Towards Path-Aware Coverage-Guided Fuzzing

- **Giacomo Priamo** (Sapienza University of Rome)
- Daniele Cono D'Elia (Sapienza University of Rome)
- Mathias Payer (EPFL)
- Leonardo Querzoni (Sapienza University of Rome)

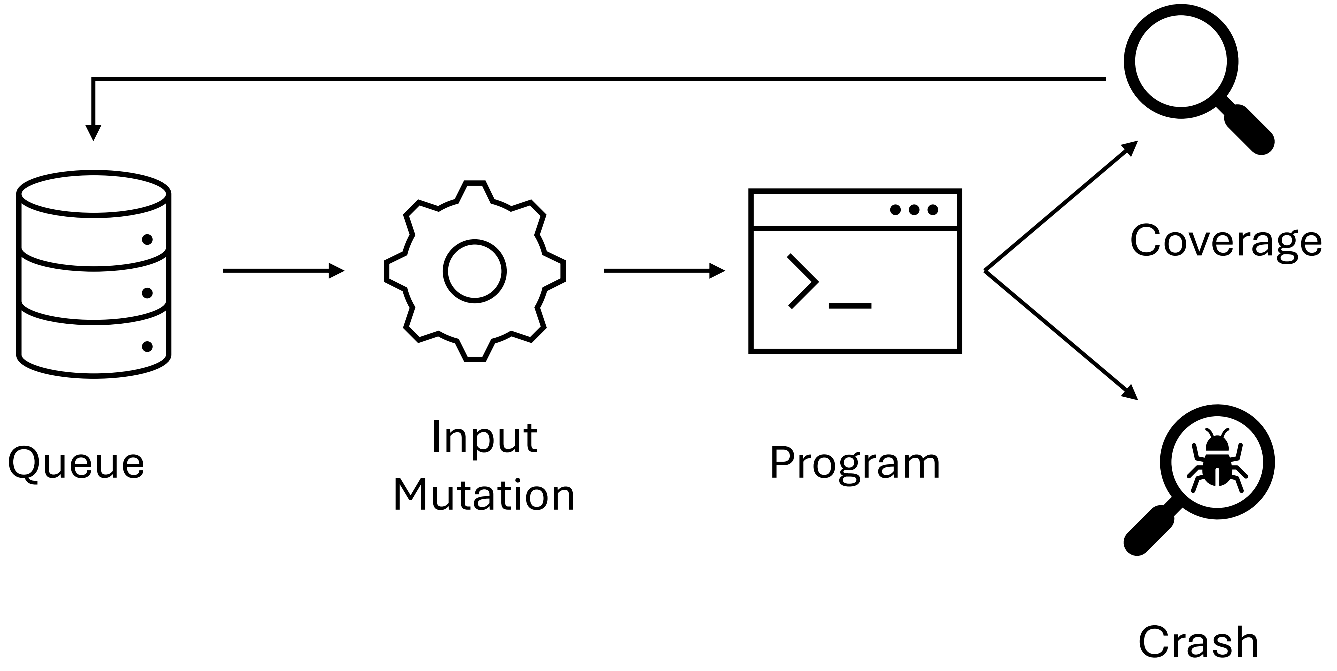
IEEE/ACM Code Generation and Optimization (CGO) 2026
Sydney, Australia

Fuzz Testing (Fuzzing)

- Software testing technique
- Feed random inputs (test cases) to the program under test
- Make the program crash unexpectedly (bug?)

Coverage-guided Fuzzing

- Evolution of the original fuzzing paradigm
- Use a coverage **feedback** to keep *interesting* inputs in a queue, and mutate them
- Most common feedback: code coverage
  - Edge coverage



The Limitation of Edge Coverage

Great trade-off between efficiency (lightweight) and effectiveness (bug finding ability), but **coarse-grained**

Cannot distinguish executions that follow **distinct** control-flow **paths** through the **same edges**

Path-dependent bugs may go **undetected**

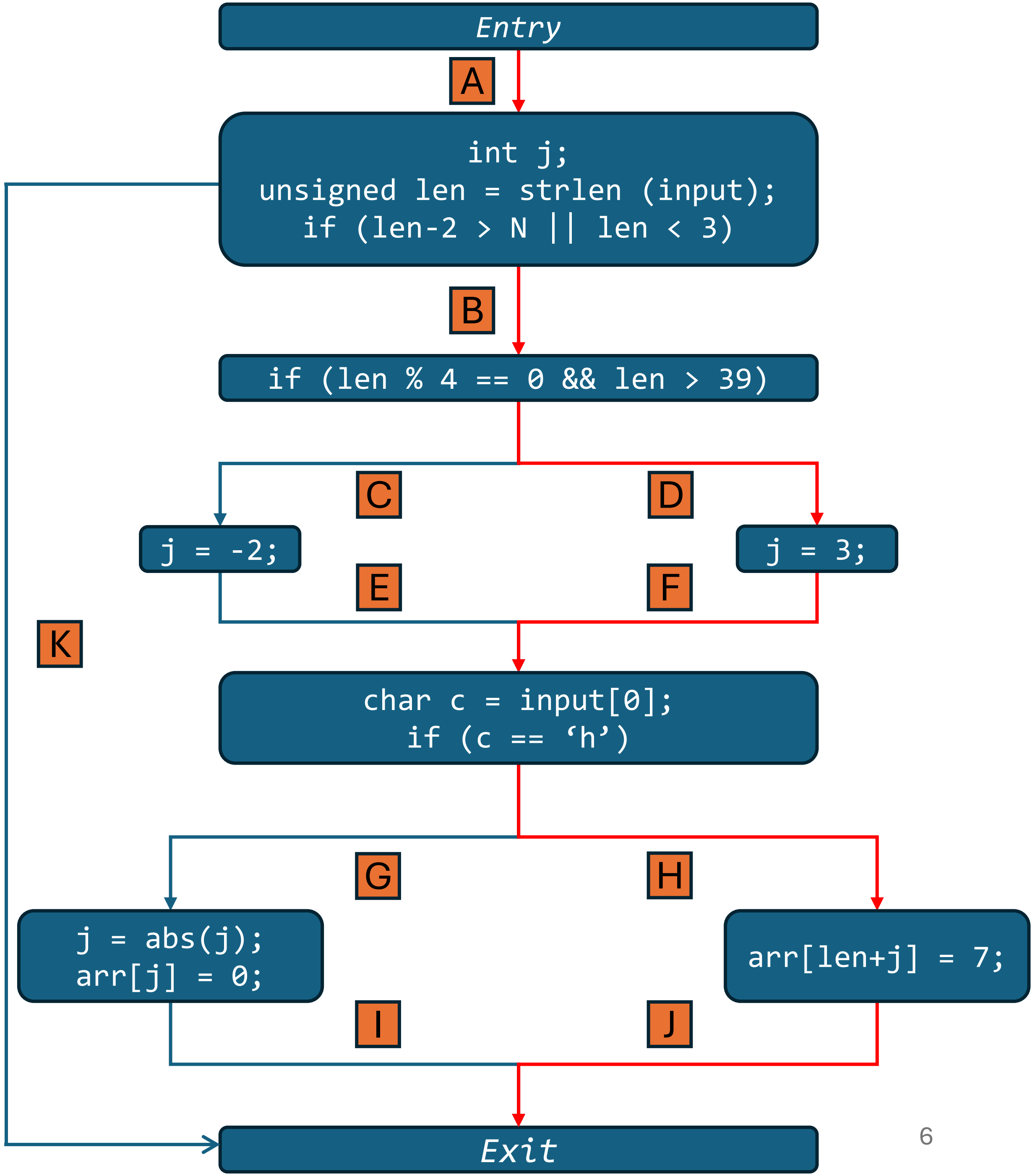
Motivating Example

```c
// arr is an heap array of size N = 54
void foo(char *input, int *arr) {
    int j;

    unsigned len = strlen(input);
    if (len-2 > N || len < 3) return;
    if (len % 4 == 0 && len > 39)
        j = 3; //rare to reach
    else
        j = -2;

    char c = input[0];

    if (c == ‘h’)
        // buffer overflow if reached via
        // ‘rare’ block and len > 50
        arr[len+j] = 7;
    else {
        j = abs(j);
        arr[j] = 0;
    }
}
```

