

Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints

Dokyung Song, Felicitas Hetzelt, Jonghwan Kim,
Brent Byunghoon Kang, Jean-Pierre Seifert, Michael Franz



Device Drivers are Still Vulnerable in 2020

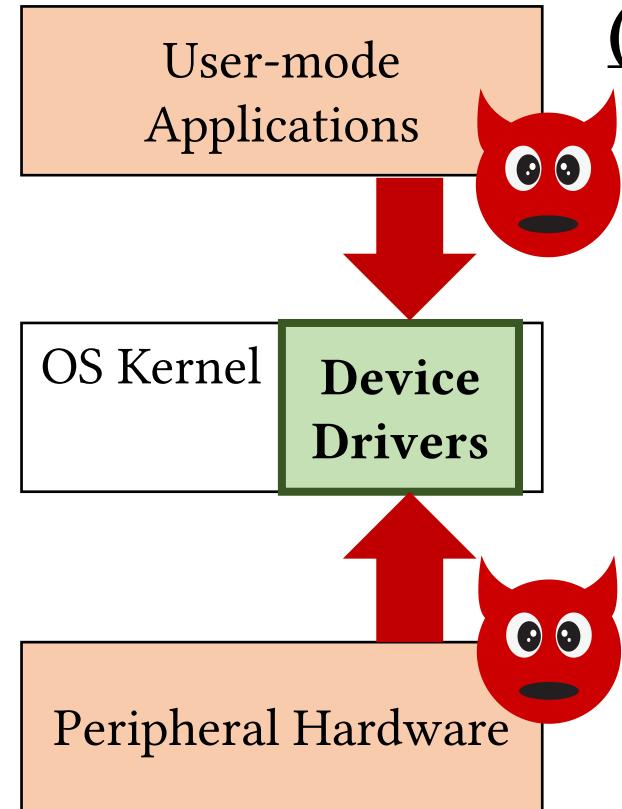
Security Bulletin: NVIDIA GPU Display Driver - June 2020

NVIDIA GPU DISPLAY DRIVER

CVE-ID	Description	Base Score	Vector
CVE-2020-5962	NVIDIA GPU Display Driver contains a vulnerability in the NVIDIA Control Panel component, in which an attacker with local system access can corrupt a system file, which may lead to denial of service or escalation of privileges.	7.8	AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H
CVE-2020-5963	NVIDIA CUDA Driver contains a vulnerability in the Inter Process Communication APIs, in which improper access control may lead to code execution, denial of service, or information disclosure.	7.8	AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H
CVE-2020-5964	NVIDIA GPU Display Driver contains a vulnerability in the service host component, in which the application resources integrity check may be missed. Such an attack may lead to code execution, denial of service or information disclosure.	6.5	AV:L/AC:L/PR:H/UI:R/S:U/C:H/I:H/A:H
CVE-2020-5965	NVIDIA GPU Display Driver contains a vulnerability in the DirectX 11 user mode driver (nvwgf2um/x.dll), in which a specially crafted shader can cause an out of bounds access, leading to denial of service.	5.5	AV:L/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:H
	NVIDIA Windows GPU Display Driver contains a vulnerability in the kernel mode		2

Why Vulnerabilities in Drivers matter?

(i) Highly Privileged



(ii) Wide Attack Surface

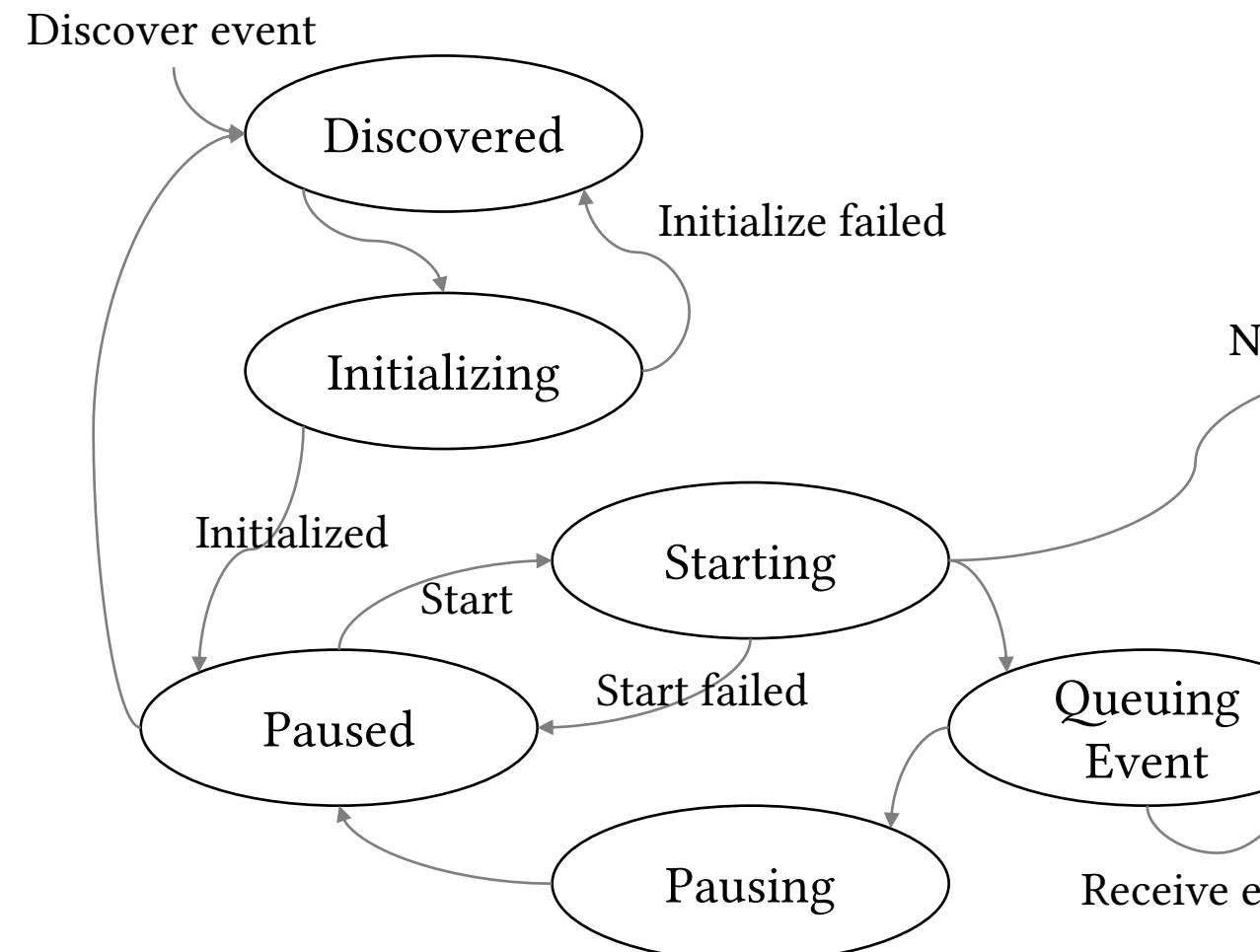
System Call Attack Surface

- `open(/dev/...)`
- `read(...)` and `write(...)`
- `ioctl(...)`
- ...

