Inside SAMVA:
illustration of the core algorithms

Sébastien Michelland, Antoine Gicquel

FORWARD Seminar @ Campus Cyber — Paris, March 30th, 2026

FRANCE 2030 PROGRAMME DE RECHERCHE CYBERSÉCURITÉ
Inria
anr
PROGRAMME DE transfert AUX campus cyber

Project context

SAMVA [Gic+23]
Static Analysis tool for Multi-fault Vulnerability Assessment
https://gitlab.inria.fr/samva-project

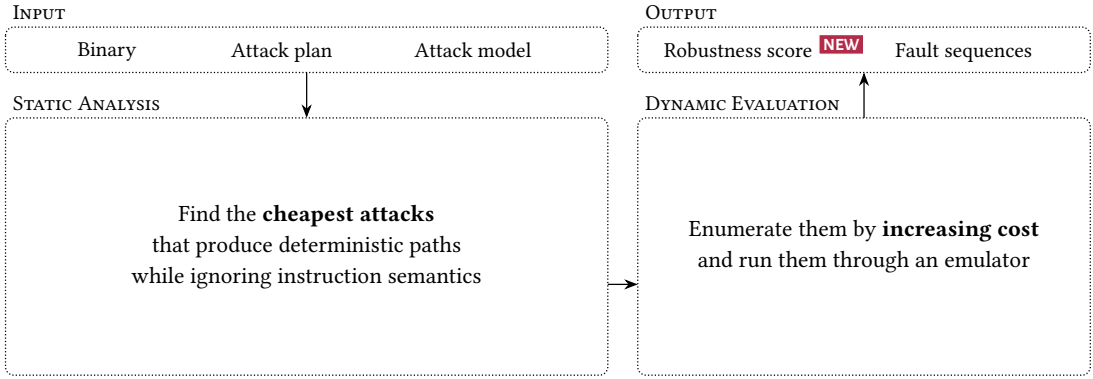
Part of Antoine Gicquel's Ph.D [Gic24]

I will mostly present the latest version of SAMVA.

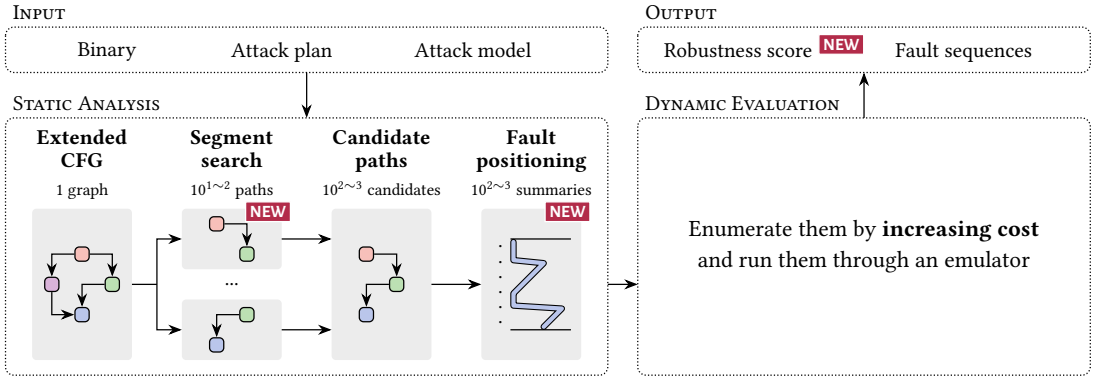
- **Original version:** finds attack paths in a best-effort manner, some heuristics
- **Current version:** deterministically finds the best attacks, reports robustness score
- My contributions are marked as NEW

