

Quantifying programs' robustness
✦ against multi-fault control-flow attacks ✦
by static analysis



Sébastien MICHELLAND (Inria, ENS de Lyon)

Antoine GICQUEL (Inria), Damien HARDY (IRISA), Erven ROHOU (Inria)



PROGRAMME
DE RECHERCHE
CYBERSÉCURITÉ

Inria

anr[®]



PROGRAMME DE
transfert aux
campus cyber



1. Overview of SAMVA

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]



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]



Two topics today:

- ▶ **SAMVA in general:** the analysis and its results
- ▶ **New algorithms:** find all attacks fast, evaluate robustness
Paper at ISSRE'26 (Sébastien, Antoine, Erven, Damien)



With enough faults you can do anything

With infinite instruction skips, you can...

- ▶ Pick and choose which instructions you want, ignore everything else
- ▶ Use any backwards jump or call main loop or backwards call to loop forever

With enough faults you can do anything

With infinite instruction skips, you can...

- ▶ Pick and choose which instructions you want, ignore everything else
- ▶ Use any backwards jump or call main loop or backwards call to loop forever

Complex exploit (⚠ purely theoretical)

Make a Turing-complete machine out of a few gadgets [Pén+20]

With enough faults you can do anything

With infinite instruction skips, you can...

- ▶ Pick and choose which instructions you want, ignore everything else
- ▶ Use any backwards jump or call main loop or backwards call to loop forever

Complex exploit (⚠ purely theoretical)

Make a Turing-complete machine out of a few gadgets [Pén+20]

Simple exploit (realistic!)

“Go from entry of verifyPIN to auth=true without triggering the countermeasure”

Fault models of the day

We are dealing with long and repeated faults.

Fault models of the day

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

Fault models of the day

We are dealing with long and repeated faults.

Instruction skip

1 fault,
width 3

~~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

Fault models of the day

We are dealing with long and repeated faults.

