SAMVA: Static Analysis for Multi-Fault Attack Paths Determination

Antoine Gicquel Univ Rennes, Inria, CNRS, IRISA, France
Damien Hardy Univ Rennes, Inria, CNRS, IRISA, France
Karine Heydemann Thales | Sorbonne Université / LIP6, France
Erven Rohou Univ Rennes, Inria, CNRS, IRISA, France

28/09/2023 - JAIF 2023

Fault injection attacks

Adversary goals

- Leak critical data
- Break cryptographic properties
- Take over a device

Fault injection for control-flow hijacking

- Execute authentication code
- Avoid countermeasures

```c
BOOL verifyPIN() {
    g_authenticated = 0;

    if(g_ptc > 0) {
        if(byteArrayCompare(...) == 1) {
            g_ptc = 3;
            g_authenticated = 1;
            return 1;
        } else {
            g_ptc--;
            return 0;
        }
    }

    return 0;
}
```

Example with a source code in C

Fault injection attacks

Adversary goals

- Leak critical data
- Break cryptographic properties
- Take over a device

Fault injection for control-flow hijacking

- Execute authentication code
- Avoid countermeasures

```c
BOOL verifyPIN() {
    g_authenticated = 0;

    if(g_ptc > 0) {
        if(byteArrayCompare(...) == 1) {
            g_ptc = 3;
            g_authenticated = 1;
            return 1;
        } else {
            g_ptc--;
            return 0;
        }
    }

    return 0;
}
```

Example with a source code in C

Fault injection attacks

Adversary goals

- Leak critical data
- Break cryptographic properties
- Take over a device

Fault injection for control-flow hijacking

- Execute authentication code
- Avoid countermeasures

```c
BOOL verifyPIN() {
    g_authenticated = 0;

    if(g_ptc > 0) {
        if(byteArrayCompare(...) == 1) {
            g_ptc = 3;
            g_authenticated = 1;
            return 1;
        } else {
            g_ptc--;
            return 0;
        }
    }

    return 0;
}
```

Example with a source code in C

Fault injection attacks

Adversary goals

- Leak critical data
- Break cryptographic properties
- Take over a device

Fault injection for control-flow hijacking

- Execute authentication code
- Avoid countermeasures

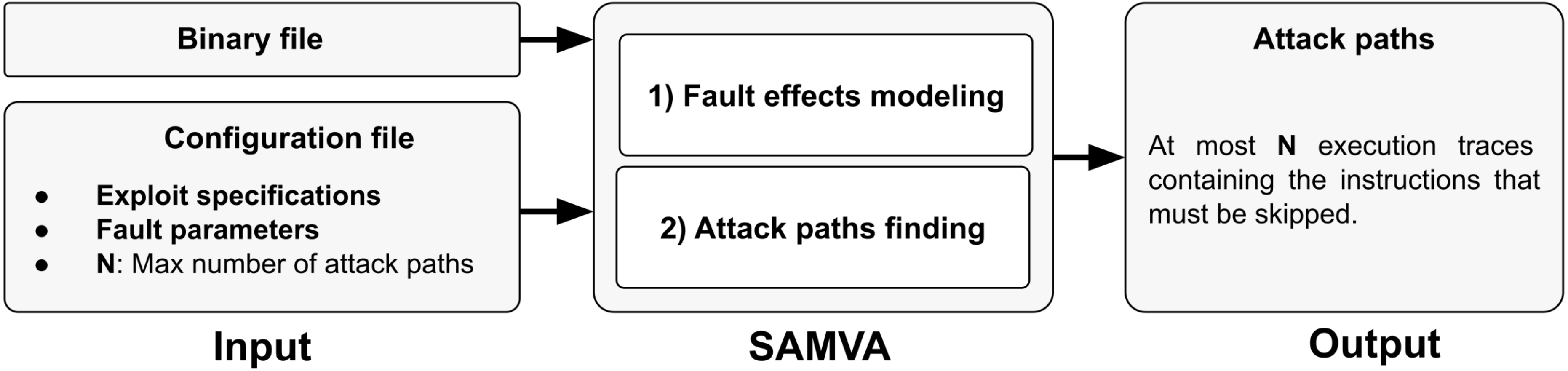
How to assess the robustness of a software against fault injection?

```c
BOOL verifyPIN() {
    g_authenticated = 0;

    if(g_ptc > 0) {
        if(byteArrayCompare(...) == 1) {
            g_ptc = 3;
            g_authenticated = 1;
            return 1;
        } else {
            g_ptc--;
            return 0;
        }
    }

    return 0;
}
```

Example with a source code in C

Contribution: SAMVA

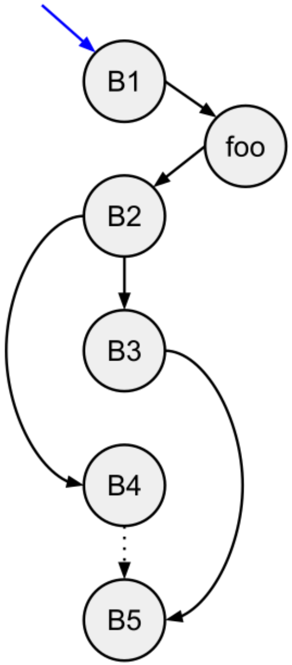
- **Static analysis method:** Finding attack paths semi-automatically
- **Multiple instruction-skip fault model**: Faults with a variable width
- **Accessibility exploit model:** Reach and avoid specified regions of code binary



