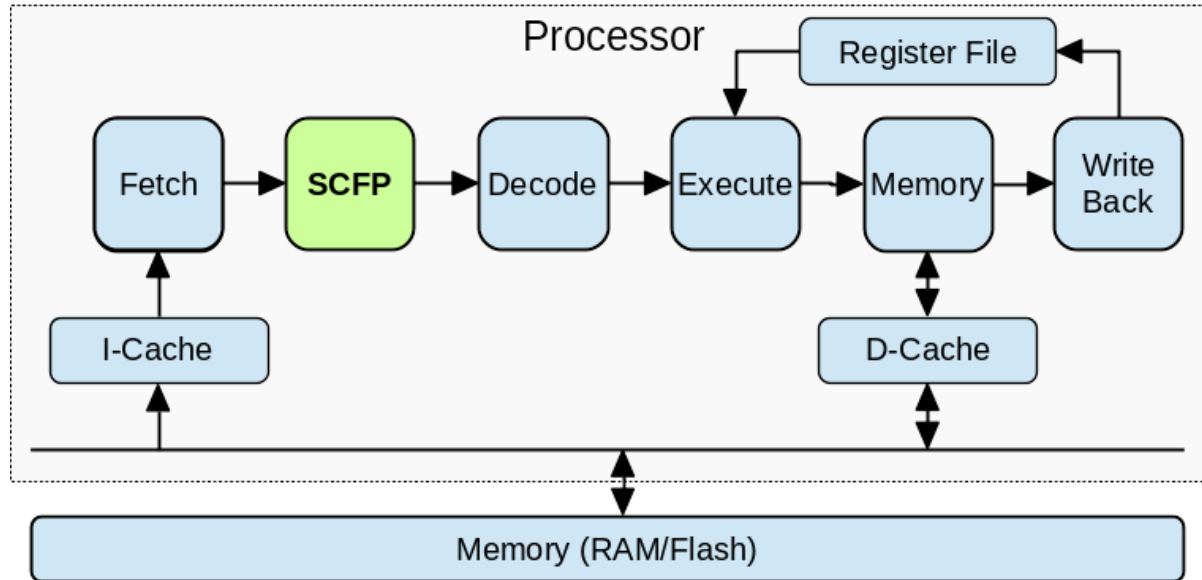
Overview

- Sponge-based Control-Flow Protection (SCFP)
 - Hardware supported CFI scheme
 - Encrypts the instruction stream with small granularity
 - Behaves like a context aware ISR
- Protects against logical and physical attacks
- Highly configurable in terms of security and cost
- RV32IM AEE-Light: ~10% runtime, ~20% code size

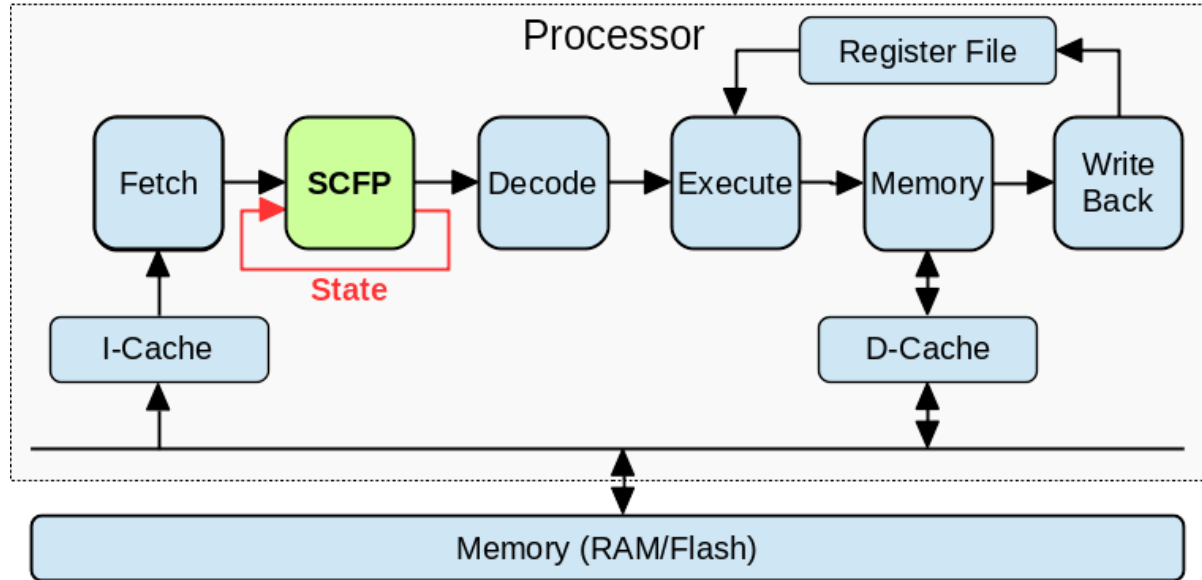
High Level Concept



High Level Concept



High Level Concept



Decryption/Execution Example

```
strcmp
```

```
      : ec d0 ee 97  
      : 28 ce 77 80  
      : 75 41 64 b1
```

```
      : 4b f4 51 75  
      : d9 a6 02 ad  
      : 51 7d 34 43
```

```
      : 4d 1b c0 0f  
      : a3 0f 21 3e
```

