CHERITech'26
CONFERENCE

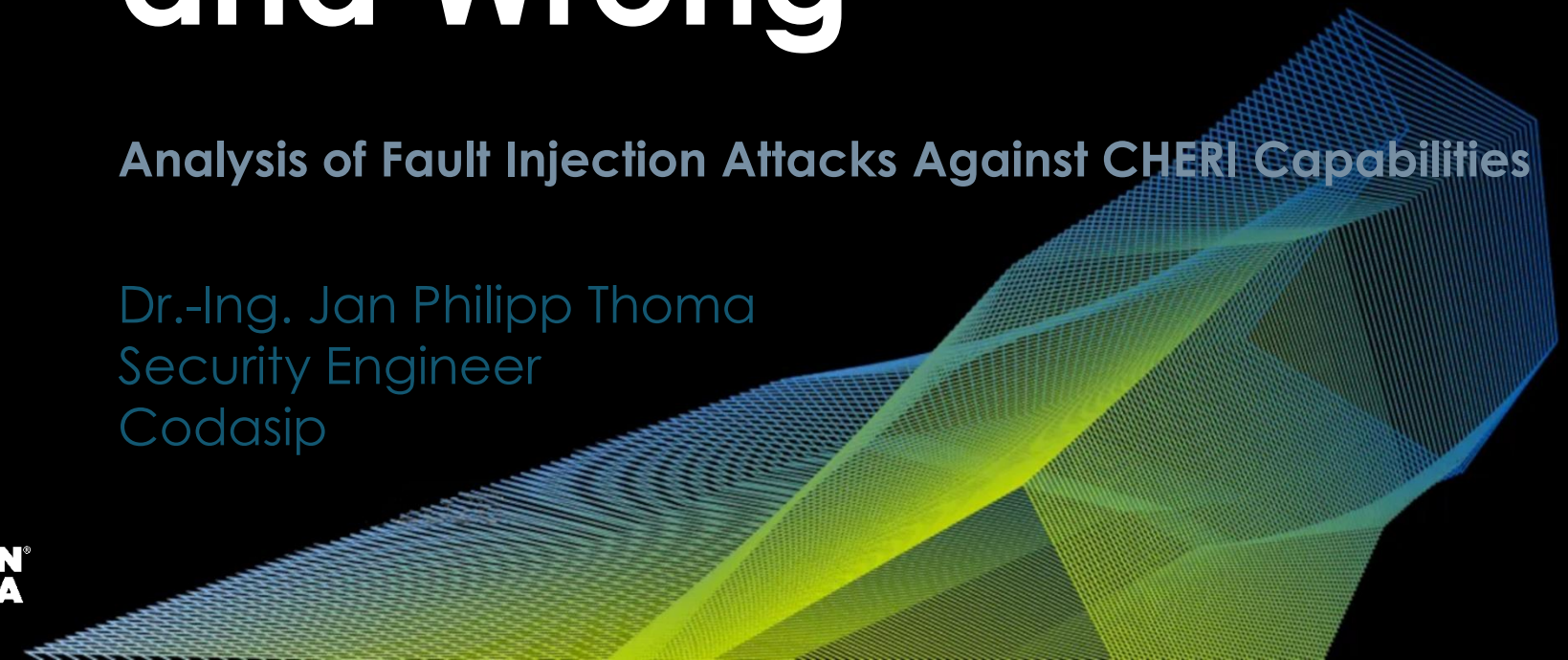
CHERI

12-13 Nov 2026
Munich, DE

Valid, In-Bounds, and Wrong

Analysis of Fault Injection Attacks Against CHERI Capabilities

Dr.-Ing. Jan Philipp Thoma
Security Engineer
Codasip



Located at semi | SEMICON EUROPA

CHERI Threat Model

- CHERI provides **hardware enforced** memory safety
  - Replaces pointers with capabilities
  - Mitigation of memory-safety vulnerabilities
  - Compartmentalization
- CHERI assumes software-based attacker



CHERI Threat Model

31%

of 2019 vulnerabilities would no longer pose a risk on CHERI

≥67%

deterministically mitigated with additional CHERI features

„We conservatively assessed the percentage of vulnerabilities reported […] in 2019 and found that ***approximately 31% would no longer pose a risk*** *[…] on a CHERI system […].* ***With additional features […], the number of deterministically mitigated vulnerabilities would be at least 67%.“***