Fault model: multiple instruction-skip

Fault Parameters

```
B2:
0x10510:    str    r0, [fp, #-12]
0x10514:    ldr    r3, [fp, #-12]
0x10518:    cmp    r3, #4
0x1051c:    bgt    10530
B3:
0x10520:    ldr    r3, [fp, #-8]
0x10524:    add    r3, r3, #1
0x10528:    str    r3, [fp, #-8]
0x1052c:    b      1053c
B4:
0x10530:    ldr    r3, [fp, #-8]
0x10534:    sub    r3, r3, #1
0x10538:    str    r3, [fp, #-8]
...
```



Fault model: multiple instruction-skip

Fault Parameters

- Minimal width of a fault **fw_min**, e.g. = 2
- Maximal width of a fault **fw_max**, e.g. = 4

B2:

fw_min

~~0x10510: str r0, [fp, #-12]~~

~~0x10514: ldr r3, [fp, #-12]~~

0x10518: cmp r3, #4

0x1051c: ~~**bgt 10530**~~

B3:

0x10520: ldr r3, [fp, #-8]

0x10524: add r3, r3, #1

0x10528: str r3, [fp, #-8]

fw_max

~~**0x1052c: b 1053c**~~

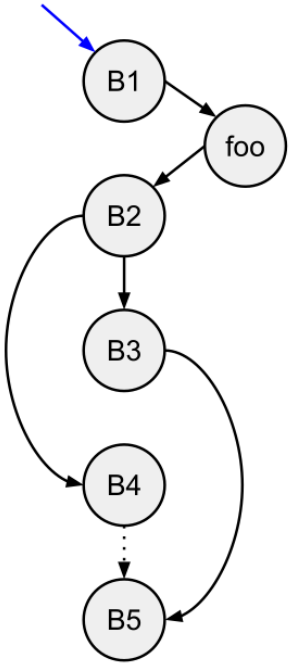
~~**B4:**~~

~~0x10530: ldr r3, [fp, #-8]~~

~~0x10534: sub r3, r3, #1~~

~~0x10538: str r3, [fp, #-8]~~

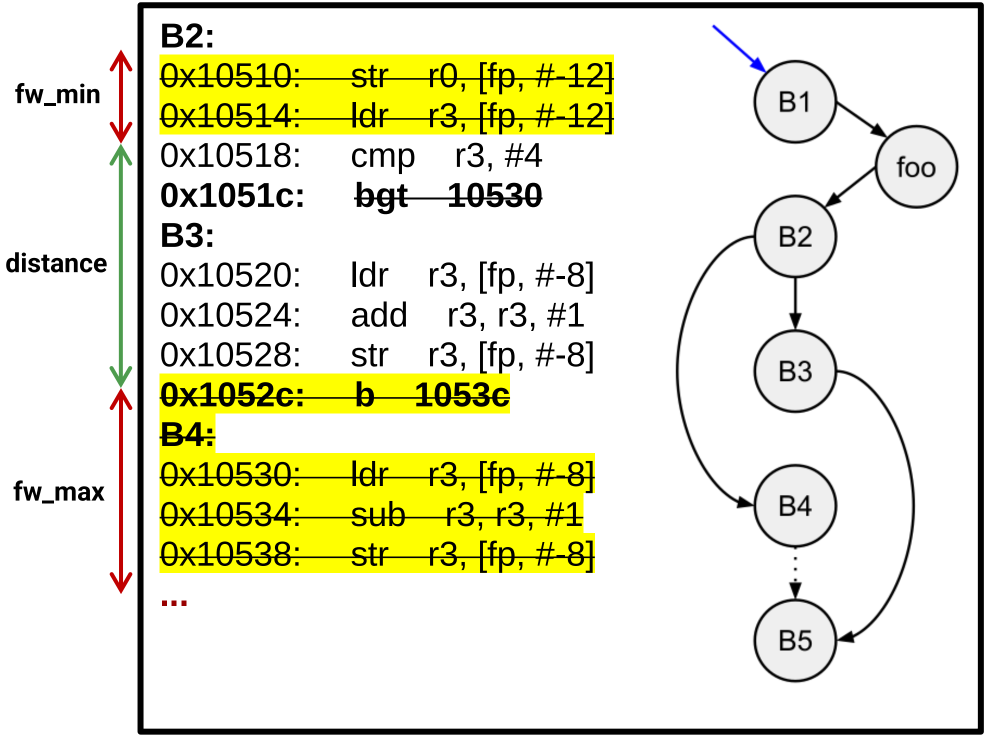
...