1. Overview, inputs and outputs

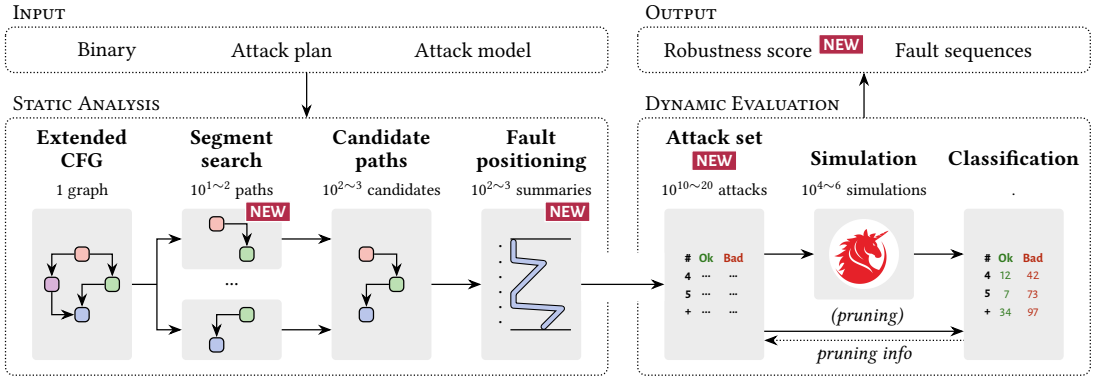
Overview of the analysis



Overview of the analysis



Overview of the analysis



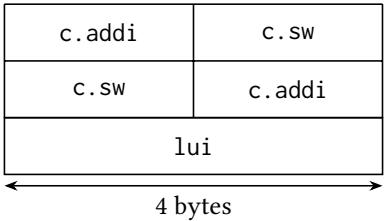
Support control-flow fault models

We are dealing with *long* and *repeated* faults.

Instruction skip

```
c.addi  sp, sp, -32
c.sw    ra, 28(sp)
c.sw    s0, 24(sp)
c.addi  s0, sp, 32
lui     a5, 0x20001
```

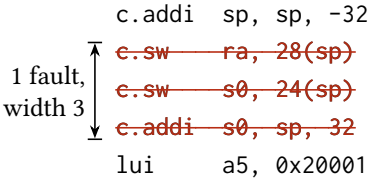
Word replay



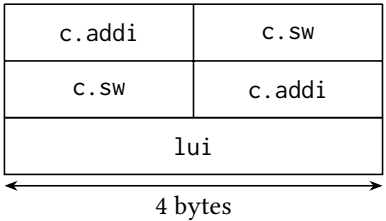
Support control-flow fault models

We are dealing with *long* and *repeated* faults.

Instruction skip



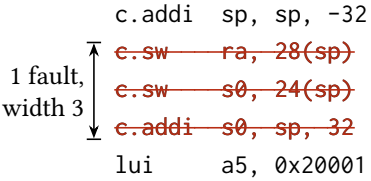
Word replay



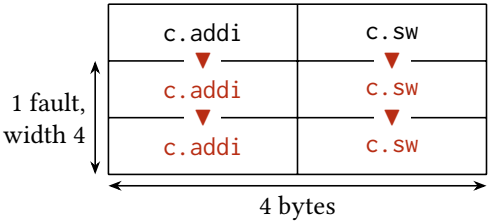
Support control-flow fault models

We are dealing with *long* and *repeated* faults.

Instruction skip



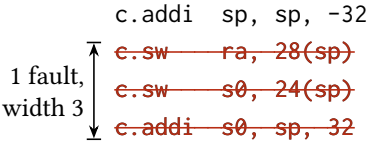
Word replay



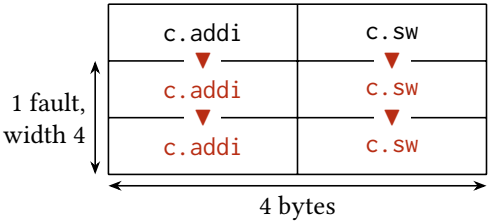
Support control-flow fault models

We are dealing with *long* and *repeated* faults.

Instruction skip



Word replay



Model is defined by:

- Possible widths $[\mathsf{MinWidth}, \mathsf{MaxWidth}]$ (in instructions)
- $\mathsf{MinDistance}$: number of instructions left alone between two faults (reloading time)

Inputs and outputs

Inputs

- **Binary**: ARM or RISC-V (so far)

```
verifypin_v5_cv32e40p.elf
```

- **Attack model**:

  Instruction skip / Word replay
  MinWidth, MaxWidth, MinDistance

- **Attack plan** (*simplified)

> *"Go from entry of* `verifyPIN` *to* `auth=true`
> *without triggering the countermeasure"*

Inputs and outputs

Inputs

- **Binary**: ARM or RISC-V (so far)

```
verifypin_v5_cv32e40p.elf
```

- **Attack model**:

  Instruction skip / Word replay
  MinWidth, MaxWidth, MinDistance

- **Attack plan** (*simplified)

> *"Go from entry of* `verifyPIN` *to* `auth=true` *without triggering the countermeasure"*

Outputs

- **Fault sequences**:

  Ex: $[(t = 2, w = 4), (t = 8, w = 2), ...]$

- **Complete fault set**: NEW

Faults	Paths	Success	Crash
2	1	6	1
3	3	10	106
4	3	22	90
More	...	88	218

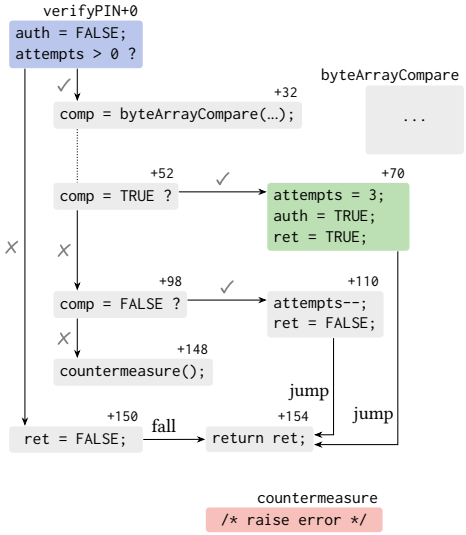
- **Robustness score**: NEW

  $\min(\text{successful sequence length}) - 1$

2. Searching for attack paths

Front-end CFG construction

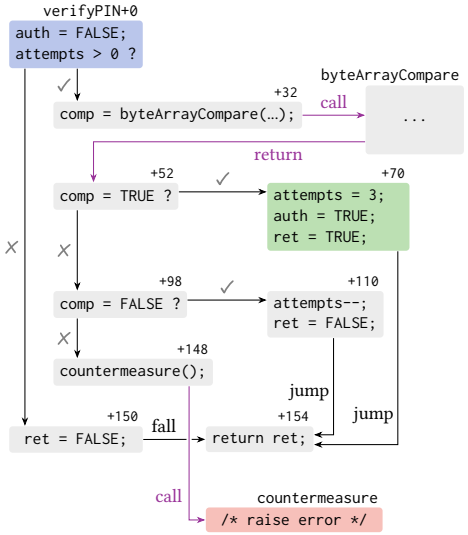
Get CFG from headless Ghidra, then:



Front-end CFG construction

Get CFG from headless Ghidra, then:

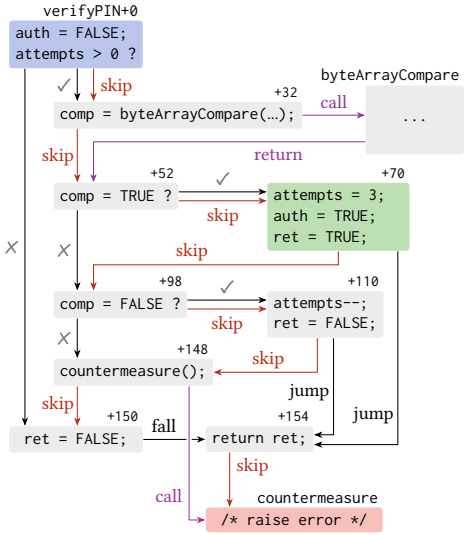
1. Add **call** and **return** edges



Front-end CFG construction

Get CFG from headless Ghidra, then:

1. Add **call** and **return** edges
2. Add **skip** edges for skipping terminators

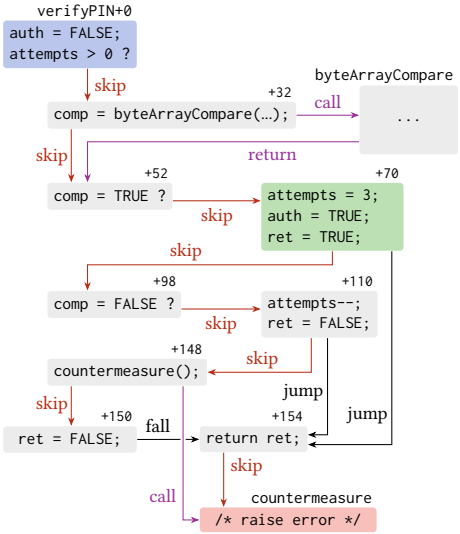


Front-end CFG construction

Get CFG from headless Ghidra, then:

1. Add **call** and **return** edges
2. Add **skip** edges for skipping terminators
3. Remove conditional edges

Only deterministic paths, so no conditionals.
(Data-flow manipulation: future work!)



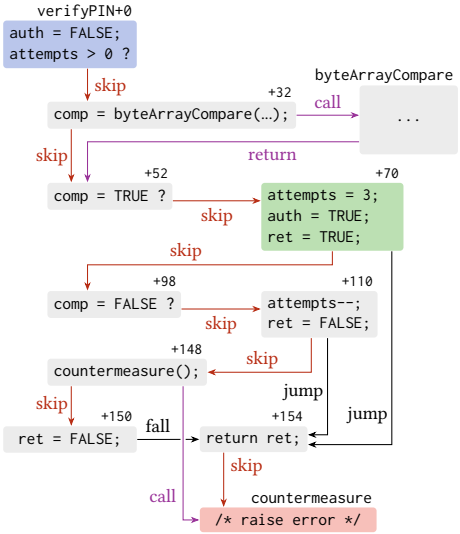
Front-end CFG construction

Get CFG from headless Ghidra, then:

1. Add **call** and **return** edges
2. Add **skip** edges for skipping terminators
3. Remove conditional edges

Only deterministic paths, so no conditionals.
(Data-flow manipulation: future work!)