Motivating Example

- Bug-triggering conditions:
  a) ‘input’ string longer than 50 characters and start with an ‘h’
  b) Execution must pass through edges D, F, H, and J
     - A → B → D → F → H → J

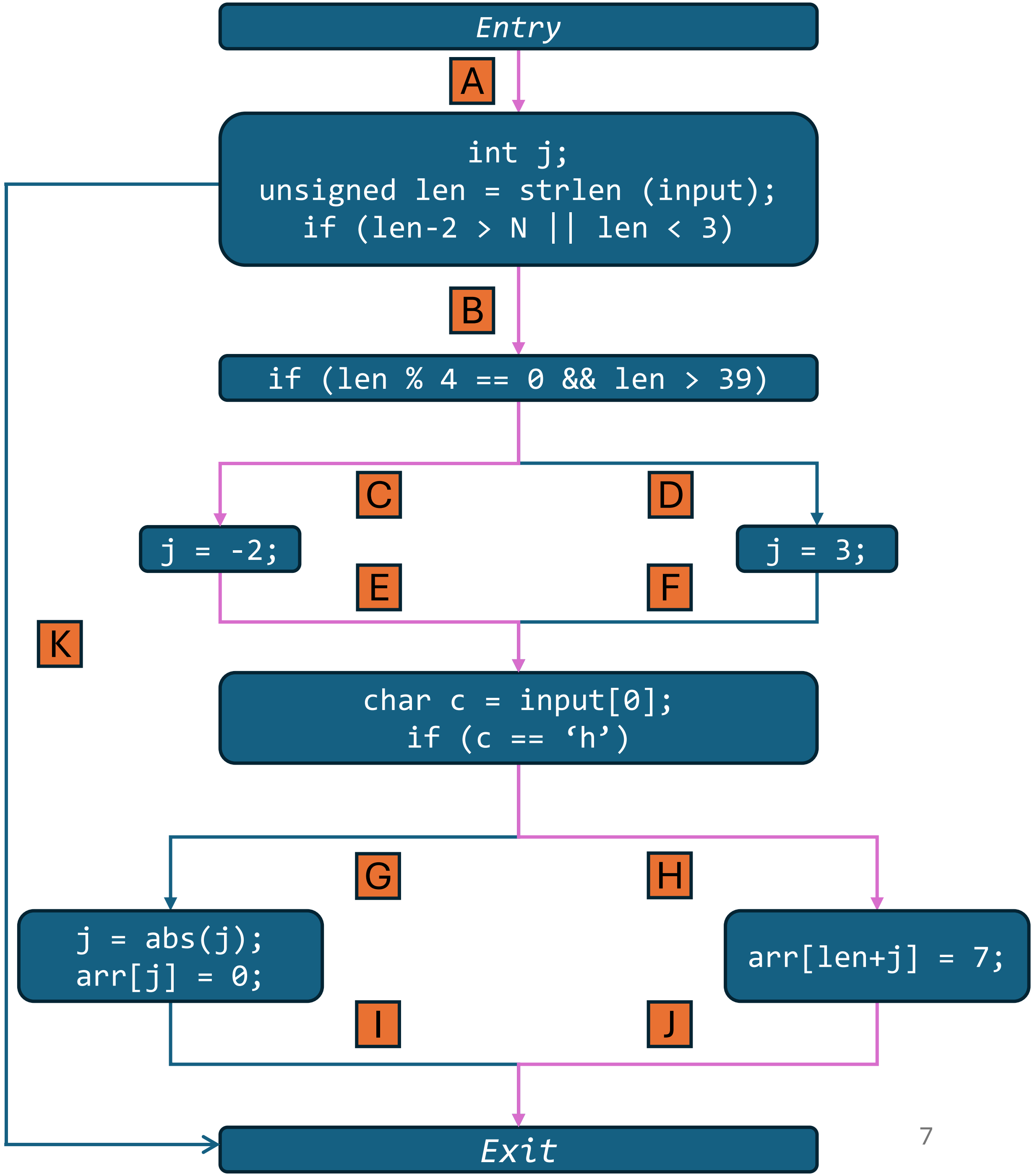
     (precise program state configuration required)
- 5 paths in total, only 1 triggers the bug (the rarest)