Fault model: multiple instruction-skip

Fault Parameters

- Minimal width of a fault
  fw_min, e.g. = 2
- Maximal width of a fault
  fw_max, e.g. = 4
- Minimal distance between two faults
  fw_min_dist, e.g. = 5



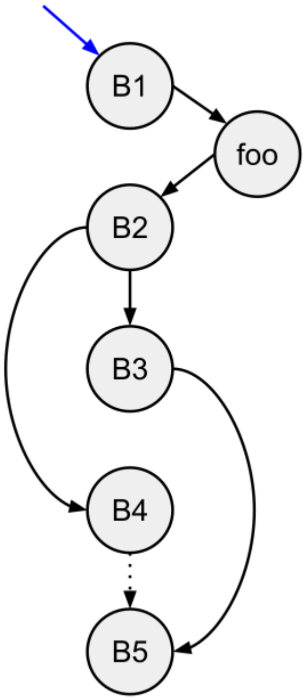
Fault effects

Fault Parameters

- Minimal width of a fault
  fw_min, e.g. = 2
- Maximal width of a fault
  fw_max, e.g. = 4
- Minimal distance between two faults
  fw_min_dist, e.g. = 5

Predictable path w.o. data-flow analysis

```
B2:
0x10510:    str    r0, [fp, #-12]
0x10514:    ldr    r3, [fp, #-12]
0x10518:    cmp    r3, #4
0x1051c:    bgt    10530
B3:
0x10520:    ldr    r3, [fp, #-8]
0x10524:    add    r3, r3, #1
0x10528:    str    r3, [fp, #-8]
0x1052c:    b      1053c
B4:
0x10530:    ldr    r3, [fp, #-8]
0x10534:    sub    r3, r3, #1
0x10538:    str    r3, [fp, #-8]
...
```



Fault effects

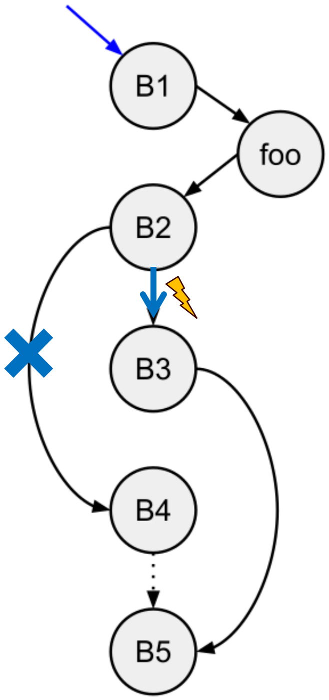
Fault Parameters

- Minimal width of a fault
  fw_min, e.g. = 2
- Maximal width of a fault
  fw_max, e.g. = 4
- Minimal distance between two faults
  fw_min_dist, e.g. = 5

Predictable path w.o. data-flow analysis

- **Conditional jumps:** systematically skipped

```
B2:
0x10510:    str    r0, [fp, #-12]
0x10514:    ldr    r3, [fp, #-12]
0x10518:    cmp    r3, #4
0x1051c:    bgt    10530
B3:
0x10520:    ldr    r3, [fp, #-8]
0x10524:    add    r3, r3, #1
0x10528:    str    r3, [fp, #-8]
0x1052c:    b      1053c
B4:
0x10530:    ldr    r3, [fp, #-8]
0x10534:    sub    r3, r3, #1
0x10538:    str    r3, [fp, #-8]
...
```



Fault effects

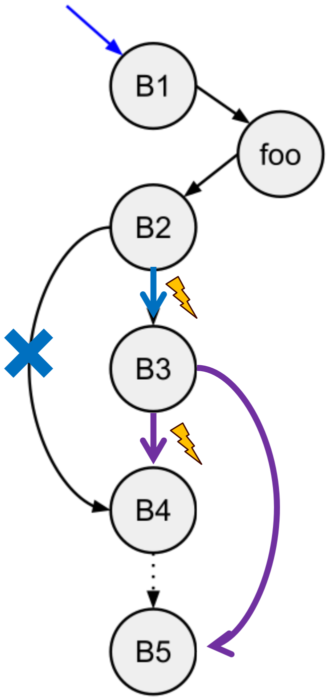
Fault Parameters

- Minimal width of a fault
  fw_min, e.g. = 2
- Maximal width of a fault
  fw_max, e.g. = 4
- Minimal distance between two faults
  fw_min_dist, e.g. = 5

Predictable path w.o. data-flow analysis

- **Conditional jumps:** systematically skipped
- **Unconditional jumps:** executed or skipped

```
B2:
0x10510:    str    r0, [fp, #-12]
0x10514:    ldr    r3, [fp, #-12]
0x10518:    cmp    r3, #4
0x1051c:    bgt    10530
B3:
0x10520:    ldr    r3, [fp, #-8]
0x10524:    add    r3, r3, #1
0x10528:    str    r3, [fp, #-8]
0x1052c:    b      1053c
B4:
0x10530:    ldr    r3, [fp, #-8]
0x10534:    sub    r3, r3, #1
0x10538:    str    r3, [fp, #-8]
...
```