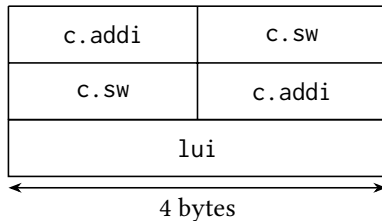
Instruction skip

1 fault,
width 3

```

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



Fault models of the day

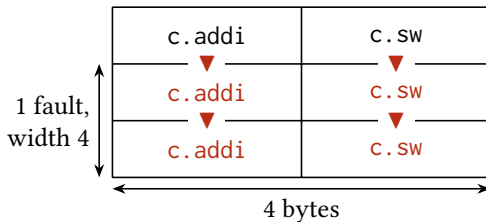
We are dealing with long and repeated faults.

Instruction skip

1 fault,
width 3

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

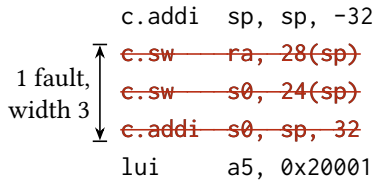
Word replay



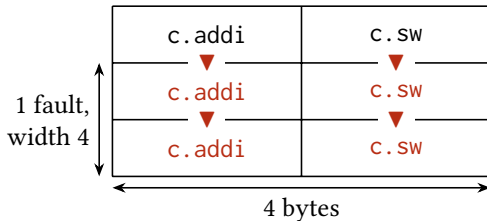
Fault models of the day

We are dealing with long and repeated faults.

Instruction skip



Word replay



Model is defined by:

- ▶ Possible widths [MinWidth, MaxWidth] (in instructions)
- ▶ MinDistance: number of instructions left alone between two faults (reloading time)

Inputs and outputs

INPUTS

- ▶ **Binary:** ARM or RISC-V (so far)
- ▶ **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)
- ▶ **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: $[(t = 2, w = 4), (t = 8, w = 2), \dots]$

