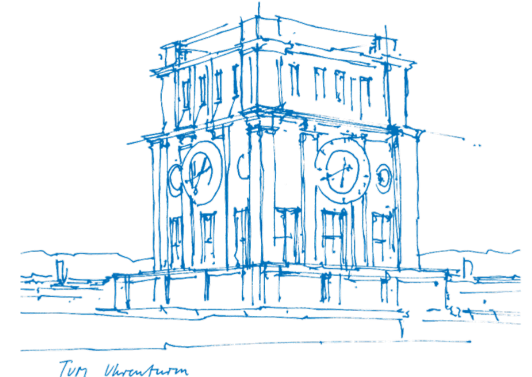
CompaSeC: A Compiler-assisted Security Countermeasure to Address Instruction Skip Fault Attacks on RISC-V

J. Geier[1], L. Auer[2], D. Mueller-Gritschneder[1], U. Sharif[1], and U. Schlichtmann[1]
[1]Technical University of Munich, Chair of Electronic Design Automation
[2]Fraunhofer Institute for Applied and Integrated Security (AISEC)

28th Asia and South Pacific Design Automation Conference
Tokyo, January 19, 2023

ASIA SOUTH PACIFIC DAC DESIGN AUTOMATION CONFERENCE



Outline

1. Motivation

2. Compiler-assisted Countermeasures Against Instruction Skip Fault Attacks

3. Evaluation and Performance Results

4. Conclusion

Outline

1. Motivation

2. Compiler-assisted Countermeasures Against Instruction Skip Fault Attacks

3. Evaluation and Performance Results

4. Conclusion

Software Implemented Hardware Fault Tolerance

Fault Model

Exploit		Bypass Secure Boot: Boot unverified software image
Manifestation	**Instruction Set Architecture**	Instruction corruption
		Architectural state corruption
		Memory op. corruption
	Imple-mentation	µArch register corruption
		Logic state corruption
Physical Attack		Optical/Laser Fault (LFI)
		Clock/Voltage Glitch

Software Implemented Hardware Fault Tolerance

Fault Model

Exploit

Bypass Secure Boot: Boot unverified software image

Manifestation

Instruction Set Architecture

Instruction corruption

Architectural state corruption

Memory op. corruption

Imple-mentation

µArch register corruption

Logic state corruption

Physical Attack

Optical/Laser Fault (LFI)

Clock/Voltage Glitch

Countermeasures

Software

Hardware

Software Implemented Hardware Fault Tolerance

Fault Model

Exploit		Bypass Secure Boot: Boot unverified software image
Manifestation	Instruction Set Architecture	Instruction corruption
		Architectural state corruption
		Memory op. corruption
	Imple-mentation	µArch register corruption
		Logic state corruption
Physical Attack		Optical/Laser Fault (LFI)
		Clock/Voltage Glitch

Countermeasures

Software

- less fault coverage
- software performance overhead

\+ common off-the-shelf components

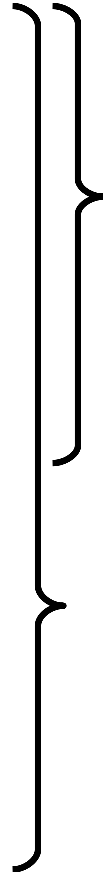
\+ deploy with software

Hardware

Software Implemented Hardware Fault Tolerance

Fault Model

Exploit		Bypass Secure Boot: Boot unverified software image
Manifestation	**Instruction Set Architecture**	Instruction corruption
		Architectural state corruption
		Memory op. corruption
	Imple-mentation	µArch register corruption
		Logic state corruption
Physical Attack		Optical/Laser Fault (LFI)
		Clock/Voltage Glitch



Countermeasures

Software

- less fault coverage
- software performance overhead
+ common off-the-shelf components
+ deploy with software

Hardware

+ (usually) better coverage
+ (usually) less software overhead
- additional hardware
- modify after shipping?

Software Implemented Hardware Fault Tolerance

Fault Model

Exploit		Bypass Secure Boot: Boot unverified software image
Manifestation	**Instruction Set Architecture**	Instruction corruption
		Architectural state corruption
		Memory op. corruption
	Imple-mentation	µArch register corruption
		Logic state corruption
Physical Attack		Optical/Laser Fault (LFI)
		Clock/Voltage Glitch

Countermeasures

Software

- \- less fault coverage
- \- software performance overhead
- \+ common off-the-shelf components
- \+ deploy with software

Hardware

- \+ (usually) better coverage
- \+ (usually) less software overhead
- \- additional hardware
- \- modify after shipping?

Software Implemented Hardware Fault Tolerance

Countermeasures

Software

- \- less fault coverage
- \- software performance overhead
- \+ common off-the-shelf components
- \+ deploy with software

Software Implemented Hardware Fault Tolerance

Countermeasures

Software

- \- less fault coverage
- \- software performance overhead
- \+ common off-the-shelf components
- \+ deploy with software

Source Level

- Target (ISA) independent
- Algorithm dependent

Example:

- Algorithm re-execution

Software Implemented Hardware Fault Tolerance

Countermeasures

Software

- \- less fault coverage
- \- software performance overhead
- \+ common off-the-shelf components
- \+ deploy with software

Source Level

- Target (ISA) independent
- Algorithm dependent

Example:

- Algorithm re-execution

Instruction-level

- Target dependent, e.g. RISC-V
- Algorithm independent

Examples:

- Instruction re-execution
- Dual module redundancy (DMR)
- Runtime signature monitoring (RSM)

Outline

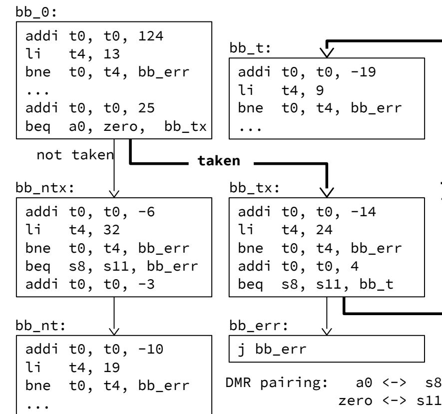
1. Motivation
2. Compiler-assisted Countermeasures Against Instruction Skip Fault Attacks
3. Evaluation and Performance Results
4. Conclusion

CompaSeC Framework

- Source code agnostic

```
if(a == 0) {
    ... // taken
} else {
    ... // not taken
}
```

Source Code Project (.h/.c/.cpp)



hardened object code (.asm)

CompaSeC Framework

- Source code agnostic
- LLVM-based

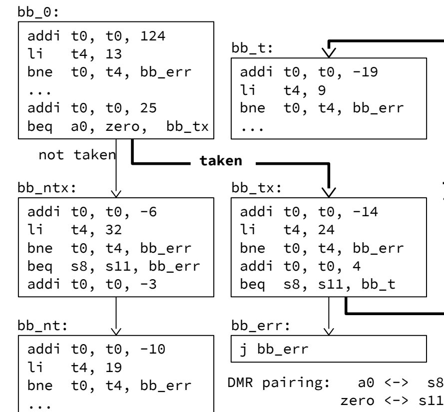
```
if(a == 0) {
    ... // taken
} else {
    ... // not taken
}
```

Source Code Project (.h/.c/.cpp)

LLVM Compiler Frontend

LLVM IR

hardened object code (.asm)



CompaSeC Framework

- Source code agnostic
- LLVM-based
- At Backend (=Machine) Level

```
if(a == 0) {
    ... // taken
} else {
    ... // not taken
}
```

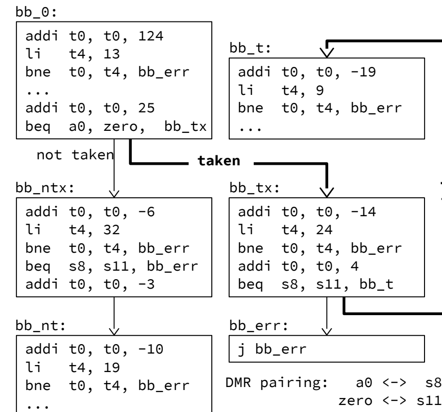
Source Code Project (.h/.c/.cpp)

LLVM Compiler Frontend

LLVM IR

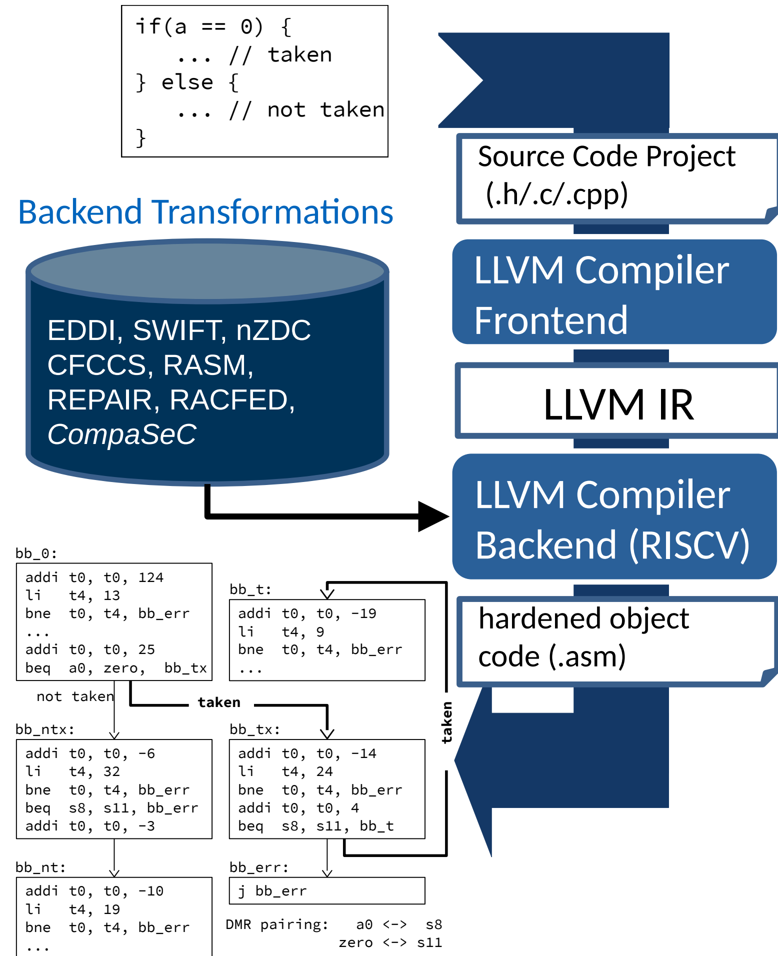
LLVM Compiler Backend (RISCV)

hardened object code (.asm)



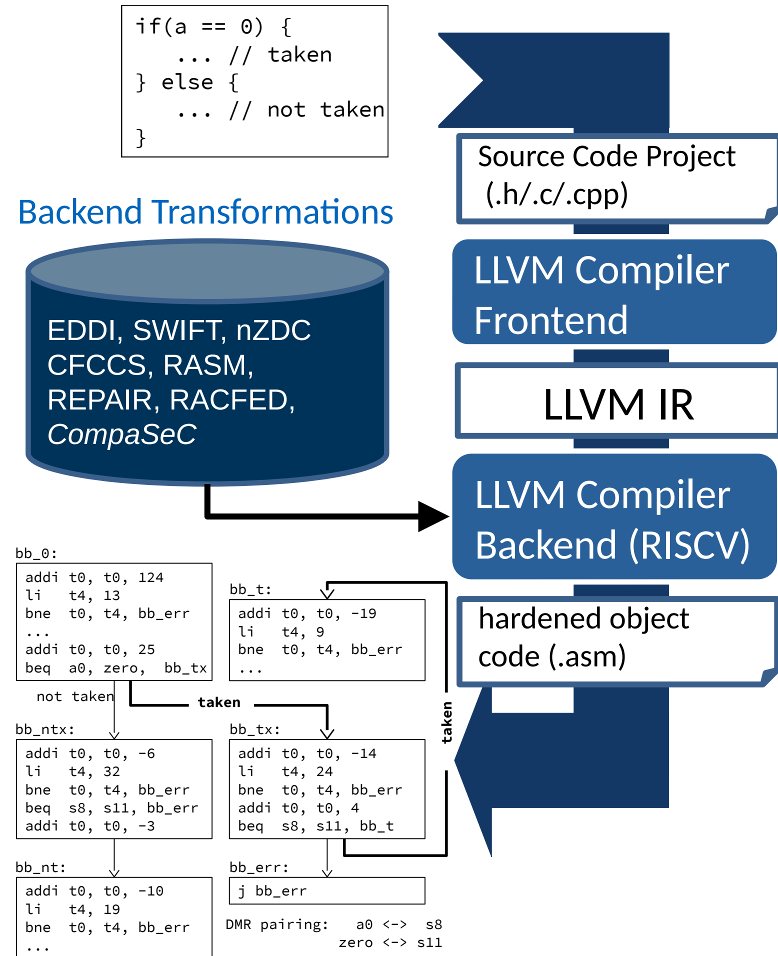
CompaSeC Framework

- Source code agnostic
- LLVM-based
- At Backend (=Machine) Level
- **Existing Transformations** (adapted for RISC-V):
  CFCSS [1], *nZDC* [2], *NEMESIS* [3], *SWIFT* [4], *EDDI* [5], *RASM* [6], *RACFED* [7], *REPAIR* [8]



CompaSeC Framework

- Source code agnostic
- LLVM-based
- At Backend (=Machine) Level
- **Existing Transformations** (adapted for RISC-V):
  CFCSS [1], *nZDC* [2], *NEMESIS* [3], *SWIFT* [4], *EDDI* [5], *RASM* [6], *RACFED* [7], *REPAIR* [8]
- **CompaSeC**: **Combination** of existing methods to eliminate Instruction Skip Faults in RISC-V



Contributions

CompaSeC Transformation

Contributions

CompaSeC Transformation

1. **Combined Transformation**: *”use what works best”*

Contributions

CompaSeC Transformation

1. **Combined Transformation**: *”use what works best”*
   - **First,** Dual Module Redundancy (DMR)

Contributions

CompaSeC Transformation

1. **Combined Transformation**: *”use what works best”*
   - **First,** Dual Module Redundancy (DMR)
   - **Then,** Runtime Signature Monitoring (RSM) over original and duplicated code

Contributions

CompaSeC Transformation

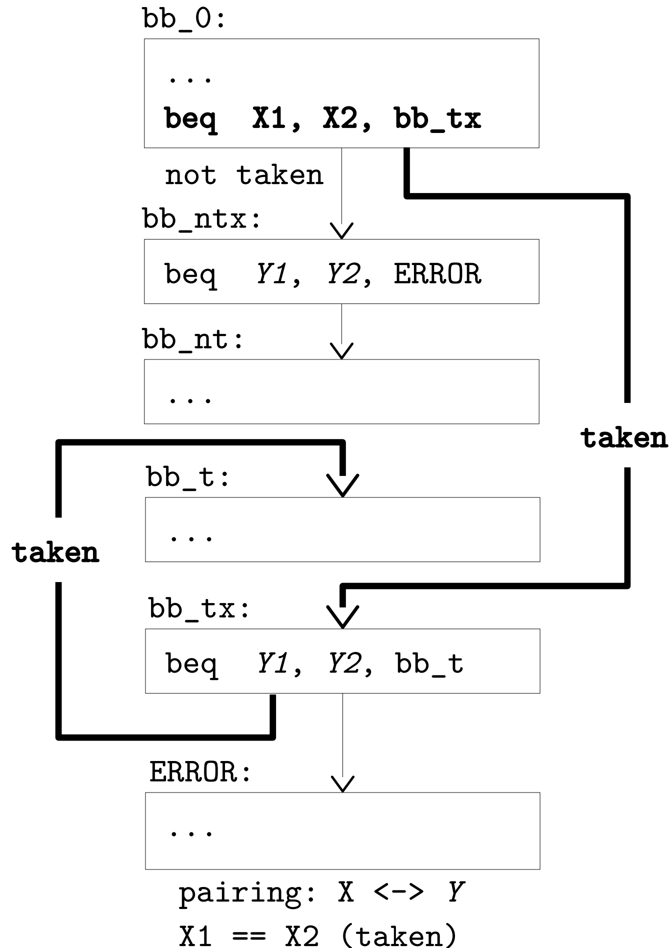
1. **Combined Transformation**: *”use what works best”*
   - **First,** Dual Module Redundancy (DMR)
   - **Then,** Runtime Signature Monitoring (RSM) over original and duplicated code
2. **Selective Hardening**: *”only, where necessary”*
   - DMR breaks Instruction Set Architecture (ISA) calling convention
   - Automated DMR domain crossing