Decryption/Execution Example

```
strcmp  
0x1b2a0645
```

```
: ec d0 ee 97  
: 28 ce 77 80  
: 75 41 64 b1
```

```
: 4b f4 51 75  
: d9 a6 02 ad  
: 51 7d 34 43
```

```
: 4d 1b c0 0f  
: a3 0f 21 3e
```

Decryption/Execution Example

```
strcmp
0x1b2a0645
0xdd3fbcce : 03 06 05 00 : 1b a2, 0(a0)
               : 28 ce 77 80
               : 75 41 64 b1
```

```
               : 4b f4 51 75
               : d9 a6 02 ad
               : 51 7d 34 43
```

```
               : 4d 1b c0 0f
               : a3 0f 21 3e
```

Decryption/Execution Example

```

strcmp
0x1b2a0645
0xdd3fbcce : 03 06 05 00 : 1b a2, 0 (a0)
0xf5a92604 : 83 86 05 00 : 1b a3, 0 (a1)
              : 75 41 64 b1

```

```

              : 4b f4 51 75
              : d9 a6 02 ad
              : 51 7d 34 43

```

```

              : 4d 1b c0 0f
              : a3 0f 21 3e

```


Decryption/Execution Example

```
strcmp  
0x1b2a0645  
0xdd3fbcce : 03 06 05 00 : 1b a2, 0 (a0)  
0xf5a92604 : 83 86 05 00 : 1b a3, 0 (a1)  
0x58c04f0a : 5b 0c 06 00 : beqz a2, 24
```


↓

```
: 4b f4 51 75  
: d9 a6 02 ad  
: 51 7d 34 43
```

```
: 4d 1b c0 0f  
: a3 0f 21 3e
```


Decryption/Execution Example

```
strcmp
0x1b2a0645
0xdd3fbcce : 03 06 05 00 : 1b a2, 0 (a0)
0xf5a92604 : 83 86 05 00 : 1b a3, 0 (a1)
0x58c04f0a : 5b 0c 06 00 : beqz a2, 24
```



```
0x58c04f0a
      : 4b f4 51 75
      : d9 a6 02 ad
      : 51 7d 34 43
```

```
0x58c04f0a
      : 4d 1b c0 0f
      : a3 0f 21 3e
```



Decryption/Execution Example

```

strcmp
0x1b2a0645
0xdd3fbcce : 03 06 05 00 : lb a2, 0(a0)
0xf5a92604 : 83 86 05 00 : lb a3, 0(a1)
0x58c04f0a : 5b 0c 06 00 : beqz a2, 24

```

↓

```

0x58c04f0a
0xe70771a6 : 13 05 15 00 : addi a0, a0, 1
               : d9 a6 02 ad
               : 51 7d 34 43

```

```

0x58c04f0a
               : 4d 1b c0 0f
               : a3 0f 21 3e

```

Decryption/Execution Example

```

strcmp
0x1b2a0645
0xdd3fbcce : 03 06 05 00 : lb a2, 0(a0)
0xf5a92604 : 83 86 05 00 : lb a3, 0(a1)
0x58c04f0a : 5b 0c 06 00 : beqz a2, 24