- Attack start
- Attack end
- Banned blocks



Graph traversal for attack paths

Standard graph traversal: Attack start Attack end Banned blocks

Example and constrained instruction trace (neutral, skip, execute):

Two aspects addressed later:

- We allow loops, so there are infinitely many paths, we need to be **lazy**;
- And so we want to find paths by **increasing cost** (number of faults).

3. Fault positioning

The positioning problem

Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Instructions:	e	n	s	n	s	n	n	s	s	e	n	n	n	n	s	n	s	e	e	s	n

The positioning problem

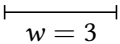
Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Instructions:	e	n	s	n	s	n	n	s	s	e	n	n	n	n	s	n	s	e	e	s	n

Width $\in [2, 5]$
MinDistance $= 2$

The positioning problem

Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Instructions:	e	n	s	n	s	n	n	s	s	e	n	n	n	n	s	n	s	e	e	s	n

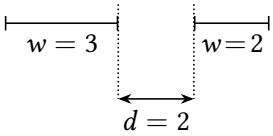
Width $\in [2, 5]$
MinDistance $= 2$



The positioning problem

Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Instructions:	e	n	s	n	s	n	n	s	s	e	n	n	n	n	s	n	s	e	e	s	n

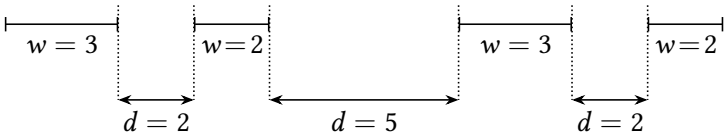
Width $\in [2, 5]$
MinDistance $= 2$



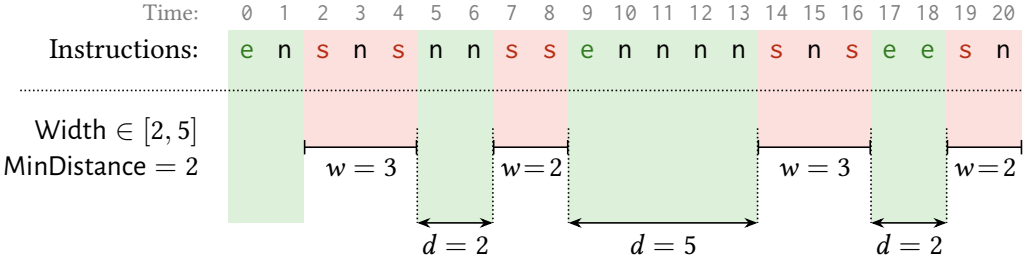
The positioning problem

Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Instructions:	e	n	s	n	s	n	n	s	s	e	n	n	n	n	s	n	s	e	e	s	n

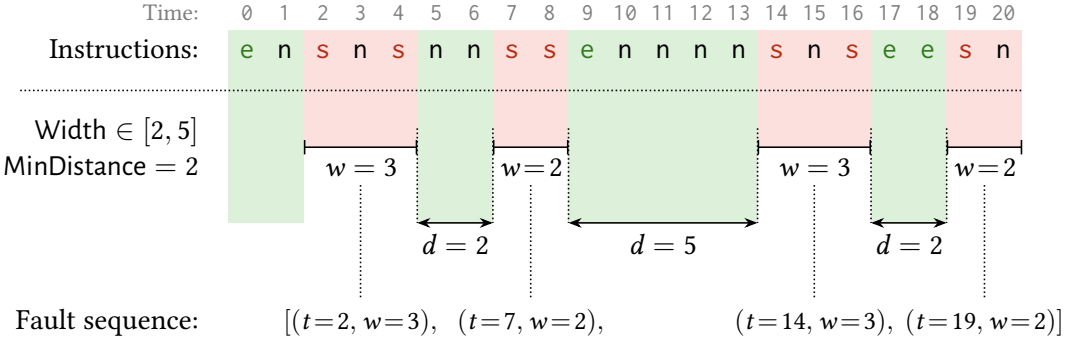
Width $\in [2, 5]$
MinDistance $= 2$



The positioning problem



The positioning problem



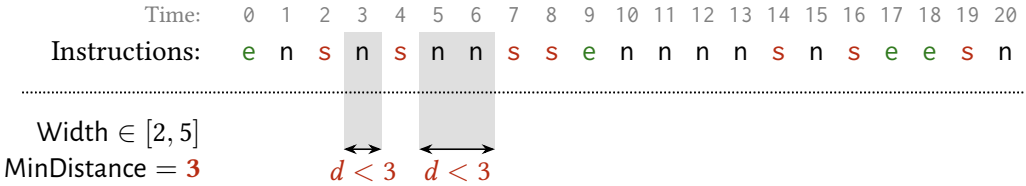
The positioning problem

Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Instructions:	e	n	s	n	s	n	n	s	s	e	n	n	n	n	s	n	s	e	e	s	n