Motivating Example

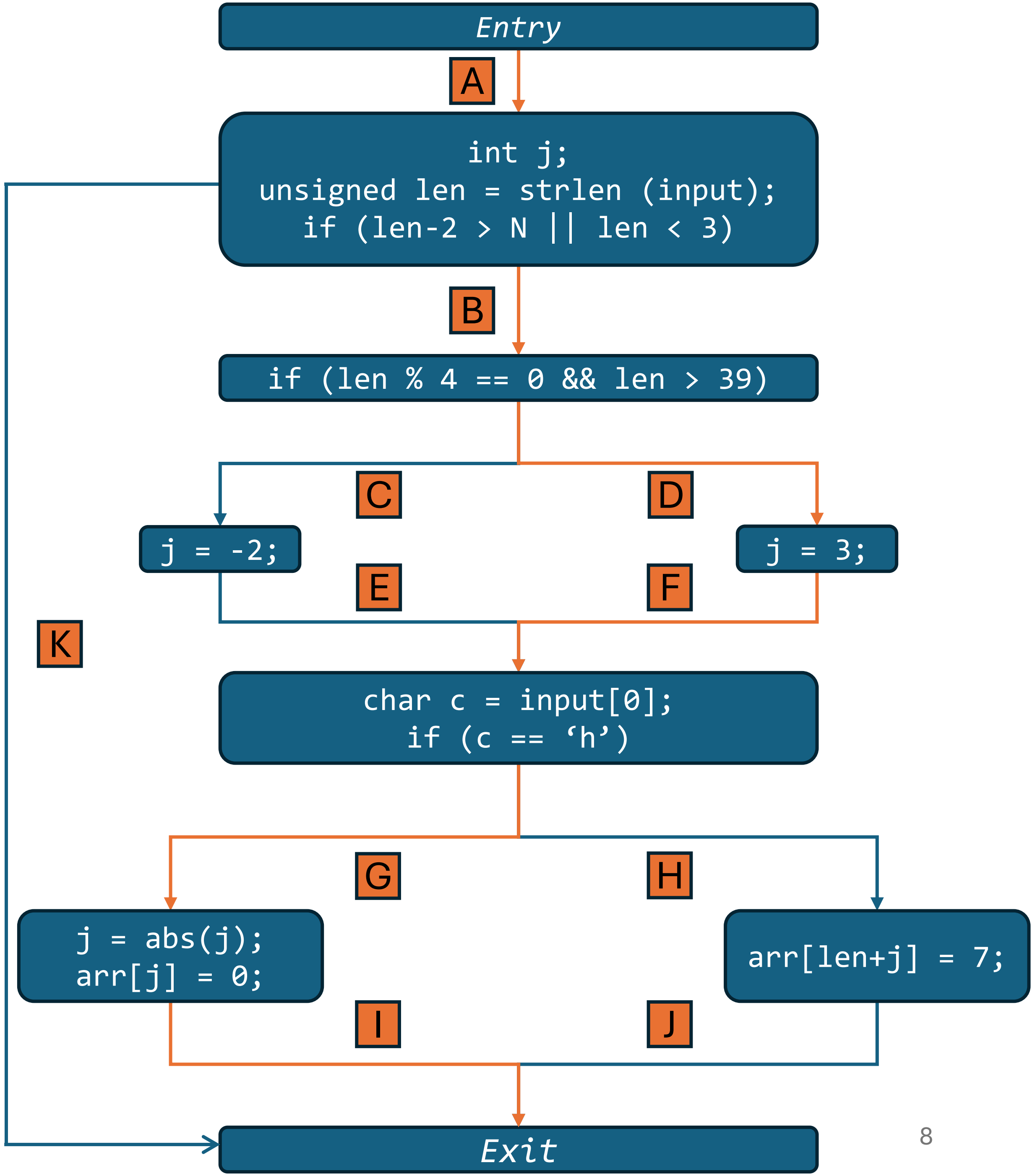
- A classic fuzzer would most likely explore first:
  - ➢ A → B → C → E → H → J

