Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis

SOSP 2021. Gong, Fonseca, Altinbucken, Maniatis

Kernel Concurrency Analysis

- Detecting kernel concurrency bugs
  - Kernels are huge (~30 million LoC) with complex interfaces (> 400 sys calls)
  - Bugs are triggered on specific inputs and specific interleavings
  - **Automation** is a need; Cannot be **exhaustive** in search

Past work

- Fuzzers: mostly for sequential executions.
  - Razzer: data-race detection **statically** and generates concurrent tests (high false positive rate)
  - Krace: No support for scheduling hint — explores a very large space.
  - Static/Dynamic data race detectors — miss other concurrency-related errors (**order** and **atomicity** violations)
- Sample mem access and randomly delay them using H/w watchpoints
- Use of PCT

Solution offered?

Snowboard

- Generates sequential tests (uses an existing fuzzer); identifies **PMC**
- Prioritises and fuse sequential test to construct concurrent tests
  - Based on a reader and writer accessing the same mem location
  - **Assumption**: potential inter-thread interactions can be predicted based on analysing memory accesses (**sequentially** and in **offline** mode)
- Uses a test selection metric that is more general than structural-coverage metrics
  - Control-flow edges, def-use, instr-pair etc.

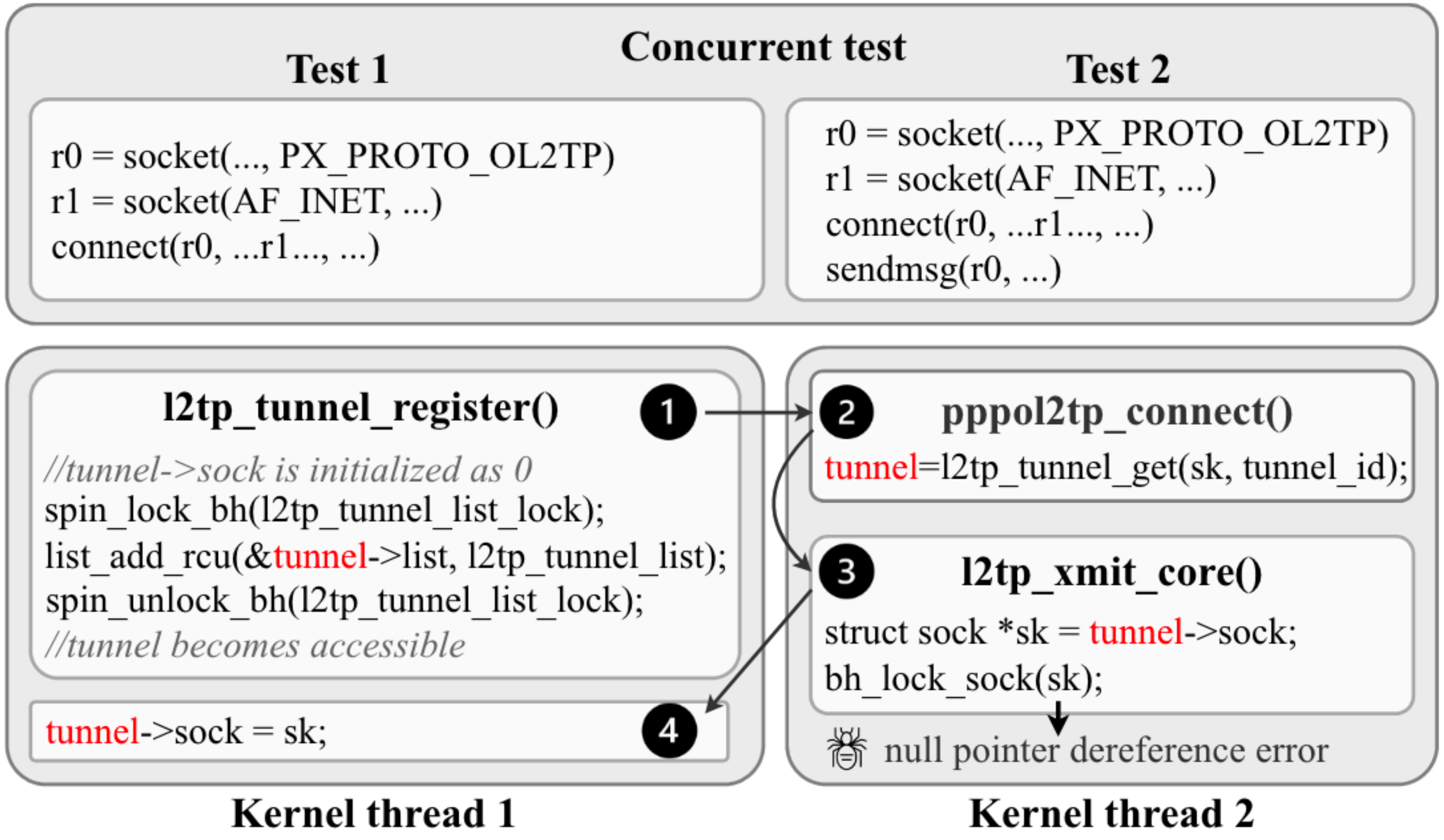
Overview of results

- 14 concurrency bugs in Linux kernels 5.3.10 and 5.12-rc3
  - Four are non-DR type causing kernel panics and file system errors
  - 12 bugs were confirmed by the developer and 6 were fixed
  - 2 have existed in the stable version of the kernel for many years.
  - Of all types — order violations, data races, and atomicity violations.

PMC — Potential Memory Communication

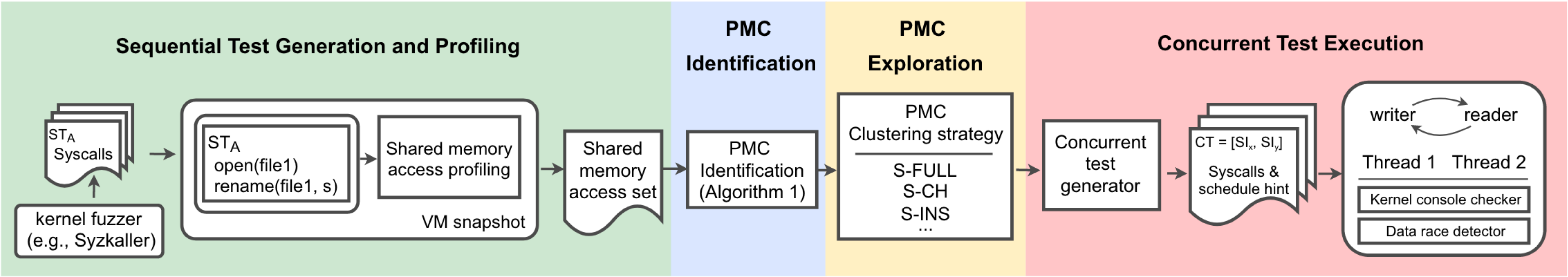
- Conditions for a PMC to occur:
  - Thread A makes a write access
  - Thread B makes a read access
  - The mem regions of the two accesses must overlap
  - The write access updates the mem area with a different value
- Note that the above conditions do not require **synchronisation** — which means more general than data races!

An example

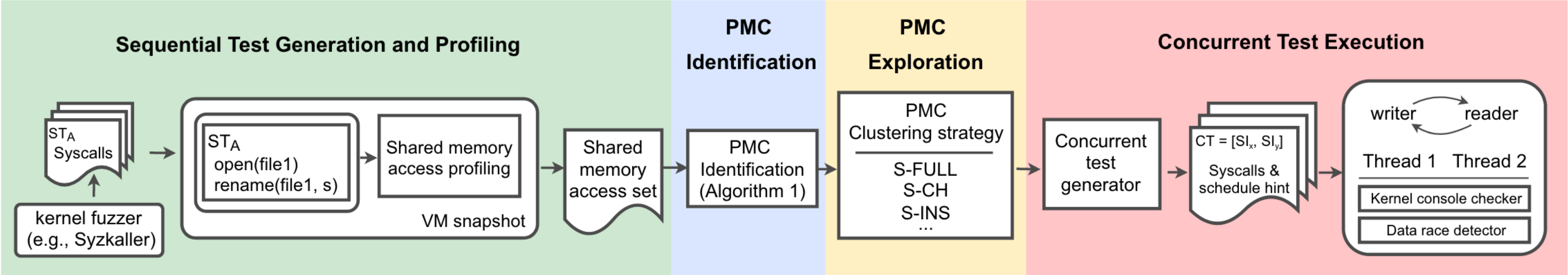


- Left kernel thread - writer
- Right kernel thread - reader
- pppol2tp_connect() - fetches the previously registered tunnel
- l2tp_tunnel_register() — registers a new tunnel
- Bug — reader accesses the tunnel before the writer has initialised the sock field