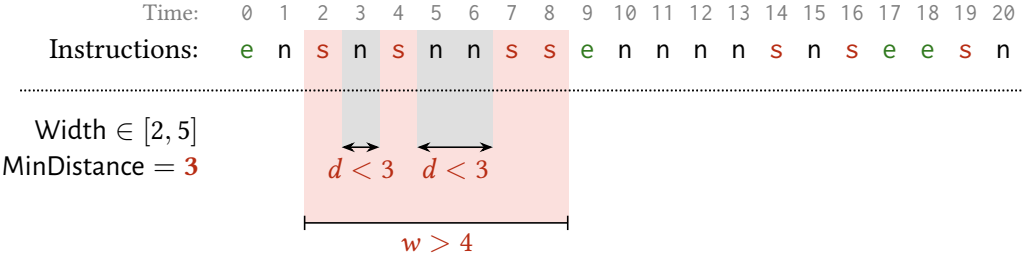
Width $\in [2, 5]$

MinDistance $= \mathbf{3}$

The positioning problem



The positioning problem



- No solution!

SAMVA's original backtracking algorithm

Generate a fault sequence in left-to-right order (increasing t):

- Find the next skip in the timeline;
- Explore branches for "good-looking" t and w;
- If there are none, backtrack; otherwise, keep going.

SAMVA's original backtracking algorithm

Generate a fault sequence in left-to-right order (increasing t):

- Find the next skip in the timeline;
- Explore branches for "good-looking" t and w;
- If there are none, backtrack; otherwise, keep going.

Quite a few limitations ⚠

1. Usually fast (100 µs; native C++) but randomly, brutally slow (2–30 minutes)
2. Returns "some" solutions, and not clear which
   - (Although I didn't find any instance where it missed the minimum)
3. Lots of linear-time pre-checks to avoid running this algorithm

Mapping out fault positioning's state space

Attacker's behavior is actually really limited.

- w_k: will wait for the next k instructions ($k \in [1, \mathsf{MinDistance}]$);
- f_k: is injecting a fault that'll last exactly k more instructions ($k \in [1, \mathsf{MaxWidth}]$).

Mapping out fault positioning's state space

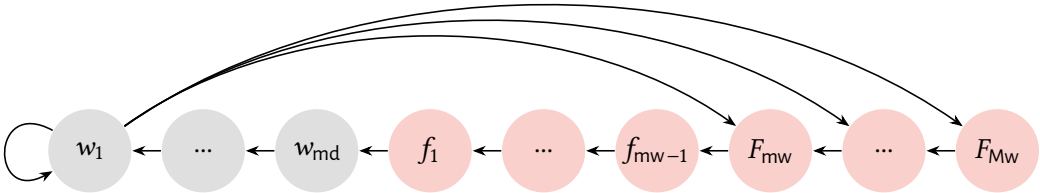
Attacker's behavior is actually really limited.

- w_k: will wait for the next k instructions ($k \in [1, \mathsf{MinDistance}]$);
- f_k: is injecting a fault that'll last exactly k more instructions ($k \in [1, \mathsf{MaxWidth}]$).

Mapping out fault positioning's state space

Attacker's behavior is actually really limited.

- w_k: will wait for the next k instructions ($k \in [1, \text{MinDistance}]$);
- f_k: is injecting a fault that'll last exactly k more instructions ($k \in [1, \text{MaxWidth}]$).

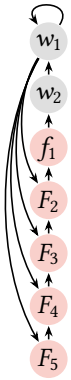


Dynamic programming for positioning

Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Instructions:	e	n	s	n	s	n	n	s	s	e	n	n	n	n	s	n	s	e	e	s	n	.

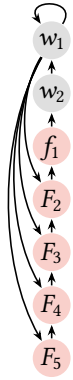
Dynamic programming for positioning

Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Instructions:	e	n	s	n	s	n	n	s	s	e	n	n	n	n	s	n	s	e	e	s	n	.