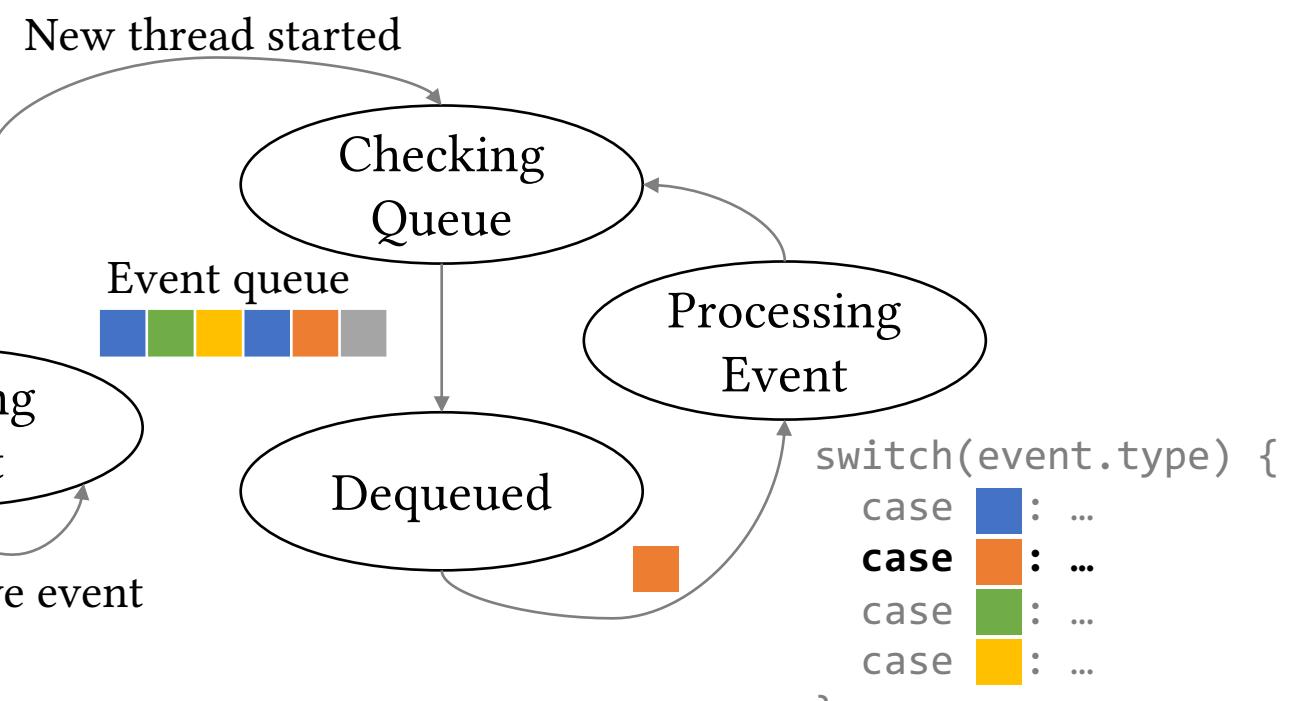
Peripheral Attack Surface

- PCI
- USB
- ...