edges	A	B	C	D	E	F	G	H	I	J	K
overall											
current	x	x	x		x			x		x	



Motivating Example

- A classic fuzzer would most likely explore first:
  - ➢ A → B → C → E → H → J

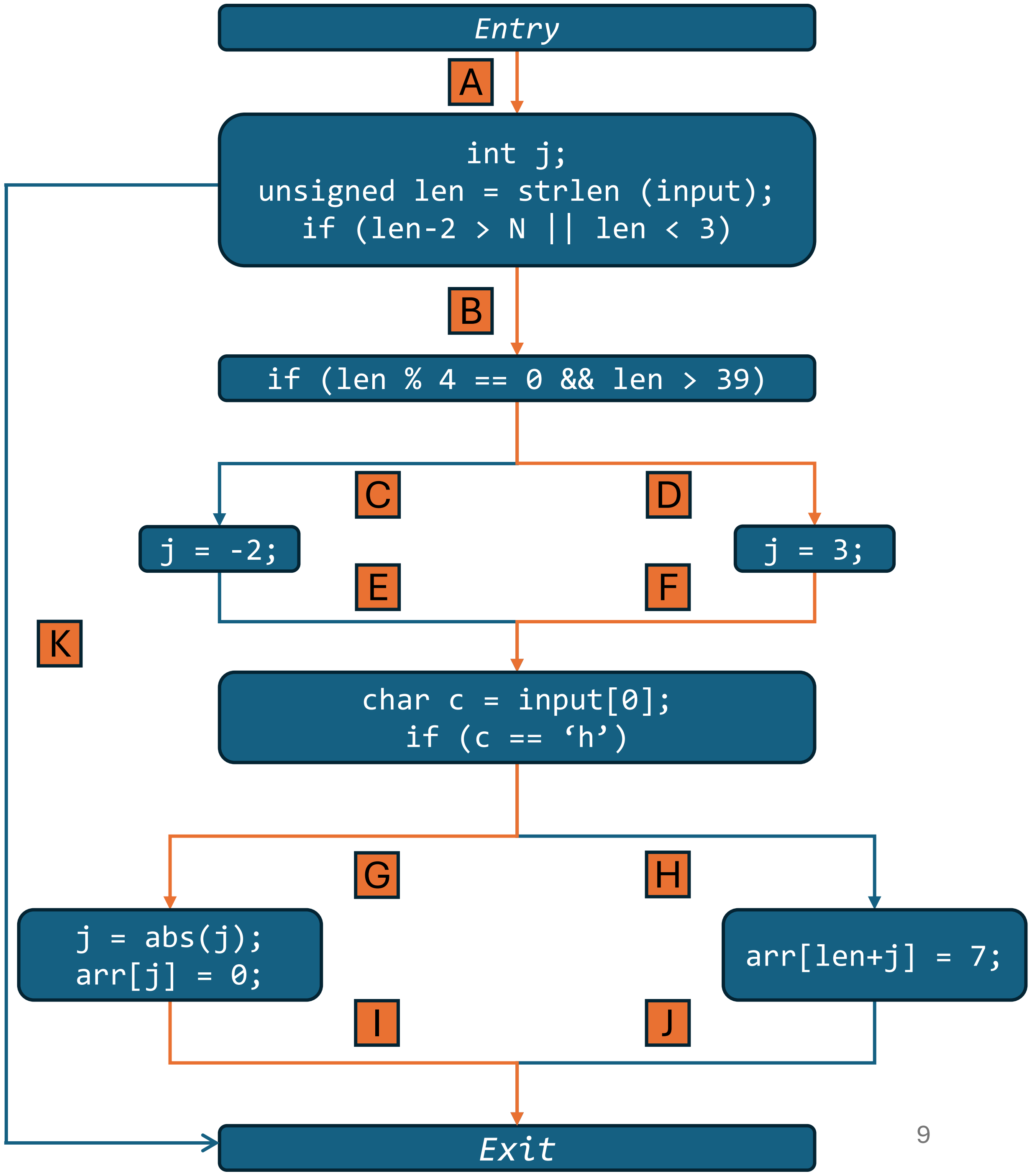


A	B	C	D	E	F	G	H	I	J	K	edges
x	x	x		x			x		x		overall
x	x	x		x			x		x		current

Motivating Example

- A classic fuzzer would most likely explore first:
  - A → B → C → E → H → J
- Then, if it discovers:
  - A → B → D → F → G → I

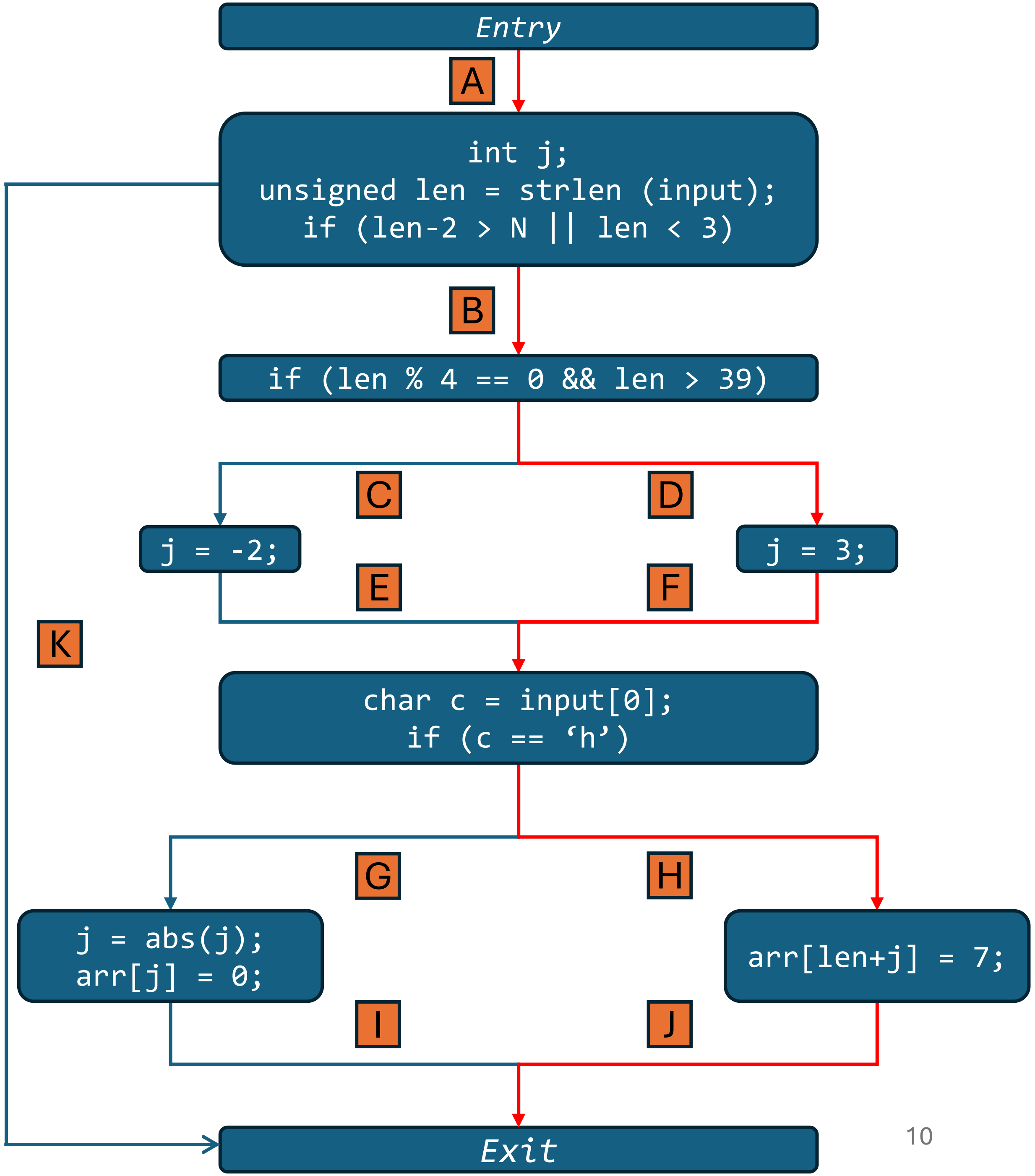
A	B	C	D	E	F	G	H	I	J	K	edges
x	**x**	**x**		**x**			**x**		**x**		overall
x	x		**x**		**x**	**x**		**x**			current



Motivating Example

- A classic fuzzer would most likely explore first:
  - ➢ A → B → C → E → H → J
- Then, if it discovers:
  - ➢ A → B → D → F → G → I