Fault effects

Fault Parameters

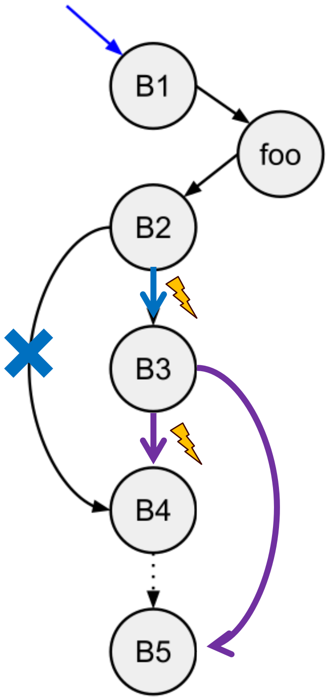
- Minimal width of a fault
  fw_min, e.g. = 2
- Maximal width of a fault
  fw_max, e.g. = 4
- Minimal distance between two faults
  fw_min_dist, e.g. = 5

Predictable path w.o. data-flow analysis

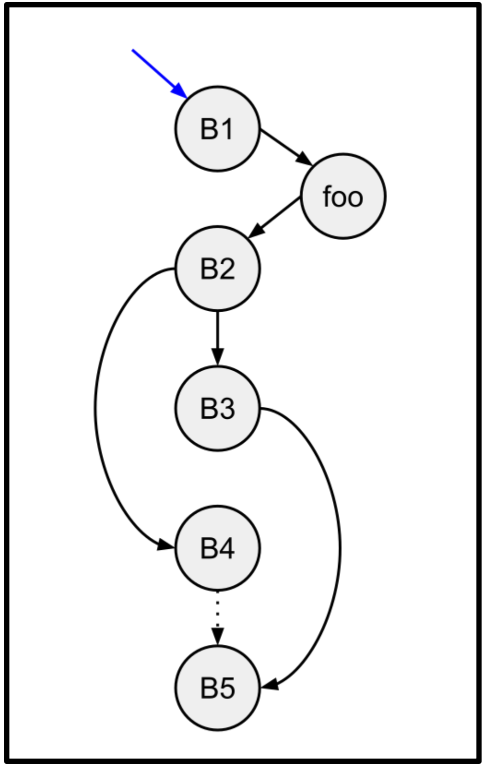
- **Conditional jumps:** systematically skipped
- **Unconditional jumps:** executed or skipped

Reflect these effects on the CFG

```
B2:
0x10510:   str   r0, [fp, #-12]
0x10514:   ldr   r3, [fp, #-12]
0x10518:   cmp   r3, #4
0x1051c:   bgt   10530
B3:
0x10520:   ldr   r3, [fp, #-8]
0x10524:   add   r3, r3, #1
0x10528:   str   r3, [fp, #-8]
0x1052c:   b     1053c
B4:
0x10530:   ldr   r3, [fp, #-8]
0x10534:   sub   r3, r3, #1
0x10538:   str   r3, [fp, #-8]
...
```

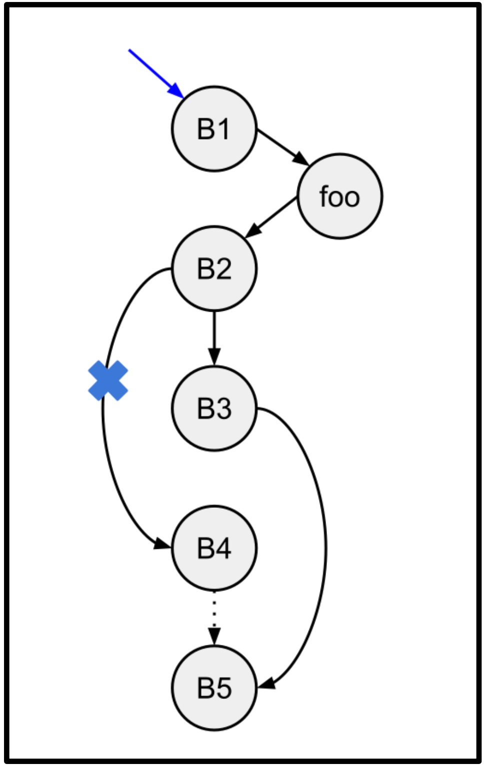


Fault effects modeling



1) Retrieve the initial control flow graph (CFG)

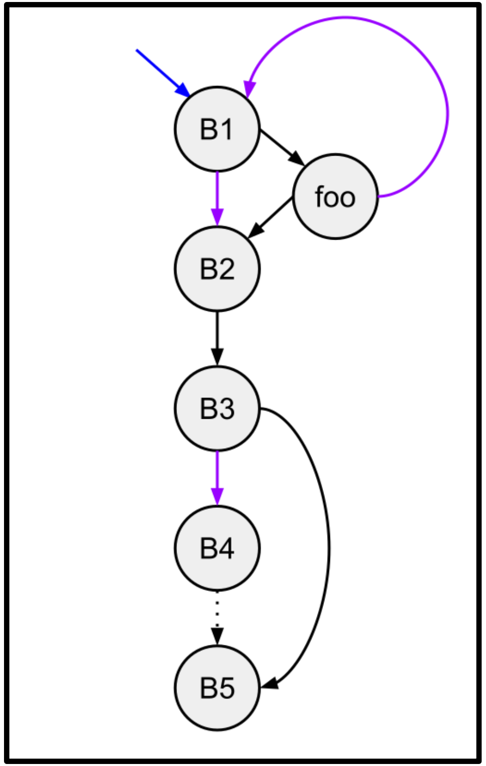
Fault effects modeling



1) Retrieve the initial control flow graph (CFG)