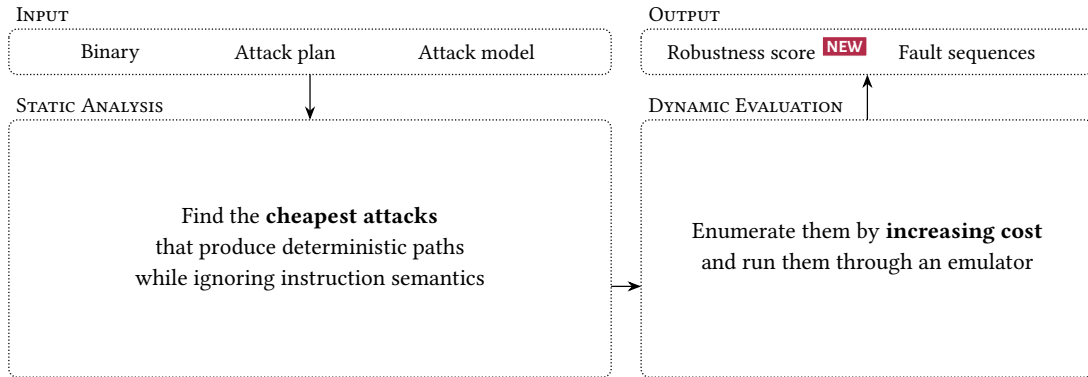
- ▶ **Complete attack set:** **NEW**

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

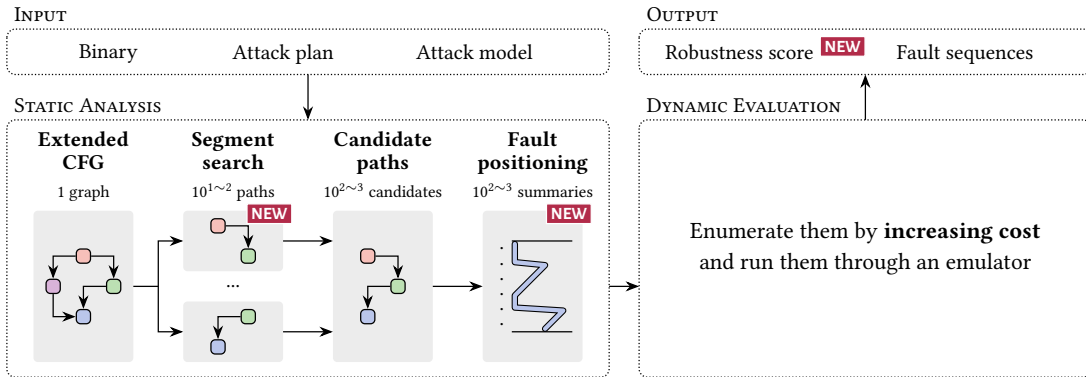
- ▶ **Robustness score:** **NEW**

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

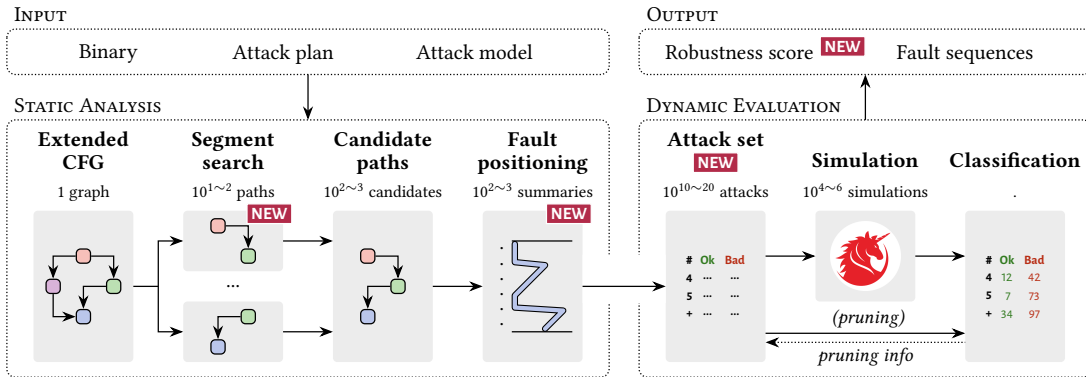
Overview of the analysis



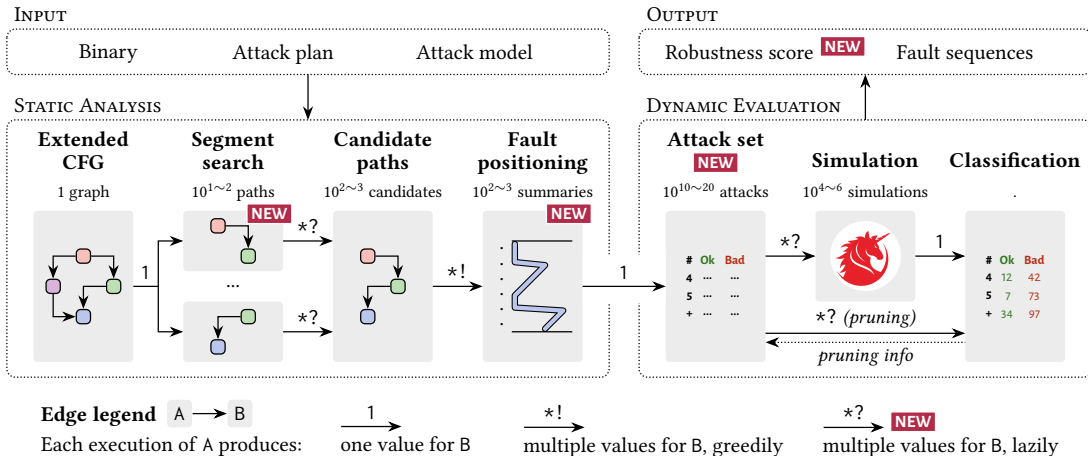
Overview of the analysis



Overview of the analysis



Overview of the analysis

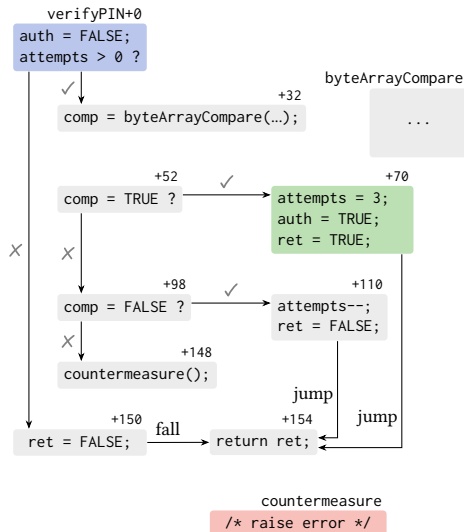




2. The analysis

Attacker's CFG construction

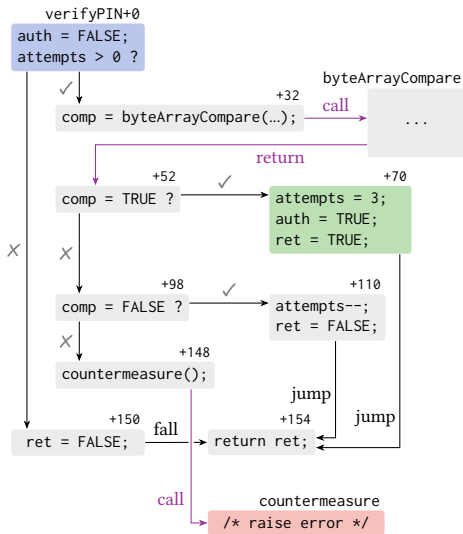
Get CFG from headless Ghidra, then:



Attacker's CFG construction

Get CFG from headless Ghidra, then:

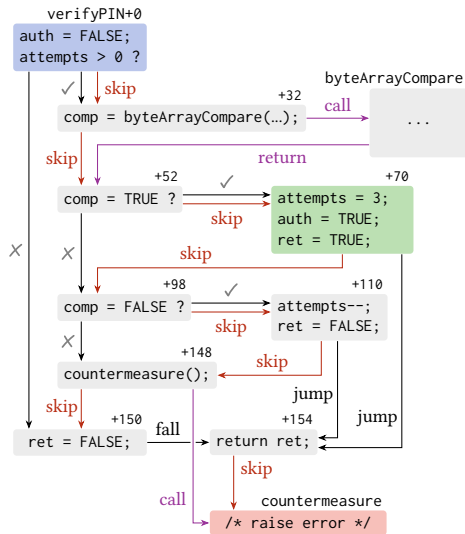
1. Add **call** and **return** edges



Attacker's CFG construction

Get CFG from headless Ghidra, then:

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

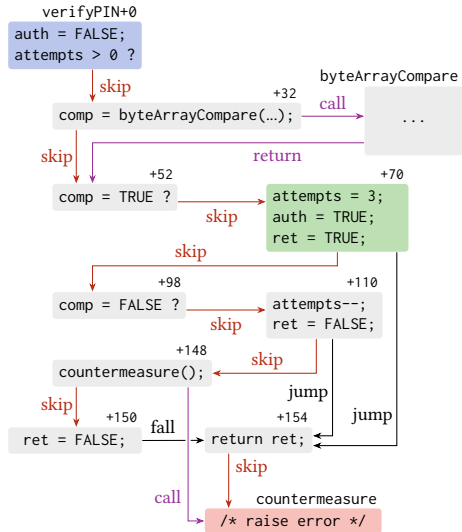


Attacker's CFG construction