Contributions

CompaSeC Transformation

1. **Combined Transformation**: *”use what works best”*
   - **First,** Dual Module Redundancy (DMR)
   - **Then,** Runtime Signature Monitoring (RSM) over original and duplicated code
2. **Selective Hardening**: *”only, where necessary”*
   - DMR breaks Instruction Set Architecture (ISA) calling convention
   - Automated DMR domain crossing
3. **Verification** in Secure Boot Scenario

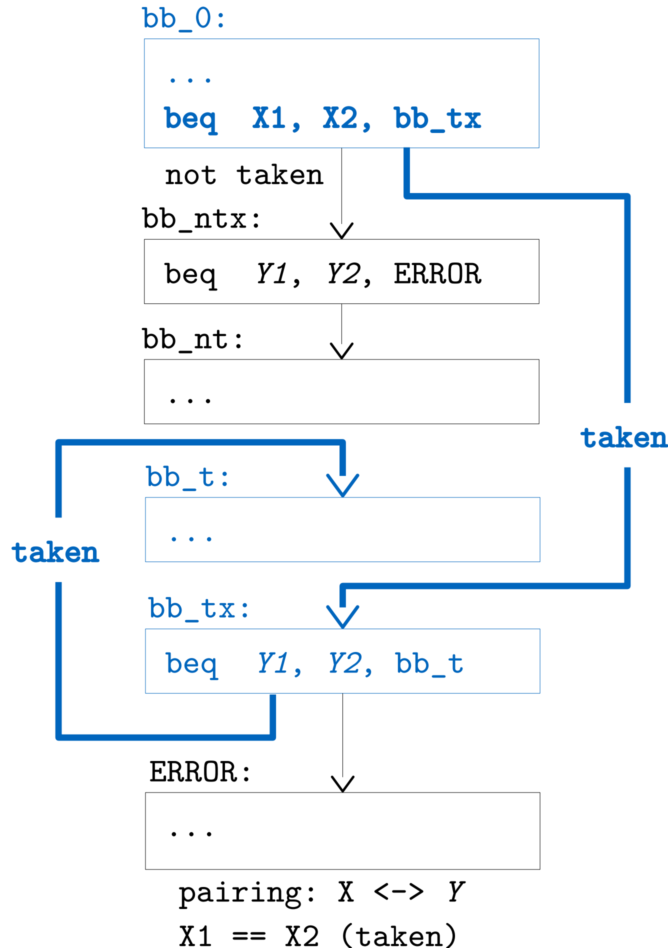
Dual Module Redundancy (DMR) Transformation



(DMR): nZDC+NEMESIS [2, 3]

1. Reserve half of ISA register file
   - Primary X
   - Shadow Y
2. Duplicate operations on 2nd half
   - Balance Checks X<->Y

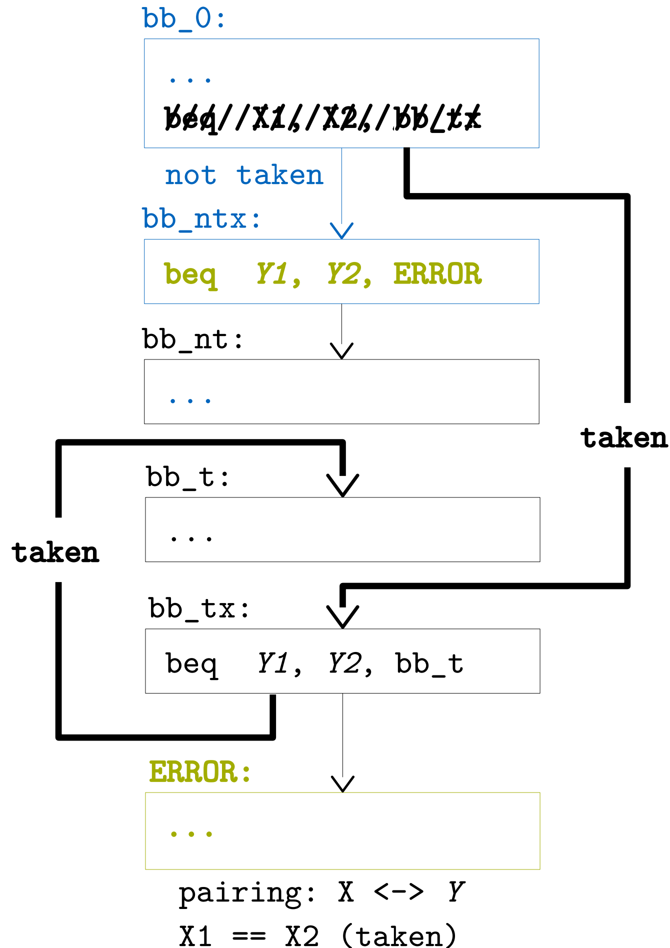
Dual Module Redundancy (DMR) Transformation



(DMR): nZDC+NEMESIS [2, 3]

1. Reserve half of ISA register file
   - Primary X
   - Shadow Y
2. Duplicate operations on 2nd half
   - Balance Checks X<->Y

Dual Module Redundancy (DMR) Transformation

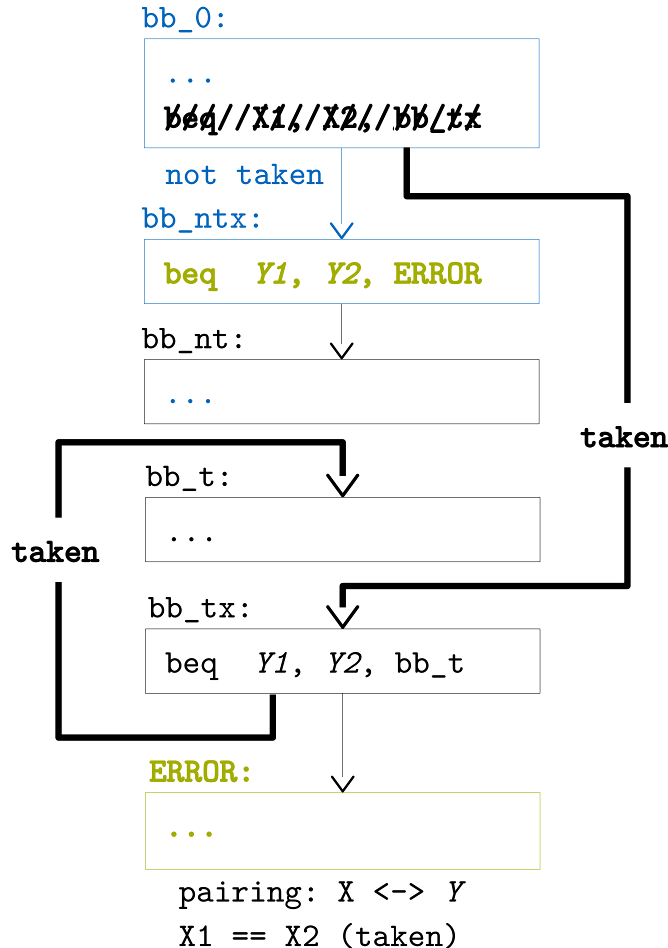


(DMR): nZDC+NEMESIS [2, 3]

1. Reserve half of ISA register file
   - Primary X
   - Shadow Y
2. Duplicate operations on 2nd half
   - Balance Checks X<->Y

- Single Instruction Skips

Dual Module Redundancy (DMR) Transformation

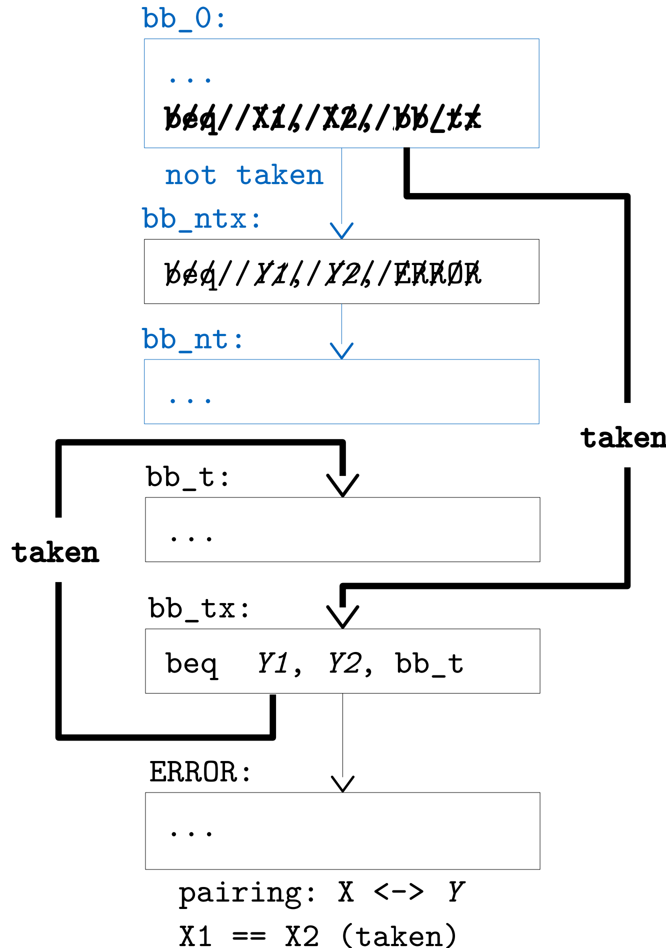


(DMR): nZDC+NEMESIS [2, 3]

1. Reserve half of ISA register file
   - Primary X
   - Shadow Y
2. Duplicate operations on 2nd half
   - Balance Checks X<->Y

- Single Instruction Skips
  - detect Branch imbalance trips
  - detect Thread imbalance trips

Dual Module Redundancy (DMR) Transformation

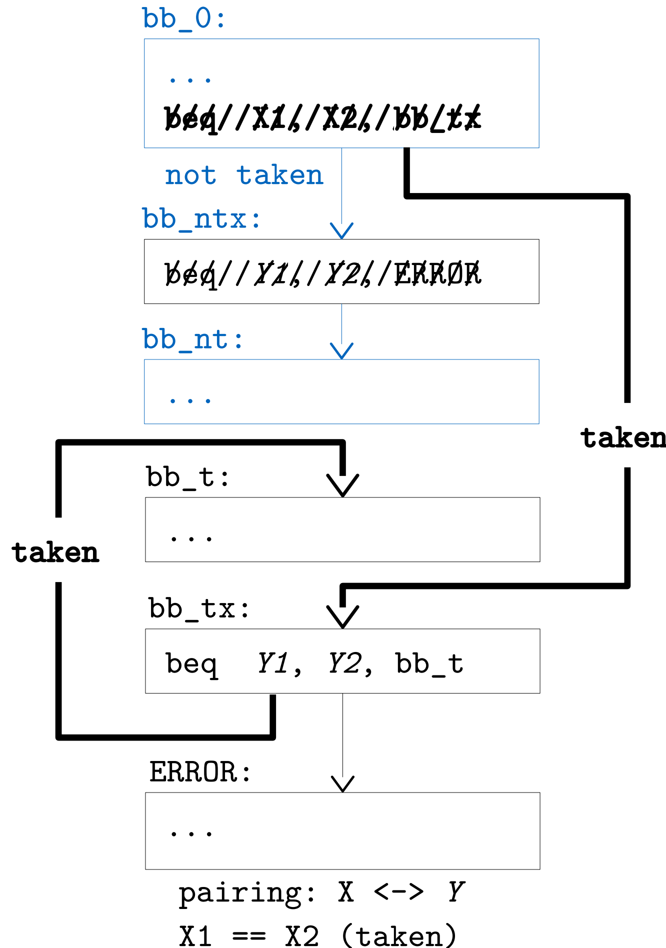


(DMR): nZDC+NEMESIS [2, 3]

1. Reserve half of ISA register file
   - Primary X
   - Shadow Y
2. Duplicate operations on 2nd half
   - Balance Checks X<->Y

- Single Instruction Skips
  - detect Branch imbalance trips
  - detect Thread imbalance trips
- Multi Instruction Skips

Dual Module Redundancy (DMR) Transformation

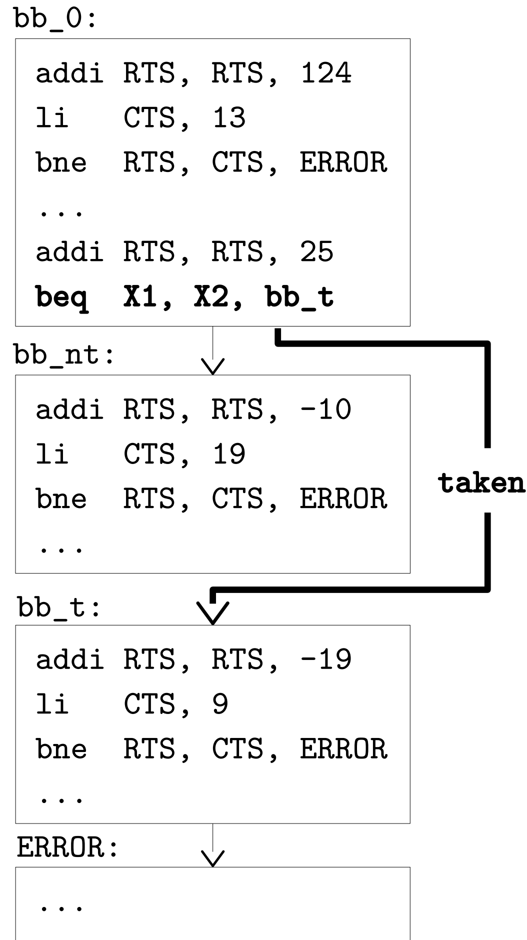


(DMR): nZDC+NEMESIS [2, 3]

1. Reserve half of ISA register file
   - Primary X
   - Shadow Y
2. Duplicate operations on 2nd half
   - Balance Checks X<->Y

- Single Instruction Skips
  - detect Branch imbalance trips
  - detect Thread imbalance trips
- Multi Instruction Skips
  - bypass Branch duplicate
  - bypass Thread duplicate

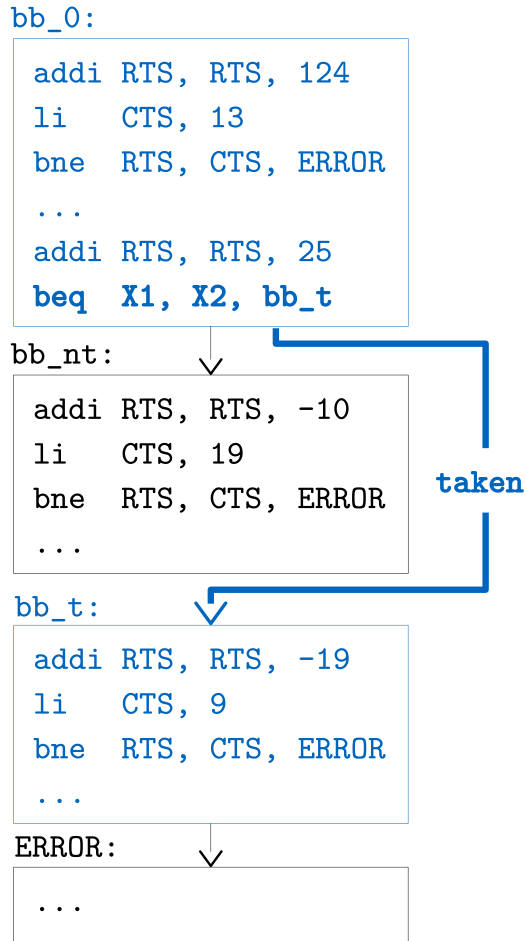
Runtime Signature Monitoring (RSM) Transformation



RSM: RACFED/RASM [7]