- Nicolas Joly, Saif ElSherei, Saar Amar – Microsoft Security Response Center

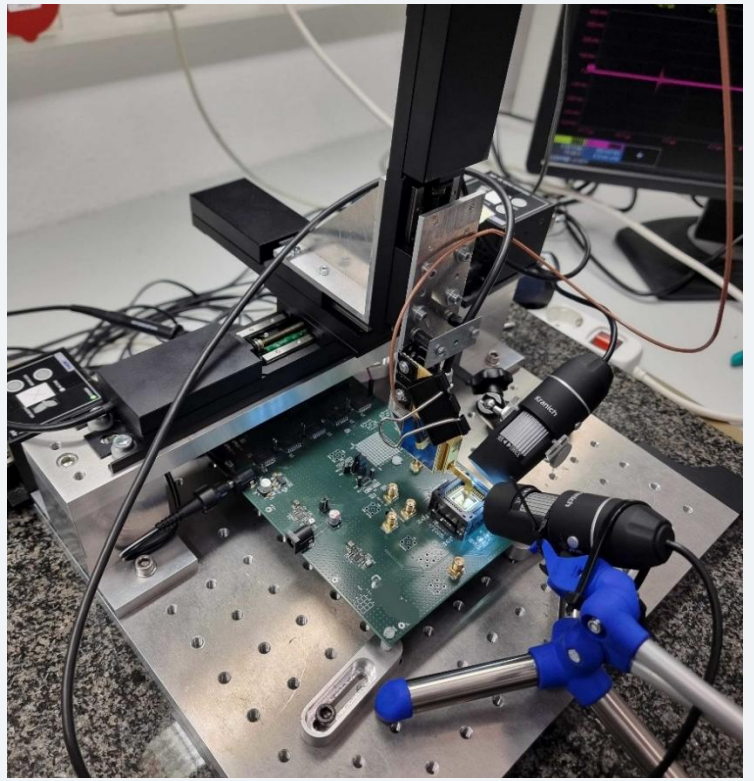
System Security in a Nutshell



AI-generated image

Introduction to Fault Attacks

- Attackers can provoke faults
  - Electromagnetic pulses
  - Clock / voltage glitches
  - Laser fault injection
- Cause errors in the calculation
  - Bit flips in registers or memory
  - Instruction skips



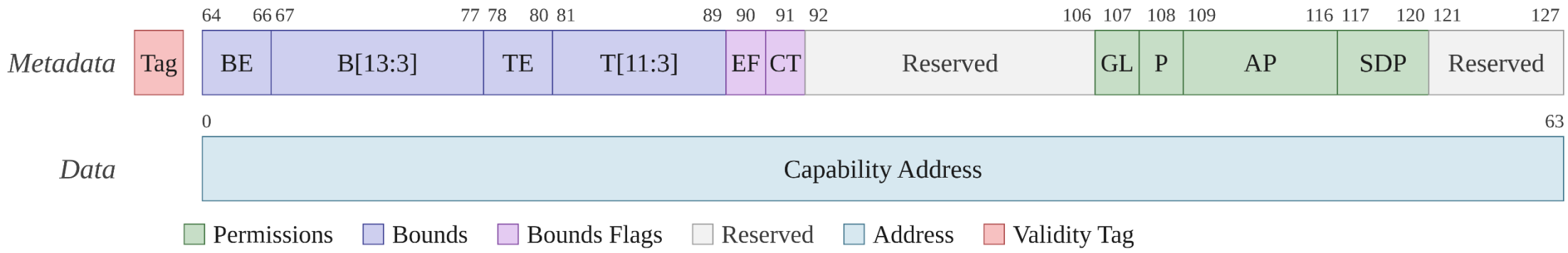
© Uhle et al., Coil-based Detection and Concurrent Error Correction against EMFI, Ruhr University Bochum

Fault Attacks

- Fault attacks give attackers plenty of capabilities
  - Recover Cryptographic secrets
  - Modify control flow
  - Cause data corruption

What happens if CHERI capabilities are targeted?

CHERI Capability Encoding



Regions of Interest:

Tag

1 validity bit

Valid or not

Bounds

Bits 64–89

BE, B[13:3], TE, T[11:3]