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!)



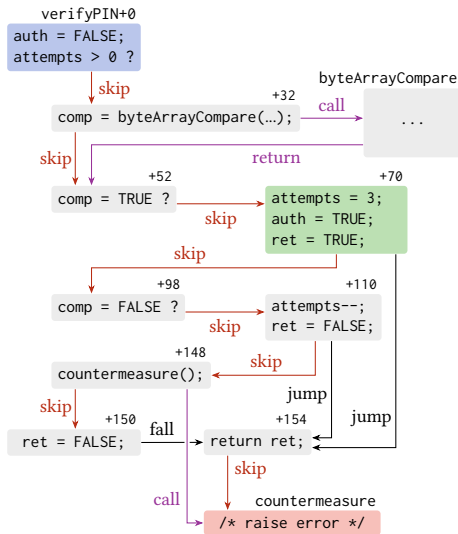
Attacker's CFG construction

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!)

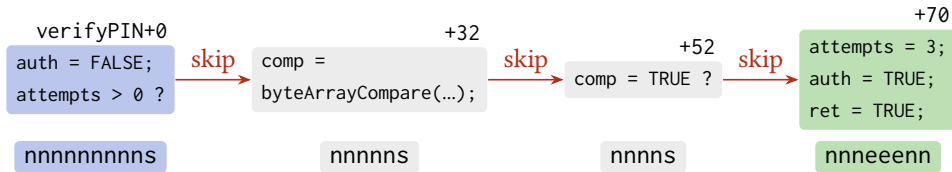
- Attack start
- Attack end
- Banned blocks



Graph traversal for attack paths

Standard graph traversal yields possible attack paths.

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



Two technical complications:

- ▶ 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).

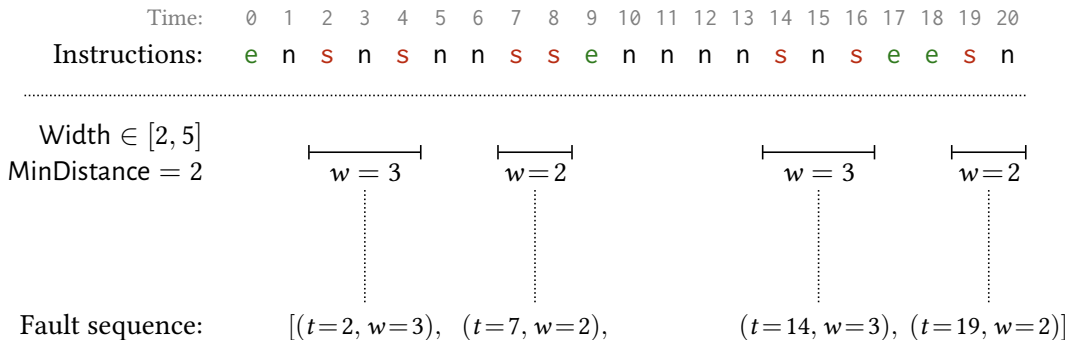
The positioning problem

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

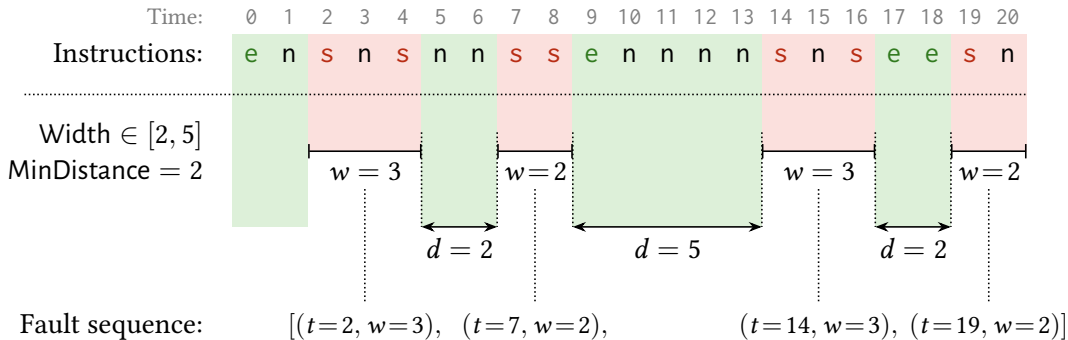
Width $\in [2, 5]$

MinDistance = 2

The positioning problem



The positioning problem



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 μ s; 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

New fault positioning algorithm

Dynamic programming approach that runs in linear time **NEW**