Stateful, Event-Driven Nature of Drivers



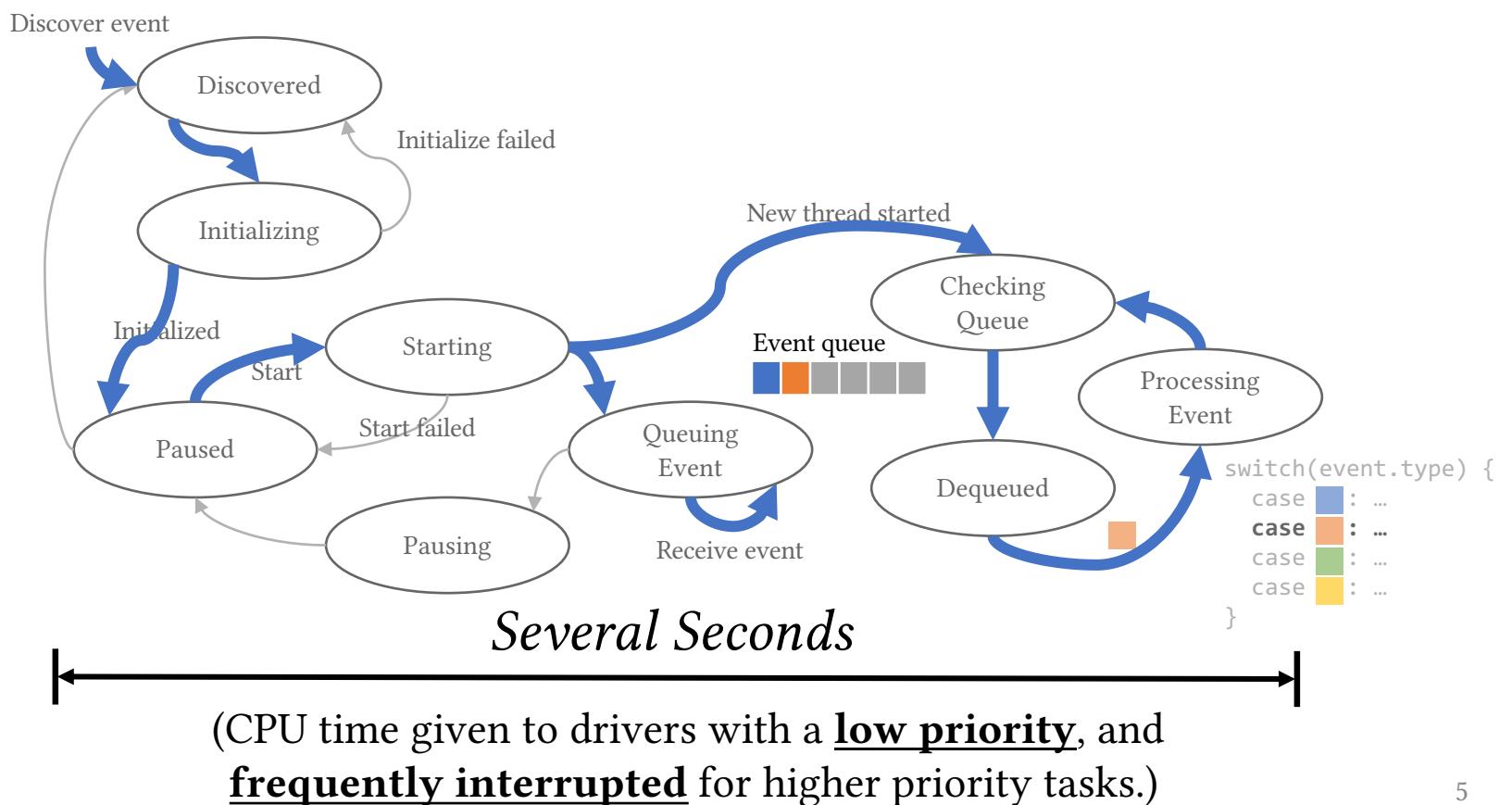
<Driver Execution Flow Example>



Problem: Fuzzing Device Drivers is Slow

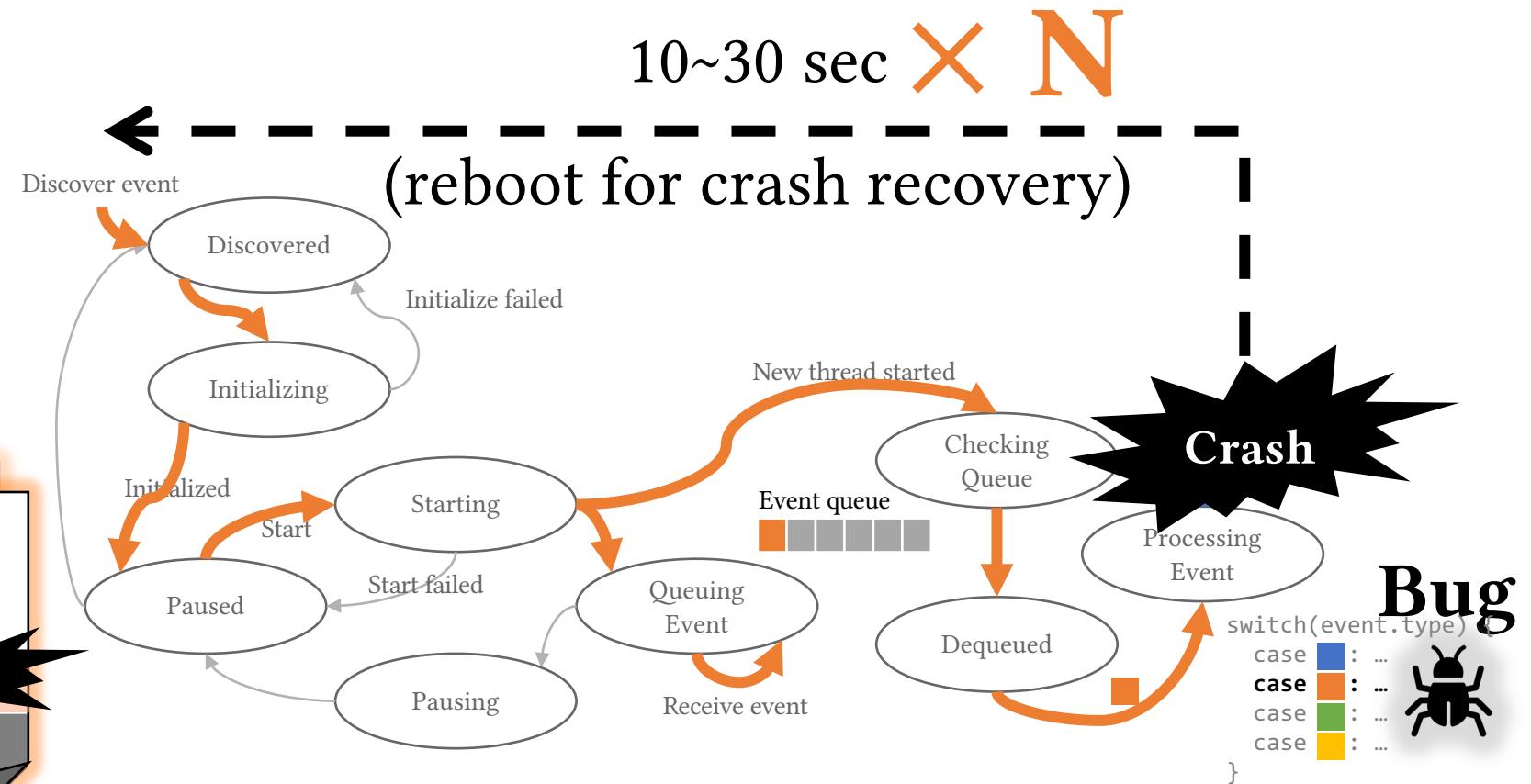
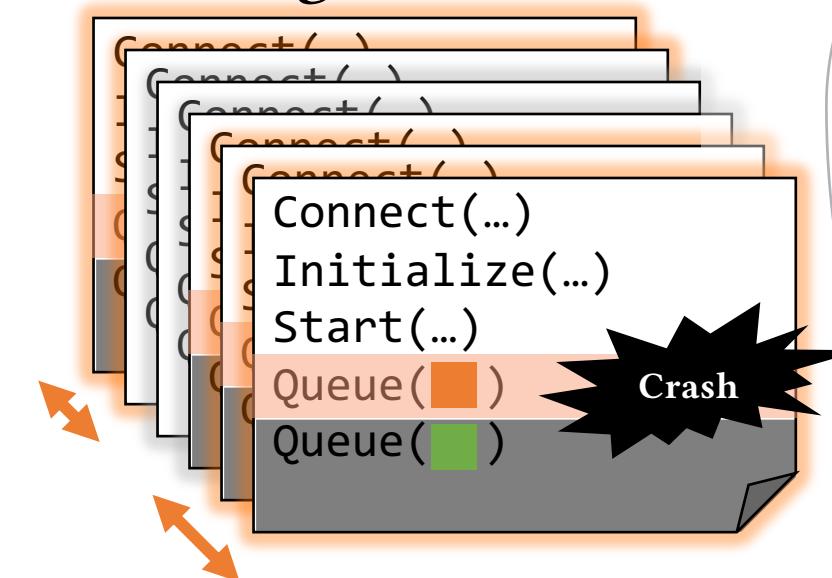
A fuzzer generated

```
Connect(...)  
Initialize(...)  
Start(...)  
Queue(█)  
Queue(█)
```



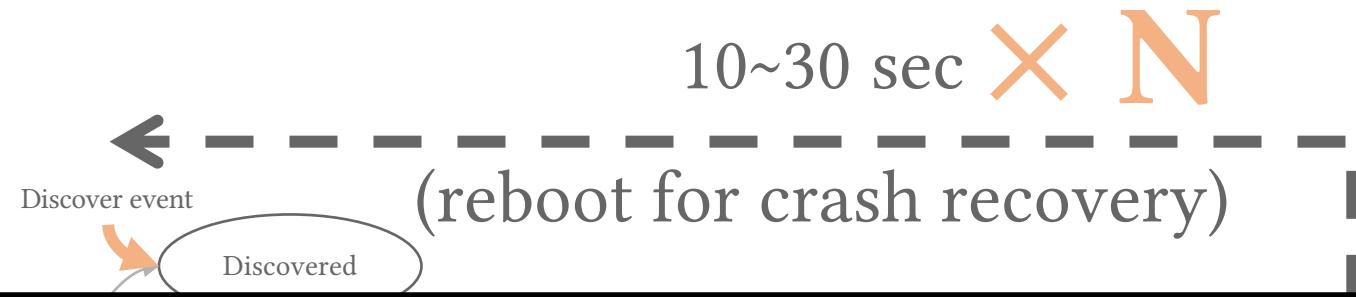
Problem: Slowed down further by Crashes

A fuzzer generated

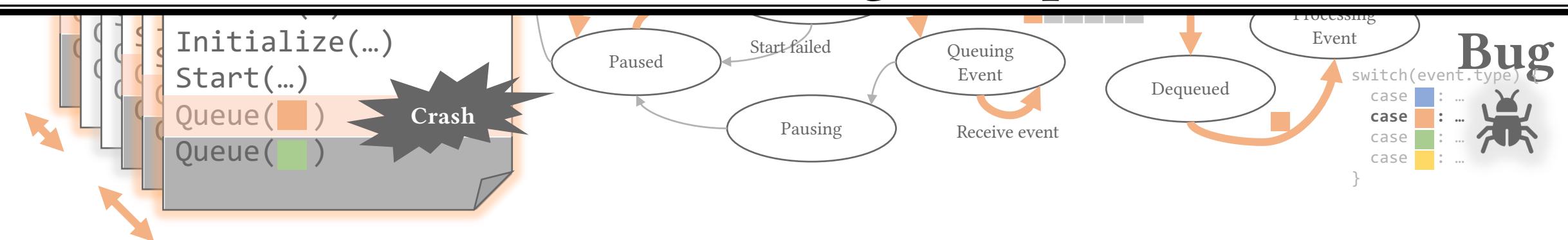


N test inputs hit the Bug (or Shallow Bug)

Problem: Slowed down further by Crashes



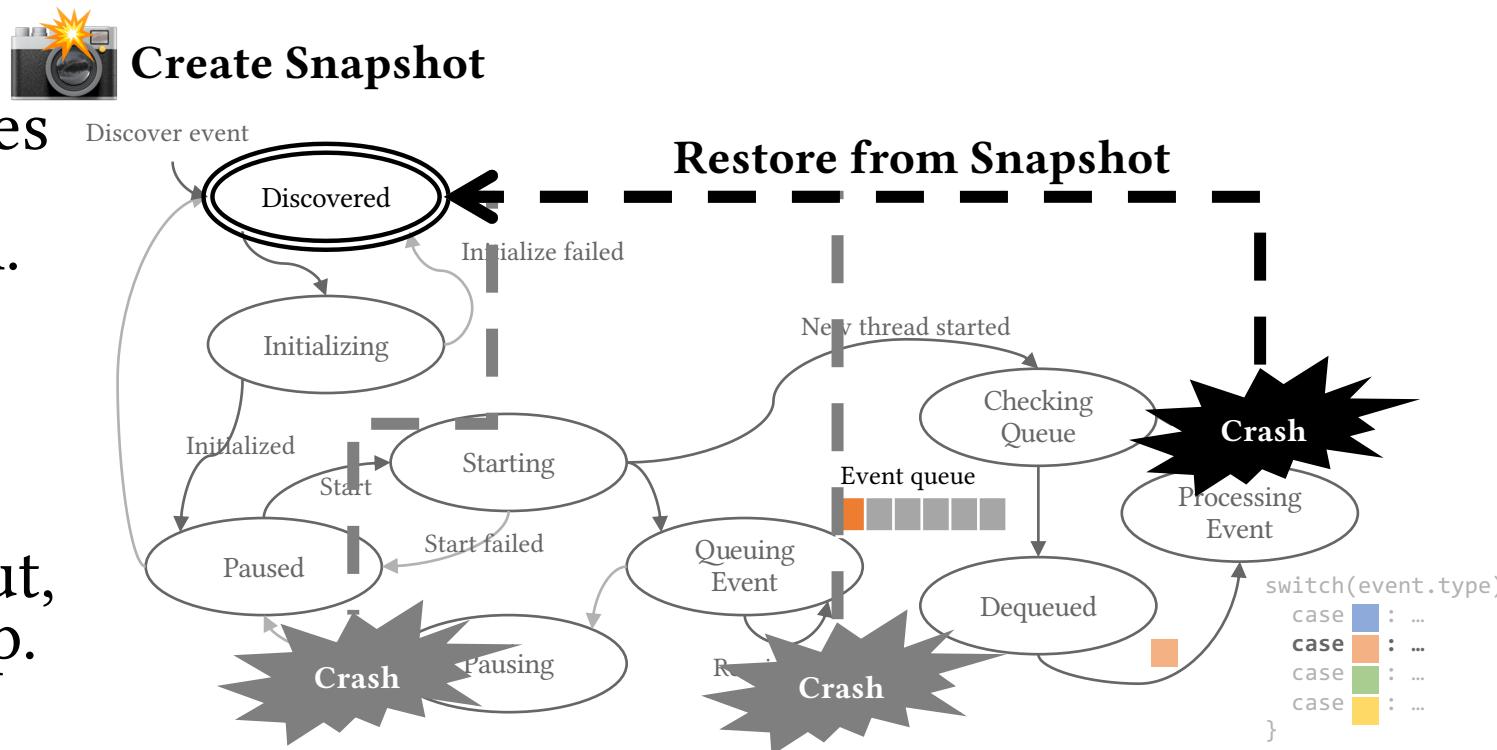
Hard to reach **deep bugs**
when shallow bugs are present.