Permissions

Bits 107–120

GL, P, AP, SDP

Address

Bits 0–63

Capability address

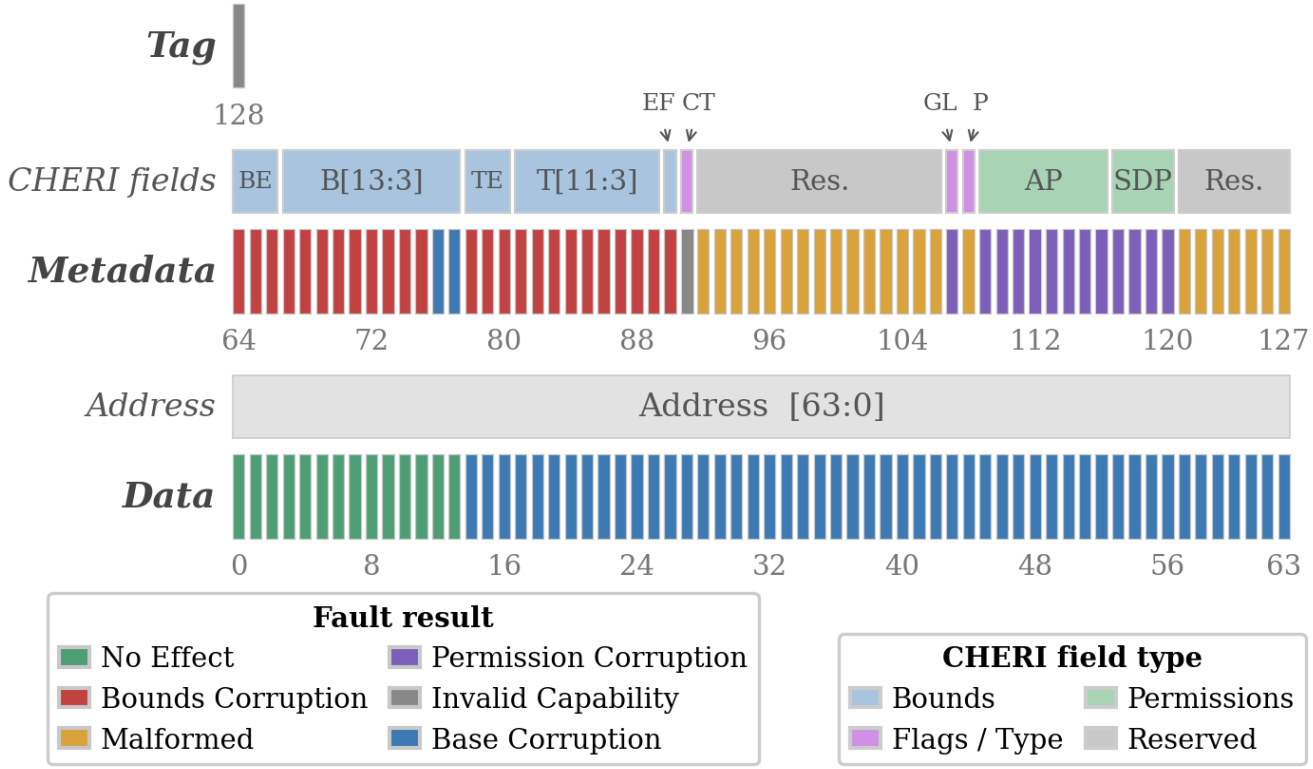
Fault Evaluation

Single-bit faults in capabilities

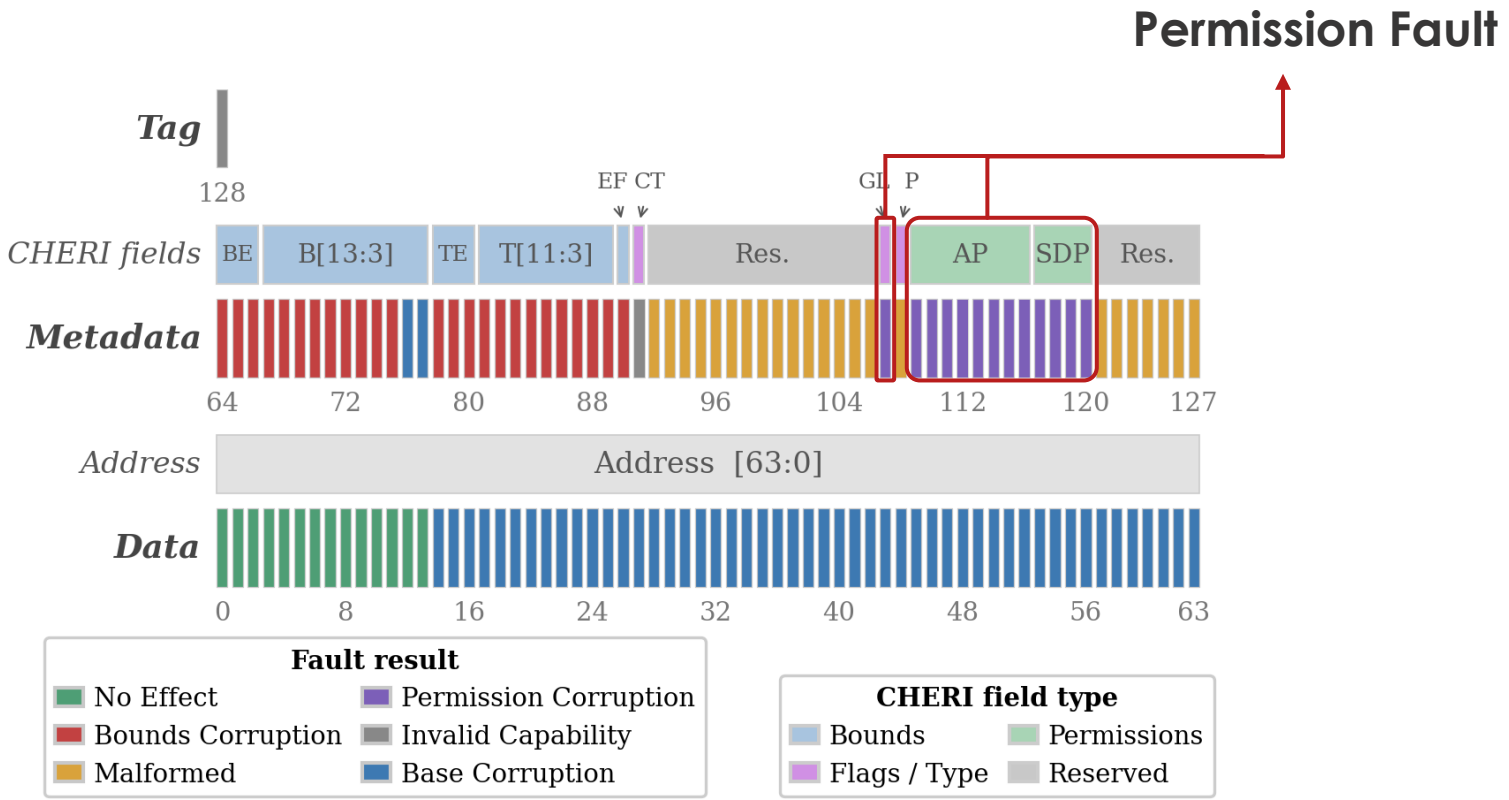
- In memory or in registers
- Independent of any specific microarchitecture