- ▶ 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

New fault positioning algorithm

Dynamic programming approach that runs in linear time **NEW**

- ▶ 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

Change of paradigm

Before: we invoked fault positioning as little as possible, relying on pre-checks instead.
Now: we use it all the time as an oracle.

- ▶ This is how we weigh the cost of paths in the CFG traversal.
 - ▶ Min cost of a solution obtained in a single linear pass with bitvectors (!)



3. Running SAMVA

Sample execution

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

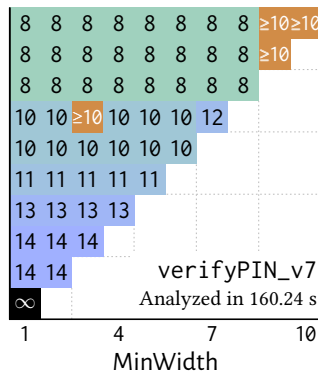
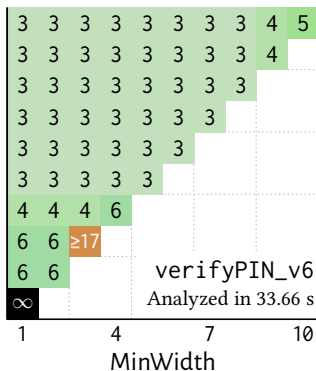
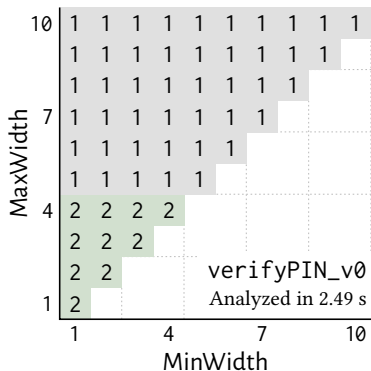
EVALUATION
Total simulations: 6600 (99.9 s)
Success X: store X: load X: LR X: pop
4647 0 0 0 0
X: move CF: ret CF: pop CF: load CF: move
0 0 0 0 0
Bad trace Analysis! Unknown!
0 0 1953

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