1. Assign Basic Block compile time signatures (CTS)
2. Compute `CTS` at runtime `RTS`
   - Control Flow Checks `CTS==RTS`

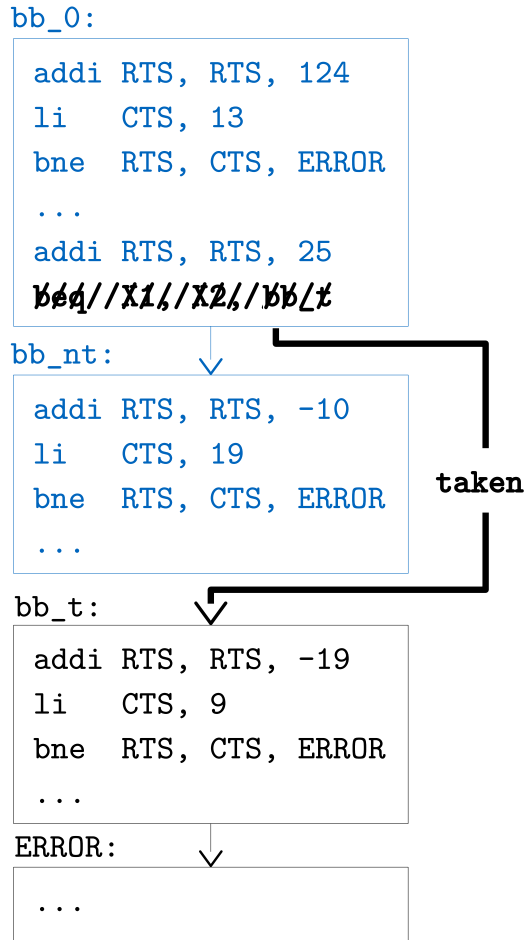
Runtime Signature Monitoring (RSM) Transformation



RSM: RACFED/RASM [7]

1. Assign Basic Block compile time signatures (CTS)
2. Compute `CTS` at runtime `RTS`
   - Control Flow Checks `CTS==RTS`

Runtime Signature Monitoring (RSM) Transformation

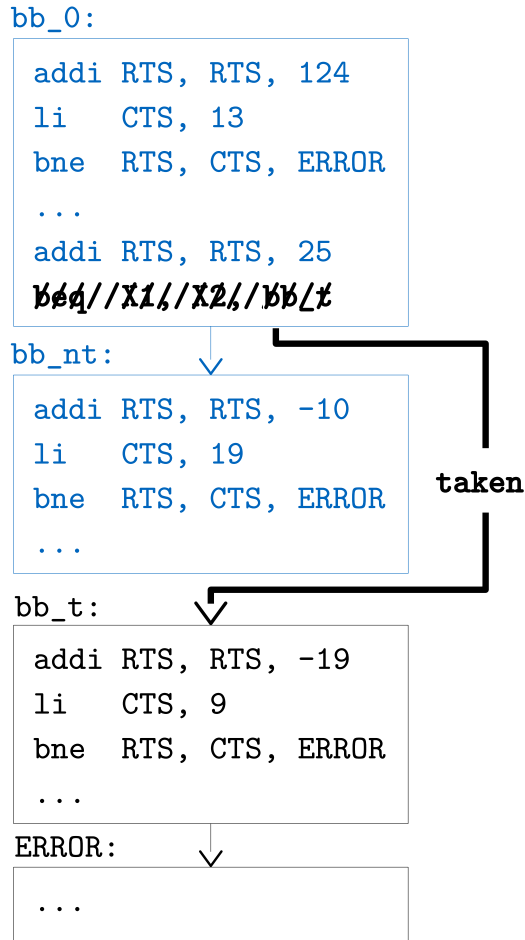


RSM: RACFED/RASM [7]

1. Assign Basic Block compile time signatures (CTS)
2. Compute CTS at runtime RTS
   - Control Flow Checks CTS==RTS

- Single Instruction Skips

Runtime Signature Monitoring (RSM) Transformation

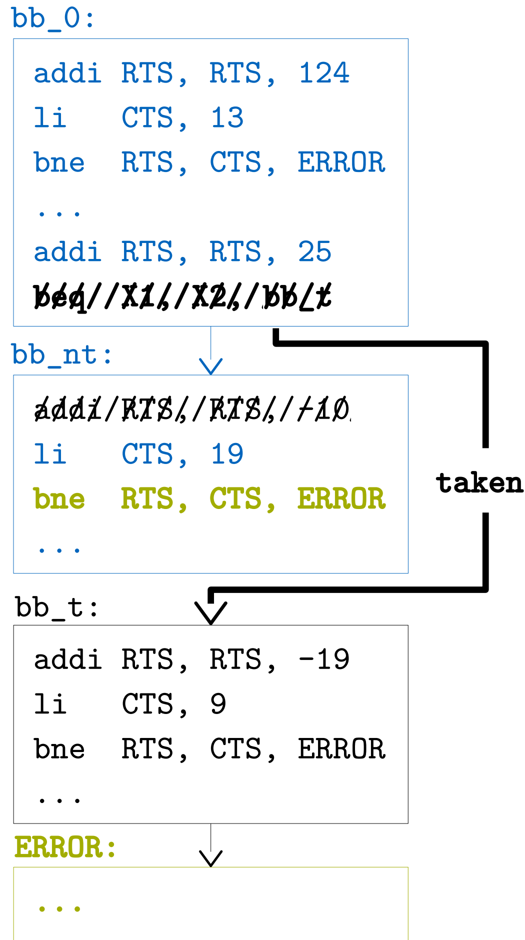


RSM: RACFED/RASM [7]

1. Assign Basic Block compile time signatures (CTS)
2. Compute `CTS` at runtime `RTS`
   - Control Flow Checks `CTS==RTS`

- Single Instruction Skips
  - bypass Control flow is **legal**

Runtime Signature Monitoring (RSM) Transformation

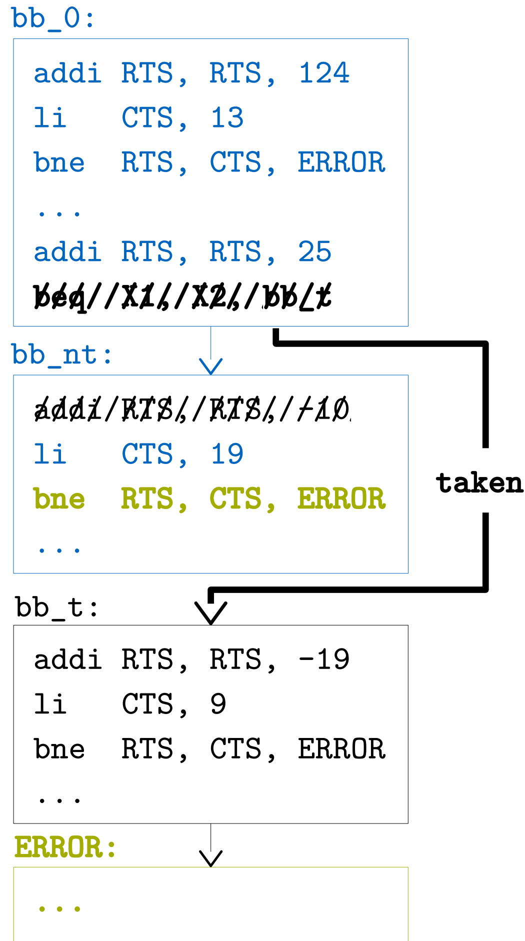


RSM: RACFED/RASM [7]

1. Assign Basic Block compile time signatures (CTS)
2. Compute CTS at runtime RTS
   - Control Flow Checks CTS==RTS

- Single Instruction Skips
  - bypass Control flow is **legal**
- Multi Instruction Skips

Runtime Signature Monitoring (RSM) Transformation

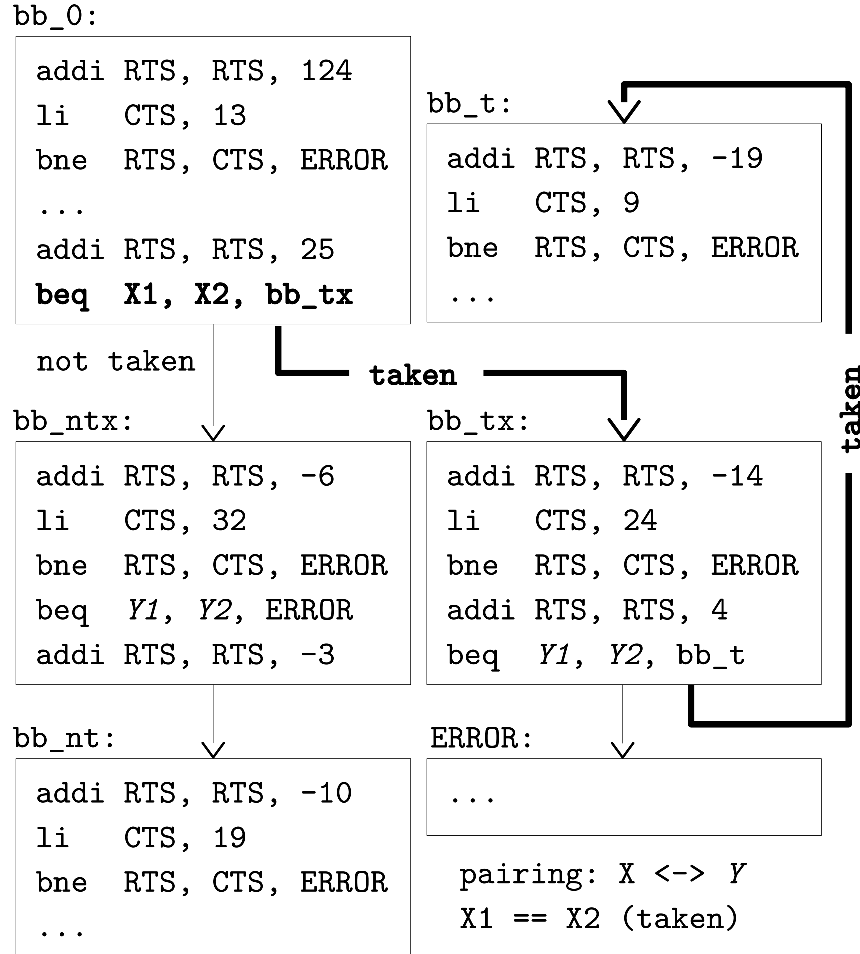


RSM: RACFED/RASM [7]

1. Assign Basic Block compile time signatures (CTS)
2. Compute CTS at runtime RTS
   - ▶ Control Flow Checks CTS==RTS

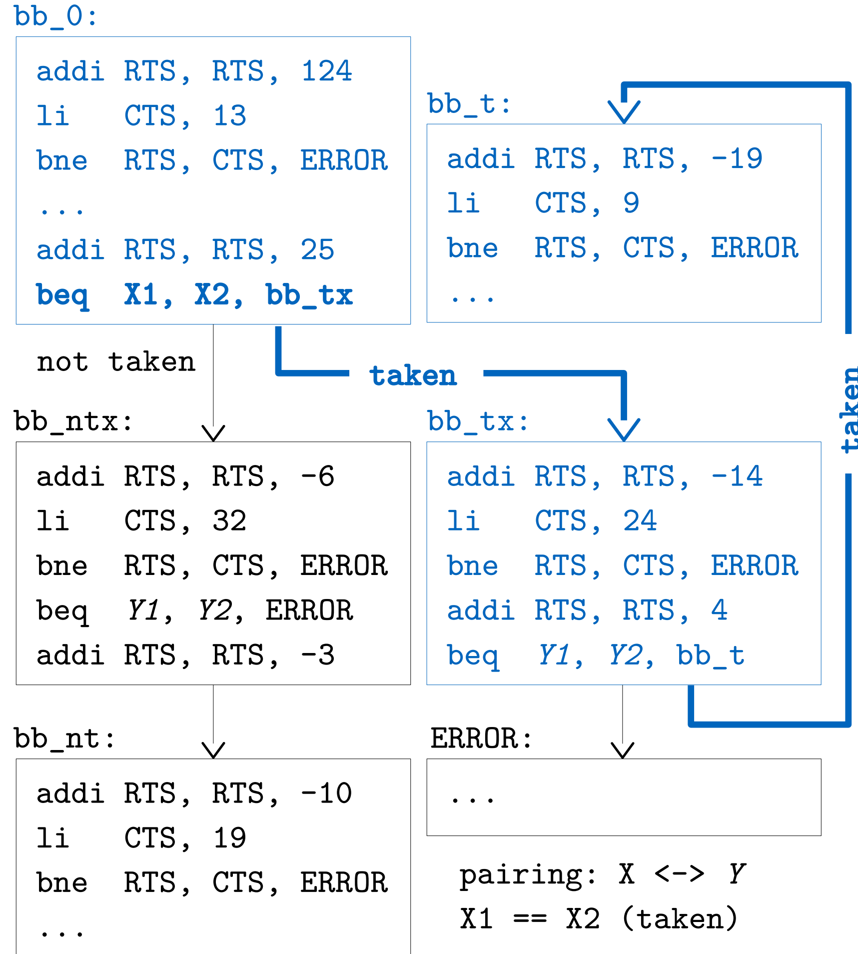
- Single Instruction Skips
  - ▶ bypass Control flow is **legal**
- Multi Instruction Skips
  - ▶ detect Breaks gradual runtime signature
    Trips ERROR with next check

Combining DMR and RSM



- Transformation process:
  - First, DMR pass over original code
  - Then, RSM pass over original and DMR code

Combining DMR and RSM



- Transformation process:
  - First, DMR pass over original code
  - Then, RSM pass over original and DMR code

Combining DMR and RSM

```
bb_0:
addi  RTS, RTS, 124
li    CTS, 13
bne   RTS, CTS, ERROR
...
addi  RTS, RTS, 25
beq   X1, X2, bb_tx   (struck out)
```

not taken / taken

```
bb_t:
addi  RTS, RTS, -19
li    CTS, 9
bne   RTS, CTS, ERROR
...
```

```
bb_ntx:
addi  RTS, RTS, -6
li    CTS, 32
bne   RTS, CTS, ERROR
beq   Y1, Y2, ERROR
addi  RTS, RTS, -3
```

```
bb_tx:
addi  RTS, RTS, -14
li    CTS, 24
bne   RTS, CTS, ERROR
addi  RTS, RTS, 4
beq   Y1, Y2, bb_t
```

taken

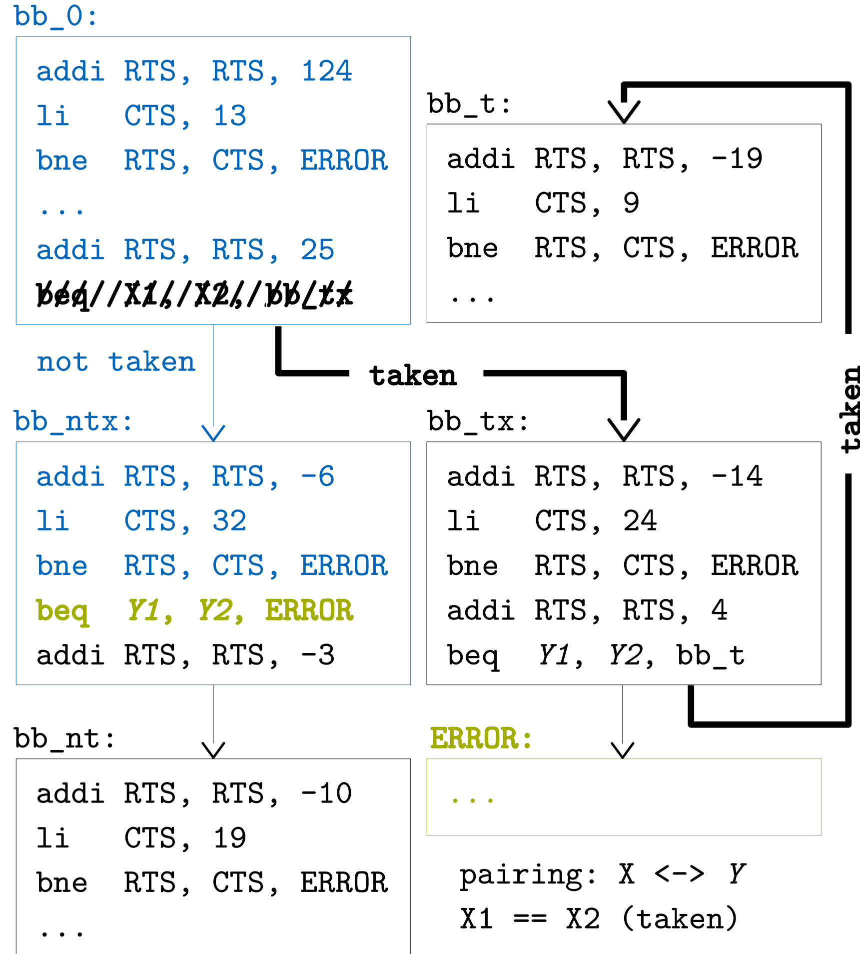
```
bb_nt:
addi  RTS, RTS, -10
li    CTS, 19
bne   RTS, CTS, ERROR
...
```

```
ERROR:
...
```

```
pairing: X <-> Y
X1 == X2 (taken)
```