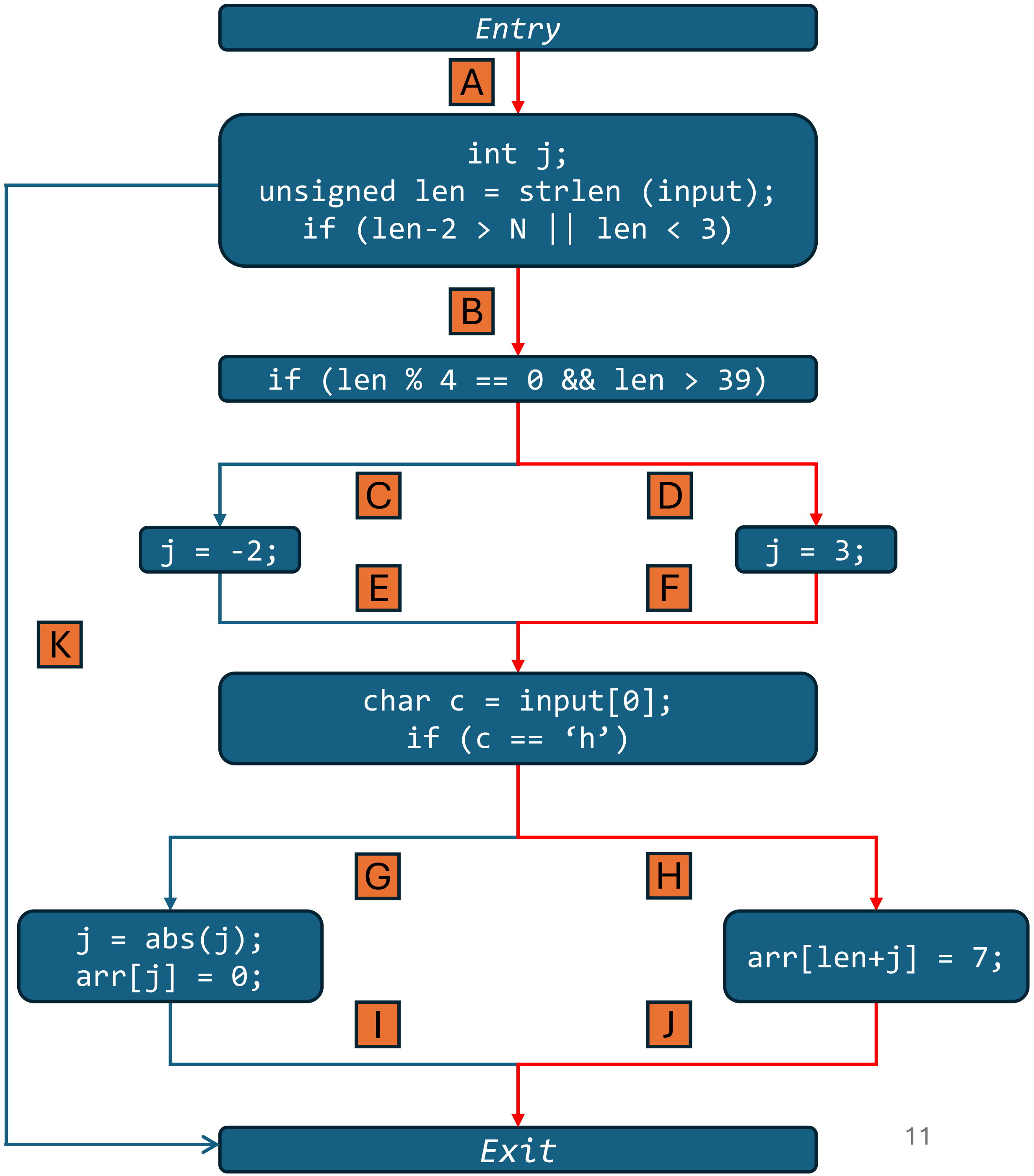
A	B	C	D	E	F	G	H	I	J	K	edges
x	x	x	x	x	x	x	x	x	x		overall
x	x		x		x	x		x			current



Motivating Example

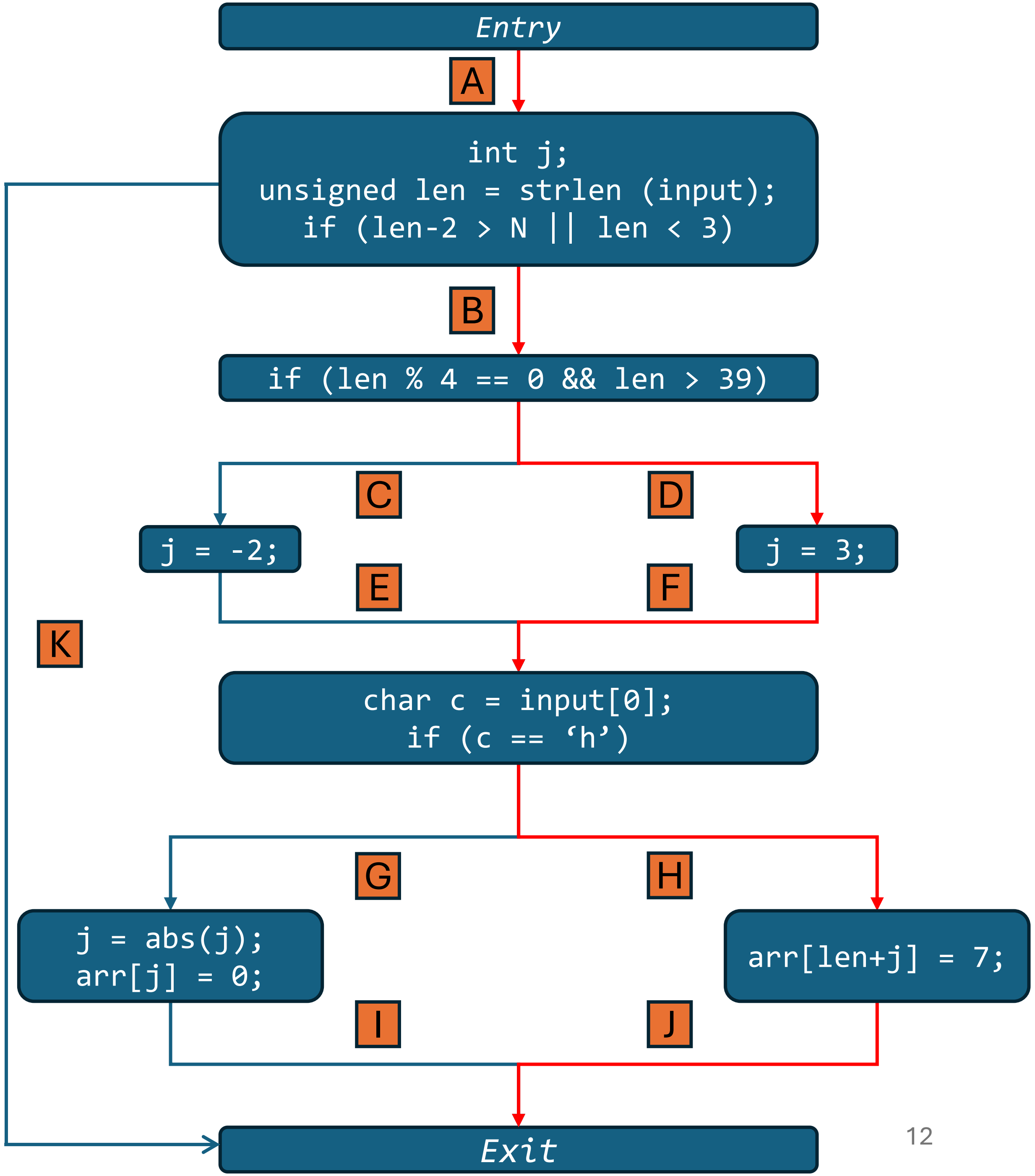
- A classic fuzzer would most likely explore first:
  - ➢ A → B → C → E → H → J
- Then, if it discovers:
  - ➢ A → B → D → F → G → I
- And only at a later point it satisfies condition (b):
  - ➢ A → B → D → F → H → J
  - The test case will not be retained in the queue (all edges already observed)