Snowboard Design Overview

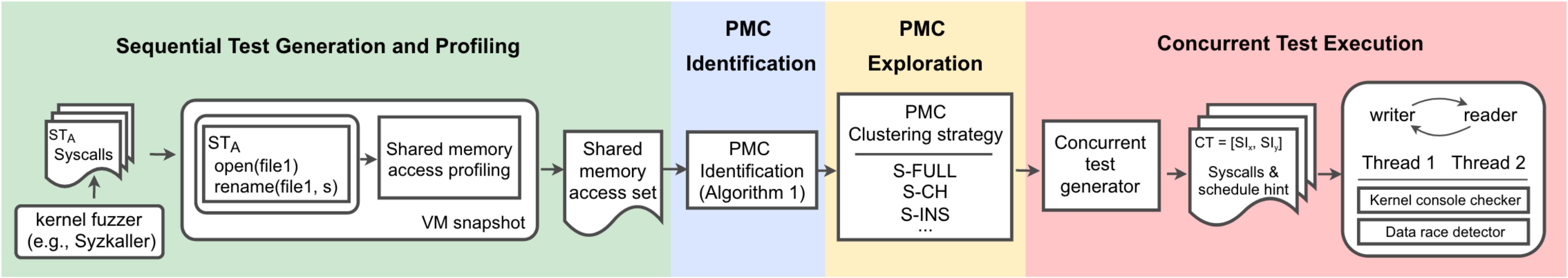


Sequential Test Generation (ST_A)



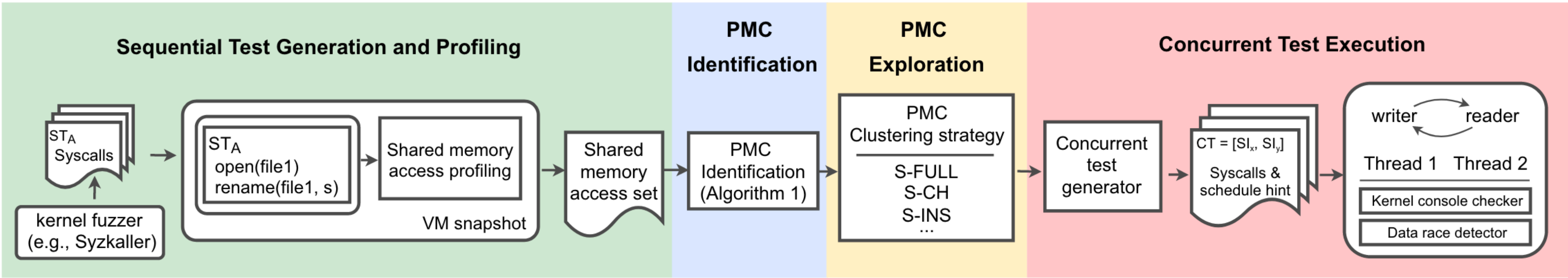
- External fuzzer, static analysis tools
- Snowboard uses the coverage metrics exported by the generator to select tests with high coverage and low overlap.
- Snowboard dynamically profiles (sequential execution) selected tests by recording
  - Type of mem access and instruction addresses, address range, vals read/written
  - Runs from the same fixed initial kernel state
  - Standard assumption: only non-stack accesses are potentially shared (uses ESP register to prune stack accesses)

PMC Identification



- Gathers all shared accesses across all sequential tests
    - Indexes them by the mem range they access
- Detect overlapping mem ranges for reads and writes
- If for each pair <W, R> the value written is different from the value read, then designated as a PMC.
- Implemented as a nested scan over the index structure