```

```

0x58c04f0a
0xe70771a6 : 13 05 15 00 : addi a0, a0, 1
0x5b26165e : 93 85 15 00 : addi a1, a1, 1
           : 51 7d 34 43

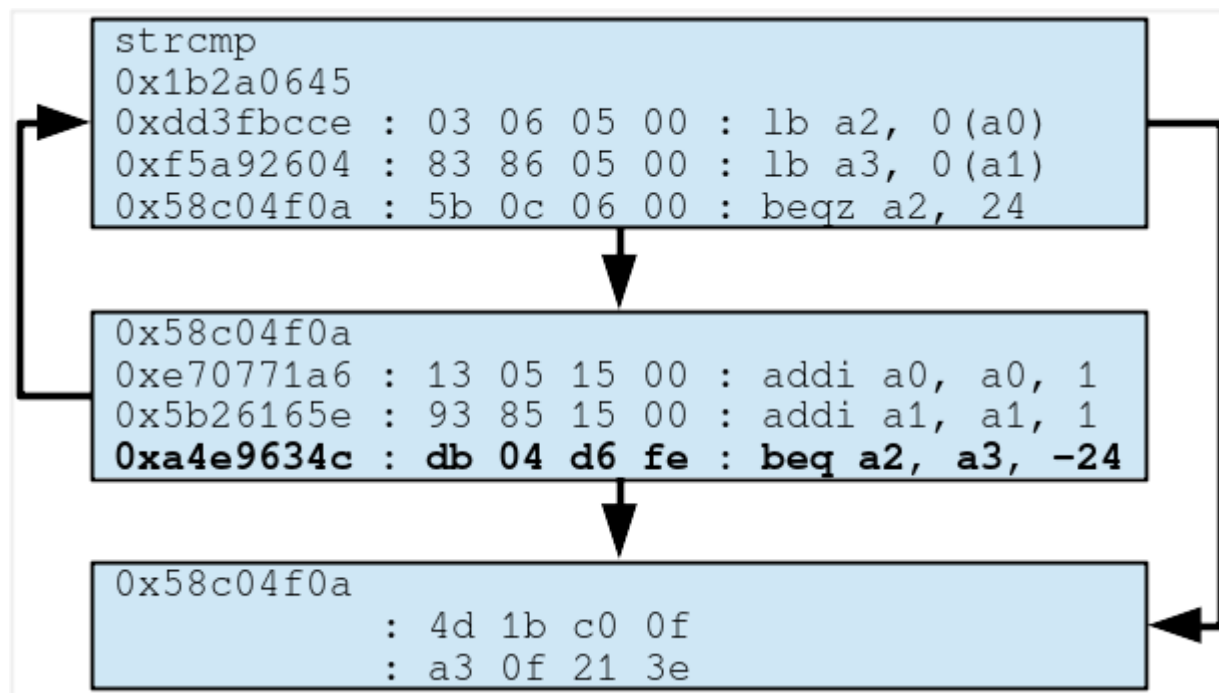
```

```

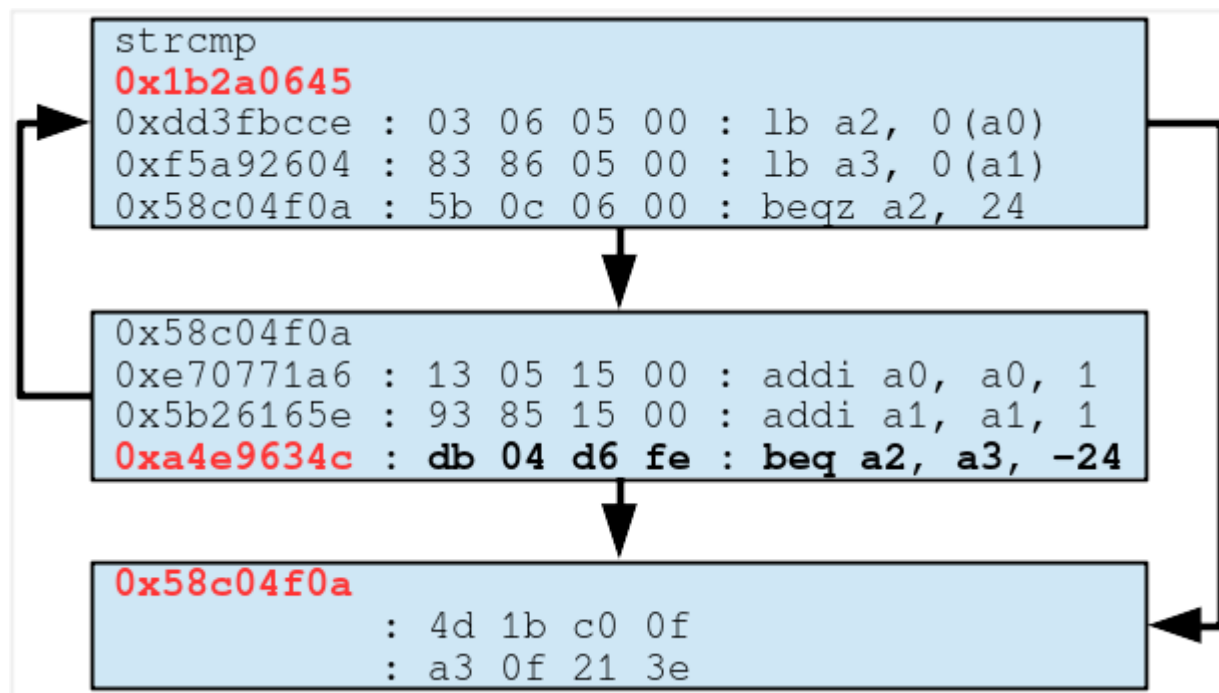
0x58c04f0a
           : 4d 1b c0 0f
           : a3 0f 21 3e