A	B	C	D	E	F	G	H	I	J	K	edges
x	x	x	x	x	x	x	x	x	x		overall
x	x		x		x		x		x		current



Motivating Example

- Solution: instead of only considering which edges are **hit**, also trace their traversal **order** (path-awareness)
- A **path-aware** fuzzer would treat each single path as **novel**:
  - A → B → C → E → G → I
  - A → B → C → E → H → J
  - A → B → D → F → G → I
  - A → B → D → F → H → J
- It will now retain the test case that takes the required path (b)
  - Mutating the bytes of this input will lead to satisfy (a), thus triggering the bug



Challenges

i. Potentially **unbounded number** of inter-procedural **execution paths** in real programs

➢ Enumeration is untenable

ii. Queue **explosion** entailed by more sensitive feedback mechanisms

➢ Causes fuzzer inefficiency

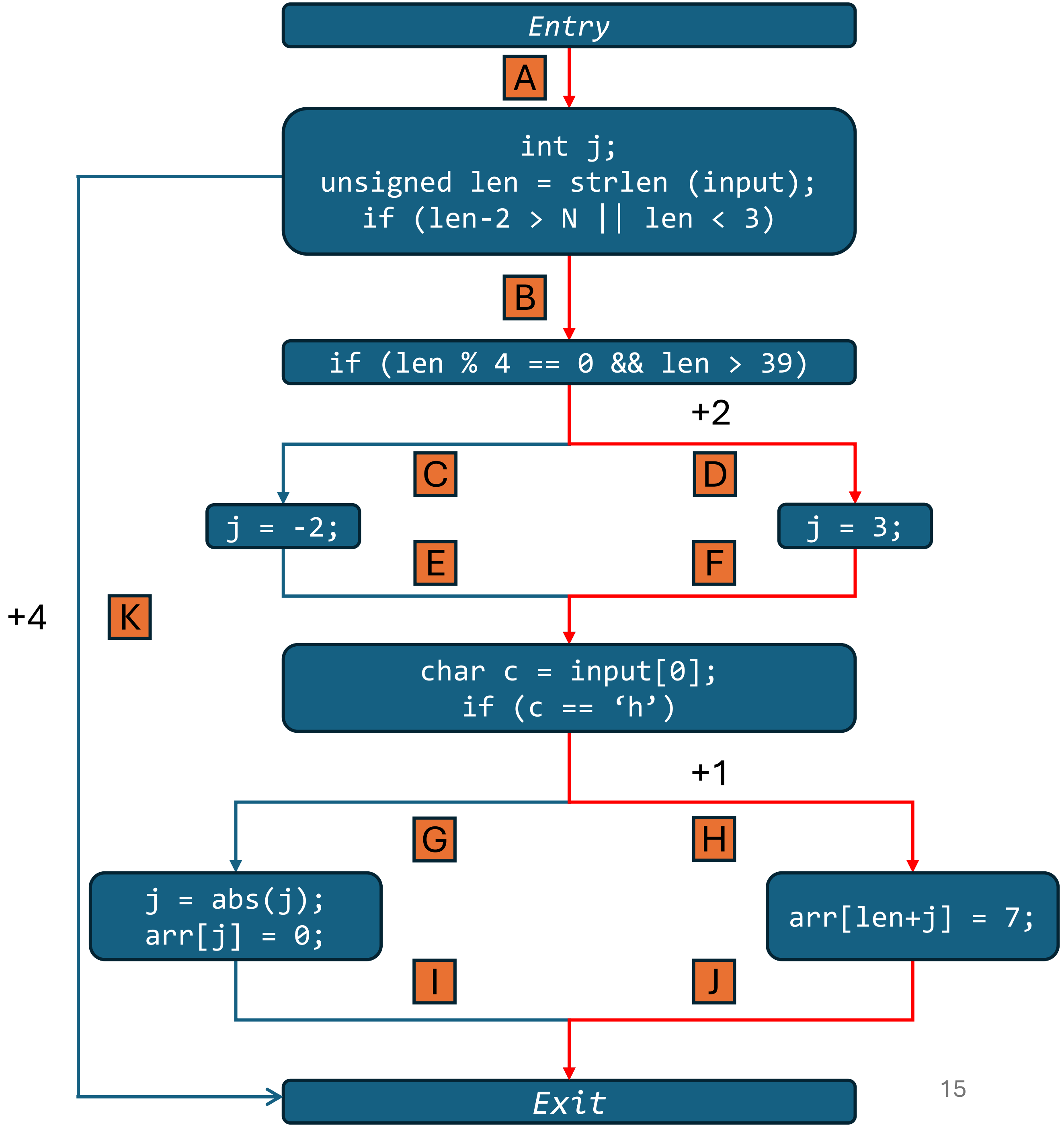
Challenge (i): unbounded number of paths

- Prior research tried studying **whole-program paths**, dismissing it as **infeasible**

- Our idea: track **intra-procedural** paths
  - Focus on function internal states
  - Inevitable information loss (but acceptable)
  - Drastic reduction of the number of paths

How: Ball-Larus algorithm

- Efficient, optimized algorithm to distinguish execution paths
- Assign a unique integer ID value to each path
- Place counters across the CFG to reconstruct said IDs
- Bug-triggering path's ID = '3'
- LLVM IR instrumentation