N test inputs hit the Bug (or Shallow Bug)

Existing Approach: Fuzzing with Snapshot

- Snapshot restoration ensures no interference between test inputs, even after crash. (Clean-state fuzzing)
- Existing tools create **a single snapshot** before start processing input, typically at program startup.
- After executing each test case, the program is restored from that snapshot.



Existing Approach: Fuzzing without Snapshot

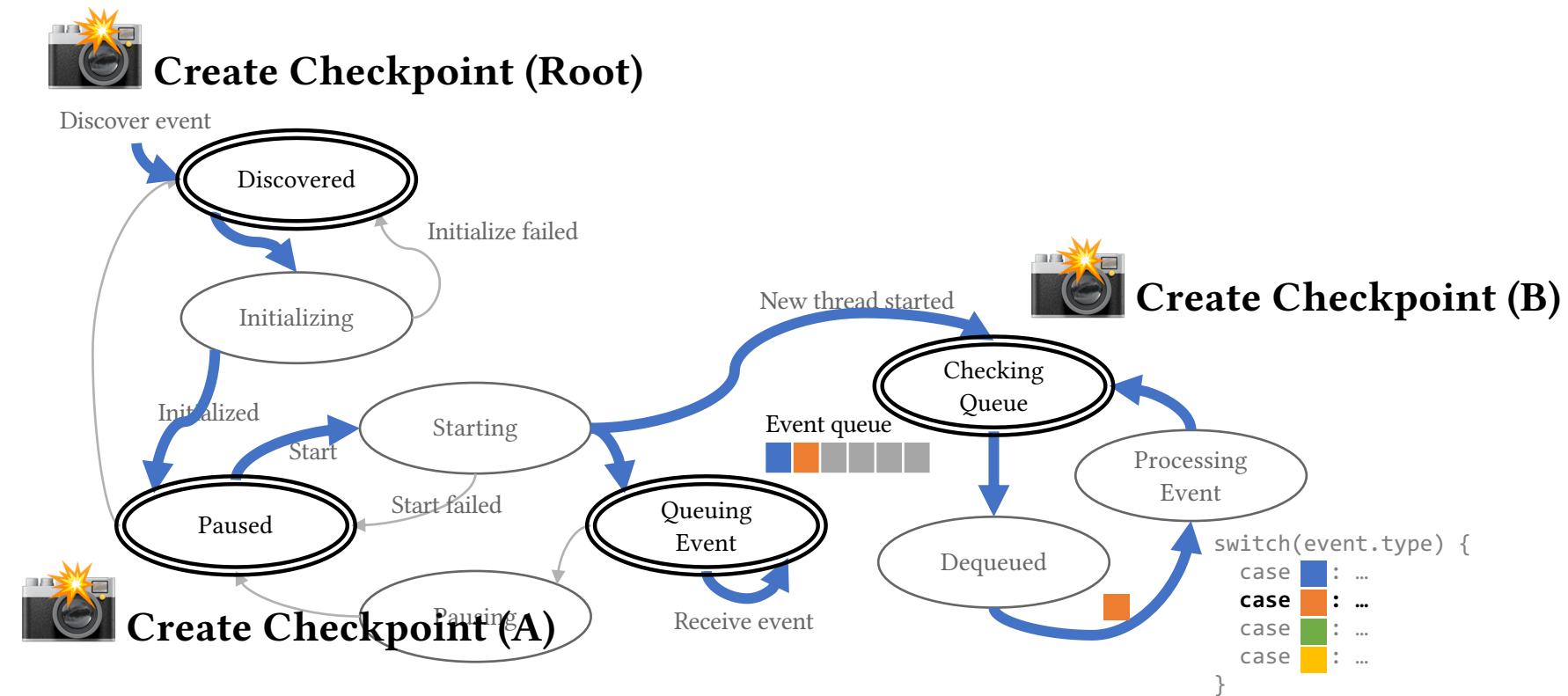
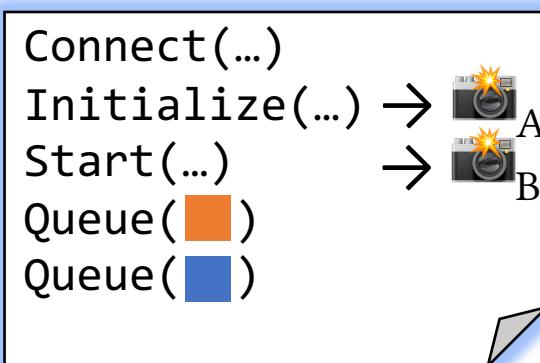
- Snapshot creation/restoration adds a run-time overhead.
- Snapshot techniques that capture kernel components can be even more costly.
 - VM Emulation + fork() → VM Emulation is slow.
 - Full VM Snapshot → VM Restore can take several seconds.
- Some fuzzers do not use snapshots.
 - [User-space] libFuzzer is an in-process fuzzer; afl has persistent mode
 - [Kernel-space] syzkaller does not use snapshots

Our Approach: Dynamic VM Checkpointing

System Initialized →  Root

:

A fuzzer generated

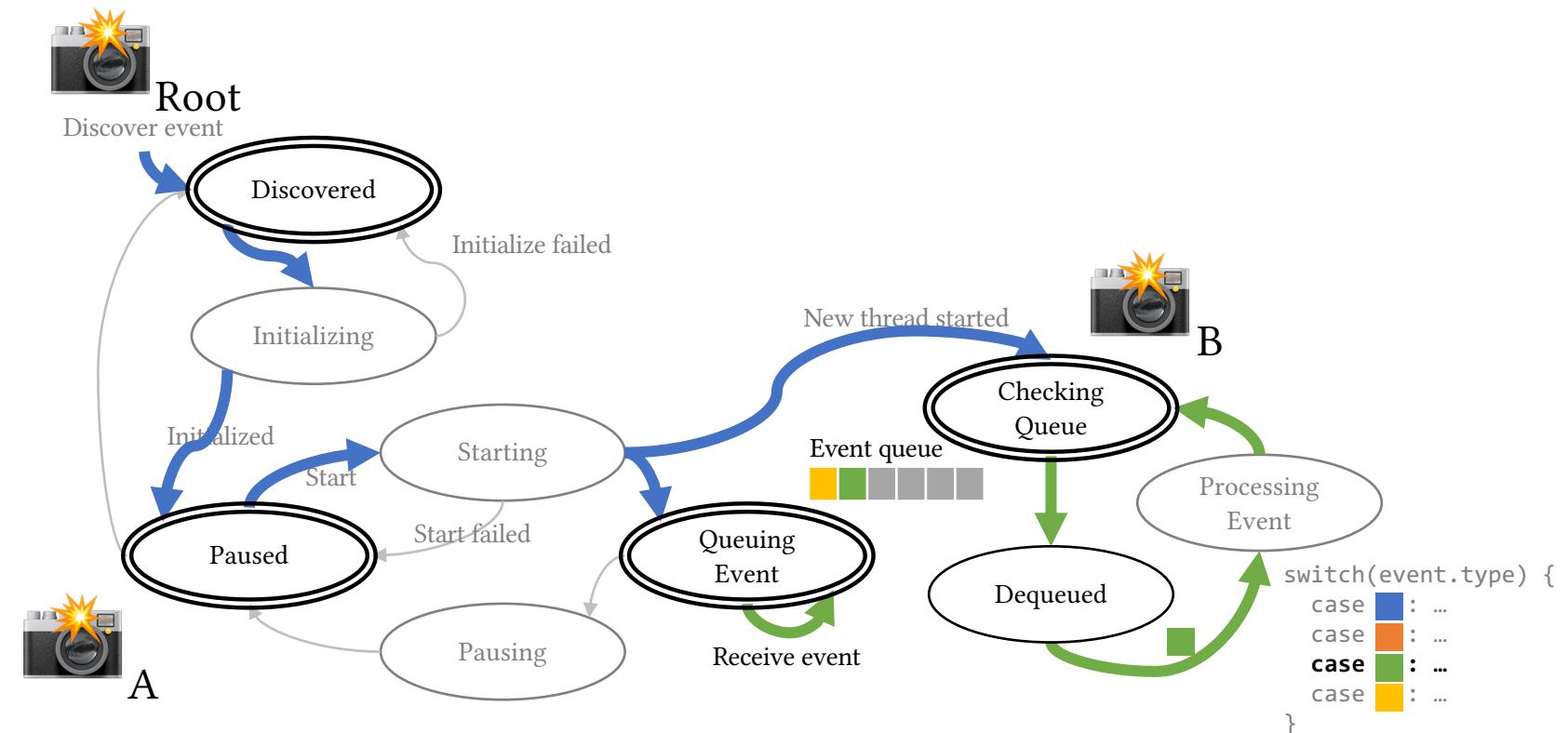
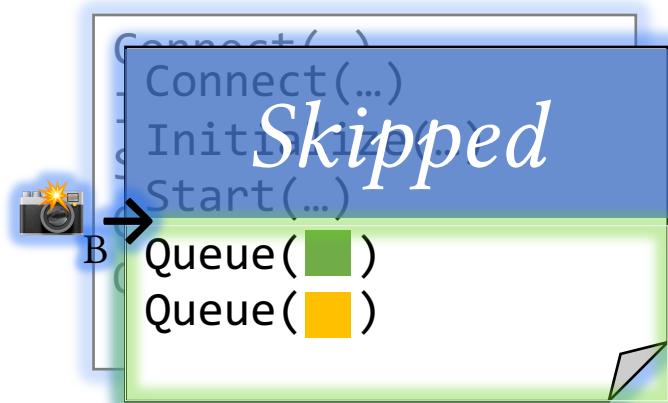