```

Robustness across settings and countermeasures

- Robustness as a function of fault width:



- Also, tens of thousands of optimal attacks in under 1 minute!



4. Data-flow manipulation

Future work: Attacking the program's data

```
/* critical function */  
int encrypt(void const *privkey, char *message) { /* ... */ }  
  
/* call site */  
int rc = encrypt(privkey, message);  
printf("status: %s\n", get_status_message(rc));
```

Future work: Attacking the program's data

```
/* critical function */  
int encrypt(void const *privkey, char *message) { /* ... */ }  
  
/* call site */  
int rc = encrypt(privkey, message);  
printf("status: %s\n", get_status_message(rc));
```

Desired attack plan

1. Start at "call encrypt", let privkey := a0
2. Now reach "call printf" with (sp) == privkey

How? Carefully execute moves on the way down.

```
mv    a0, privkey
mv    a1, messageF
call  encrypt
# a0: rc

call  get_status_message
# a0: get_status_message(rc)

sw    a0, (sp)          # variadic arg
# (sp): get_status_message(rc)
mv    a0, "status: %s\n" # (simplified)
call  printf           # -> "status: success"
```

How? Carefully execute moves on the way down.

```
mv    a0, privkey
mv    a1, message
call encrypt
# a0: privkey

call get_status_message
# a0: privkey

sw    a0, (sp)           # variadic arg
# (sp): privkey
mv    a0, "status: %s\n"  # (simplified)
call printf             # -> "status: b4c072e832..."
```



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>.
- [Pén+20] Pierre-Yves Péneau et al. “NOP-Oriented Programming: Should we Care?” In: *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2020, pp. 694–703. DOI: 10.1109/EuroSPW51379.2020.00100.

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 ($k \in [1, \text{MaxWidth}]$).



Mapping out fault positioning's state space

Attacker's behavior is actually really limited.

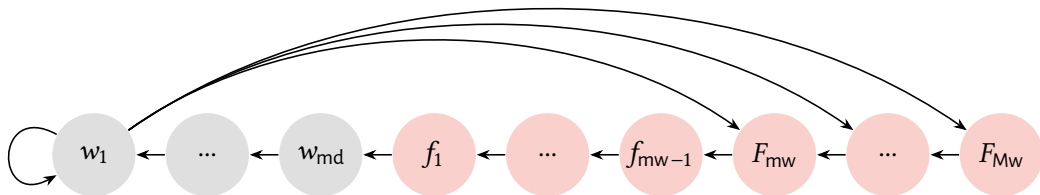
- ▶ w_k : will wait for the next k instructions ($k \in [1, \text{MinDistance}]$);
- ▶ f_k : is injecting a fault that'll last exactly k more instructions ($k \in [1, \text{MaxWidth}]$).



Mapping out fault positioning's state space

Attacker's behavior is actually really limited.

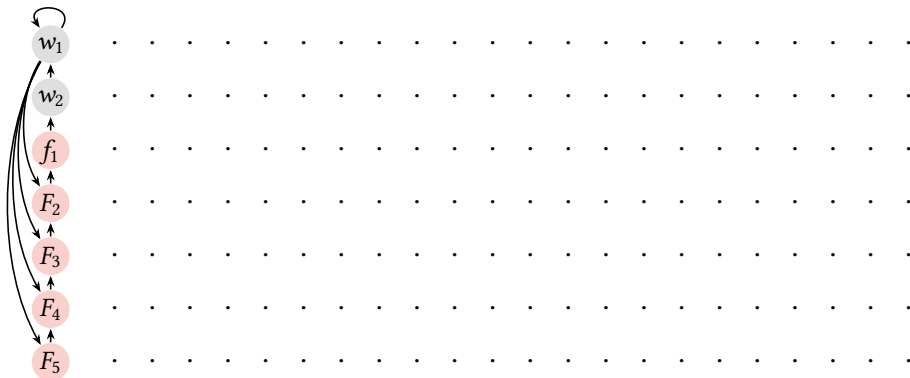
- ▶ w_k : will wait for the next k instructions ($k \in [1, \text{MinDistance}]$);
- ▶ f_k : is injecting a fault that'll last exactly k more instructions ($k \in [1, \text{MaxWidth}]$).



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

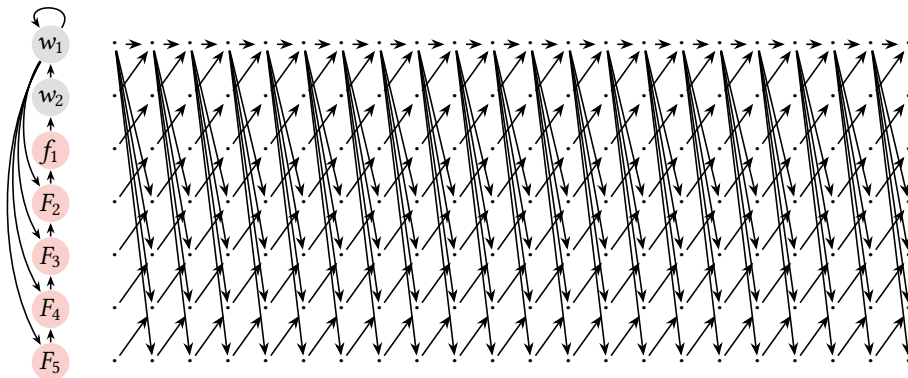
Instructions: e n s n s n n s s e n n n n s n s e e s n .