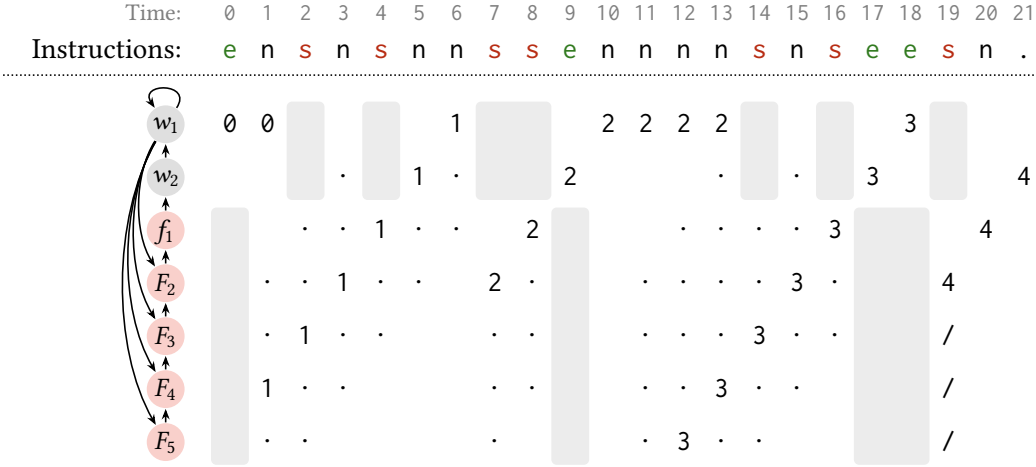


Dynamic programming for positioning

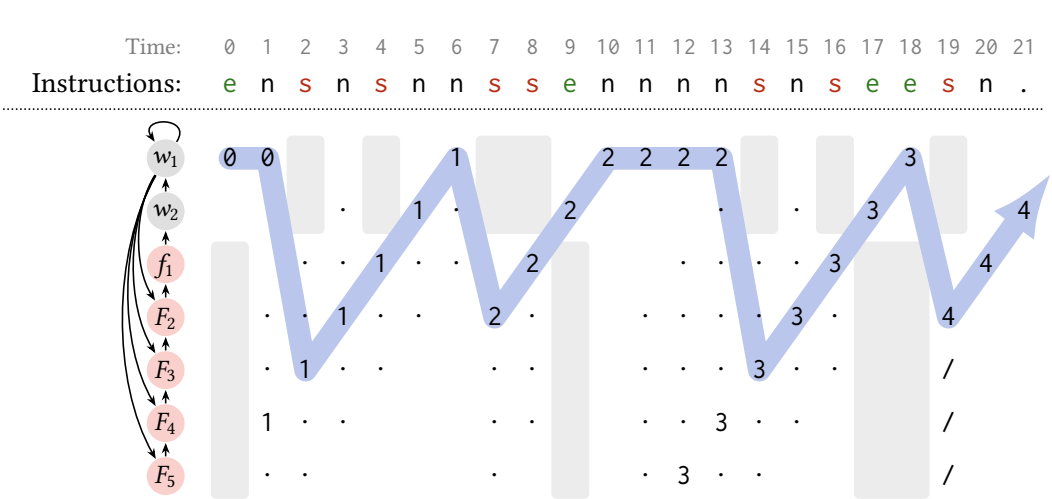
Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Instructions:	e	n	s	n	s	n	n	s	s	e	n	n	n	n	s	n	s	e	e	s	n	.



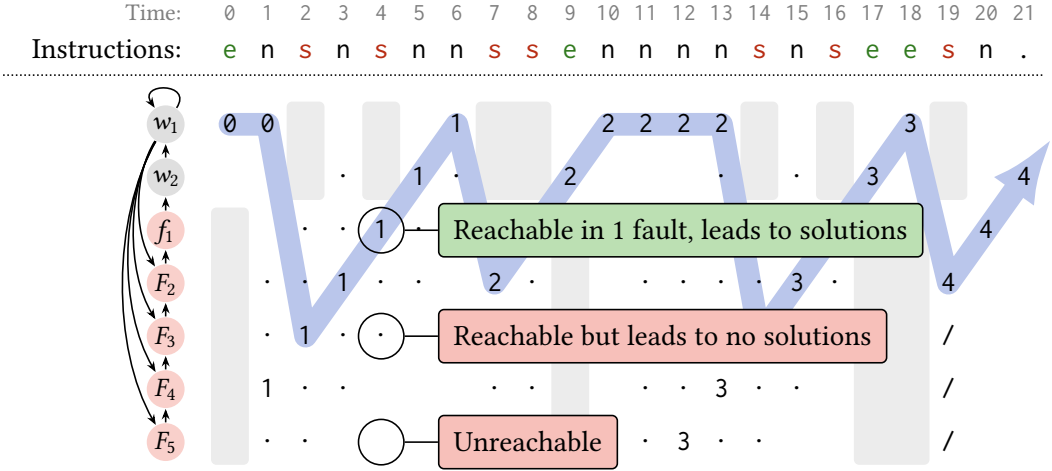
Dynamic programming for positioning



Dynamic programming for positioning



Dynamic programming for positioning



Dynamic programming metrics

Pass #1 (left-to-right)

- InputPathCount: Number of paths from start to a node (s, t).