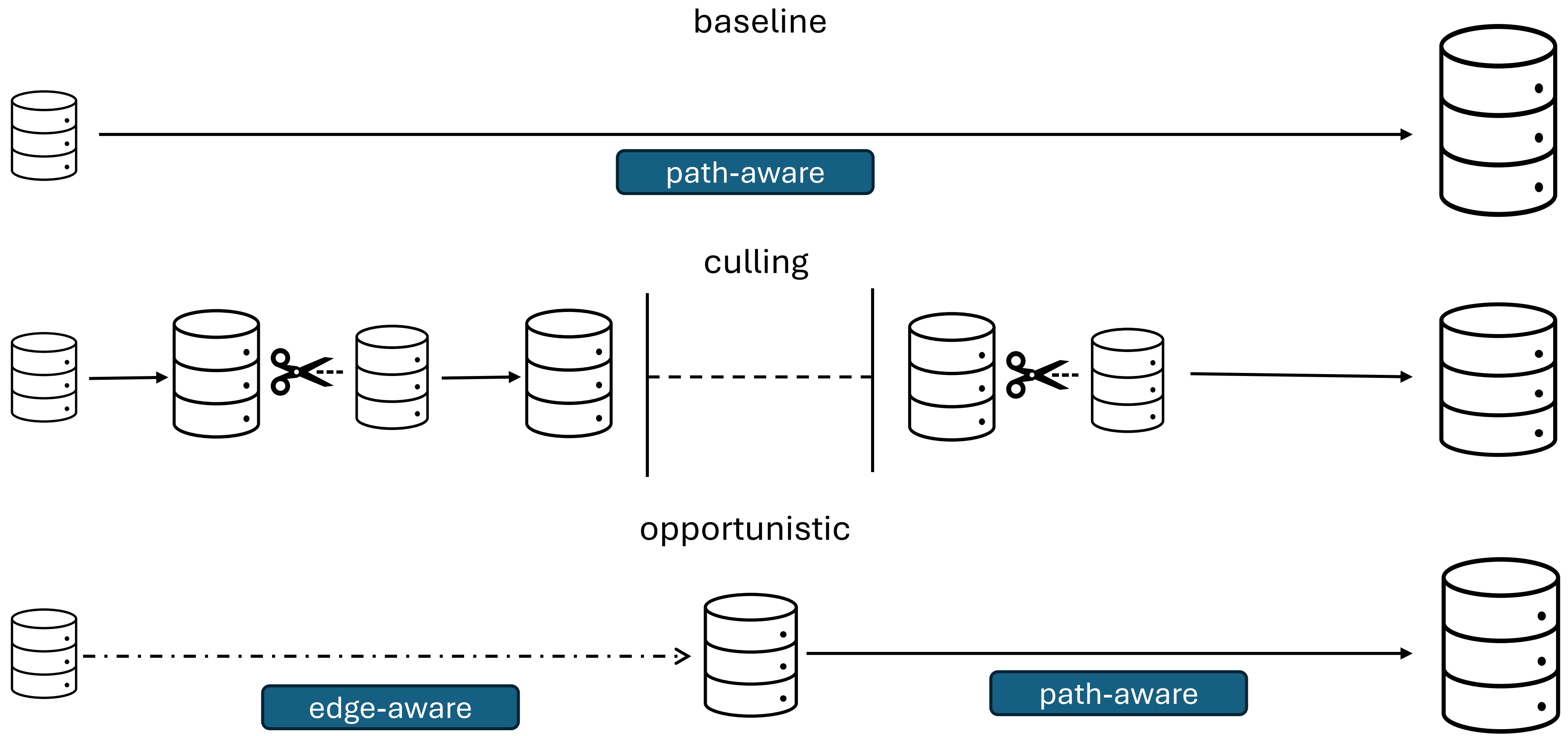
Challenge (ii): queue explosion

- Our path-aware fuzzer finds **many** bugs missed by the edge coverage counterpart
- Inefficiency due to **queue explosion** (expected)
    - Typical of more sensitive feedbacks
    - Many more *interesting* inputs retained in the queue
    - **Hinders** overall fuzzing **efficiency** (less bugs and lower code coverage)
    - 4.5x larger queues on average, peaks of 62.3x and 37.6x

Challenge (ii): queue explosion

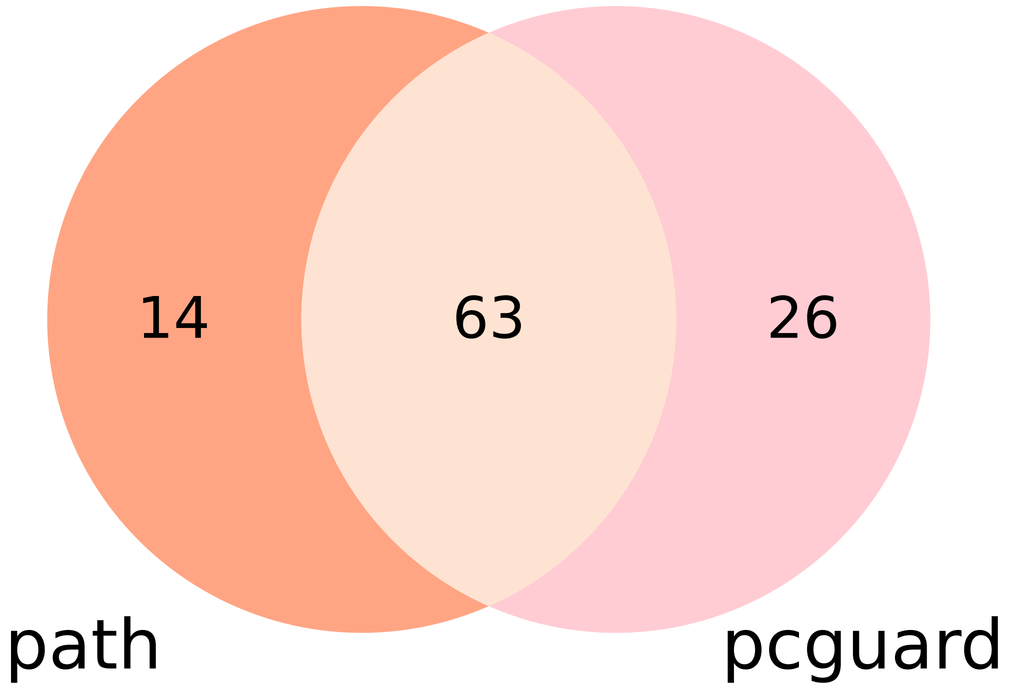
- Our proposal: 2 exploration-biasing strategies
  - **Culling**: periodically trim the fuzzer’s queue and resume the exploration
  - **Opportunistic**: capitalize on the code coverage wealth from a more efficient feedback, used as a baseline for path-aware exploration