```

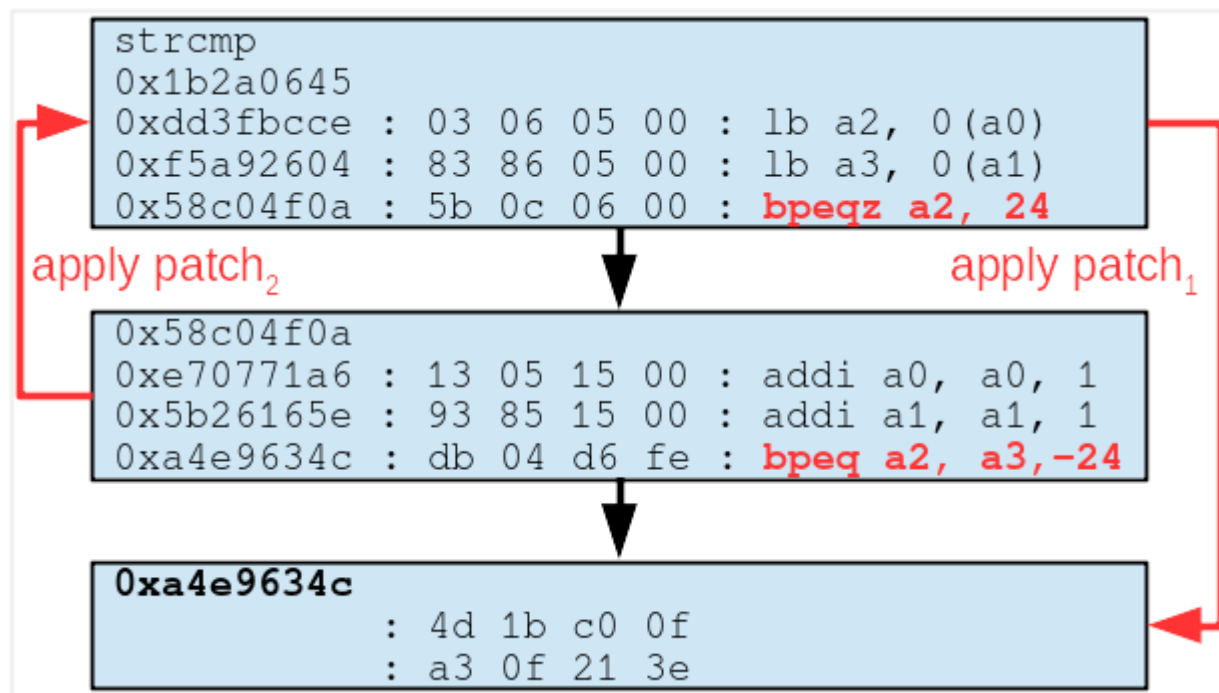
Decryption/Execution Example



Decryption/Execution Example



Decryption/Execution Example



RISC-V ISA Integration - Branches

- Branches additionally have an associated patch that is applied conditionally
- New BPEQ, BPNE, BPLT, BPLTU, BPGE, and BPGEU instructions

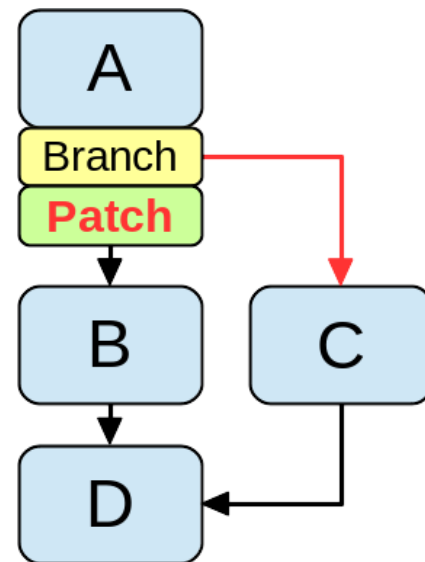
Listing 1 Pseudo code for the BP_{xxx} instructions.

Note: *SPC* denotes the SCFP state, *PC* the program counter

Note: *PatchValue* is located at *PC + 4*

```

1: if Reg[rs1] {=,≠,<,≥} Reg[rs2] then
2:   SPC ← SPC ⊕ PatchValue           // apply patch
3:   PC ← PC + signExtend(imm)        // perform branch
4: else
5:   PC ← PC + 8                      // fall-through but skip patch
  
```



RISC-V ISA Integration – Direct Calls

- Patch is applied when returning from the direct call
- Additional JALRP instruction for returns, JAL for calls

Listing 2 Pseudo code for the JALRP instruction.

Note: *SPC* denotes the SCFP state, *PC* the program counter

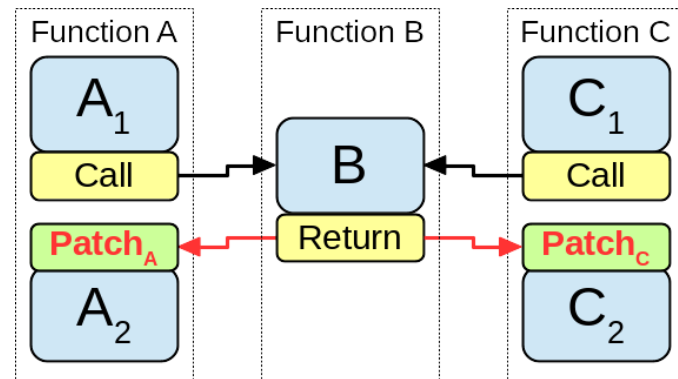
Note: *SrcPatch* is located at $PC + 4$

Note: *TargetPatch* is located at *AlignedTarget*

```

1:  $Target \leftarrow Reg[rs1] + signExtend(imm)$ 
2:  $AlignedTarget \leftarrow Target \& \sim 3$            // determine target
3:
4:
5:
6:
7:
8:
9:  $SPC \leftarrow SPC \oplus TargetPatch$            // apply patch
10:  $PC \leftarrow AlignedTarget + 4$              // perform jump

```



RISC-V ISA Integration – Indirect Calls

- Up to two patches need to be applied during call and return
- Direct to the same functions work too

Listing 2 Pseudo code for the JALRP instruction.