Spike-based simulator

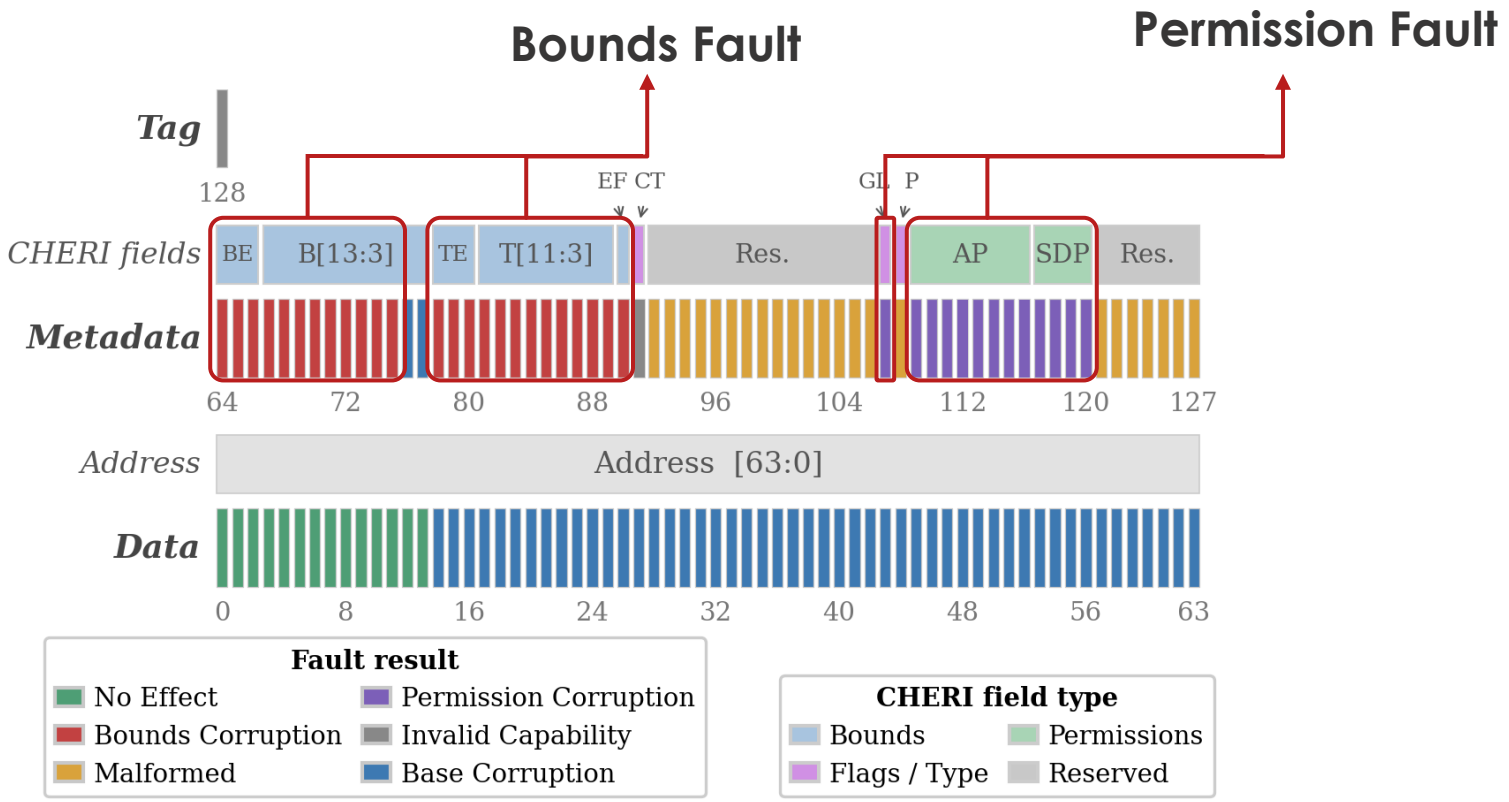
- Fault trigger instructions flip bits in registers
- Trigger instructions have no other effect