Visual Comparison



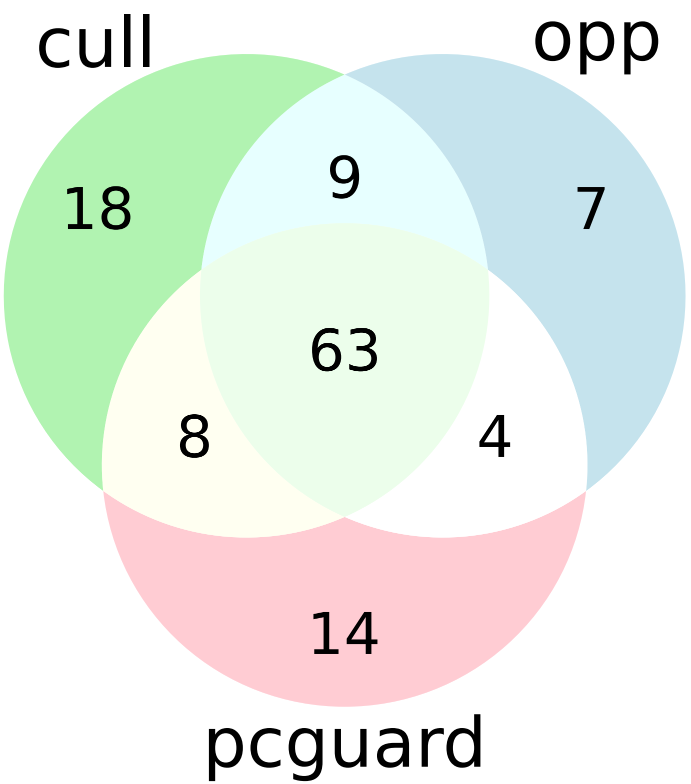
Evaluation

- **AFL++**-based prototype
- **UniFuzz** benchmark (18 subjects)
- **3 path-aware configurations (fuzzers)**: baseline (*path*), culling (*cull*), opportunistic (*opp*)
- **pcguard** as the state-of-the-art implementation of edge coverage (AFL++ default)
- **10 runs** per <subject, fuzzer> pair
- Automated crash clustering, followed by **manual analysis** to identify **unique bugs**



Main Results (1/3)

- The baseline path-aware fuzzer (77 bugs) already finds many **bugs uncaught** by pcguard (89 bugs)
  - 14 (18%) entirely missed by pcguard
- **Inefficiency** due to queue explosion
  - 12 fewer bugs than pcguard (-13.5%)
  - 7% lower fuzzing throughput (execs/sec) on average
  - 4.5x larger queues on average
  - Lower code coverage (87.3% of the code reached by pcguard)

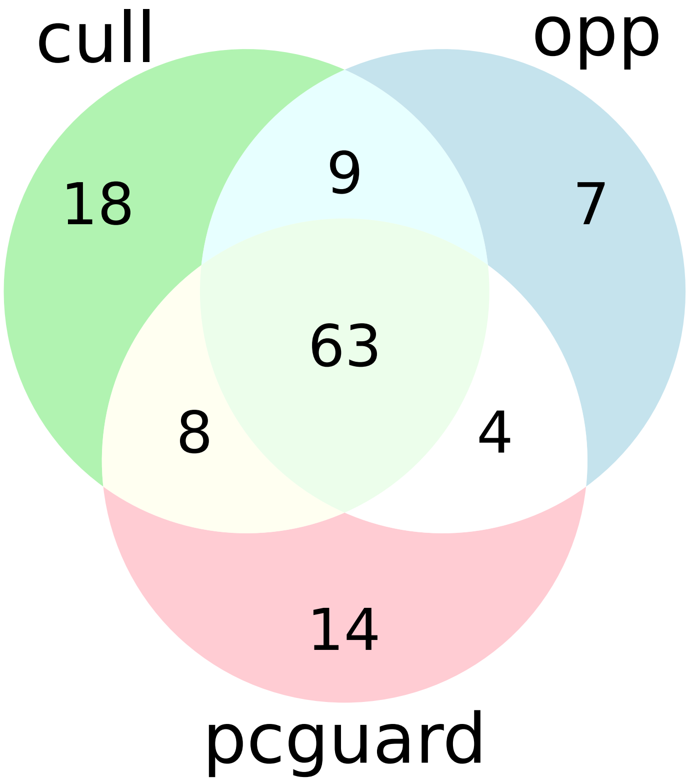


Main Results (2/3)

- Exploration-biasing strategies **boost bug-finding** abilities of *path* by tackling queue explosion

- cull (98 bugs) **surpasses** pcguard (89 bugs)
  - 9 more bugs than pcguard in total (+10.1%)
  - 27 of them (27.5%) missed by pcguard
  - Mitigates queue explosion (2.2x on average)
  - Covers new code w.r.t. path and pcguard

Main Results (3/3)



- Exploration-biasing strategies **boost bug-finding** abilities of *path* by tackling queue explosion

- opp (83 bugs) is another profitable design point
  - 6 more bugs than path (+7.8%)
  - 6 fewer bugs than pcguard (-6.7%)
  - 16 bugs missed by pcguard (19.3%)
  - 11 bugs missed by cull (13.2%)
  - Covers more code than path and cull, and different code than pcguard

Takeaways

Path-awareness grants the fuzzer an **increased visibility** over the program, revealing many **subtle bugs** that may go undetected

Higher sensitivity entails **runtime costs**, which can be mitigated with exploration-biasing strategies

No single right way of wielding this increased visibility (different strategies yield unique bugs not found by the other ones)

All the bugs detected by our path-aware fuzzers were **within reach of pcguard**, but it failed to uncover them due to its **coarse-grained** nature

Conclusions

- **Path-aware feedback** for fuzzing is both **feasible** and **profitable** (many bugs found, within reach of the current state-of-the-art implementation for edge coverage)

- Significant **untapped potential** for fuzzing research
  - First time this new dimension is studied with intra-procedural paths
  - Warrants further investigation in this new direction

github.com/**Sap4Sec**/path-aware-fuzzing/

giacomo.priamo@diag.uniroma1.it

Thank you