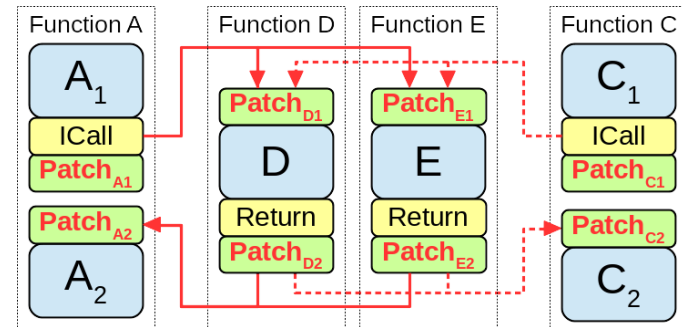
Note: *SPC* denotes the SCFP state, *PC* the program counter

Note: *SrcPatch* is located at $PC + 4$

Note: *TargetPatch* is located at *AlignedTarget*

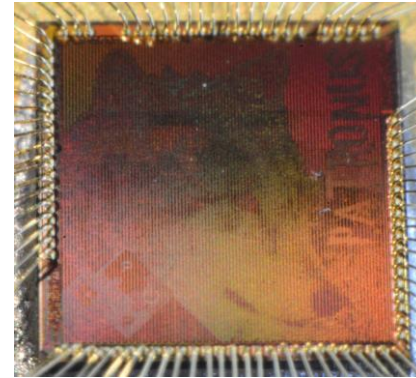
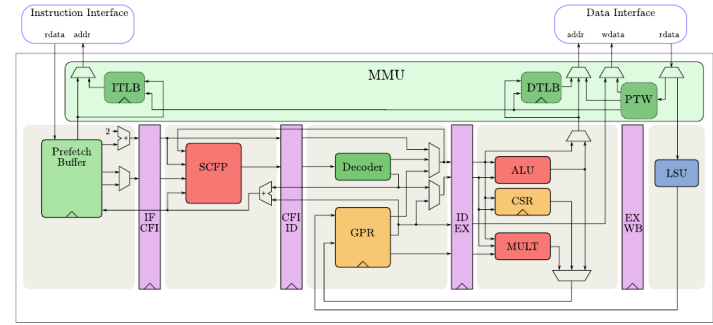
```

1:  $Target \leftarrow Reg[rs1] + signExtend(imm)$ 
2:  $AlignedTarget \leftarrow Target \& \sim 3$            // determine target
3: if  $Target \& 1$  then
4:    $SPC \leftarrow SPC \oplus SrcPatch$            // apply patch
5:    $SPC \leftarrow permute(SPC, AlignedTarget)$ 
6:    $Reg[rd] \leftarrow PC + 9$                  // set link reg.
7: else
8:    $Reg[rd] \leftarrow PC + 5$                  // set link reg.
9:    $SPC \leftarrow SPC \oplus TargetPatch$        // apply patch
10:  $PC \leftarrow AlignedTarget + 4$            // perform jump
  
```



Prototype Implementation

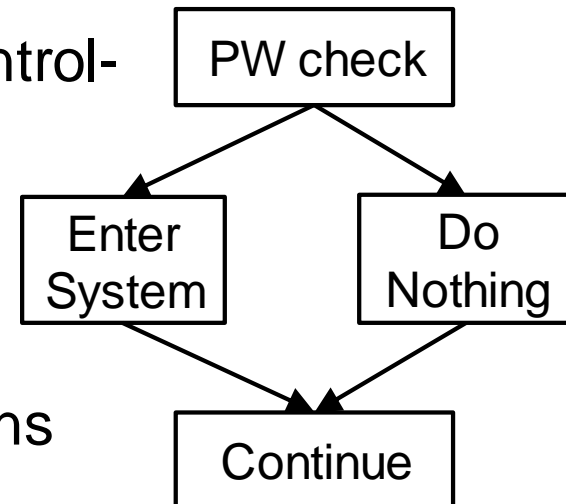
- LLVM-based toolchain
- RI5CY-based hardware
 - AEE-Light with PRINCE in APE-like mode
 - ~30kGE of area for SCFP at 100MHz in UMC65
 - ~10% runtime and ~20% code size overhead



Protected Conditional Branches [4]