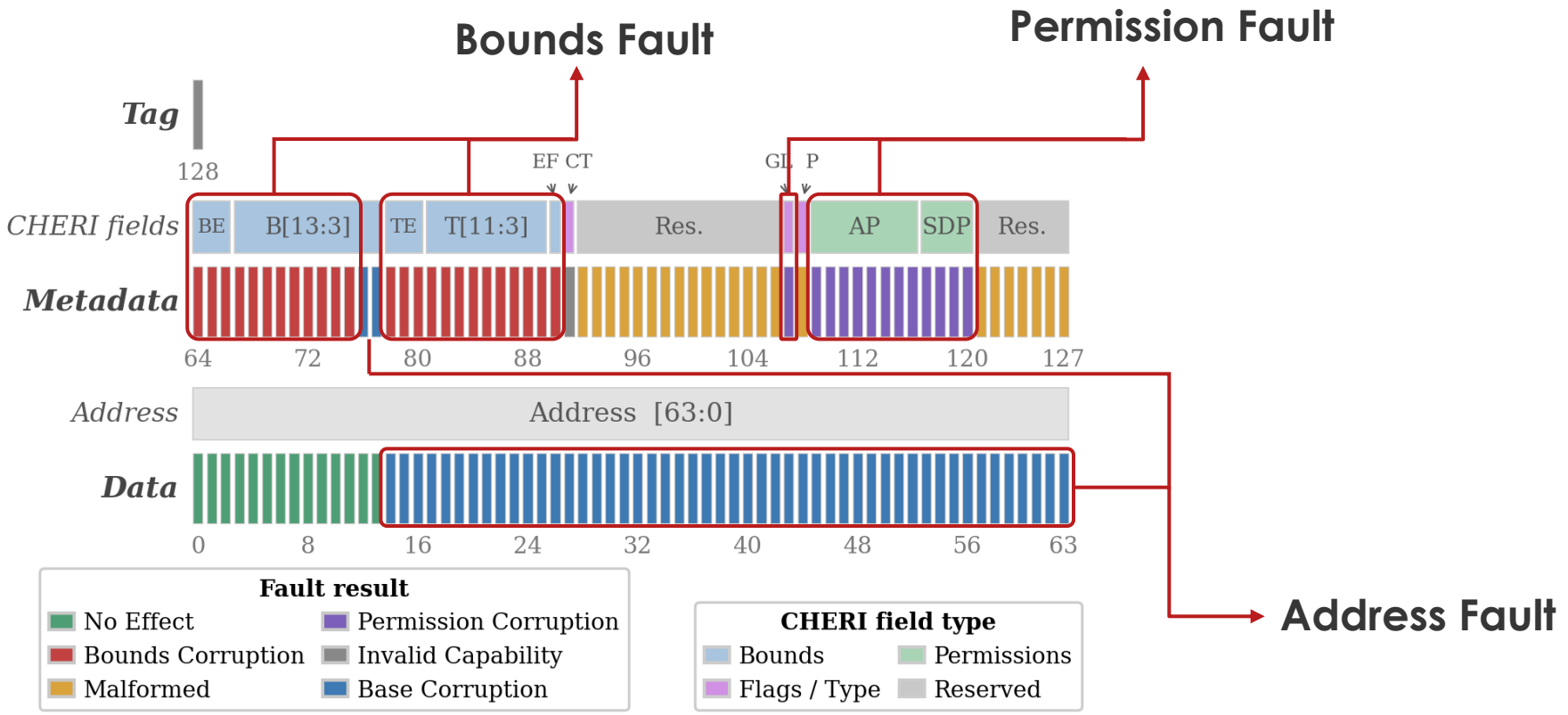
Fault Evaluation



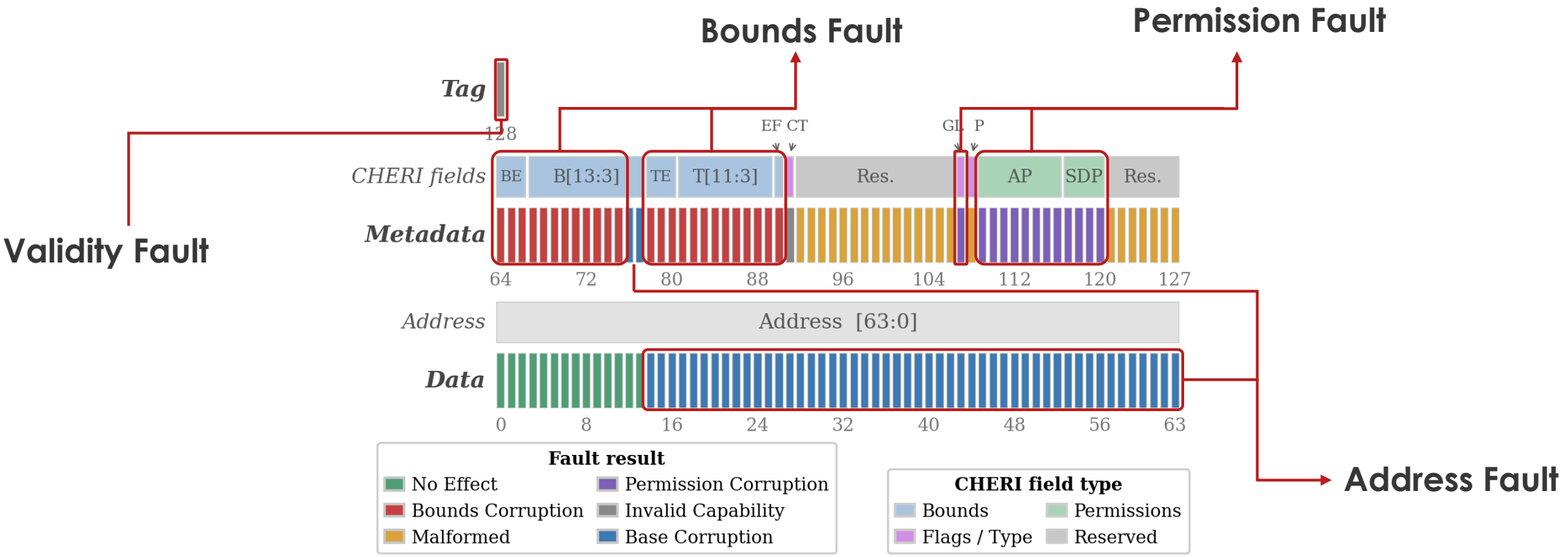
Fault Evaluation



Fault Evaluation



Fault Evaluation



Faults in CHERI RISC-V

Bounds corruption
Increase or decrease top or bottom bounds — the bounds almost always remain valid

Permission corruption
Grant or revoke permissions on a capability

(In)validating corruption
Seal or unseal capabilities; set or unset the tag bit

Base corruption
Move a capability by a fixed offset, length unchanged

Faults in CHERI RISC-V

Bounds corruption
Increase or decrease top or bottom bounds — the bounds almost always remain valid

Permission corruption
Grant or revoke permissions on a capability

(In)validating corruption
Seal or unseal capabilities; set or unset the tag bit

Faulting a capability requires the program to use the widened capability.

The attacker needs both a fault injection and code execution.

Base corruption
Move a capability by a fixed offset, length unchanged

Single-bit fault that changes base and bounds

This is **actually** the most dangerous one

Faults in CHERI RISC-V

- Fault attacks can cause instruction skips
  - Permission changes and bounds reduction are critical instructions in CHERI
  - LLVM often overwrites the source register

DANGEROUS — destination overwrites the source

```
YBNDSW c10, c10, len
```

A skipped instruction leaves the wide capability sitting in c10, and the program keeps using it.

SAFE — destination differs from the source

```
YBNDSW c11, c10, len
```

The narrowed capability lands in c11, so a skip cannot widen what the program already holds.