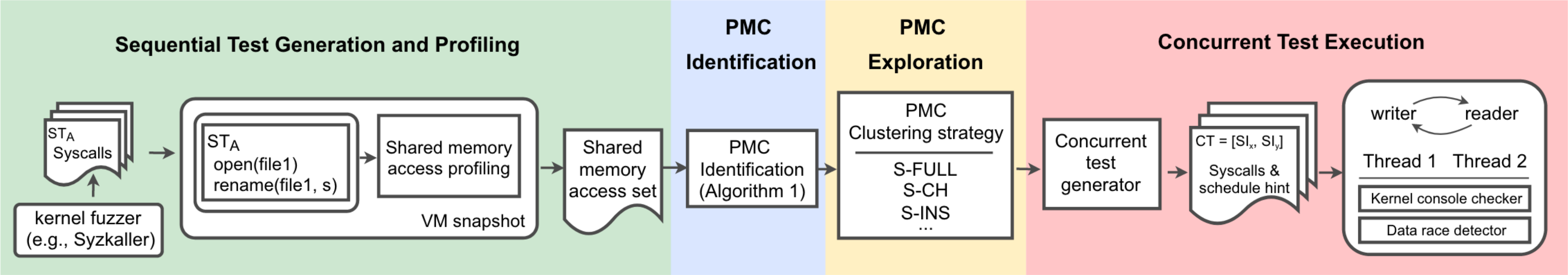
PMC Identification



- 169 billion PMCs in Linux kernel 5.12-rc3 — too large!
- Insights — many PMCs are equivalent under some criteria.
  - Form clusterings of PMCs on such criteria
- Not sound but complete
  - It may club two PMCs even when they expose distance misbehaviours

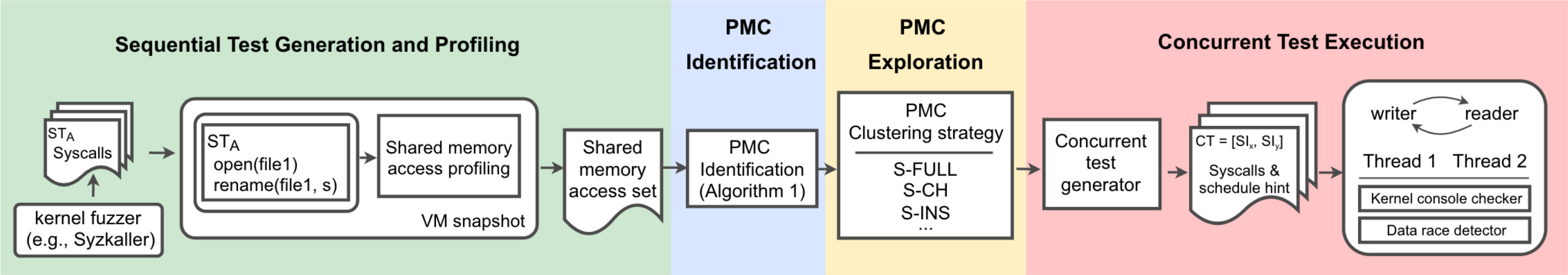
Clustering strategy	Clustering Key / [Filter Predicate]
S-FULL	$(\text{ins}_w, \text{addr}_w, \text{byte}_w, \text{value}_w, \text{ins}_r, \text{addr}_r, \text{byte}_r, \text{value}_r)$ / [*True*]
S-CH	$(\text{ins}_w, \text{addr}_w, \text{byte}_w, \text{ins}_r, \text{addr}_r, \text{byte}_r)$ / [*True*]
S-CH NULL	$(\text{ins}_w, \text{addr}_w, \text{byte}_w, \text{ins}_r, \text{addr}_r, \text{byte}_r)$ / [$value_w$=0]
S-CH UNALIGNED	$(\text{ins}_w, \text{addr}_w, \text{byte}_w, \text{ins}_r, \text{addr}_r, \text{byte}_r)$ / [($addr_r$!= $addr_w$ or $byte_r$!= $byte_w$)]
S-CH DOUBLE	$(\text{ins}_w, \text{addr}_w, \text{byte}_w, \text{ins}_r, \text{addr}_r, \text{byte}_r)$ / [*df_leader*]
S-INS	$(\text{ins}_{w/r})$ / [*True*]
S-INS-PAIR	$(\text{ins}_w, \text{ins}_r)$ / [*True*]
S-MEM	$(\text{addr}_w, \text{byte}_w, \text{addr}_r, \text{byte}_r)$ / [*True*]

Concurrent Test Execution



- PMC can map to multiple test pairs
  - Randomly choose one to construct CT
  - CT = <t1, t2, sched-hint>
  - Scheduling component
    - Trigger the PMC and not trigger the PMC
    - Avoid deadlocks/livelocks