$$\begin{cases} f(s,0) = 1 \\ f(s',t+1) = \displaystyle\sum_{(s,t)\to(s',t+1)} f(s,t) \end{cases}$$

- InputCost: Mininum cost of any input path.

$$\begin{cases} f(s,0) = \mathsf{EdgeCost}(w_1, s) \\ f(s',t+1) = \displaystyle\min_{(s,t)\to(s',t+1)} \Big(f(s,t) + \mathsf{EdgeCost}(s,s')\Big) \end{cases}$$

Dynamic programming metrics

Pass #1 (left-to-right)

- InputPathCount: Number of paths from start to a node (s, t).

$$\begin{cases} f(s,0) = 1 \\ f(s', t+1) = \displaystyle\sum_{(s,t)\to(s',t+1)} f(s,t) \end{cases}$$

- InputCost: Mininum cost of any input path.

$$\begin{cases} f(s,0) = \mathsf{EdgeCost}(w_1, s) \\ f(s', t+1) = \displaystyle\min_{(s,t)\to(s',t+1)} \Big(f(s,t) + \mathsf{EdgeCost}(s,s')\Big) \end{cases}$$

Pass #2 (right-to-left)

- OutputPathCount: Number of paths from a node (s, t) to end.

$$\begin{cases} f(s, t_{\mathsf{max}} - 1) = 1 \\ f(s,t) = \displaystyle\sum_{(s,t)\to(s',t+1)} f(s,t) \end{cases}$$

- OutputCost: Minimum cost of any output path.

$$\begin{cases} f(s, t_{\mathsf{max}} - 1) = 0 \\ f(s,0) = \mathsf{EdgeCost}(0, s) \\ f(s,t) = \displaystyle\min_{(s,t)\to(s',t+1)} \Big(f(s,t) + \mathsf{EdgeCost}(s,s')\Big) \end{cases}$$

Applications of the new algorithm

- Performance: takes a consistent 1–10 ms on any input (Python)
- Guaranteed to find the minimum number of faults needed for a path
- Finds *all* attacks, classified by cost

This is how we determine the cost of partial paths in the CFG traversal!

Applications of the new algorithm

- Performance: takes a consistent 1–10 ms on any input (Python)
- Guaranteed to find the minimum number of faults needed for a path
- Finds *all* attacks, classified by cost

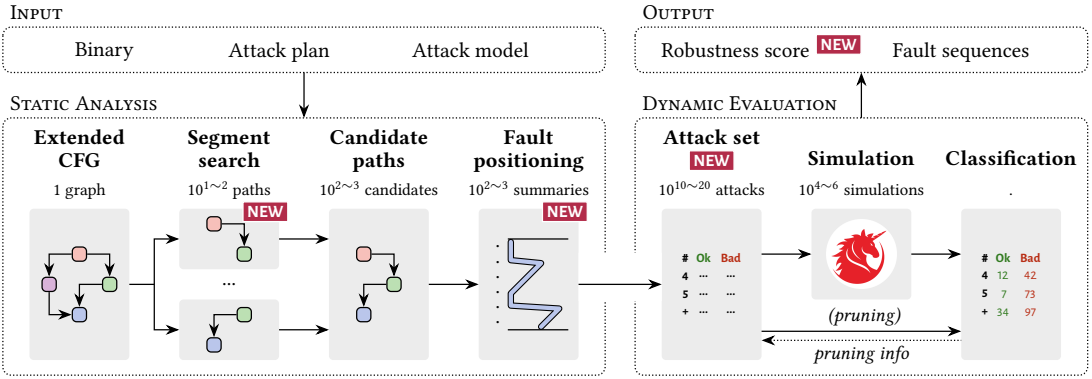
This is how we determine the cost of partial paths in the CFG traversal!

And... optimizing further leads to **linear-time, constant-space** bitvector algorithms for:

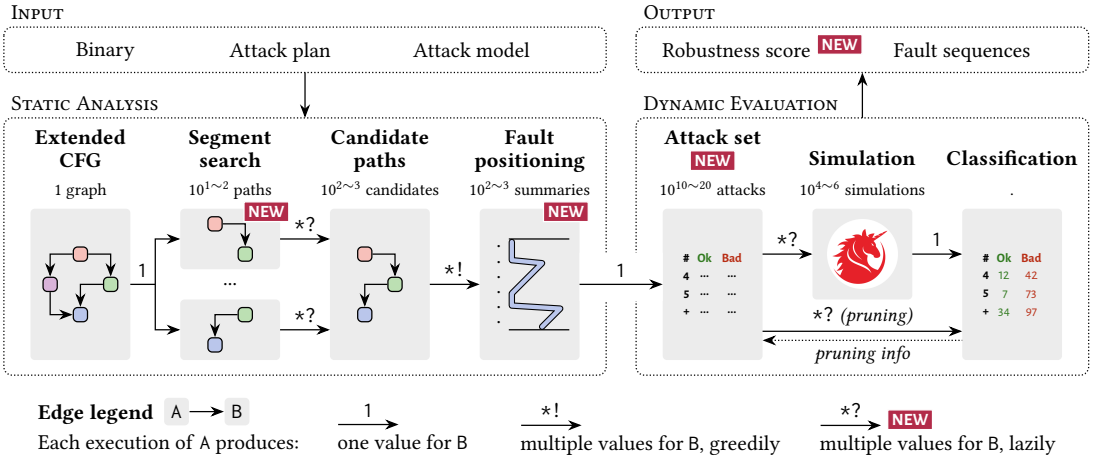
1. Deciding the existence of solutions to the fault positioning problem!
   - Currently implemented (Python).
2. Finding the minimum fault cost of any given path!
   - That one is WIP.