Example: Fault Injection + Buffer Overflow

- A faulted top bound enables buffer overflows with CHERI
  - **`fill()`** writes four bytes past the buffer
  - With bounds intact, CHERI traps the write
  - **Fault the top bound** and the write passes
  - No exception is raised; the privileged path runs

overflow.c

```
volatile uint32_t privileged = 0;   // adjacent flag
char              buf[BUFSIZE];  // input buffer

static void fill(char *buf, uint32_t n)
{
    for (uint32_t i = 0; i < n; i++)
        buf[i] = 0xFF;
}

fill(buf, BUFSIZE + 4);   // 4 bytes past the end

if (privileged != 0) {
    return SUCCESS;
} else {
    return FAIL;
}
```

Example: Fault Injection + Buffer Overflow

- A faulted top bound enables buffer overflows with CHERI
  - **`fill()`** writes four bytes past the buffer
  - With bounds intact, CHERI traps the write
  - **Fault the top bound** and the write passes
  - No exception is raised; the privileged path runs

overflow.c

```
volatile uint32_t privileged = 0;   // adjacent flag
char              buf[BUFSIZE];  // input buffer

static void fill(char *buf, uint32_t n)
{
    for (uint32_t i = 0; i < n; i++)
        buf[i] = 0xFF;
}

fill(buf, BUFSIZE + 4);   // 4 bytes past the end

if (privileged != 0) {
    return SUCCESS;
} else {
    return FAIL;
}
```

Enlarge Bounds

Example: Fault Injection + Buffer Overflow

- A faulted top bound enables buffer overflows with CHERI
  - **`fill()`** writes four bytes past the buffer
  - With bounds intact, CHERI traps the write
  - **Fault the top bound** and the write passes
  - No exception is raised; the privileged path runs

CAUTION!

CHERI may omit traditional protections like stack canaries, so exploits like this might get easier.

overflow.c

```
volatile uint32_t privileged = 0;   // adjacent flag
char              buf[BUFSIZE];  // input buffer

static void fill(char *buf, uint32_t n)
{
    for (uint32_t i = 0; i < n; i++)
        buf[i] = 0xFF;
}

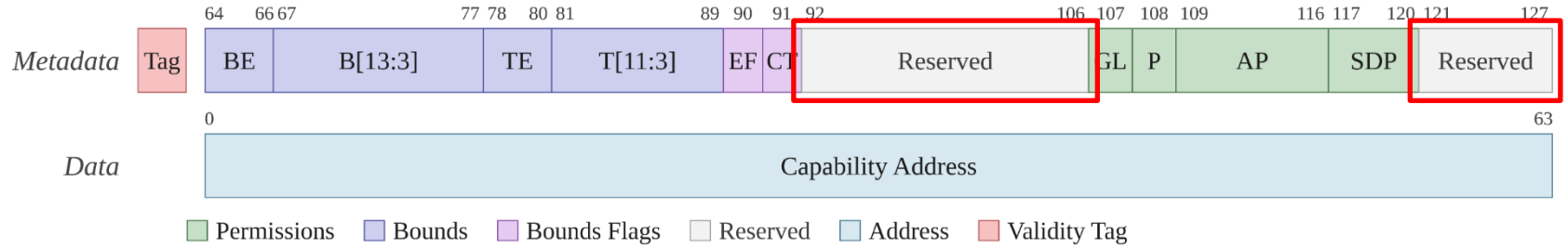
fill(buf, BUFSIZE + 4);   // 4 bytes past the end

if (privileged != 0) {
    return SUCCESS;
} else {
    return FAIL;
}
```



Countermeasures

- Fault attacks are usually prevented by adding redundancy
  - Error correction or detection (ECC, CRC32)
  - Usually complicated – where do you store the additional bits?
  - CHERI makes implementation easier!



Countermeasures

- Compiler / Software-based mitigations
  - Prevent cases like `YBNDSW c10, c10, len`
    - Bounds reduction instruction
    - If skipped, `c10` will remain large
    - **Instead:** Use a different target register
  - Double check critical operations
    - Verify capability modifications
    - Run critical code multiple times

The Bigger Picture

CHERI limits what a fault attack can achieve.

CONVENTIONAL — data can become a pointer

`data value → usable pointer`

In a conventional architecture, corruption that redirects an ordinary data value into an address or control-flow operand may immediately create a usable pointer.

CHERI — data cannot become authority

`data value → no tag, no access`

In CHERI, ordinary data does not carry a valid capability tag and therefore cannot directly become memory or control-flow authority (multiple faults required).

Conclusion

- Fault attacks can affect CHERI capabilities
  - Multi step attacks can craft malicious capabilities
  - Single fault attacks can shift capability ranges
- … but we can fix it!
  - Software / Compiler-based countermeasures for instruction skips
  - Error detection / correction against bit flips
  - The encoding space is already there!