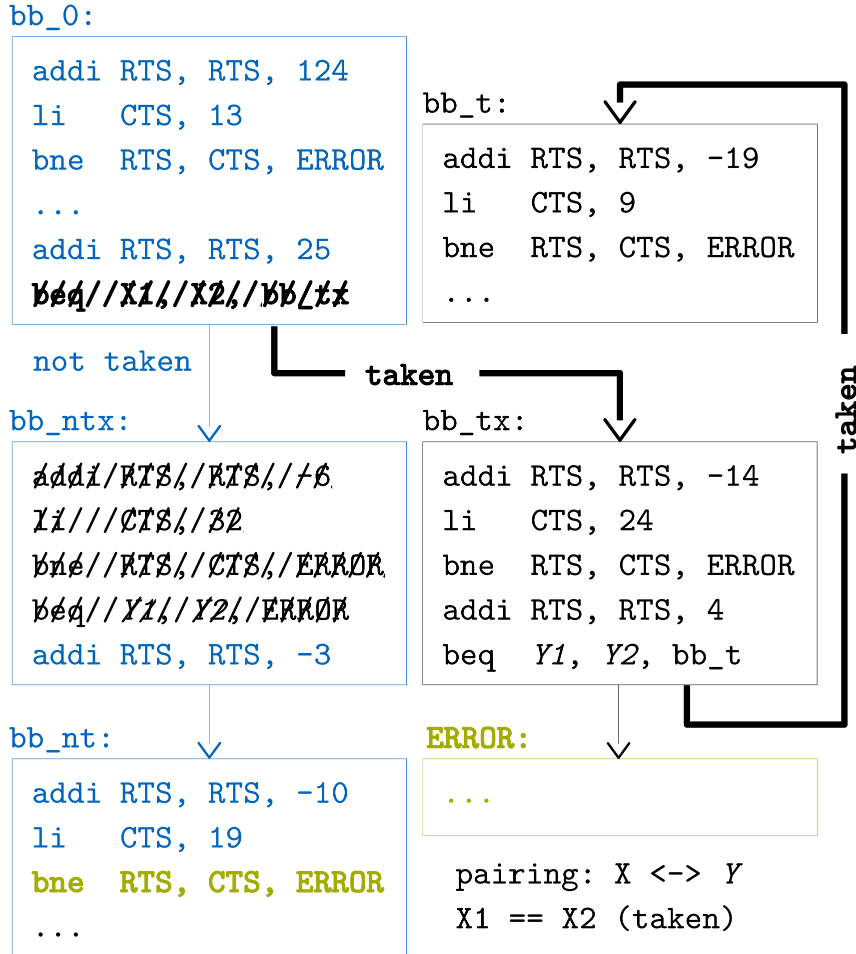
- Transformation process:
  - First, DMR pass over original code
  - Then, RSM pass over original and DMR code
- Single Instruction Skips

Combining DMR and RSM



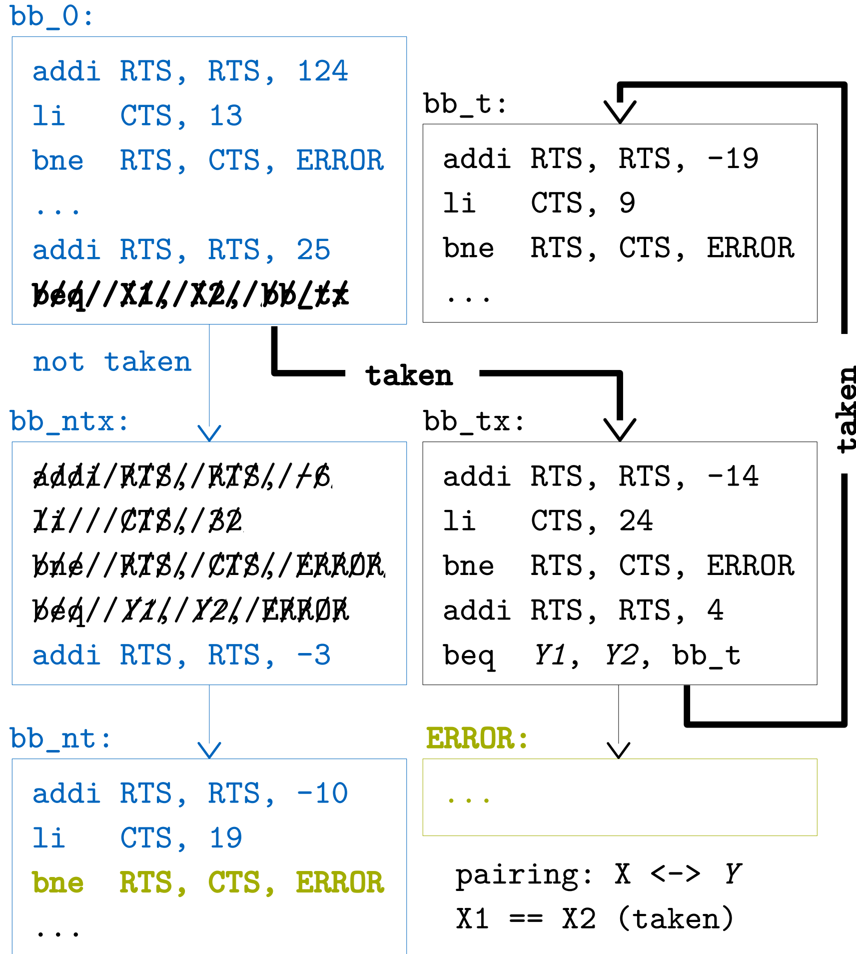
- Transformation process:
  - First, DMR pass over original code
  - Then, RSM pass over original and DMR code
- Single Instruction Skips
  - DMR detect

Combining DMR and RSM



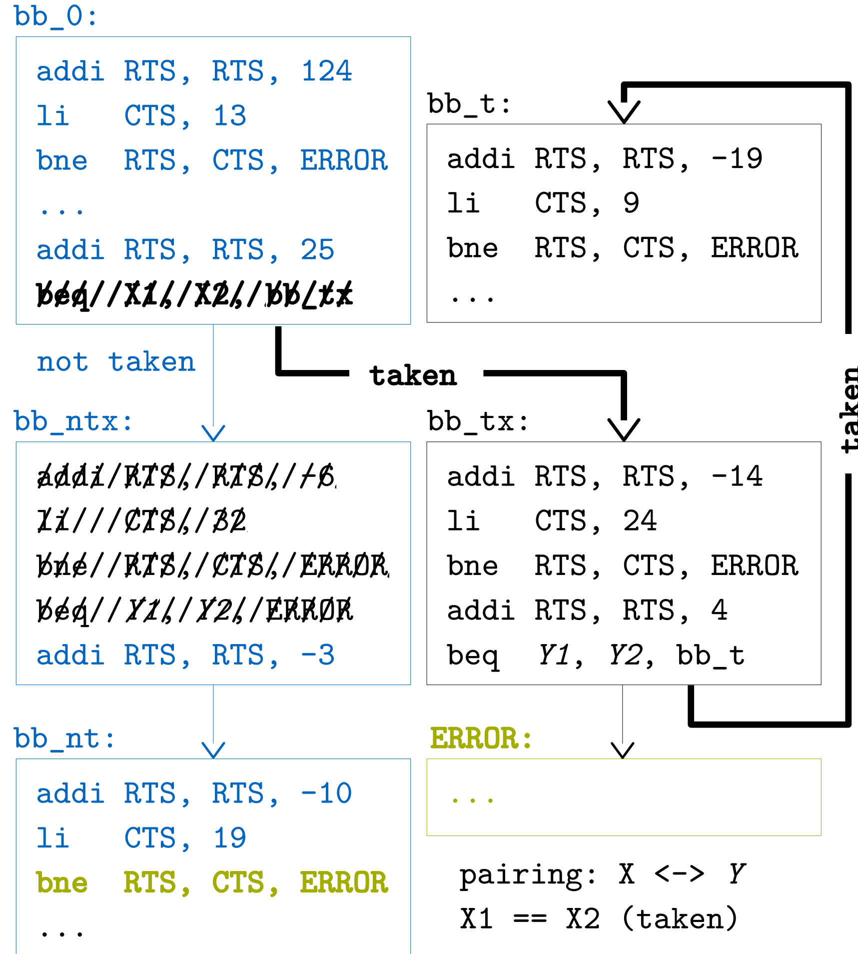
- Transformation process:
  - First, DMR pass over original code
  - Then, RSM pass over original and DMR code
- Single Instruction Skips
  - DMR detect
- Multi Instruction Skips

Combining DMR and RSM



- Transformation process:
  - First, DMR pass over original code
  - Then, RSM pass over original and DMR code
- Single Instruction Skips
  - DMR detect
- Multi Instruction Skips
  - RSM detect

Combining DMR and RSM

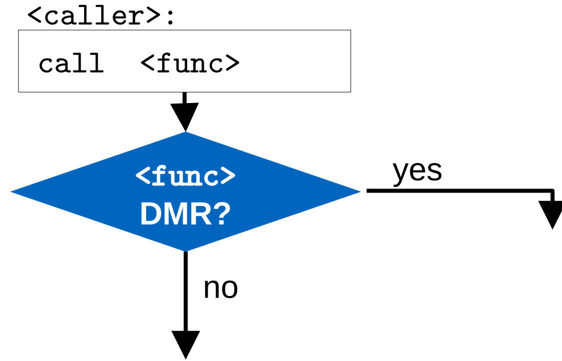


- Transformation process:
  - First, DMR pass over original code
  - Then, RSM pass over original and DMR code
- Single Instruction Skips
  - DMR detect
- Multi Instruction Skips
  - RSM detect

DMR↔RSM Symbiosis:

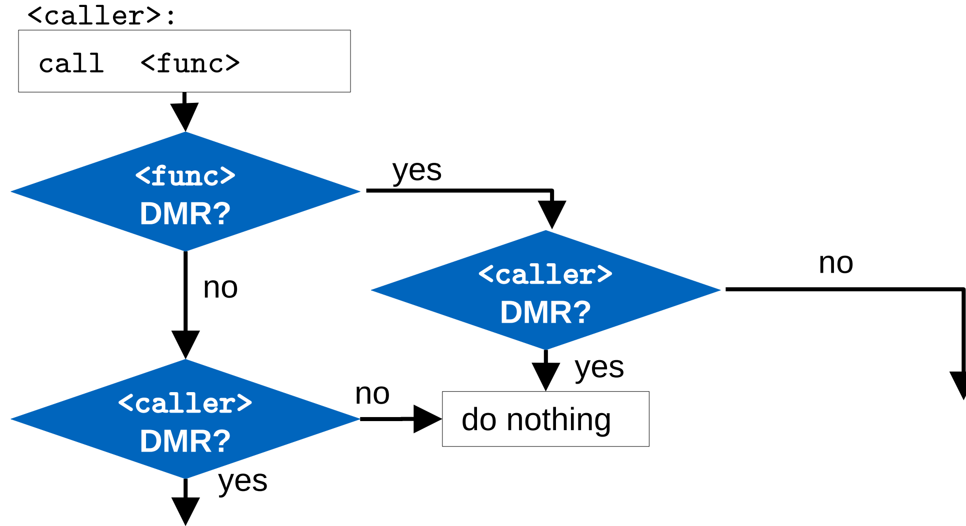
- DMR dataflow integrity
- RSM control flow integrity

Selective Hardening



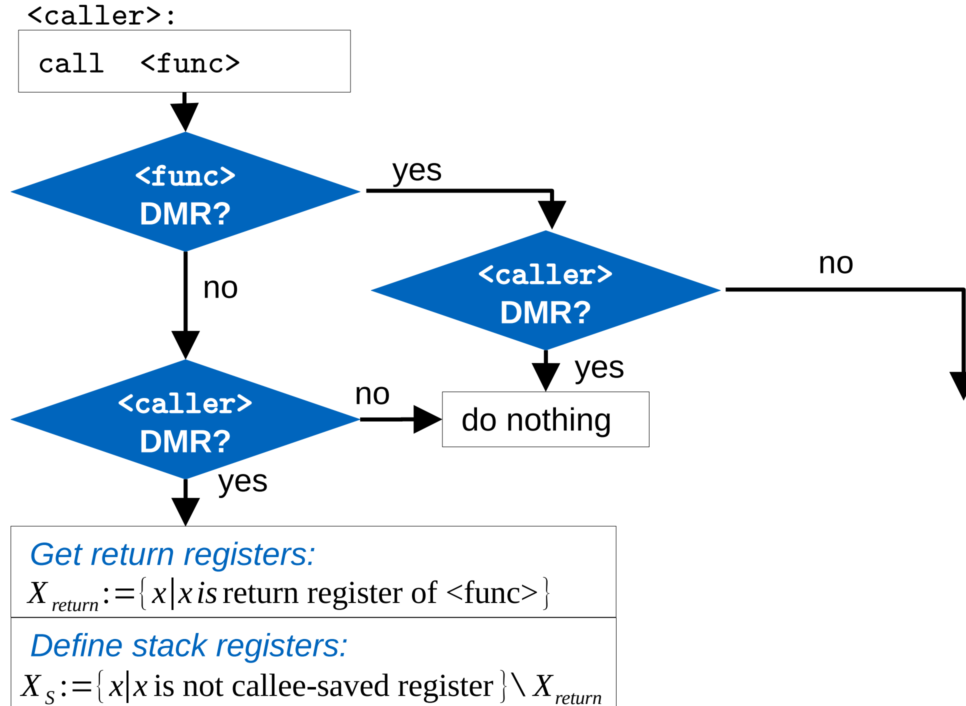
- For all function calls during transformation

Selective Hardening



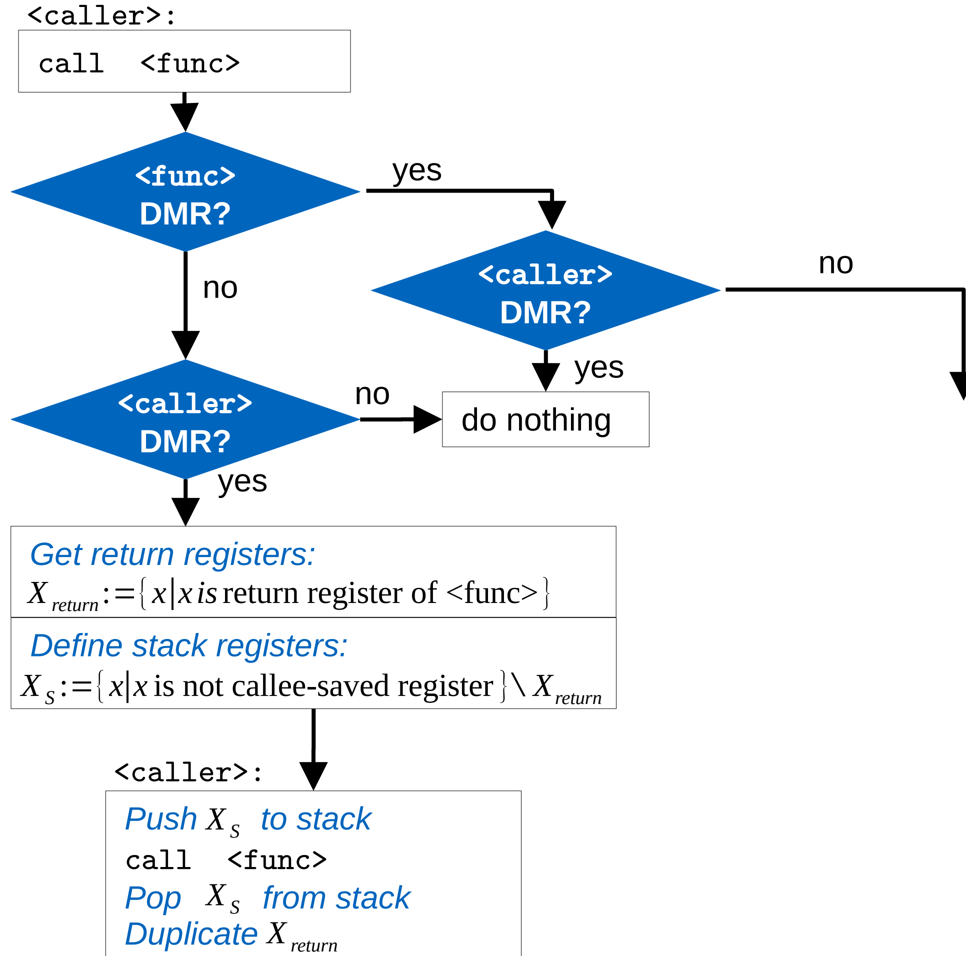
- For all function calls during transformation
- Modifaction needed for function calls crossing DMR domains