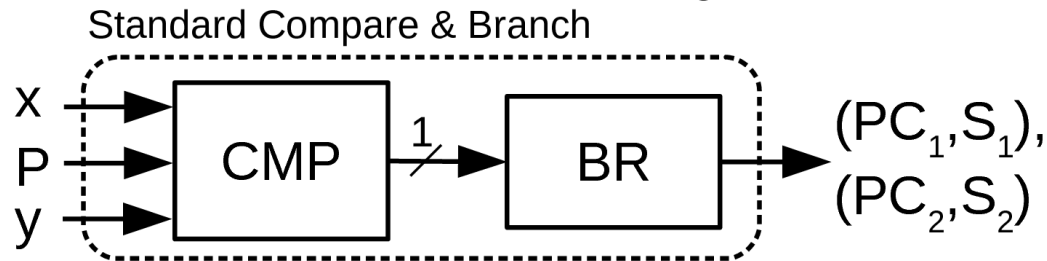
Motivation

- Fault attacks can modify the code and data
- Control-flow integrity (CFI) restricts the control-flow to valid execution traces
- Data encoding to protect data
- No protection for conditional branches
- Conditional branches are critical instructions
 - Password checks, signature verification depend on that



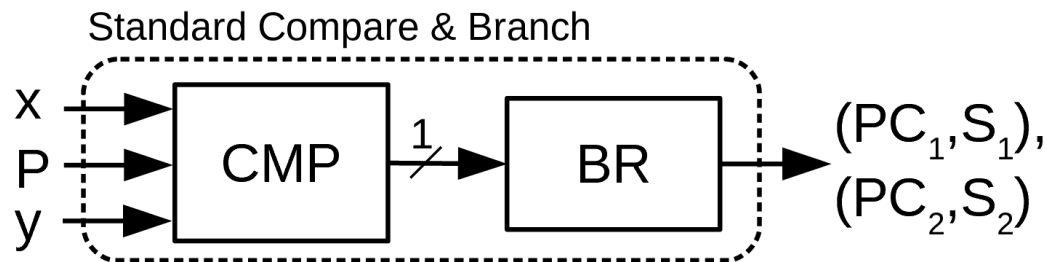
What is a Conditional Branch

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



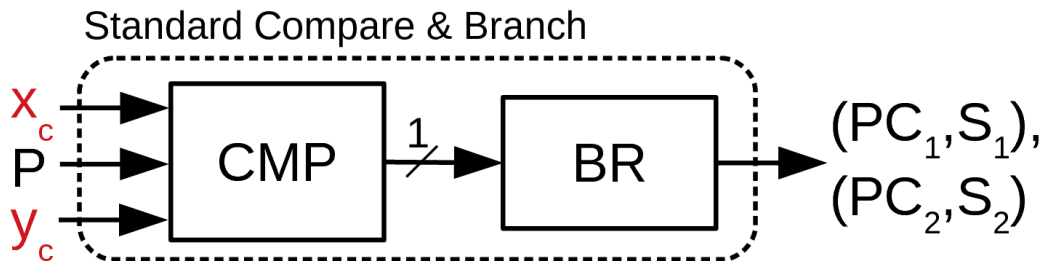
Generic Protected Conditional Branches

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



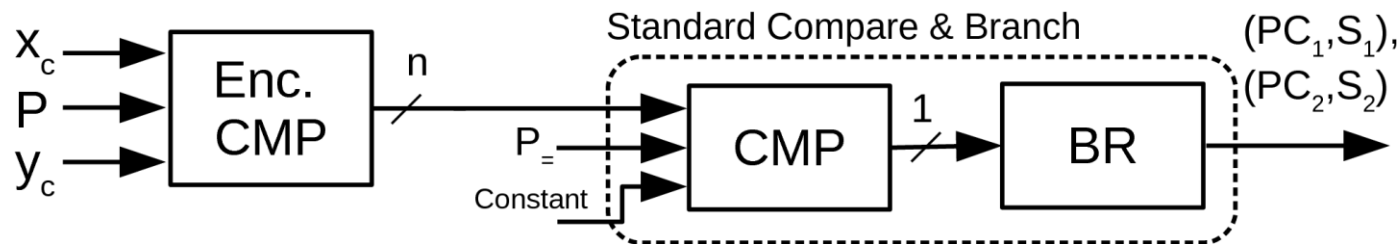
Generic Protected Conditional Branches

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



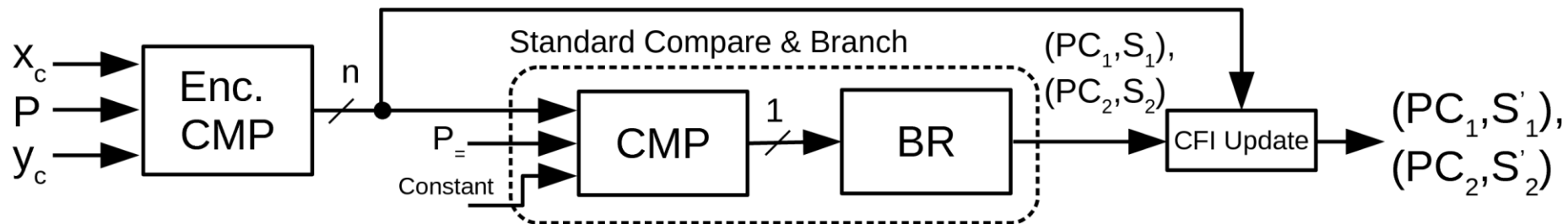
Generic Protected Conditional Branches

- Multiple attack vectors to bypass conditional branches
 - Faulting the operands \rightarrow Add redundancy to x and y (AN-codes)
 - Faulting the comparison \rightarrow **Encoded comparison** in software
 - Faulting the branch