Our Approach: Dynamic VM Checkpointing

System Initialized → Root

:

A fuzzer generated

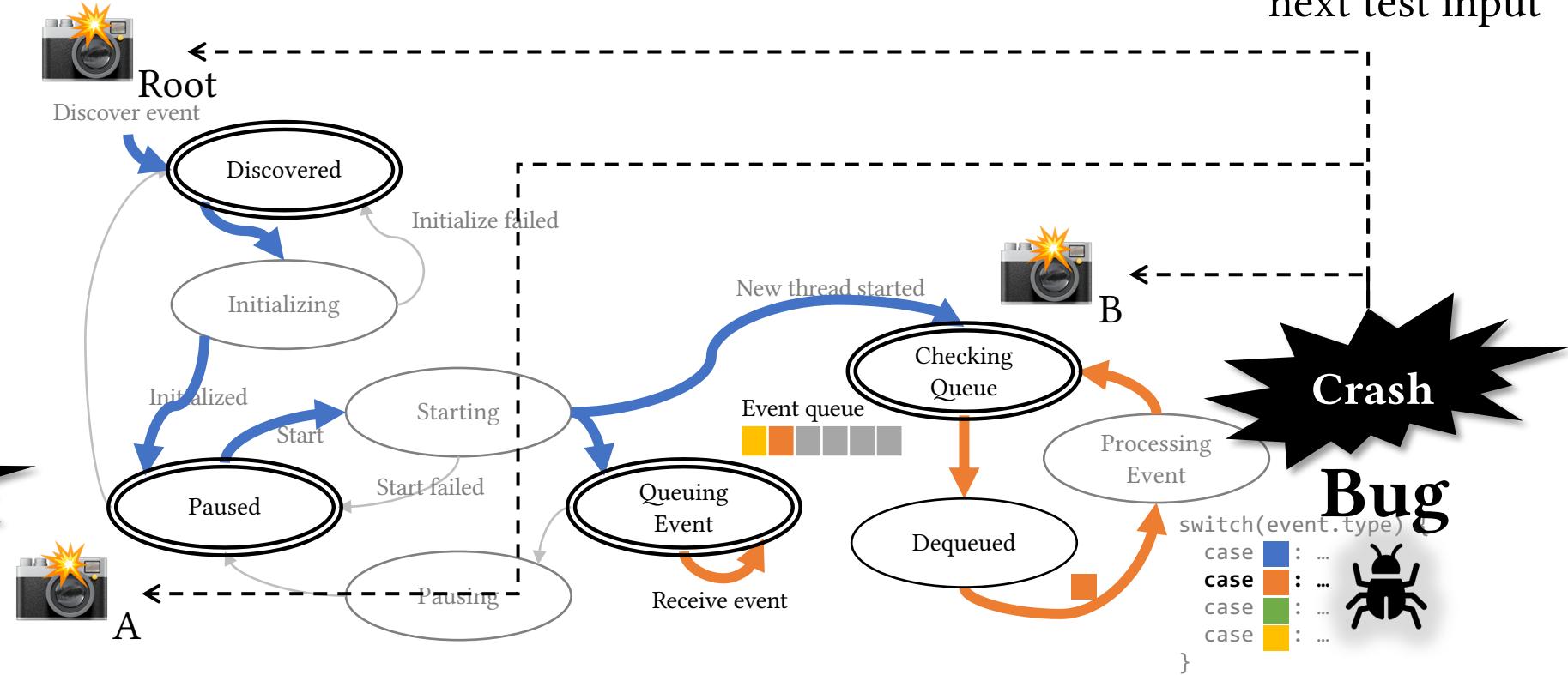
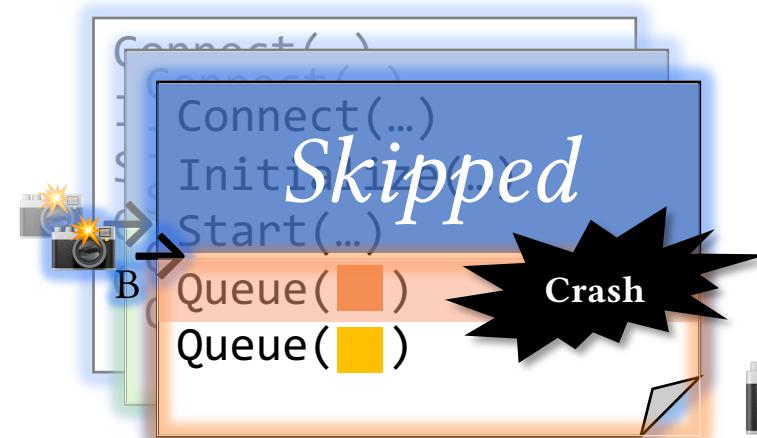