Selective Hardening



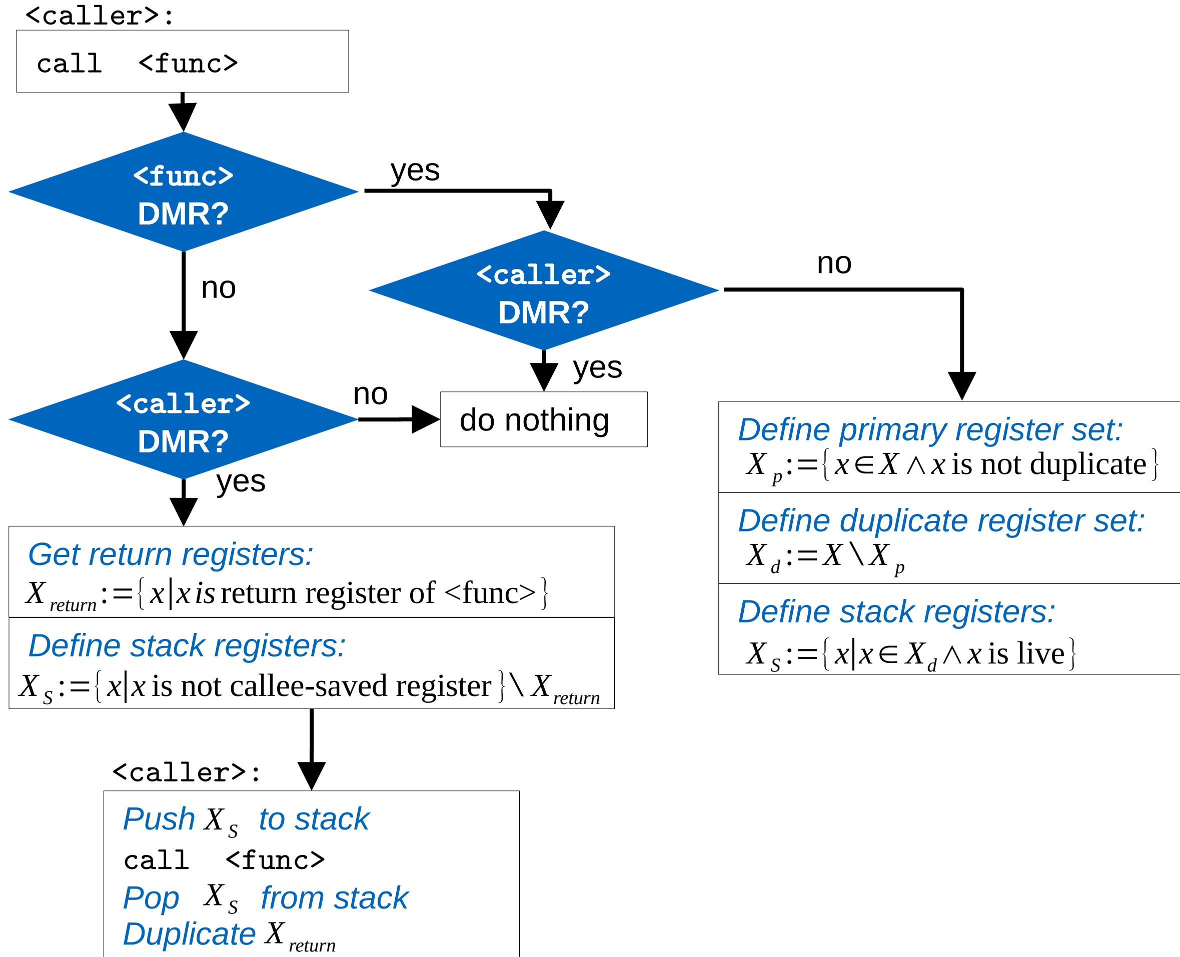
- For all function calls during transformation
- Modifaction needed for function calls crossing DMR domains
- DMR→**non-DMR** save duplicates before call, restore after

Selective Hardening



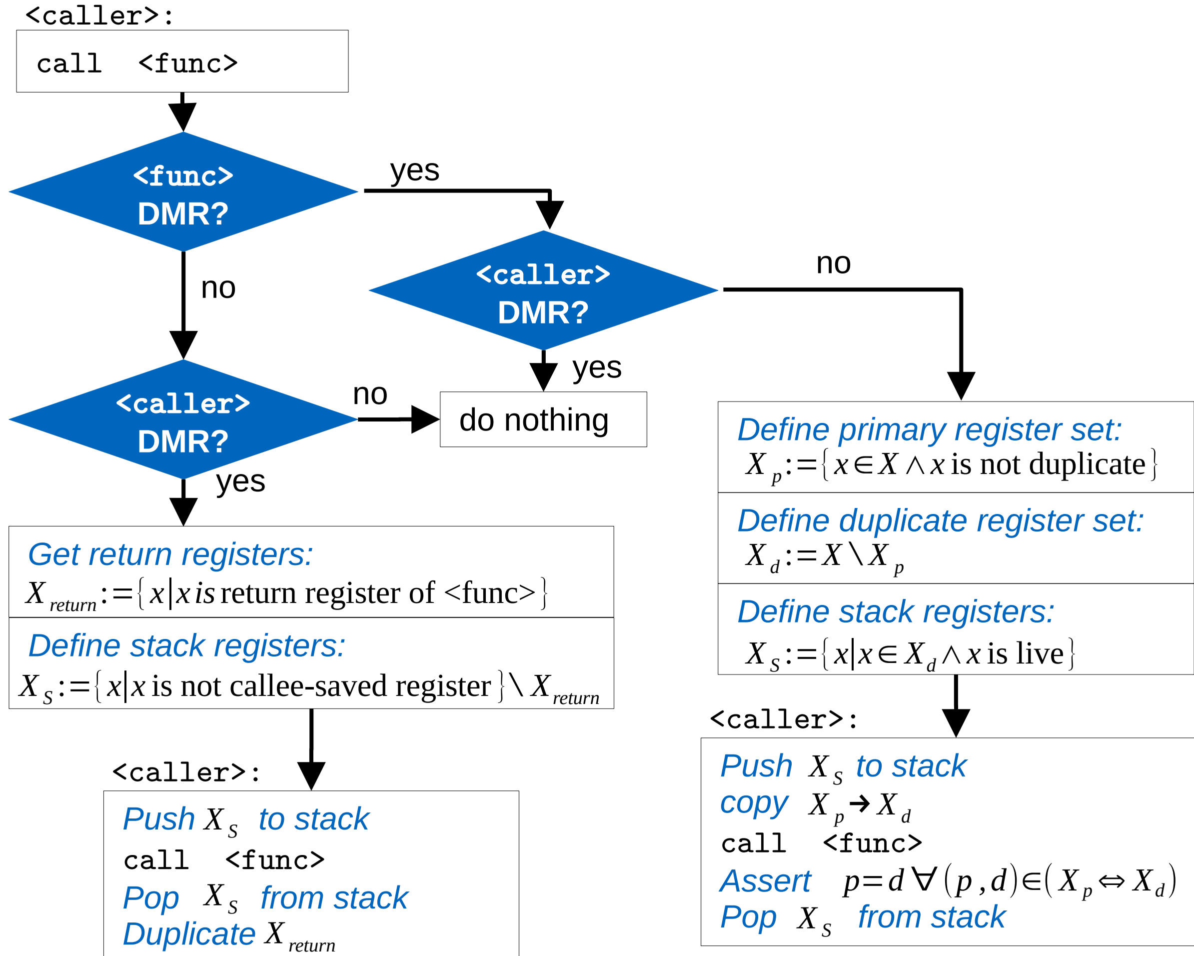
- For all function calls during transformation
- Modifaction needed for function calls crossing DMR domains
- DMR→**non-DMR** save duplicates before call, restore after

Selective Hardening



- For all function calls during transformation
- Modifaction needed for function calls crossing DMR domains
- DMR→**non-DMR** save duplicates before call, restore after
- **non-DMR**→DMR prepare duplicates and asserts balance after return

Selective Hardening



- For all function calls during transformation
- Modifaction needed for function calls crossing DMR domains
- DMR→**non-DMR** save duplicates before call, restore after
- **non-DMR**→DMR prepare duplicates and asserts balance after return

Outline

1. Motivation

2. Compiler-assisted Countermeasures Against Instruction Skip Fault Attacks

3. Evaluation and Performance Results

4. Conclusion

Scenario: Bypassing Secure Boot

- **Goal:** Boot a malicious Software Image bypassing security checks.

Scenario: Bypassing Secure Boot

- **Goal:** Boot a malicious Software Image bypassing security checks.
- Tool: ARCHIE [9]: QEMU-based fault injection simulation

Scenario: Bypassing Secure Boot

- **Goal:** Boot a malicious Software Image bypassing security checks.
- Tool: ARCHIE [9]: QEMU-based fault injection simulation

1. Identify Fault Candidates
   - All **executed instructions**
   - Fault Model: **Single**, **double**, **triple**, and **quadruple** instruction skip
   - One experiment for each Fault Candidate and Fault Model

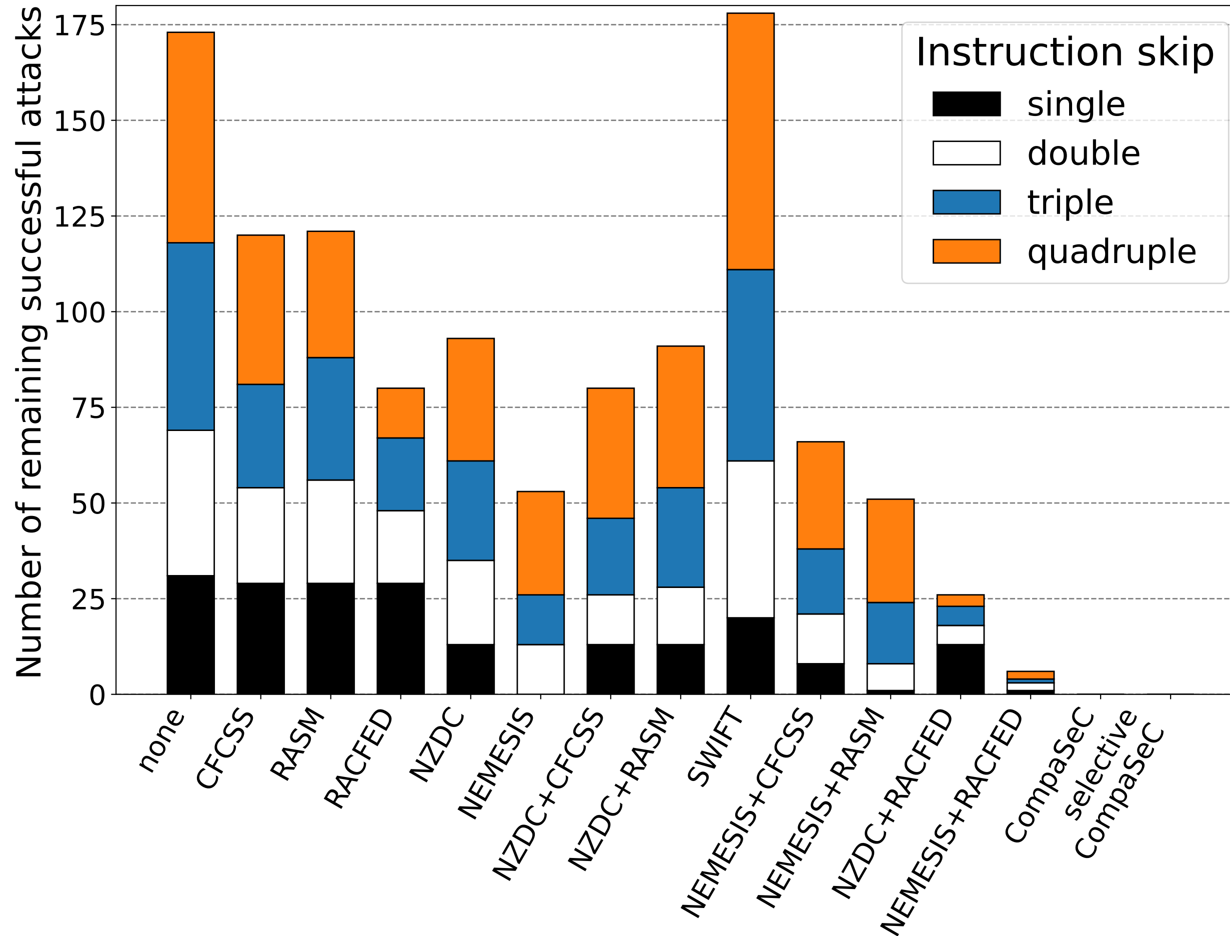
Scenario: Bypassing Secure Boot

- **Goal:** Boot a malicious Software Image bypassing security checks.
- Tool: ARCHIE [9]: QEMU-based fault injection simulation