4. Lazy, monotonic enumeration

Flow of data between SAMVA's algorithms



Flow of data between SAMVA's algorithms



Taming the search space: just go by increasing cost

Guiding principle 💡

Lazy algorithms should output values by increasing order of cost.
Or, more relaxed: the lower bound on cost for future values should increase over time.

Taming the search space: just go by increasing cost

Guiding principle 💡

Lazy algorithms should output values by increasing order of cost.
Or, more relaxed: the lower bound on cost for future values should increase over time.

1. CFG traversal: produces the exact cost order
2. Cartesian product of segments: lower bound increases over time
   - (sum of segments' costs is also a lower bound on real cost)
3. Fault positioning: produces the exact cost order
4. Simulation: picks cheapest attacks available at any time

And... that's it! This'll find the cheapest attacks.

The end result: running SAMVA

FISSC [Dur+16]'s `verifyPIN_v5` (CV32E40P build)
Width $\in [2, 20]$, MinDistance $= 2$

- Schedules CFG traversal, fault positioning and simulation "in parallel"
  - (currently round-robin but *extremely* independent)
- Enumerates the cheapest attacks *roughly* in monotonic order
- If left running, logs more and more of them

```
SEGMENT PATHS
  Segment #1 (7.68 s)      Segment #2 (39.3 ms)
  34/150 (running)         1/150 (complete)
  Cost: 6                  Cost: 0

CANDIDATE PATHS
  Candidate count: 34 (326 µs)
  Est. cost: 6 (max inversion: -1)
  Candidates: 17 analyzed (189 ms) (0 unsat) + 17 doomed
  Real cost: 5 (max inversion: -1) (max bias: 0)
  Attacks: 2.07e+18

ATTACK SOLUTIONS
   #F  #C |       Found     Pending      Worked      Failed      Pruned
    2   1 |        1850           0        1647         203           0
    3   7 |      190907      186157        3000        1750           0
    4  13 |    26458694    26458694           0           0           0
    5  17 | 1664514305 1664514305           0           0           0
    *  -- |    2.07e+18    2.07e+18           0           0           0

EVALUATION
  Total simulations: 6600 (99.9 s)
    Success      X: store     X: load      X: LR        X: pop
    4647         0            0            0            0
    X: move      CF: ret      CF: pop      CF: load     CF: move
    0            0            0            0            0
    Bad trace    Analysis!    Unknown!
    0            0            1953

Best attack: Found in 2 faults
Best exclusion: Impossible in ≤ 1 fault
Robustness against this attack is 1 fault 😎
```

5. Conclusion

Conclusion

Contributions

- Revamp SAMVA with a new fault positioning algorithm and lazy data-flow
- Result: complete analysis for the attacks considered; pretty dang fast

Takeaway message!

- Combo of static analysis and dynamic evaluation is very flexible
- Always worth it to pin down the state space in search problems

---

SAMVA [Gic+23]
Static Analysis tool for Multi-fault Vulnerability Assessment
https://gitlab.inria.fr/samva-project

References I

[Dur+16] Louis Dureuil et al. “FISSC: a fault injection and simulation secure collection”. In: *Computer Safety, Reliability and Security*. Vol. 9922. Lecture Notes in Computer Science. Trondheim, Norway: Springer, Sept. 2016, pp. 3–11. DOI: 10.1007/978-3-319-45477-1_1. URL: https://hal.science/hal-03784107.

[Gic+23] Antoine Gicquel et al. “SAMVA: Static Analysis for Multi-Fault Attack Paths Determination”. In: *COSADE 2023-14th International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer. 2023, pp. 3–22.

[Gic24] Antoine Gicquel. “Étude de vulnérabilité d’un programme au format binaire en présence de fautes précises et nombreuses”. 2024URENS059. PhD thesis. 2024. URL: http://www.theses.fr/2024URENS059/document.