2) Remove edges: BB ending with conditional jump

Fault effects modeling

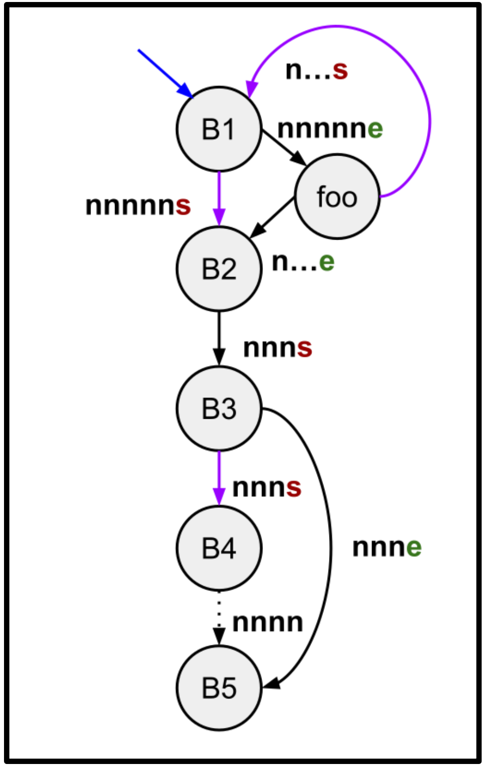


1) Retrieve the initial control flow graph (CFG)

2) Remove edges: BB ending with conditional jump

3) Add new edges: BB ending with unconditional jump

Fault effects modeling



1) Retrieve the initial control flow graph (CFG)

2) Remove edges: BB ending with conditional jump

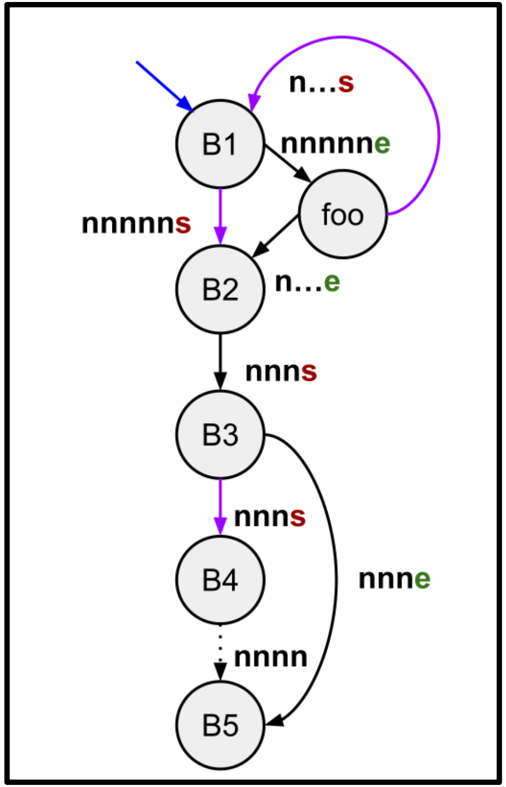
3) Add new edges: BB ending with unconditional jump

4) Annotate edges with a sequence of "types"

- One type per instruction of the source BB
  - ***execute*** (**e**): must be executed
  - ***skip*** (**s**): must be skipped
  - ***neutral*** (**n**): can be either skipped or executed

Attack paths finding

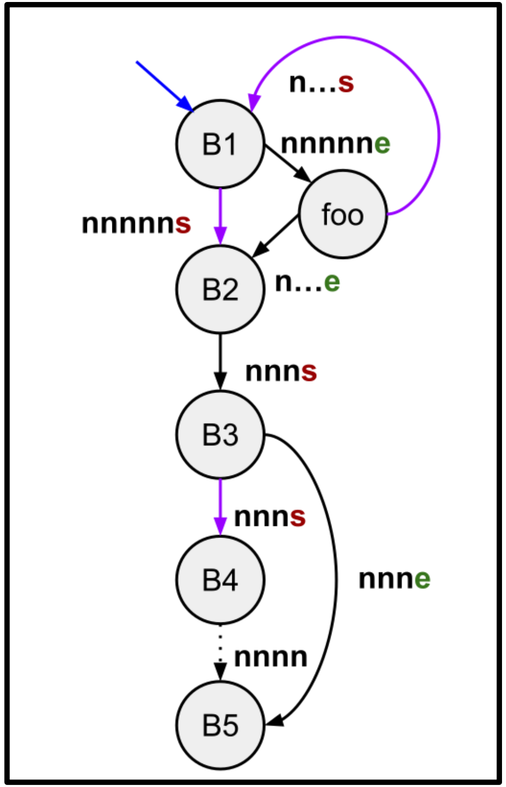
- **Build a set of candidate paths**
  Example: Targeted basic blocks = [foo; B4]
  B1 – foo – B2 – B3 – B4