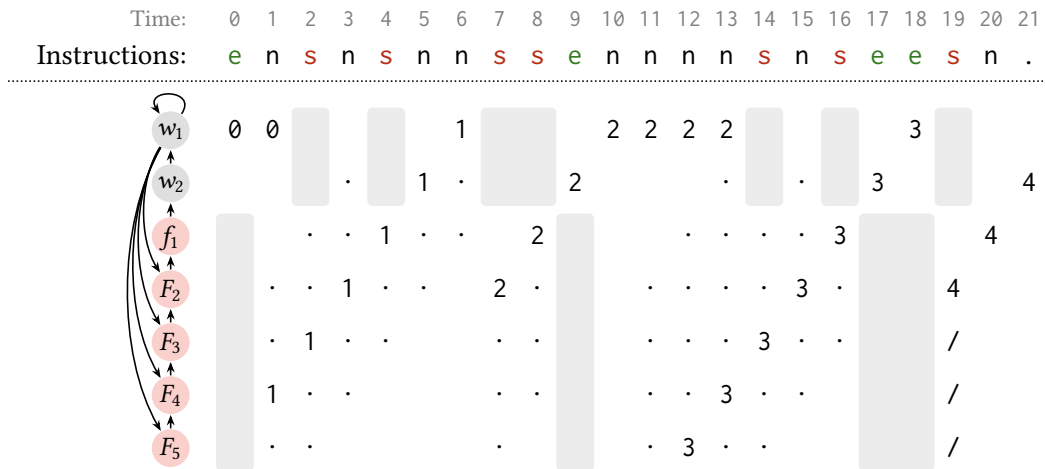
Dynamic programming for positioning

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

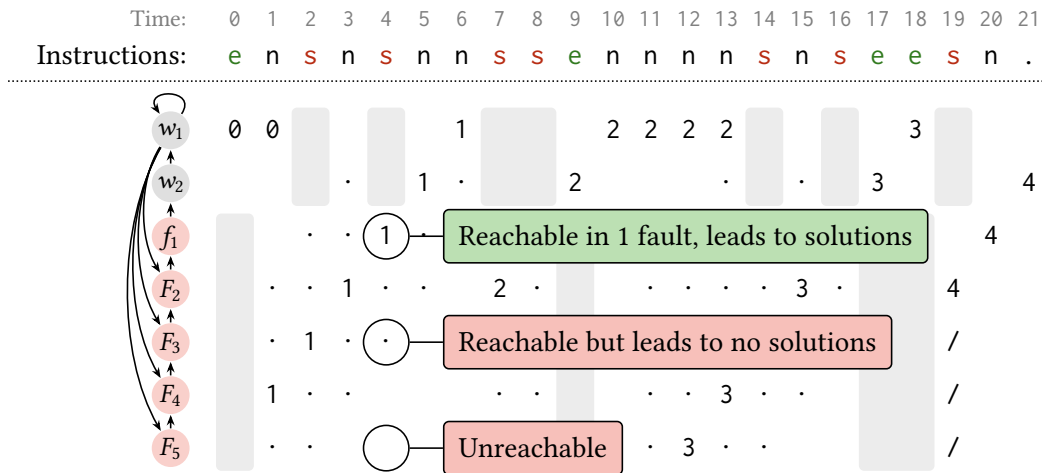
Instructions: e n s n s n n s s e n n n n s n s e e s n .