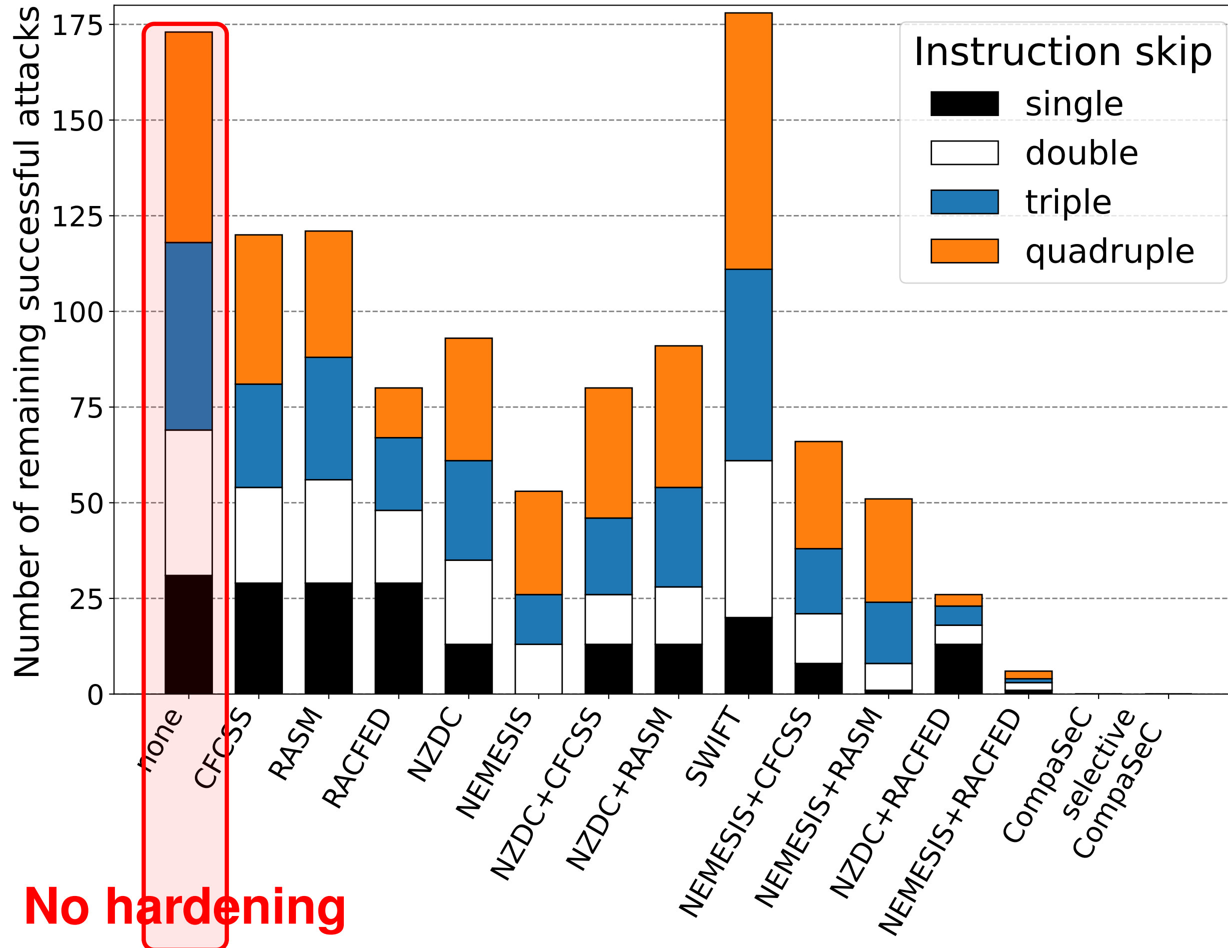
1. Identify Fault Candidates
   - All **executed instructions**
   - Fault Model: **Single**, **double**, **triple**, and **quadruple** instruction skip
   - One experiment for each Fault Candidate and Fault Model
2. Simulate and Categorize

DMR	RSM	Fault Candidates [10^3]	Successful Faults	Detection Rate [%]
none		91	173	-
CompaSeC		685	0	85.9

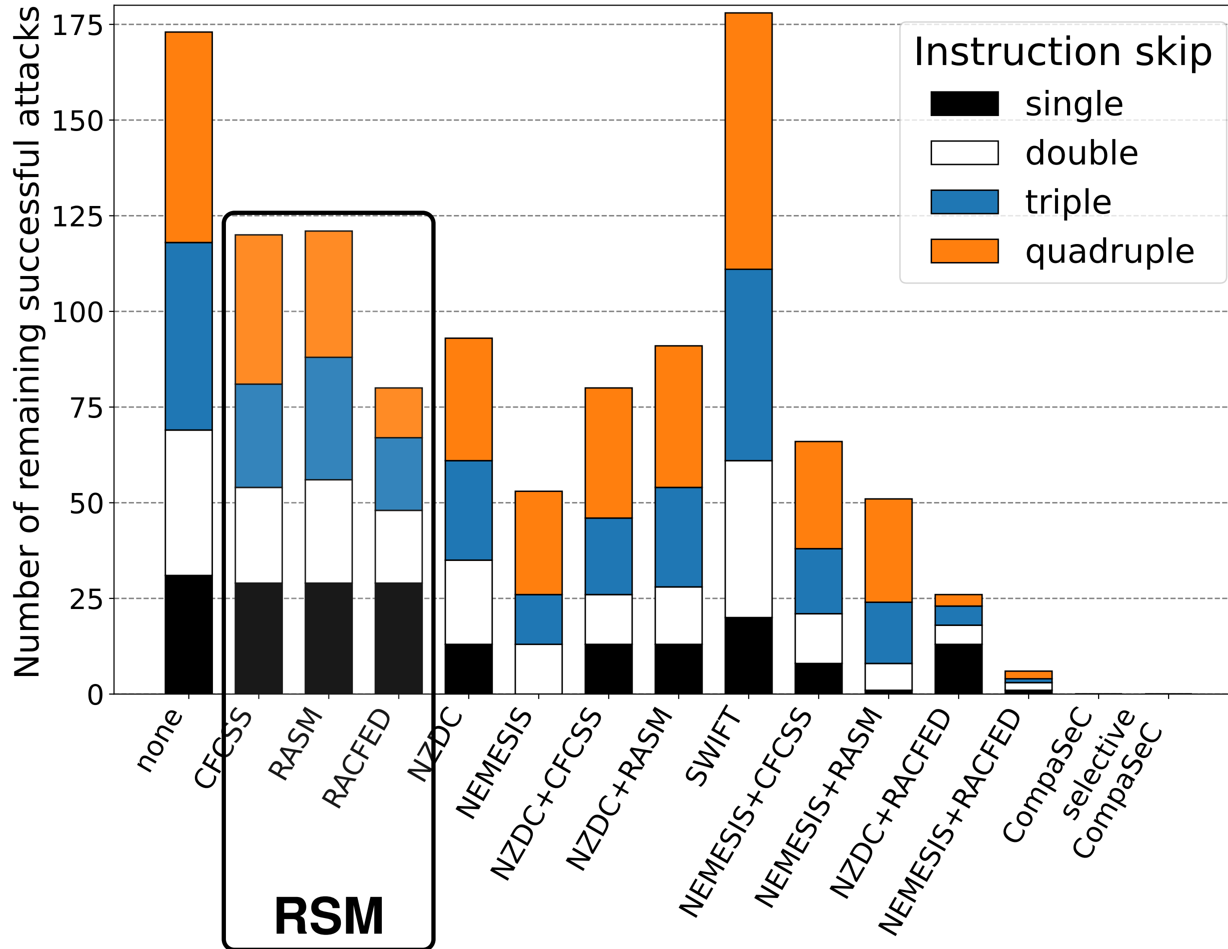
Efficacy: Remaining Successful Fault Attacks



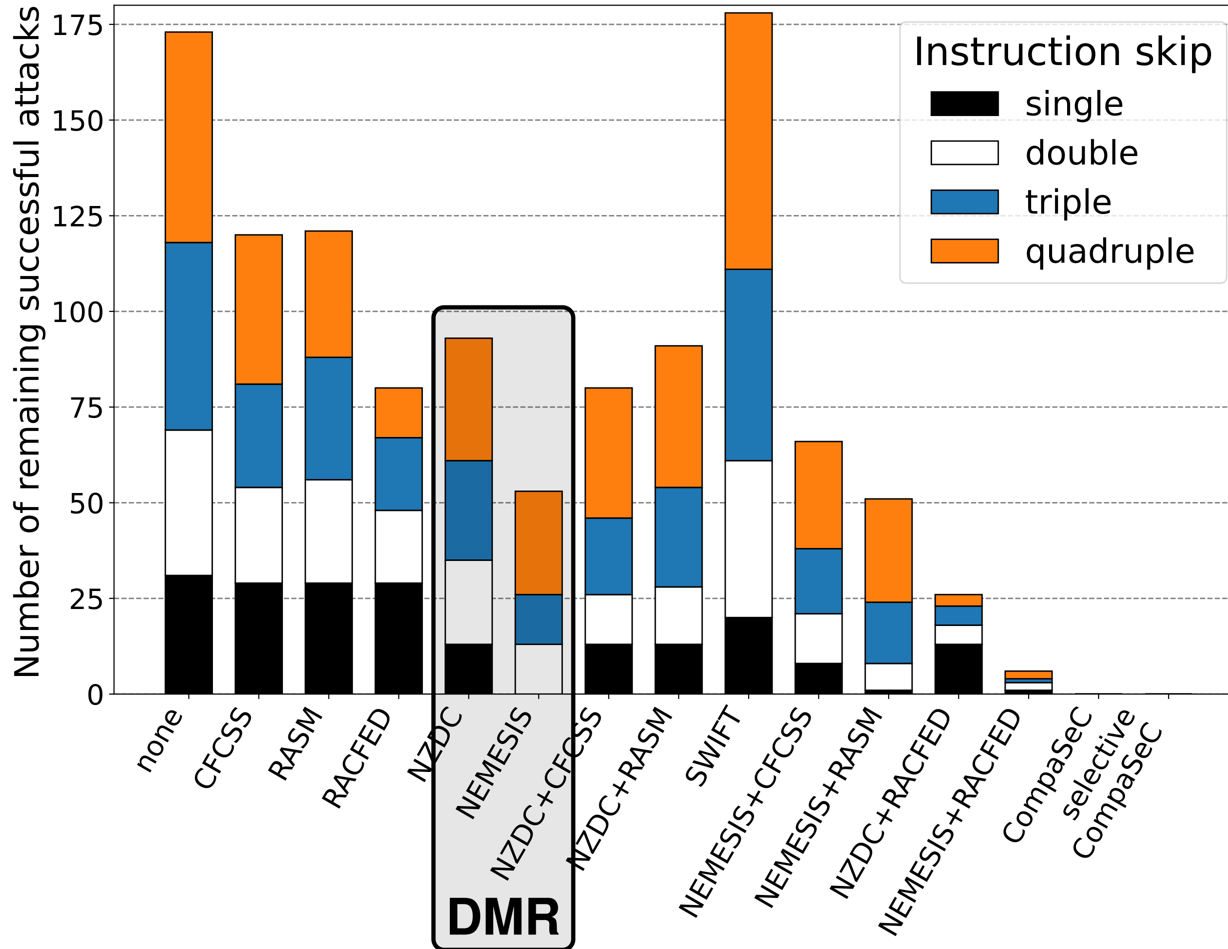
Efficacy: Remaining Successful Fault Attacks



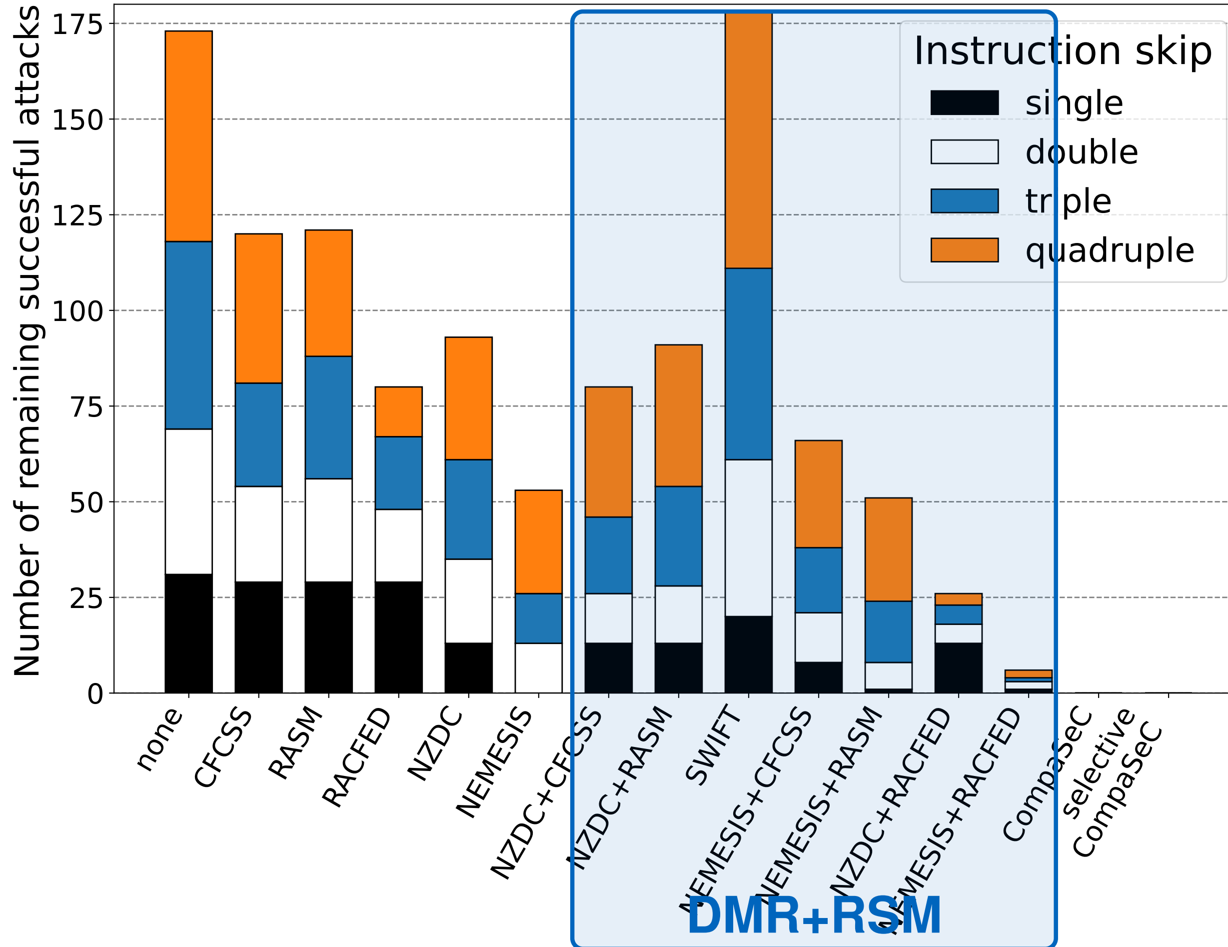
Efficacy: Remaining Successful Fault Attacks



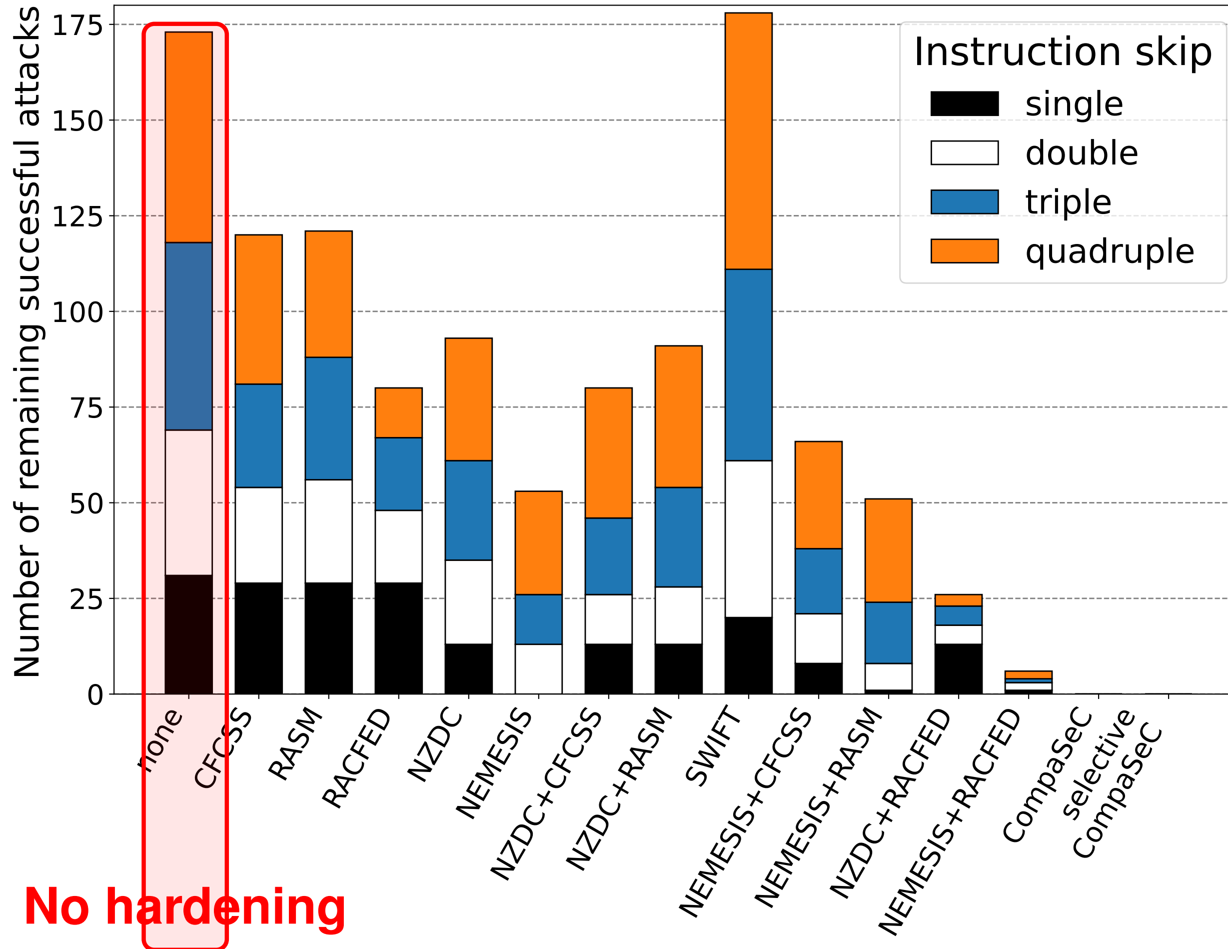
Efficacy: Remaining Successful Fault Attacks