Attack paths finding

- **Build a set of candidate paths**
  Example: Targeted basic blocks = [foo; B4]
  B1 – foo – B2 – B3 – B4

- **Build an execution trace**
  - List of tuples <address, type>
  - **nnnnne + nnnnnnnne + nnnns + nnns**

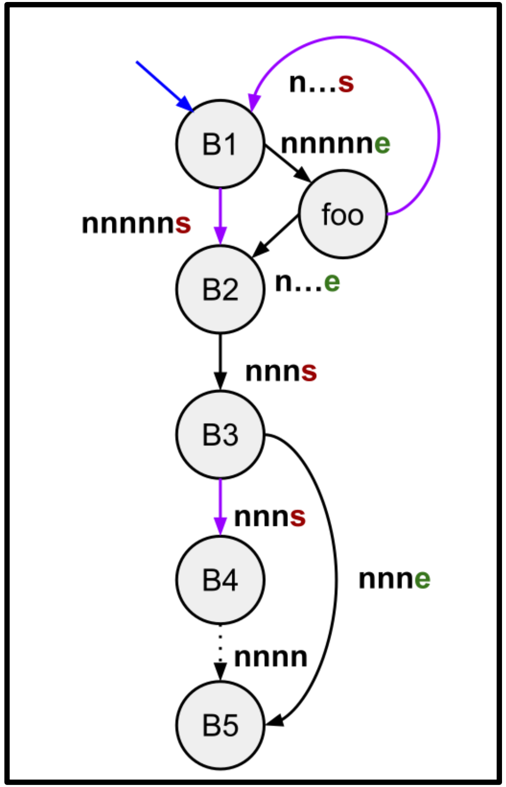


Attack paths finding

- **Build a set of candidate paths**
  Example: Targeted basic blocks = [foo; B4]
  B1 – foo – B2 – B3 – B4

- **Build an execution trace**
  - List of tuples <address, type>
  - **nnnnne + nnnnnnnne + nnnns + nnns**

- **Fault positioning**
  Determine N set of faults {(position, width)} making the path feasible



Fault positioning algorithm

0x104F8:	execute
0x104FC:	neutral
0x10500:	neutral
0x10504:	skip
0x10508:	skip
0x1050C:	neutral
0x10510:	execute
0x10514:	neutral
0x10518:	neutral
0x1051C:	skip
0x10520:	skip
0x10524:	neutral
0x10528:	neutral
0x1052C:	skip
0x10530:	neutral
0x10534:	neutral
0x10538:	neutral

Execution trace example
(*fw_min* = 4, *fw_max* = 8
***fw_min_dist* = 4)**

- **Conditions** for set of faults (a.k.a solution) to be valid
  - All instructions typed **skip** are covered by a fault
  - No instruction typed **execute** is covered by a fault
  - Faults widths ∈ [***fw_min***, ***fw_max***]
  - Distances between two faults ≥ ***fw_min_dist***

Fault positioning algorithm

0x104F8:	execute
0x104FC:	neutral
0x10500:	neutral
0x10504:	skip
0x10508:	skip
0x1050C:	neutral
0x10510:	execute
0x10514:	neutral
0x10518:	neutral
0x1051C:	skip
0x10520:	skip
0x10524:	neutral
0x10528:	neutral
0x1052C:	skip
0x10530:	neutral
0x10534:	neutral
0x10538:	neutral

Execution trace example
(*fw_min* = 4, *fw_max* = 8
***fw_min_dist* = 4)**

- **Conditions** for set of faults (a.k.a solution) to be valid
  - All instructions typed **skip** are covered by a fault
  - No instruction typed **execute** is covered by a fault
  - Faults widths ∈ [***fw_min***, ***fw_max***]
  - Distances between two faults ≥ ***fw_min_dist***
- Solutions are built **incrementally** with a **backtracking** approach
- At the end, we obtain **N execution traces along with the position and the width of the faults for each**

Experimentation: Objective & Setup

Experimentation objective

Ensure that the attack paths found by SAMVA are really effective

gem5

Modified version of gem5 allowing the simulation of instruction-skips

Experimentation: Objective & Setup

Experimentation objective

Ensure that the attack paths found by SAMVA are really effective

Modified version of gem5 allowing the simulation of instruction-skips

- **Benchmark: VerifyPIN suite**
  - FISCC [Dureuil et al. SAFECOMP 2016]
  - 8 implementations with increasing level of countermeasures
- **Tested with a large set of fault parameters**
  - **3366 different faults parameters** per binary
  - At most **30 attack paths per fault parameters**
  - Simulate attacks until one succeed