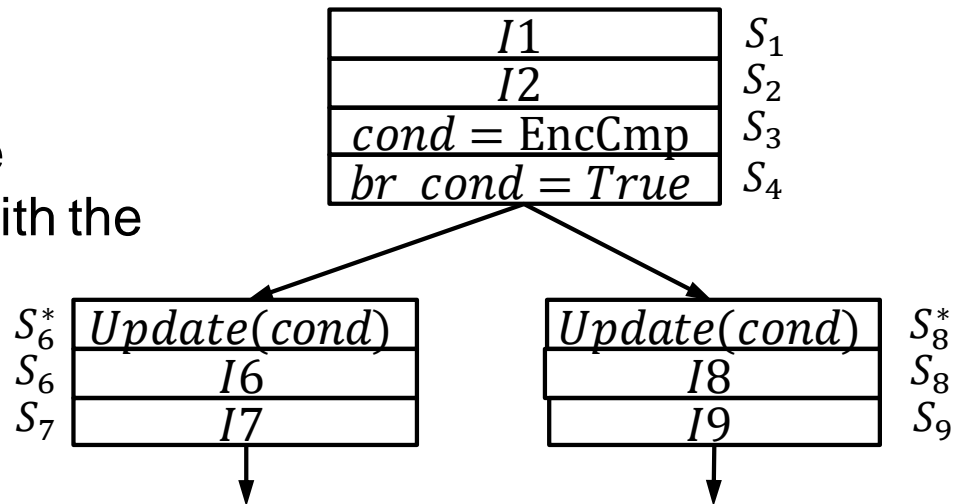
Generic Protected Conditional Branches

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



Example: Protected Conditional Branch

1. Compute the encoded comparison
2. Perform a conditional branch
3. At the branch target: **Link** the redundant **condition** value with the CFI state



Wrong branch and wrong condition lead to invalid CFI state

Encoded Comparison using AN-codes

- AN-codes natively support arithmetic operations
- Designed new comparison algorithms based on the arithmetic properties of AN-codes
- Return a syndrome $\{C_1, C_2\}$ with sufficiently large Hamming distance
- Available for all comparison predicates

Algorithm 1: AN-encoded $<$ comparison.

Data: $x_c, y_c \in \text{AN-code}$, $0 < C < A$.

Result: $cond \in \{C_1, C_2\}$.

begin

$diff \leftarrow (\text{unsigned}) x_c - y_c + C$
 $cond \leftarrow diff \% A$

end

Prototype Evaluation

- Added new branch instruction **bpdeq** to inject first operand to the CFI state
- LLVM-based toolchain
 - Automatically identifies conditional branches
 - Encodes dependent data-flow graph to AN-code domain
 - Inserts software-based comparison algorithm
- Overhead on par with state-of-the-art duplication approaches

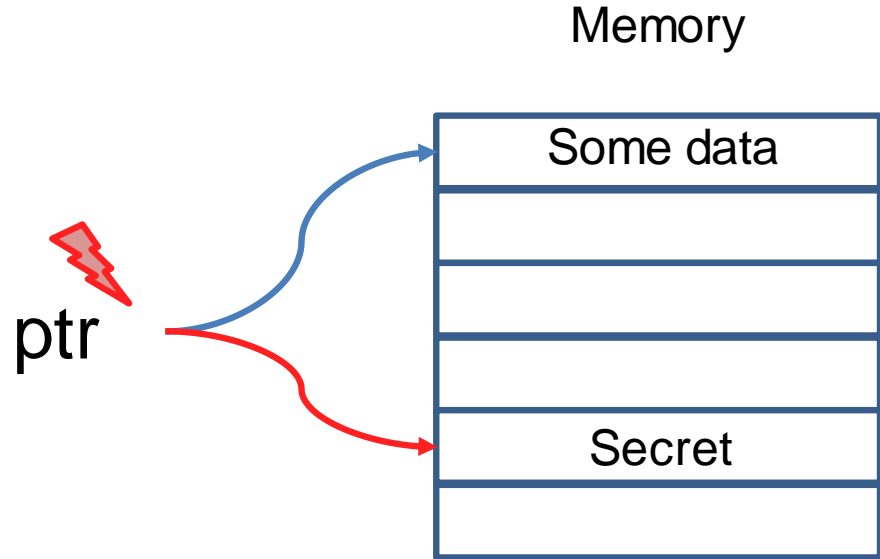
Secure Memory Accesses [5]

Motivation

- **No protection for memory accesses**
- Memory accesses are critical
 - There is a lot of critical information in the memory
 - **How** to ensure we read from the correct location?