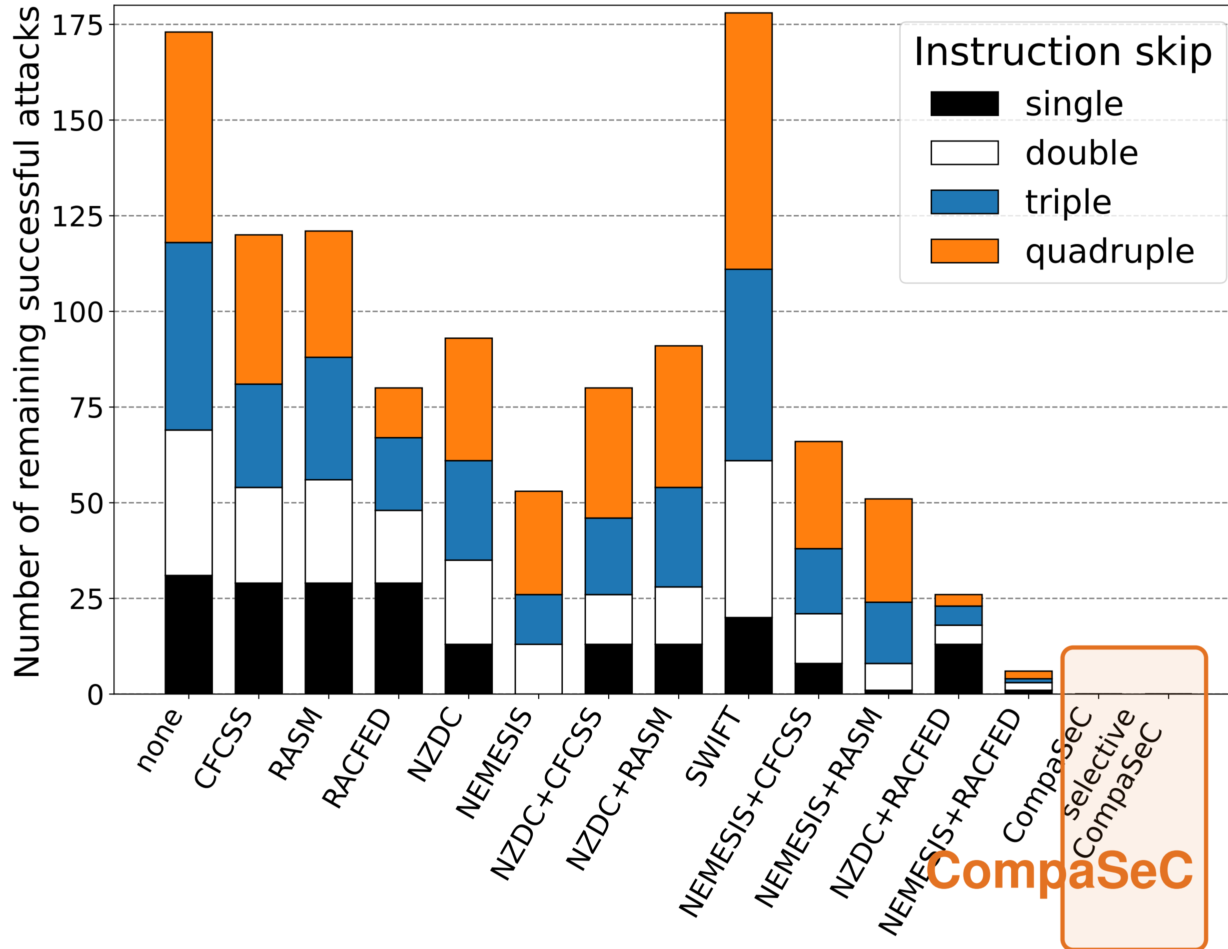
Efficacy: Remaining Successful Fault Attacks



Efficacy: Remaining Successful Fault Attacks

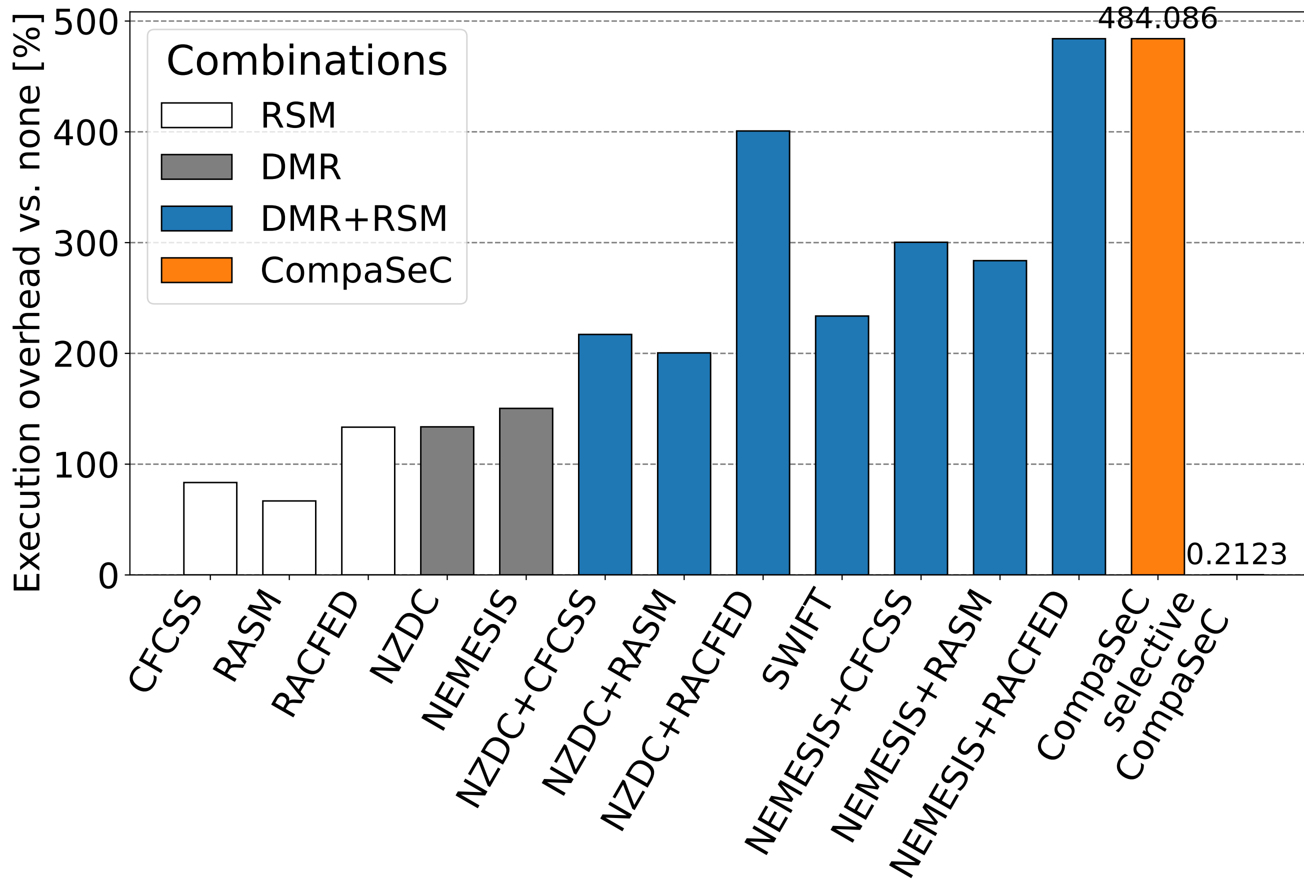


Efficacy: Remaining Successful Fault Attacks



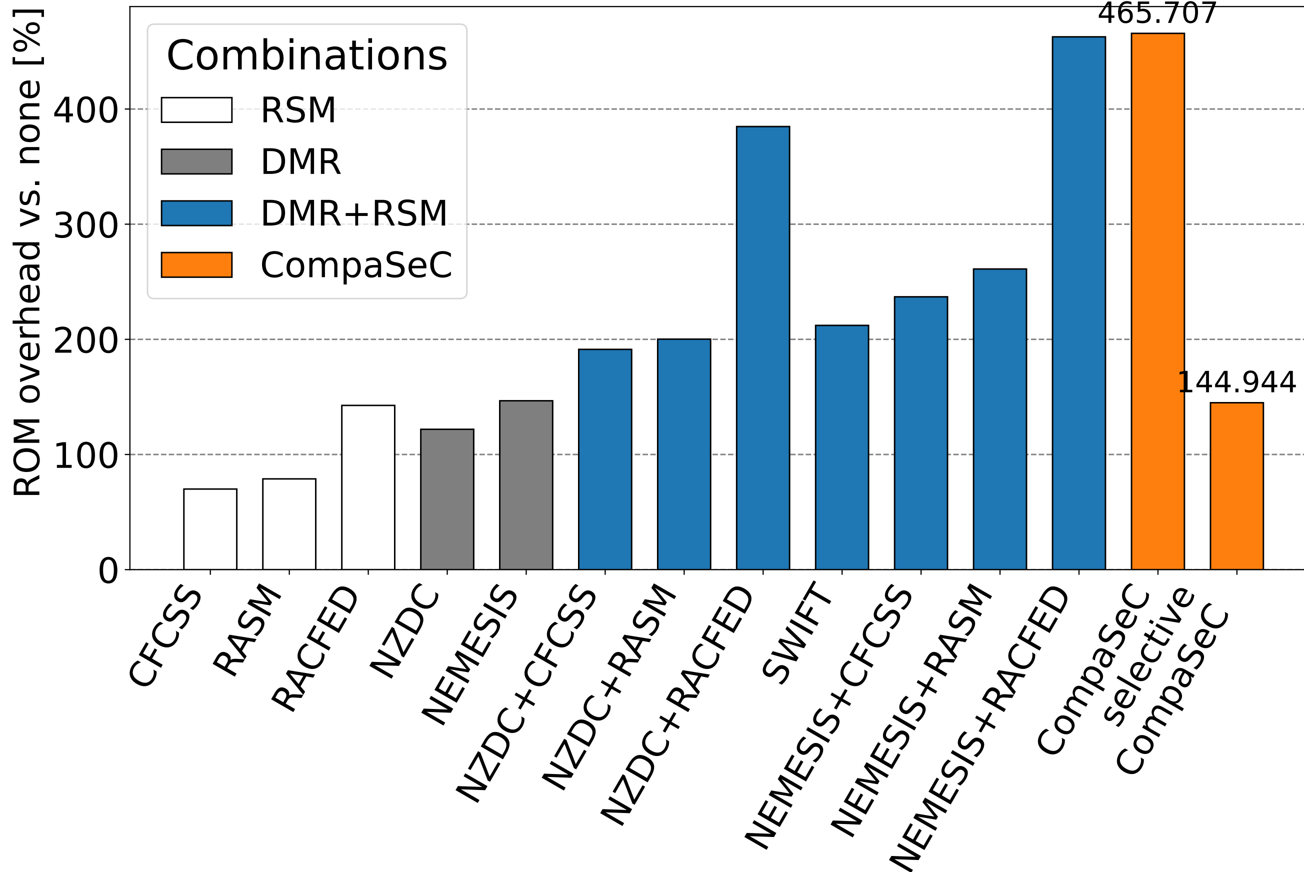
Performance: Execution Overhead

Metric: Number of executed instructions



Performance: ROM Size Overhead

Metric: ROM footprint for Secure Boot



Outline

1. Motivation

2. Compiler-assisted Countermeasures Against Instruction Skip Fault Attacks

3. Evaluation and Performance Results

4. Conclusion

Summary

Summary

- Security-aware compiler-assisted countermeasure
  - strong vs. single skip
  - weak vs. multi skip

  - weak vs. single skip
  - strong vs. multi skip

Summary

- Security-aware compiler-assisted countermeasure
  Dual Module Redundancy (DMR):
  - strong vs. single skip
  - weak vs. multi skip

  - weak vs. single skip
  - strong vs. multi skip

Summary

- Security-aware compiler-assisted countermeasure

  Dual Module Redundancy (DMR):
  - strong vs. single skip
  - weak vs. multi skip

  Runtime Signature Monitoring (RSM):
  - weak vs. single skip
  - strong vs. multi skip

Summary

- Security-aware compiler-assisted countermeasure
  Dual Module Redundancy (DMR):
  - strong vs. single skip
  - weak vs. multi skip

  Runtime Signature Monitoring (RSM):
  - weak vs. single skip
  - strong vs. multi skip
- Combining DMR and RSM:
  - *"Symbiotic Interplay"* at large overhead and high protection

Summary

- Security-aware compiler-assisted countermeasure

  Dual Module Redundancy (DMR):
  - strong vs. single skip
  - weak vs. multi skip

  Runtime Signature Monitoring (RSM):
  - weak vs. single skip
  - strong vs. multi skip
- Combining DMR and RSM:
  - *"Symbiotic Interplay"* at large overhead and high protection
- Verification against known Fault Model allows:

Summary

- Security-aware compiler-assisted countermeasure

  Dual Module Redundancy (DMR):
  - strong vs. single skip
  - weak vs. multi skip

  Runtime Signature Monitoring (RSM):
  - weak vs. single skip
  - strong vs. multi skip
- Combining DMR and RSM:
  - *"Symbiotic Interplay"* at large overhead and high protection
- Verification against known Fault Model allows:
  - Identifying vulnerable code sections

Summary

- Security-aware compiler-assisted countermeasure

  Dual Module Redundancy (DMR):
  - strong vs. single skip
  - weak vs. multi skip

  Runtime Signature Monitoring (RSM):
  - weak vs. single skip
  - strong vs. multi skip
- Combining DMR and RSM:
  - *"Symbiotic Interplay"* at large overhead and high protection
- Verification against known Fault Model allows:
  - Identifying vulnerable code sections
  - Selective Hardening: small overhead and high protection

Contact

Johannes Geier

johannes.geier@tum.de

Chair of Electronic Design Automation, Technical University of Munich

Lukas Auer

lukas.auer@aisec.fraunhofer.de

Fraunhofer Institute for Applied and Integrated Security (AISEC)

Open Source:

github.com/tum-ei-eda/compas-ft-riscv

Compas [10]/CompaSeC [11] LLVM-based Compiler

References I

[1] N. Oh, P.P. Shirvani, and E.J. McCluskey. 2002. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51, 1, 111–122. DOI: 10.1109/24.994926.

[2] Moslem Didehban and Aviral Shrivastava. 2016. nZDC: a compiler technique for near zero silent data corruption. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 1–6. DOI: 10.1145/2897937.2898054.

[3] Moslem Didehban, Aviral Shrivastava, and Sai Ram Dheeraj Lokam. 2017. NEMESIS: a software approach for computing in presence of soft errors. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 297–304. DOI: 10.1109/ICCAD.2017.8203792.

[4] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. 2005. SWIFT: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, 243–254. DOI: 10.1109/CGO.2005.34.

[5] N. Oh, P.P. Shirvani, and E.J. McCluskey. 2002. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51, 1, 63–75. DOI: 10.1109/24.994913.

[6] Jens Vankeirsbilck, Niels Penneman, Hans Hallez, and Jeroen Boydens. 2017. Random additive signature monitoring for control flow error detection. *IEEE Transactions on Reliability*, 66, 4, 1178–1192. DOI: 10.1109/TR.2017.2754548.