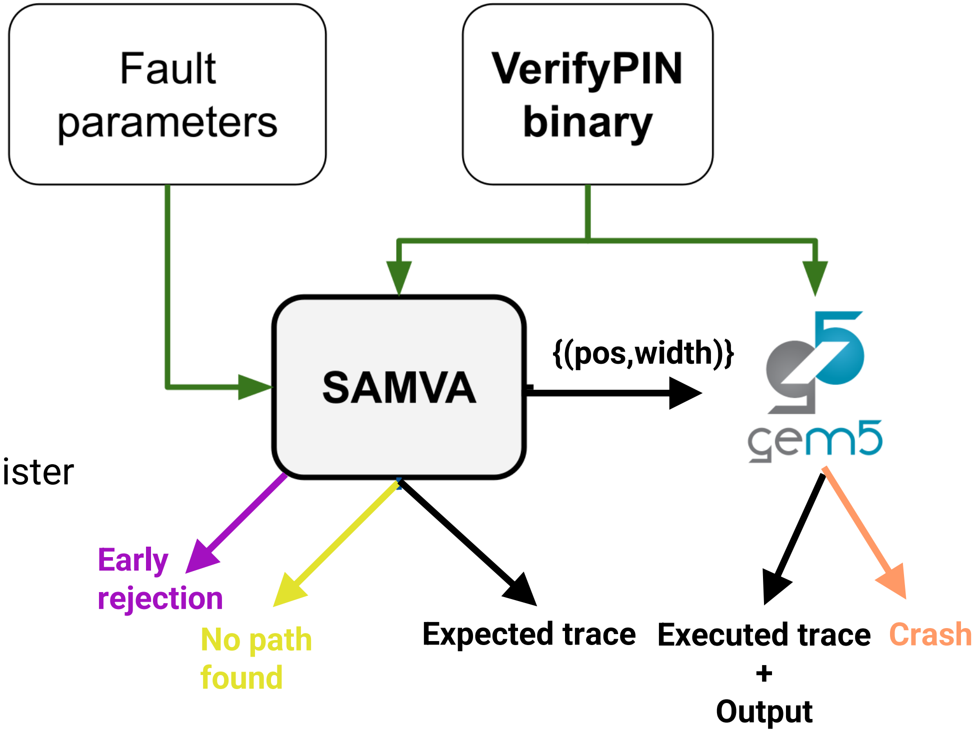
Validation methodology

Before Simulation

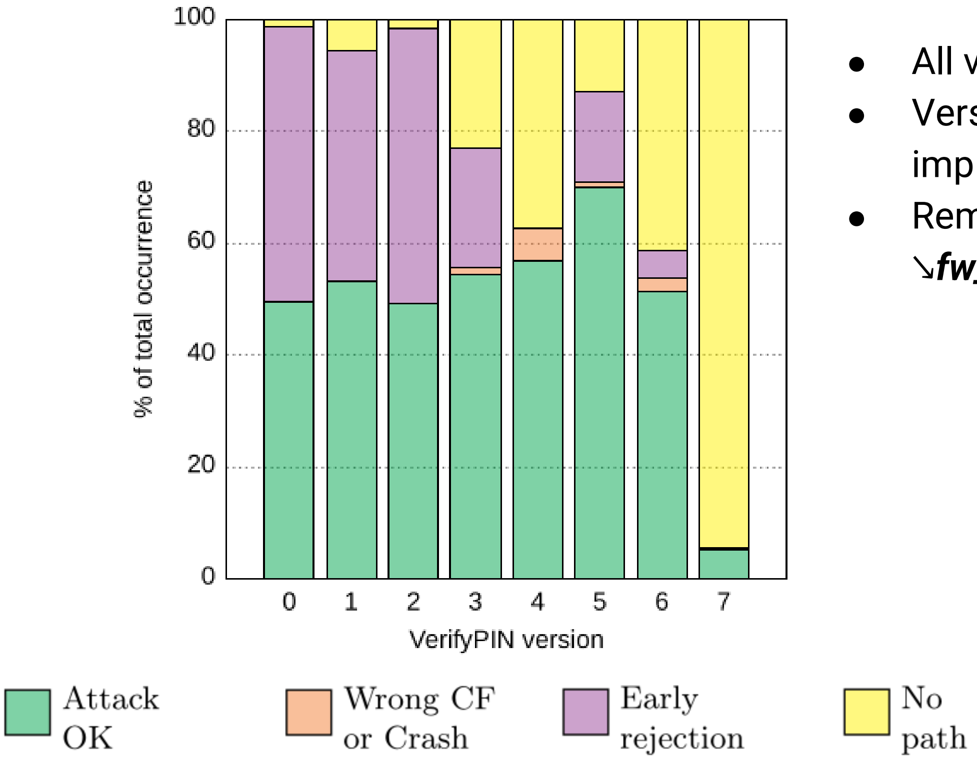
- **Early rejection**
  Fault parameters not suited for the binary
- **No Path found**

After Simulation

- **Execution crashes due to faults**
  e.g. Illegal load from address stored in a register
- **Validated attack**
  Authenticated && Traces are equal