Our Approach: Dynamic VM Checkpointing

System Initialized → Root

⋮

A fuzzer generated

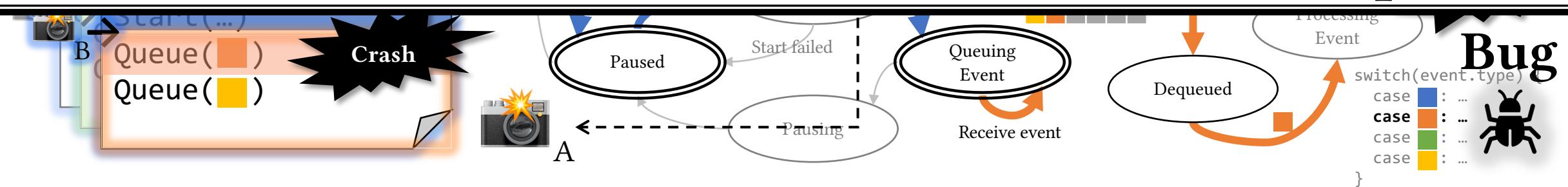


💡 Our Approach: Dynamic VM Checkpointing

Depending on
next test input

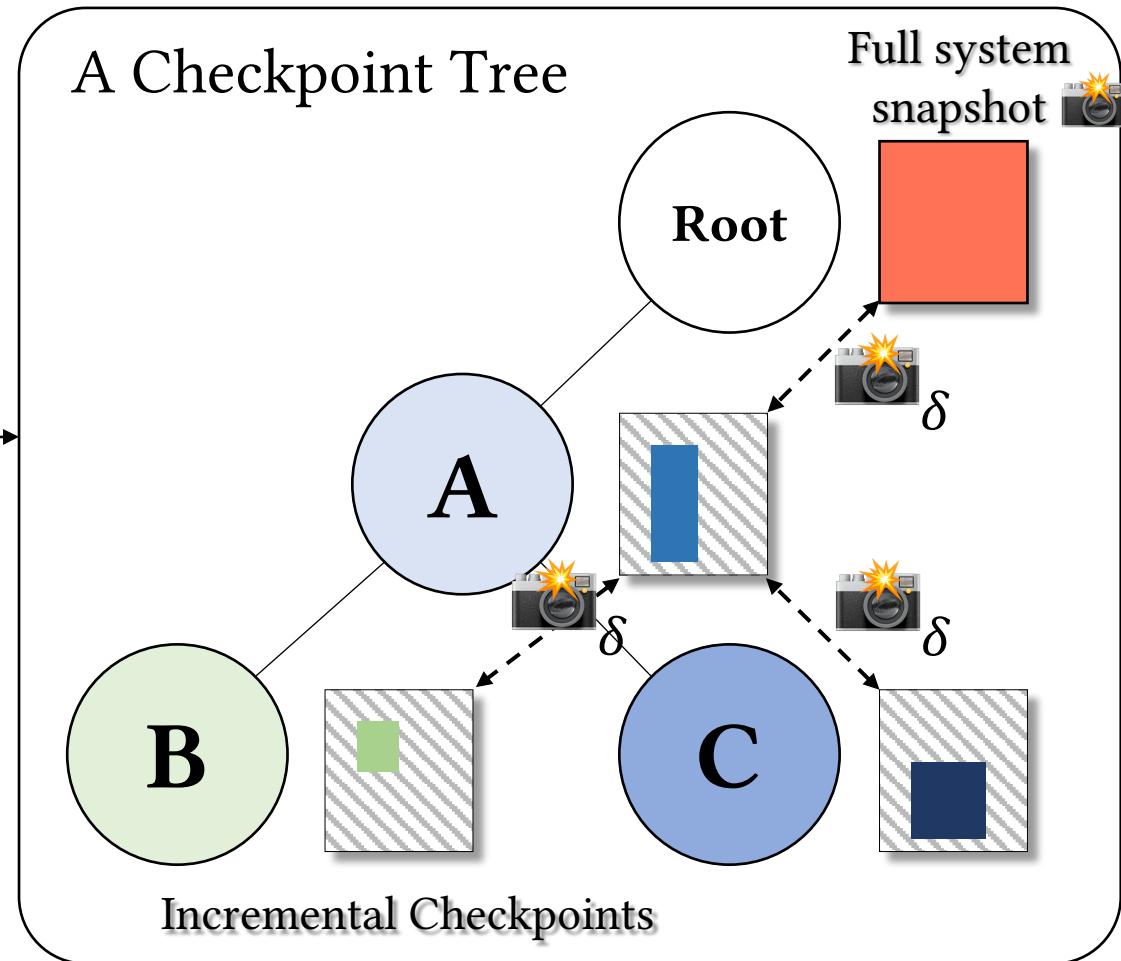


Deep code paths can be fuzzed (i) much faster,
and (ii) with no interference between test inputs.



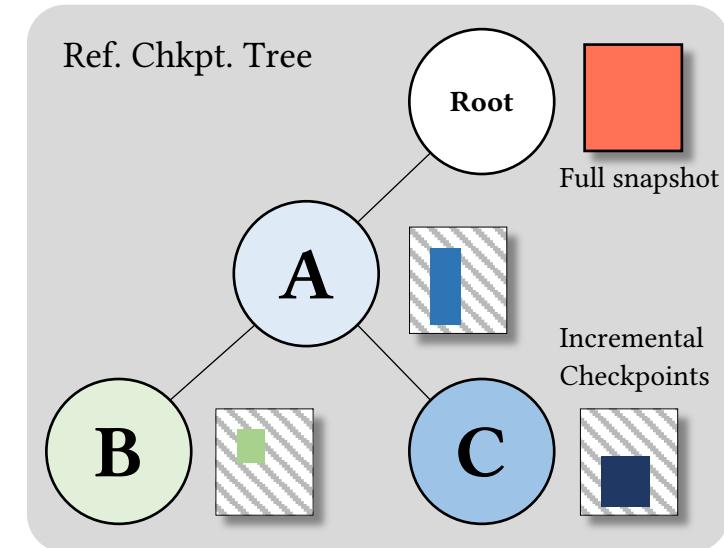
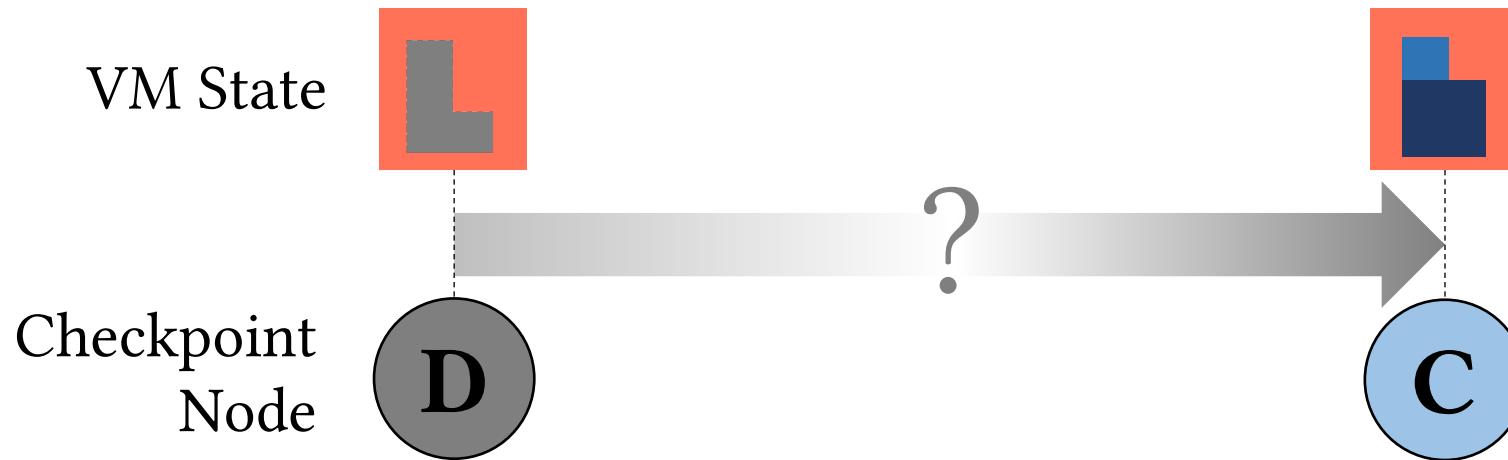
Lightweight Incremental Checkpointing

- Minimizes both the **run-time & memory overhead** of checkpoint creation
- Incremental checkpoints stored in **the checkpoint tree**
- Each tree node represents an incremental checkpoint which stores **only the pages modified w.r.t. its parent** (or “dirty pages”)