[7] Jens Vankeirsbilck, Niels Penneman, Hans Hallez, and Jeroen Boydens. 2018. Random additive control flow error detection. In *Computer Safety, Reliability, and Security*. Barbara Gallina, Amund Skavhaug, and Friedemann Bitsch, editors. Springer International Publishing, Cham, 220–234. DOI: 10.1007/978-3-319-99130-6.

[8] Uzair Sharif, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. 2021. REPAIR: control flow protection based on register pairing updates for SW-implemented HW fault tolerance. *ACM Trans. Embed. Comput. Syst.*, 20, 5s, Article 70, (Sept. 2021), 22 pages. DOI: 10.1145/3477001.

References II

[9] Florian Hauschild, Kathrin Garb, Lukas Auer, Bodo Selmke, and Johannes Obermaier. 2021. ARCHIE: A QEMU-Based framework for architecture-independent evaluation of faults. In *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, 20–30. DOI: 10.1109/FDTC53659.2021.00013.

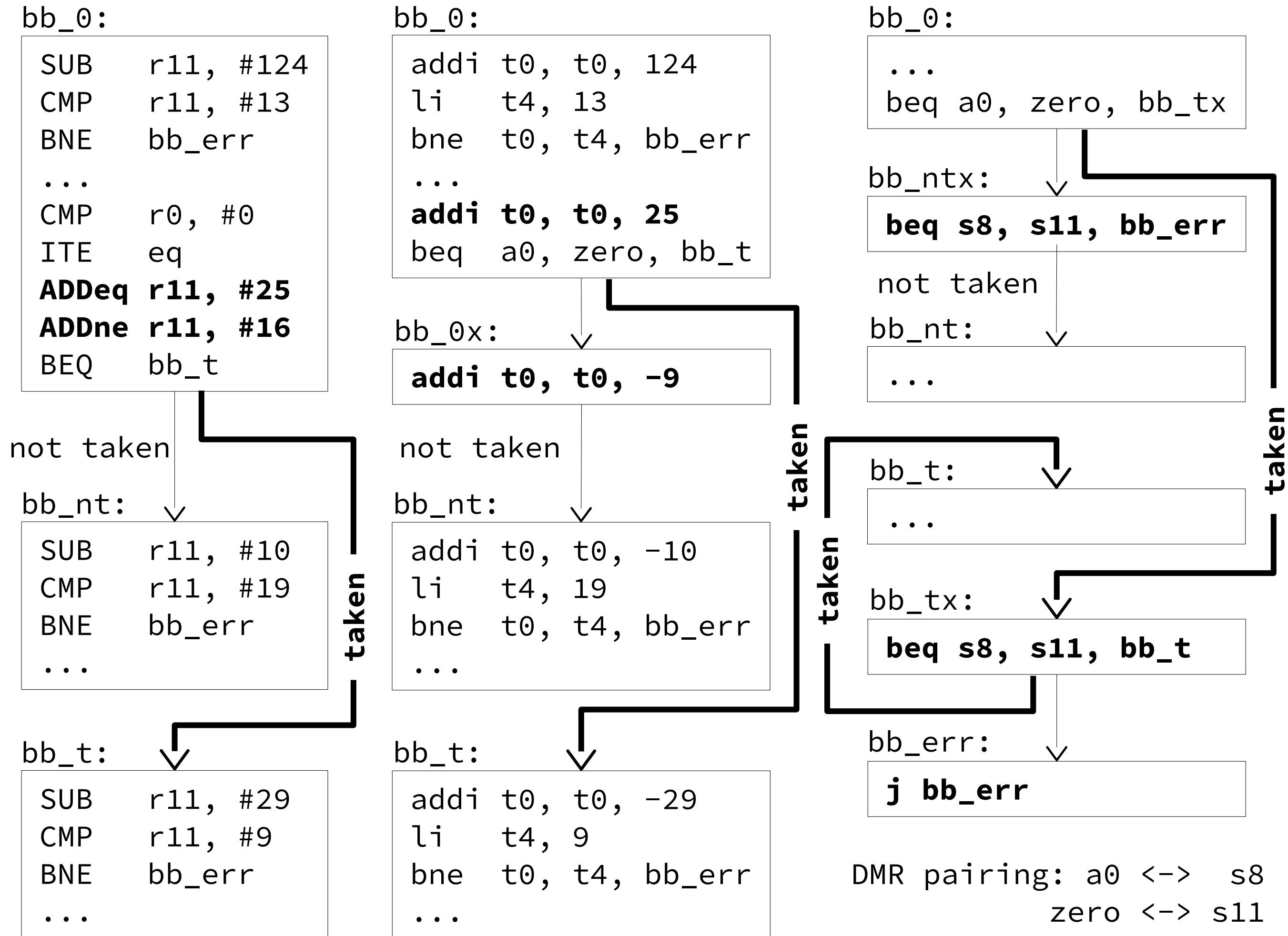
[10] Uzair Sharif, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. 2022. COMPAS: compiler-assisted software-implemented hardware fault tolerance for RISC-V. In *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, 1–4. DOI: 10.1109/MECO55406.2022.9797144.

[11] Johannes Geier, Lukas Auer, Daniel Mueller-Gritschneder, Uzair Sharif, and Ulf Schlichtmann. 2023. CompaSeC: a compiler-assisted security countermeasure to address instruction skip fault attacks on RISC-V. In *Will be published in: Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASP-DAC 2023)* (ASPDAC ’23). Tokyo, Japan, 7 pages.

Speaker

Johannes Geier received the B.Eng. (2018) in Electrical Engineering from OTH Regensburg and M.Sc. (2020) in Electrical Engineering from Technical University of Munich (TUM), Germany. Currently, he is a doctoral candidate at the Chair of Electronic Design Automation at TUM.
His research interests include virtual prototypes, fault injection simulations, and instruction set architectures.

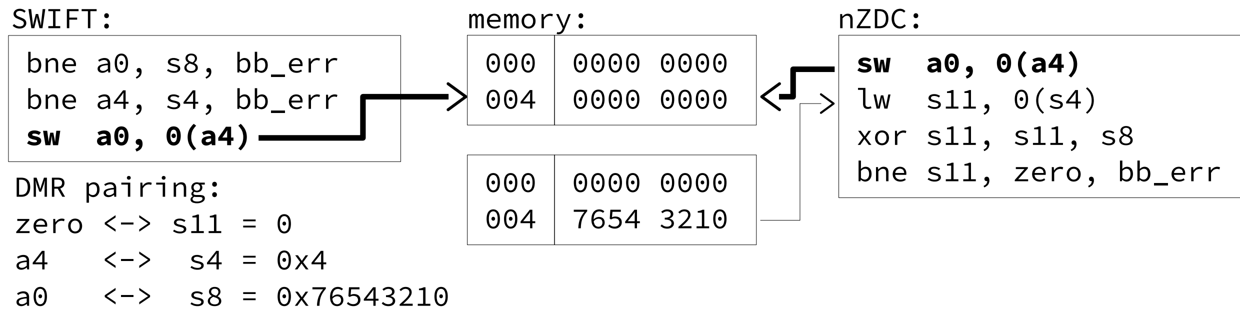
Backup - Predicated Problem



Backup - CompaSeC Combination Details

DMR

- *nZDC+NEMESIS* [2, 3] duplication, memory store-load-back loop, and balance check schedule
- *SWIFT* [4] **volatile** memory store protection
- Duplicated branches as basic block entities



RSM

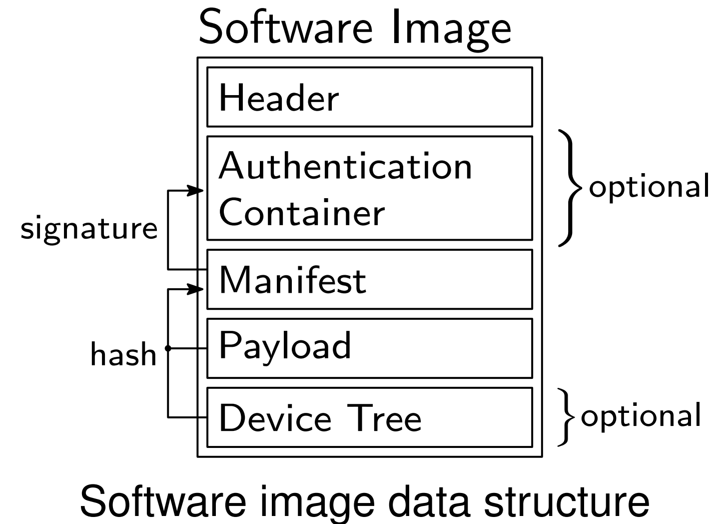
- *RACFED* [7] instruction level additive signature, basic block level checks. Adapted algorithm for minimal basic blocks size
- Pass includes DMR code

Backup - Scenario: Bypass Secure Boot

Goal: Boot a malicious Software Image, that has ...

- "No signature"
  - No authentication container is included
  - Verification of SW image is not possible
  - Not accepted by Secure Boot

- "Incorrect signature"
  - Authentication container is included
  - Signing key is not trusted
  - Not accepted by Secure Boot



Software image data structure

→ Secure Boot binary hardened with combinations of DMR and RSM techniques

Backup - Efficacy Details

DMR	RSM	Fault Candidates [10^3]	Successful Faults	Detection Rate [%]
none		91	173	-
-	cfcss	202	120	43.2
-	rasm	205	121	52.8
-	racfed	250	80	68.1
nzdc	-	268	93	37.6
nemesis	-	282	53	43.9
nzdc	cfcss	374	80	48.0
nzdc	rasm	376	91	53.1
nzdc	racfed	608	26	85.2
nemesis	cfcss	391	66	52.2
nemesis	rasm	421	51	60.9
nemesis	racfed	673	6	85.8
swift		369	178	48.3
CompaSeC		685	0	85.9
select CompaSeC		142	0	32.5

Backup - Statistic Evaluation

Safety vs. Security

Safety:

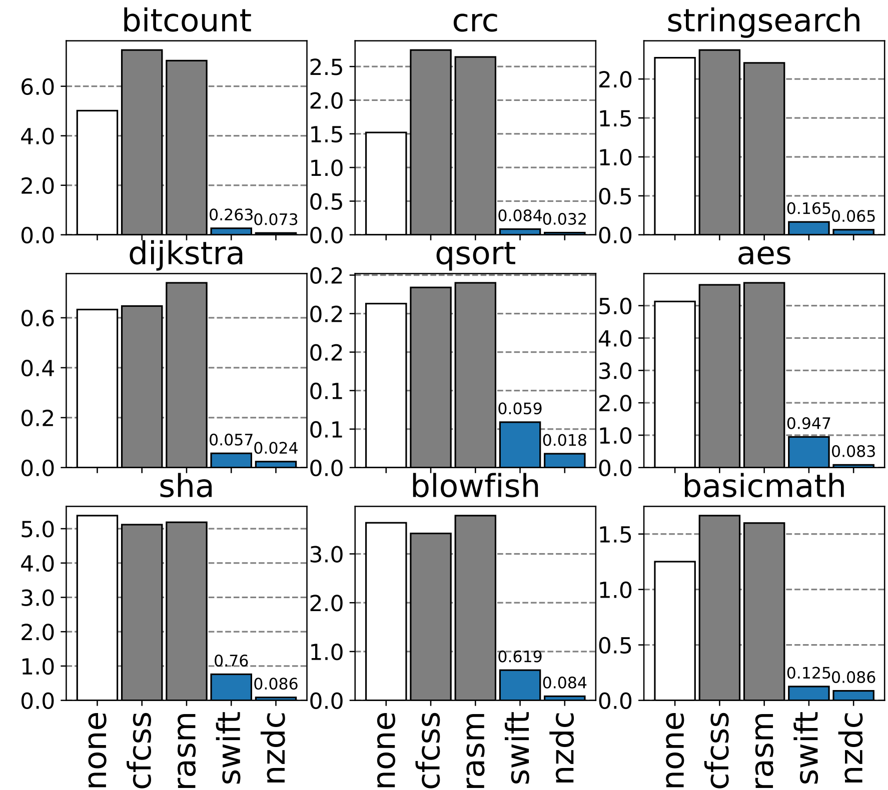
- Fault Tolerance
- Fault Model: **Random**
- Example: Cosmic rays

→ Stochastic Evaluation

Security:

- Fault Detection
- Fault Model: **Targeted**
- Example: Differential Fault Analysis

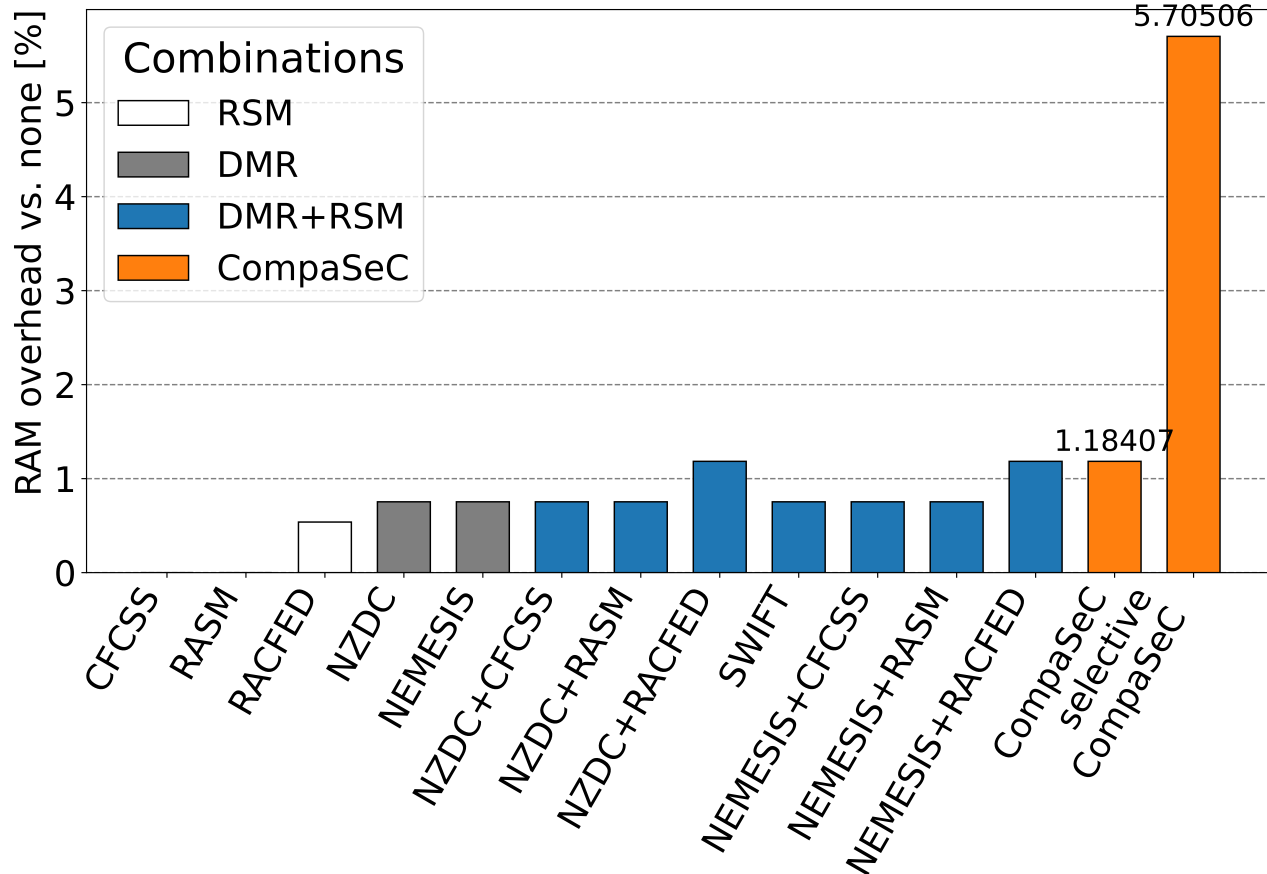
→ Verification



Silent Data Corruption Rates in MiBench programs [10]

Backup - Performance: RAM Size Overhead

Metric: RAM footprint for Secure Boot



Backup - Performance: Memory Traffic Overhead

Metric: Number of RAM memory transactions