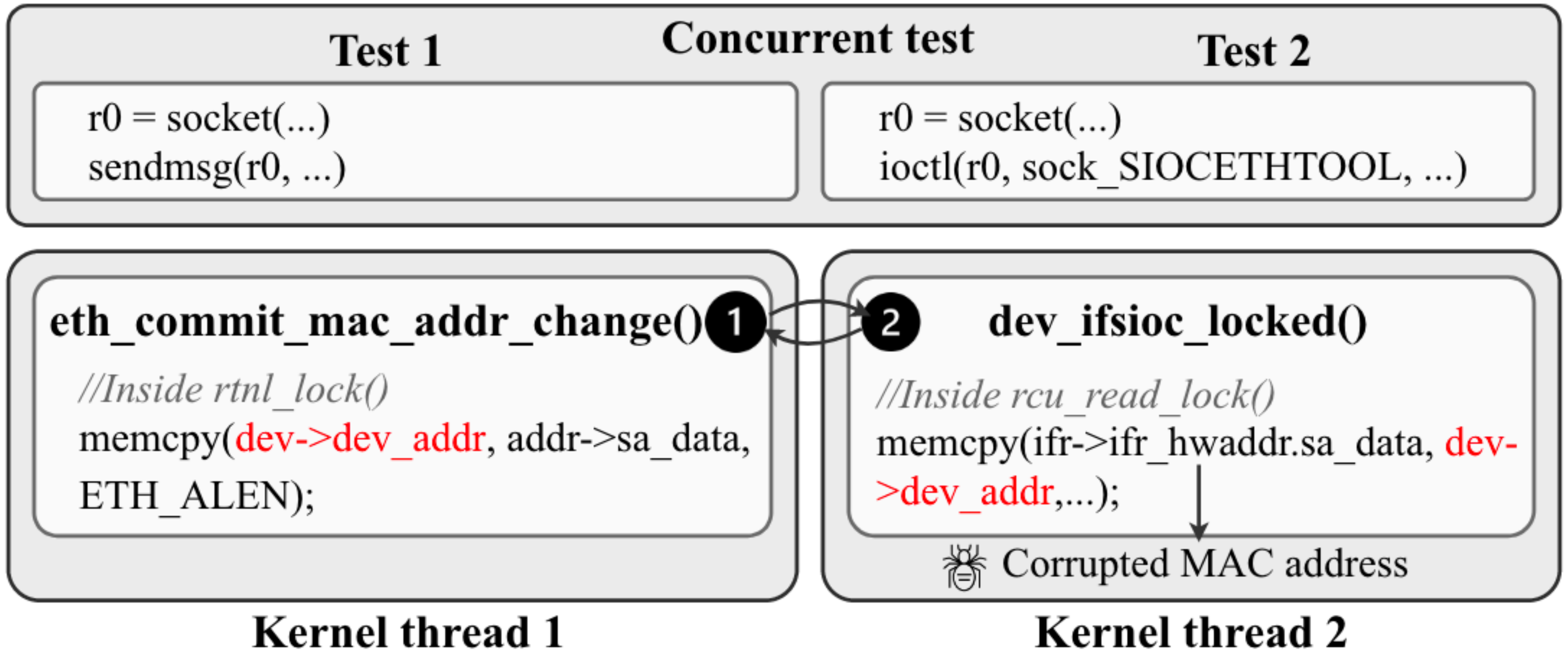
Concurrent Test Execution



- Algorithm
  - Check if thread is live
    - If not then yield control to other thread
  - For each access in the currently executing instruction
    - Switch to nondet scheduling if pmc access is arriving (for future trials)
    - If pmc access performed then note the previous access to the PMC one
    - Now switch to nondet scheduling
  - If the current execution ends in a bug — record it.
  - Check if other PMCs were observed in the trial - if yes then record them for future trials

Real harmful bugs detected

- Reader and writer have different locks
- A bad MAC address can be read by the reader
- Was unreported for 10 years.



Discussion

- Low precision (36 %) yet was able to find subtle errors.
- What about bugs involving more than 2 threads or more than one variable?
  - Why did they leave out deadlocking executions from consideration?
- Weak memory orderings?
- Breaking of assumptions for PMC computation: If instructions touch large memory segments (DBMS updating in-memory indexes).
- Guided test generation
- Any other?