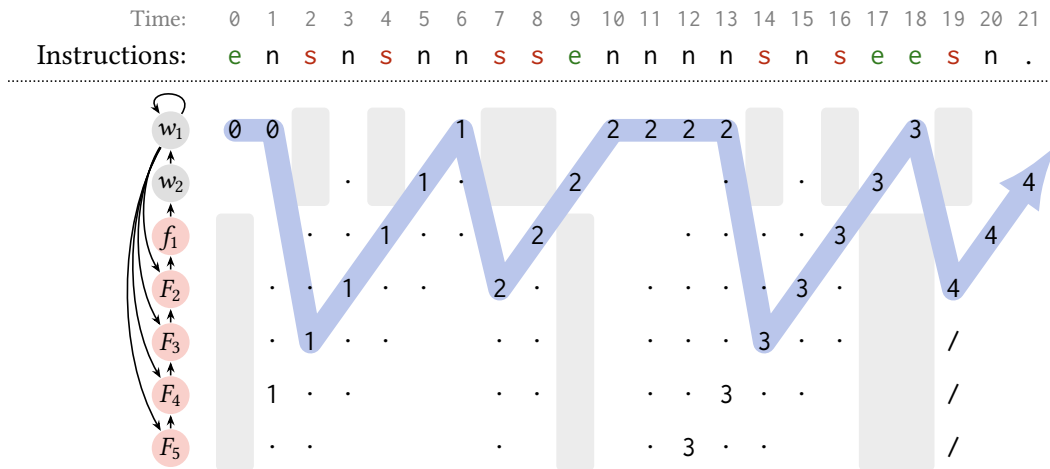
Dynamic programming for positioning



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) .

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

- **InputCost**: Minimum cost of any input path.

$$\begin{cases} f(s, 0) = \text{EdgeCost}(w_1, s) \\ f(s', t + 1) = \min_{(s, t) \rightarrow (s', t+1)} \left(f(s, t) + \text{EdgeCost}(s, s') \right) \end{cases}$$

Dynamic programming metrics

Pass #1 (left-to-right)

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

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

Pass #2 (right-to-left)

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

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

- ▶ **InputCost**: Minimum cost of any input path.

$$\begin{cases} f(s, 0) = \text{EdgeCost}(w_1, s) \\ f(s', t + 1) = \min_{(s, t) \rightarrow (s', t+1)} (f(s, t) + \text{EdgeCost}(s, s')) \end{cases}$$

- ▶ **OutputCost**: Minimum cost of any output path.

$$\begin{cases} f(s, t_{\max} - 1) = 0 \\ f(s, 0) = \text{EdgeCost}(0, s) \\ f(s, t) = \min_{(s, t) \rightarrow (s', t+1)} (f(s, t) + \text{EdgeCost}(s, s')) \end{cases}$$

Avoiding crashes: not worth it, yet

Countermeasures		Cheapest attack	Enumeration of min-fault solutions			
			Found	% Success	% Explored	Time
v0	<i>(nothing)</i>	2 faults	486	89.3 %	100 %	165 ms
v1	Hardened Booleans (HB)	2 faults	483	88.6 %	100 %	162 ms
v2	HB, Loop bound check (LBC)	2 faults	507	90.4 %	100 %	173 ms
v3	v2 + Inlining	3 faults	2817	52.6 %	100 %	1.85 s
v4	v3 + Loop counter, Backup copy	5 faults	≥ 63335	39.9 %	20.3 %	60 s
v5	v2 + Double call, Double test	3 faults	12575	64.8 %	100 %	4.68 s
v6	v3 + Double test	4 faults	40180	44.5 %	100 %	33.2 s
v7	v3 + Step counter	9 faults	≥ 98901	60.1 %	0.006 %	60 s

Avoiding crashes: not worth it, yet

Countermeasures		Cheapest attack	Enumeration of min-fault solutions			
			Found	% Success	% Explored	Time
v0	(<i>nothing</i>)	2 faults	486	89.3 %	100 %	165 ms
v1	Hardened Booleans (HB)	2 faults	483	88.6 %	100 %	162 ms
v2	HB, Loop bound check (LBC)	2 faults	507	90.4 %	100 %	173 ms
v3	v2 + Inlining	3 faults	2817	52.6 %	100 %	1.85 s
v4	v3 + Loop counter, Backup copy	5 faults	≥ 63335	39.9 %	20.3 %	60 s
v5	v2 + Double call, Double test	3 faults	12575	64.8 %	100 %	4.68 s
v6	v3 + Double test	4 faults	40180	44.5 %	100 %	33.2 s
v7	v3 + Step counter	9 faults	≥ 98901	60.1 %	0.006 %	60 s

- Nearly half the simulations succeed! Not much to gain with increased precision there.