Checkpoint Restoration

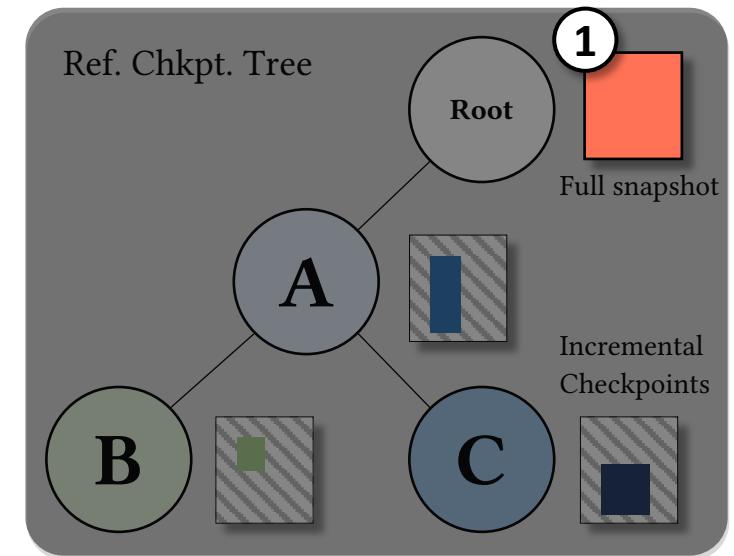
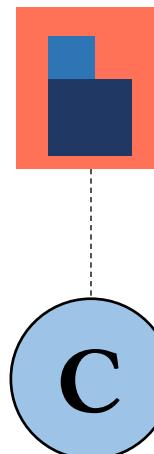
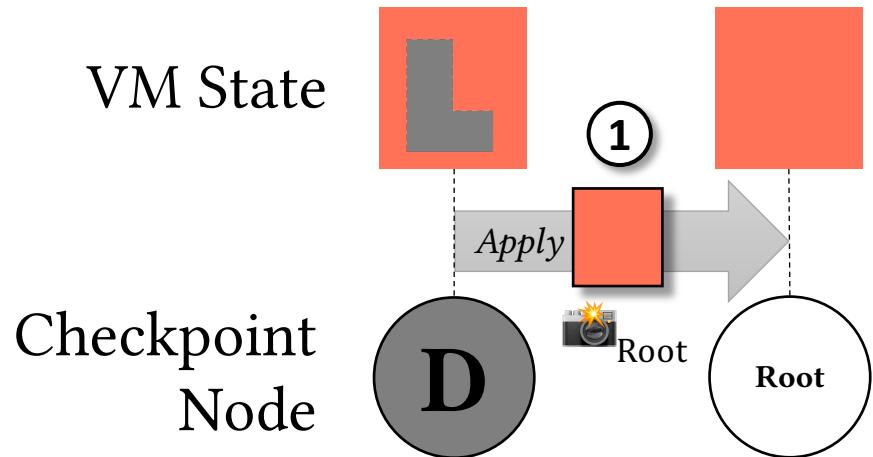
- Restoring VM from a dirty VM state **D** To Node **C**



cf. *Naïve* Checkpoint Restoration

- Top-Down

- Restoring VM from a dirty VM state **D** To Node **C**

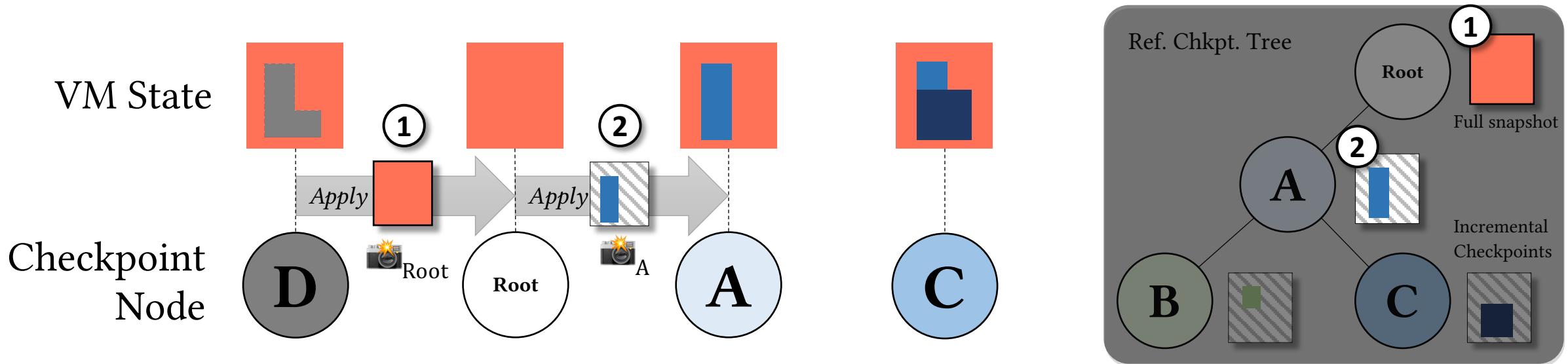


Overhead =

cf. *Naïve* Checkpoint Restoration

- Top-Down

- Restoring VM from a dirty VM state **D** To Node **C**

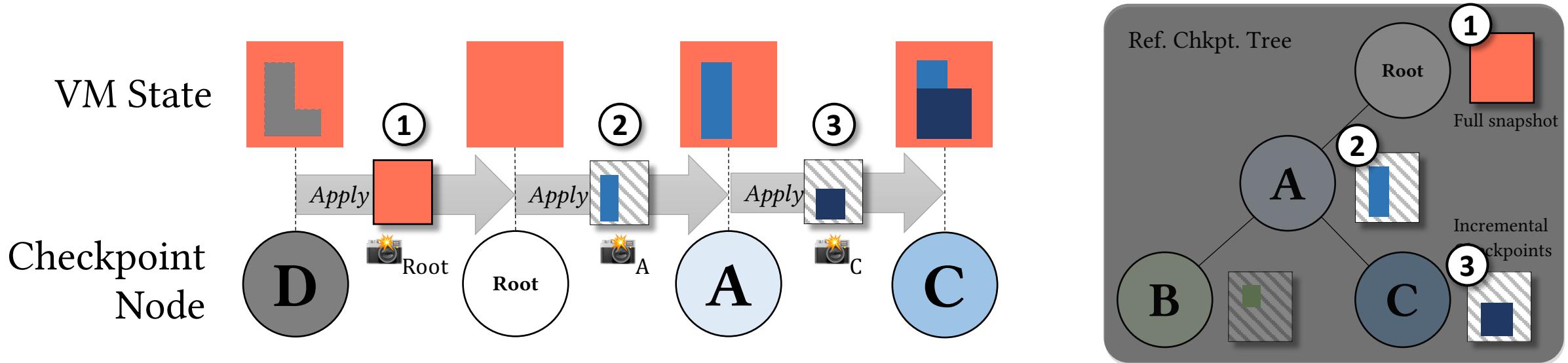


$$\text{Overhead} = \boxed{\textcolor{red}{\square}} + \boxed{\textcolor{blue}{\text{|||}}}$$

cf. *Naïve* Checkpoint Restoration

- Top-Down

- Restoring VM from a dirty VM state **D** To Node **C**

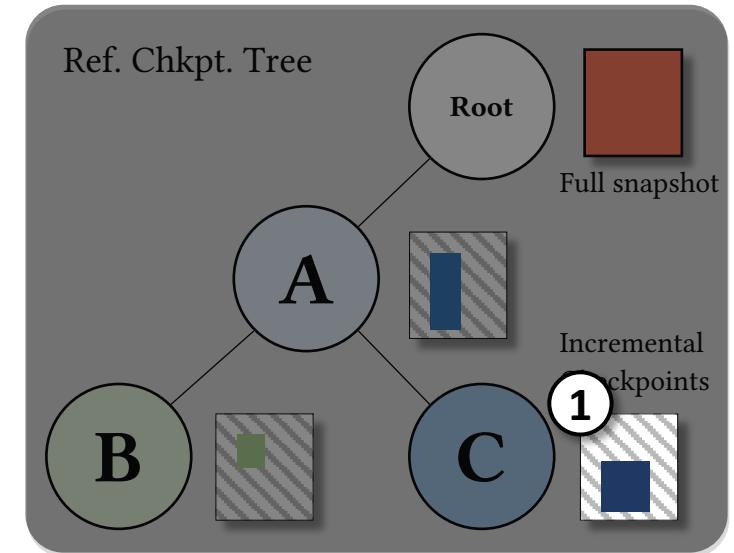
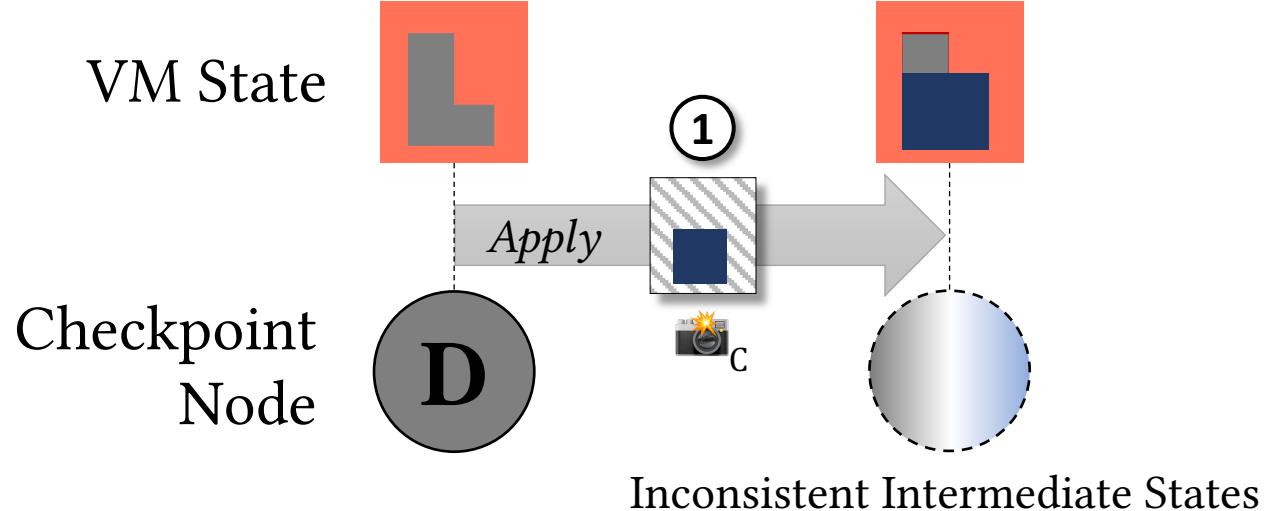