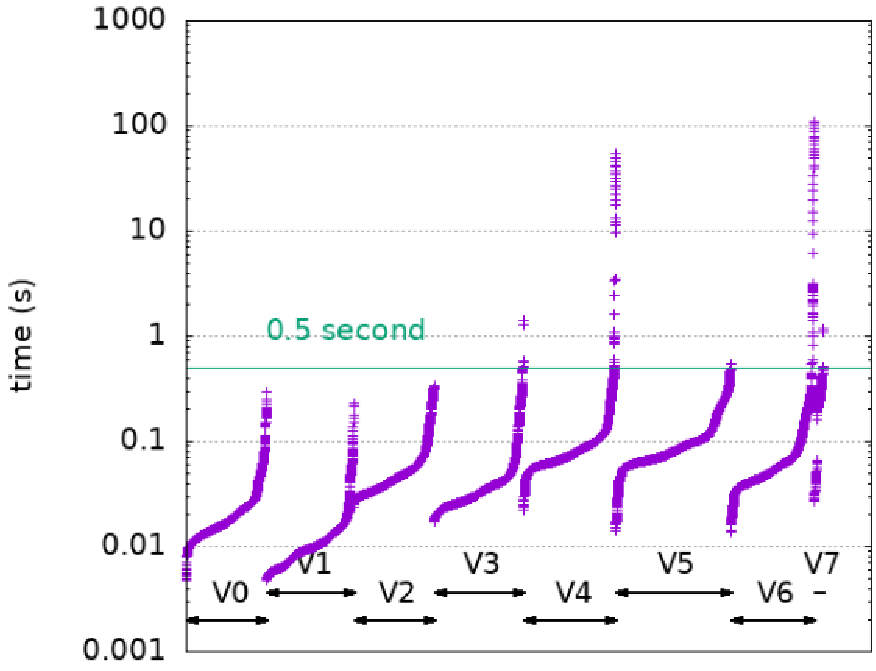
Classification of attack path searches



- All versions are vulnerable
- Versions **V4** and **V7** are the most robust implementations
- Remark: Facilitate fault positioning by ↘***fw_min***, ↗***fw_max*** and ↘***fw_min_dist***

→ **SAMVA is able to find numerous attacks paths for all versions**

Analysis time

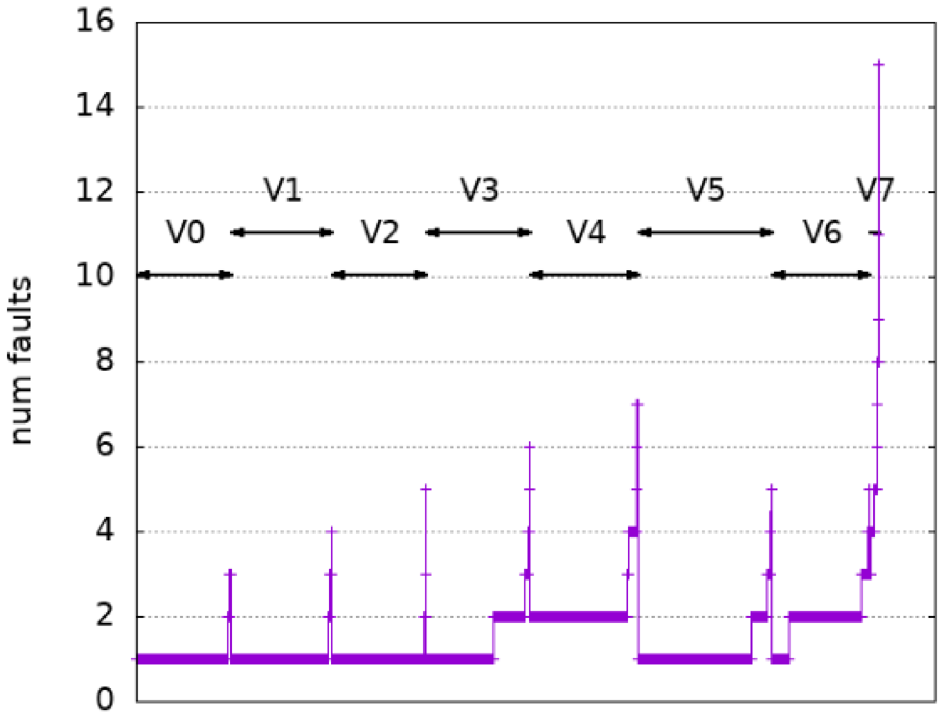


Time needed to generate the paths for each considered fault parameter

- Time taken to find **up to 30 attack paths**
- Xeon Gold 5218 2.3 GHz - 32 physical cores

→ Most of the results are under the threshold of half a second

Number of faults needed for each successful attack



- Only **1 fault**: V0 to V3, V5 and V6
- At least **2 faults**: V4
- At least **3 faults**: V7

→ **SAMVA is able to find attack paths with multiple faults when it is required**

Conclusion

- **SAMVA**
  Framework based only on static analysis for determining attack paths in presence of multiple instruction-skip faults

- **Evaluation**
  Attack paths found for all the 1 + 7 hardened versions of PIN code verification

- **Future work**
  - Extension of supported fault models
  → "instruction replay"
  - Make the link with fault injection platform

- **More details in the COSADE 23 paper !**