CHERITech'26 CONFERENCE

12–13 Nov 2026
Munich, DE

Located at semi | SEMICON EUROPA

CHERI

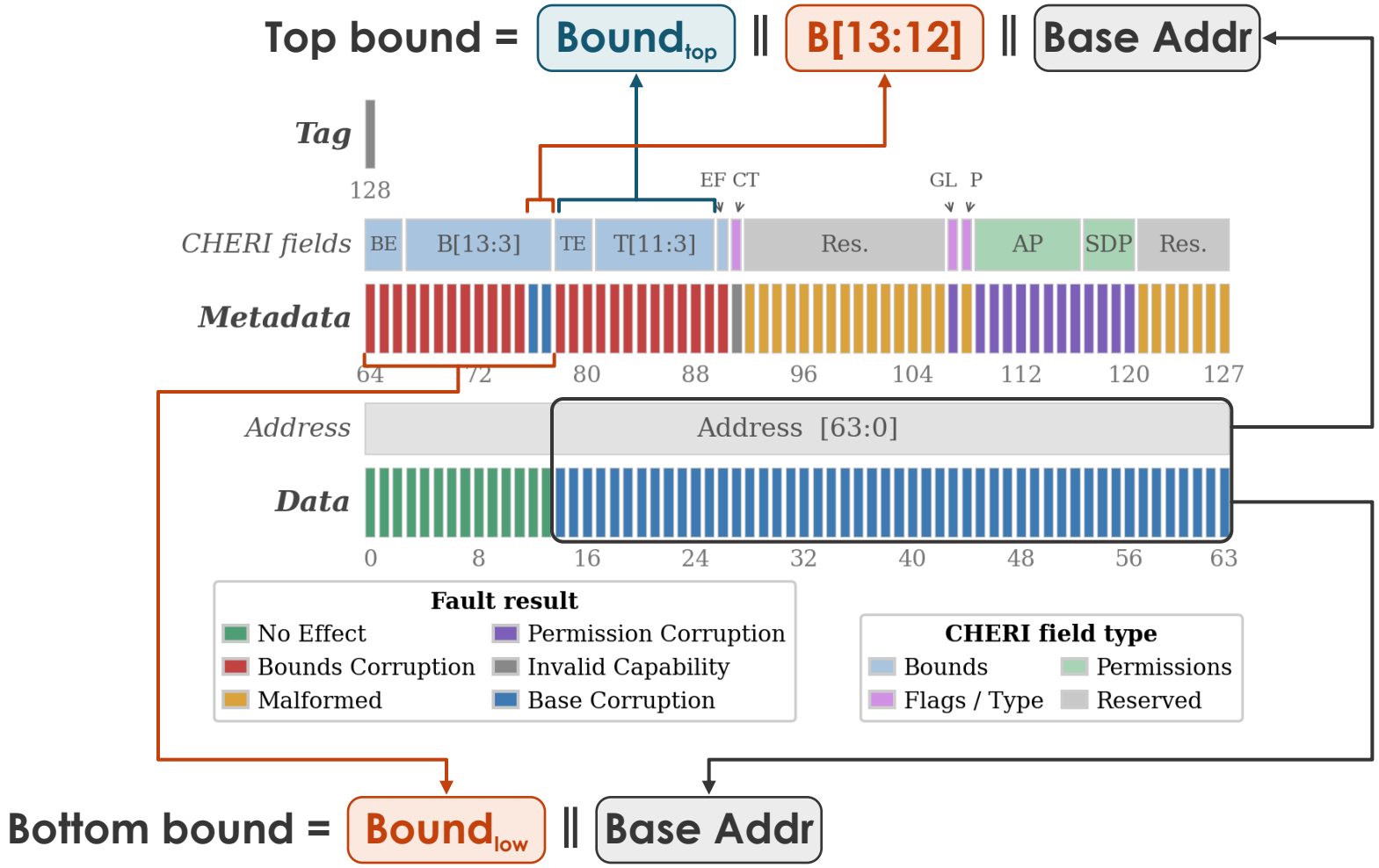
THANK YOU

Contact: jan.thoma@codasip.com

CHERI™ and the CHERI logo are protected trademarks. Use of these marks is subject to the CHERI Alliance Trademark Policy – https://cheri-alliance.org/legal-and-policies/trademark/

This work © 2026 by CHERI Alliance is licensed under CC BY-SA 4.0 (Creative Commons Attribution-ShareAlike 4.0 International) – https://creativecommons.org/licenses/by-sa/4.0/

Fault Evaluation



Example 2: Single-fault Attack

- One bit flip exposes the secret byte
  - **`scbndsi`** narrows **`buf`** to a 64-byte capability
  - The secret byte sits 2^{14} bytes into the buffer
  - **A single-bit fault in bit 14** shifts **`reduced_cap`** by 2^{14}, landing directly on the secret byte

single_fault.c

```
char* reduced_cap;

// narrow buf to a 64-byte capability
asm volatile (
    "scbndsi %[out], %[in], 64\n\t"
    : [out]"=C"(reduced_cap)
    : [in]  "C"((char*) buf)
);

buf[1<<14] = SECRET_BYTE;

if (*reduced_cap == SECRET_BYTE) {
    return 7;
}

return 1;
```