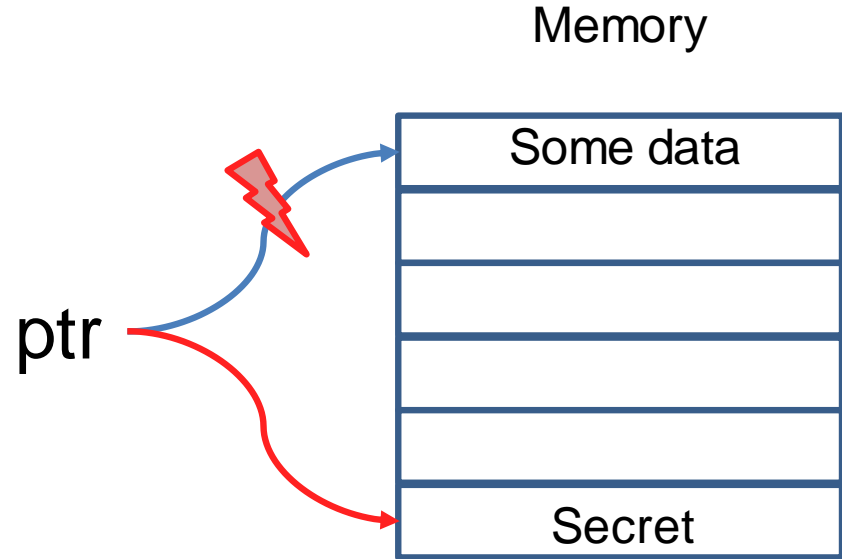
Attack Vector for Memory Accesses

- Faulted pointer redirects the memory access



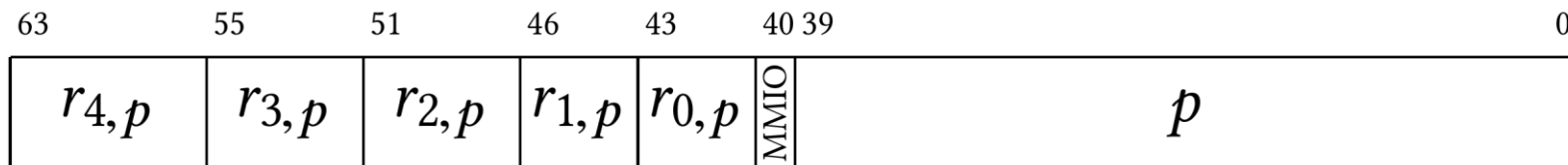
Attack Vector for Memory Accesses

- Faulted pointer redirects the memory access
- Faulting the memory access itself leads to a wrong access



Pointer Protection with Residue Codes

- Use multi-residue code to protect the pointer
 - Gives direct access to the functional value → no expensive decoding required
 - Supports pointer arithmetic
- Redundancy stored in the pointer



Secure Memory Accesses

- Pointers are protected but memory access still can be redirected
- Establish a link between the redundant address and redundant data
- Perform a linking overlay on top of encoded data
- Unlinking operation only successful when using the correct pointer and correct memory access
 - Translate addressing errors to data errors

Linking Approach

- Write memory in the form $mem[p] = l_p(D_{Reg})$
- Inverse to read data back $D_{Reg} = l_p^{-1}(mem[p])$
- Xor operation \rightarrow chosen for low-overhead
 - $mem[p] = p \oplus D_{Reg}, \quad D_{Reg} = p \oplus mem[p]$
 - **Problems** with granularity
- Use a byte-wise linking granularity to support arbitrary accesses

Prototype Evaluation

- FPGA prototype based PULP by ETH Zurich with 5% overhead
- ISA extension residue arithmetic and linked memory accesses
- Custom LLVM compiler prototype transforms all pointers
- Transformed all data pointers, protected all pointer arithmetic, replaced all memory accesses
- ~7% runtime and ~10% code size overhead

References

- [1] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. “TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V”. In: NDSS 2019.
- [2] Hannes Groß, Manuel Jelinek, Stefan Mangard, Thomas Unterluggauer, Mario Werner. “Concealing Secrets in Embedded Processors Designs”. In: CARDIS 2016
- [3] Mario Werner, Thomas Unterluggauer, David Schaffenrath, Stefan Mangard. “Sponge-Based Control-Flow Protection for IoT Devices”. In: Euro S&P 2018
- [4] Robert Schilling, Mario Werner, Stefan Mangard. “Securing Conditional Branches in the Presence of Fault Attacks”. In: DATE 2018
- [5] Robert Schilling, Mario Werner, Pascal Nasahl, Stefan Mangard. “Pointing in the Right Direction-Securing Memory Accesses in a Faulty World”. In: ACSAC 2018