$$\text{Overhead} = \boxed{\text{Red}} + \boxed{\text{Blue}} + \boxed{\text{Dark Blue}} = O(\text{The number of all + dirty pages})$$

Lightweight Checkpoint Restoration

- Bottom-up, Delta Restore

- Restoring VM from a dirty VM state **D** To Node **C**

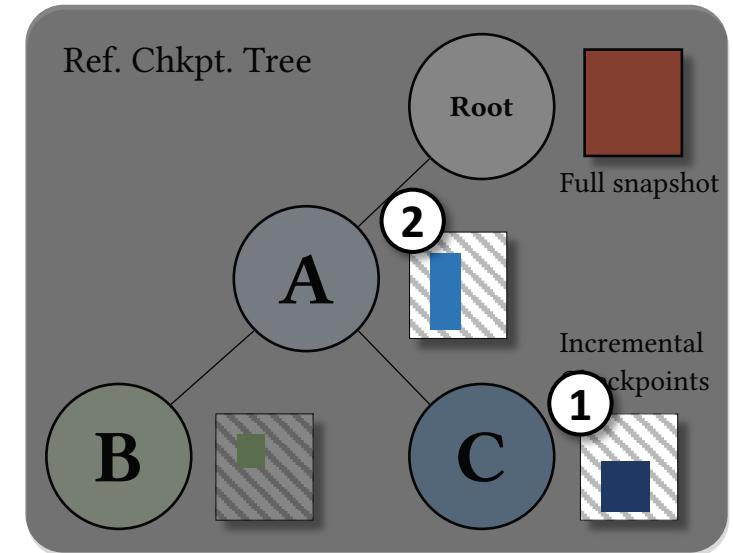
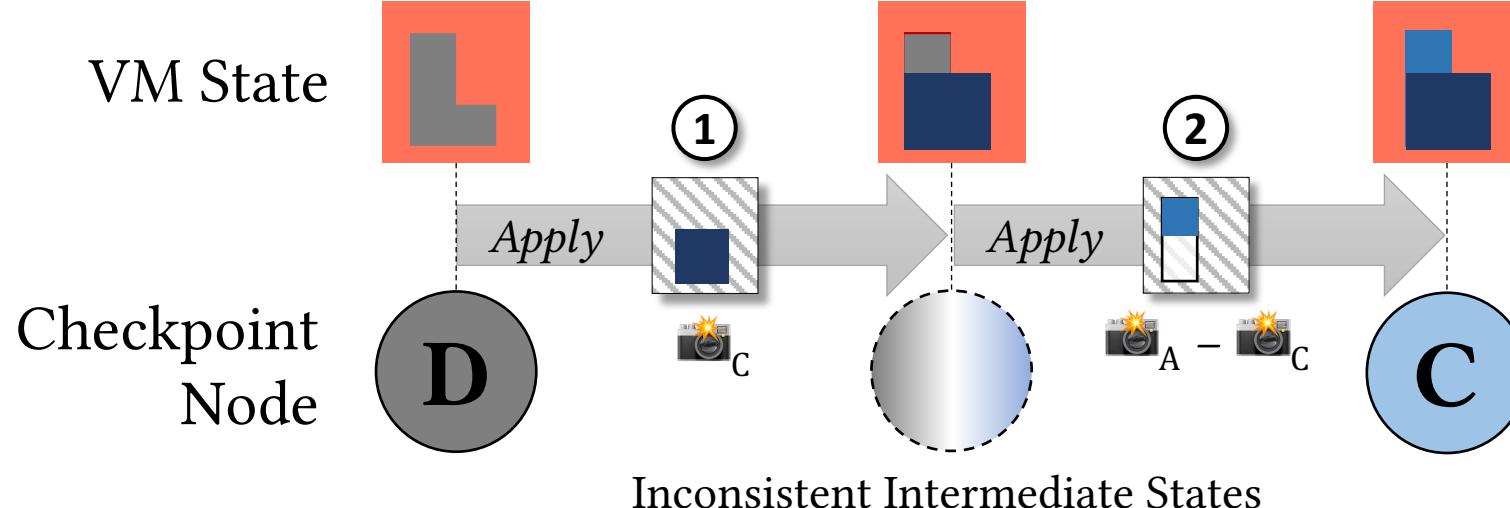


Overhead =

Lightweight Checkpoint Restoration

- Bottom-up, Delta Restore

- Restoring VM from a dirty VM state **D** To Node **C**



$$\text{Overhead} = \text{[diagonal hatched box]} + \text{[diagonal hatched box with white bar]} = O(\text{The number of dirty pages})$$

Lightweight Checkpoint Restoration

- Bottom-up, Delta Restore

- Restoring VM from a dirty VM state  To Node 



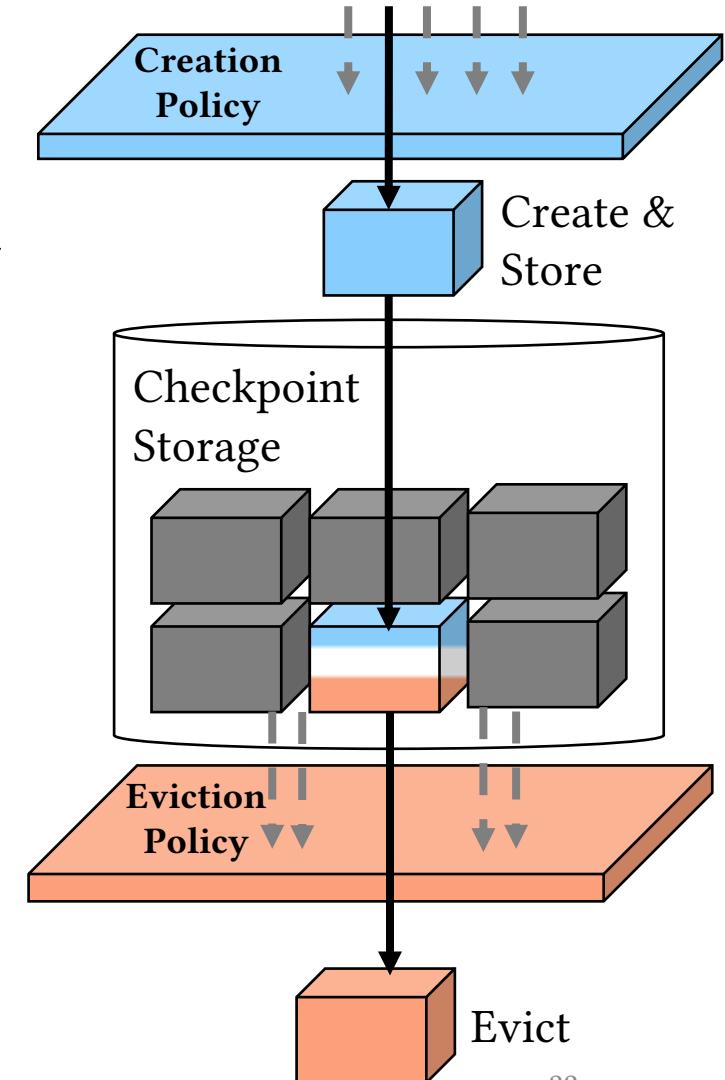
This approach achieves up to *8.9x faster* VM restoration
than the naïve top-down approach.



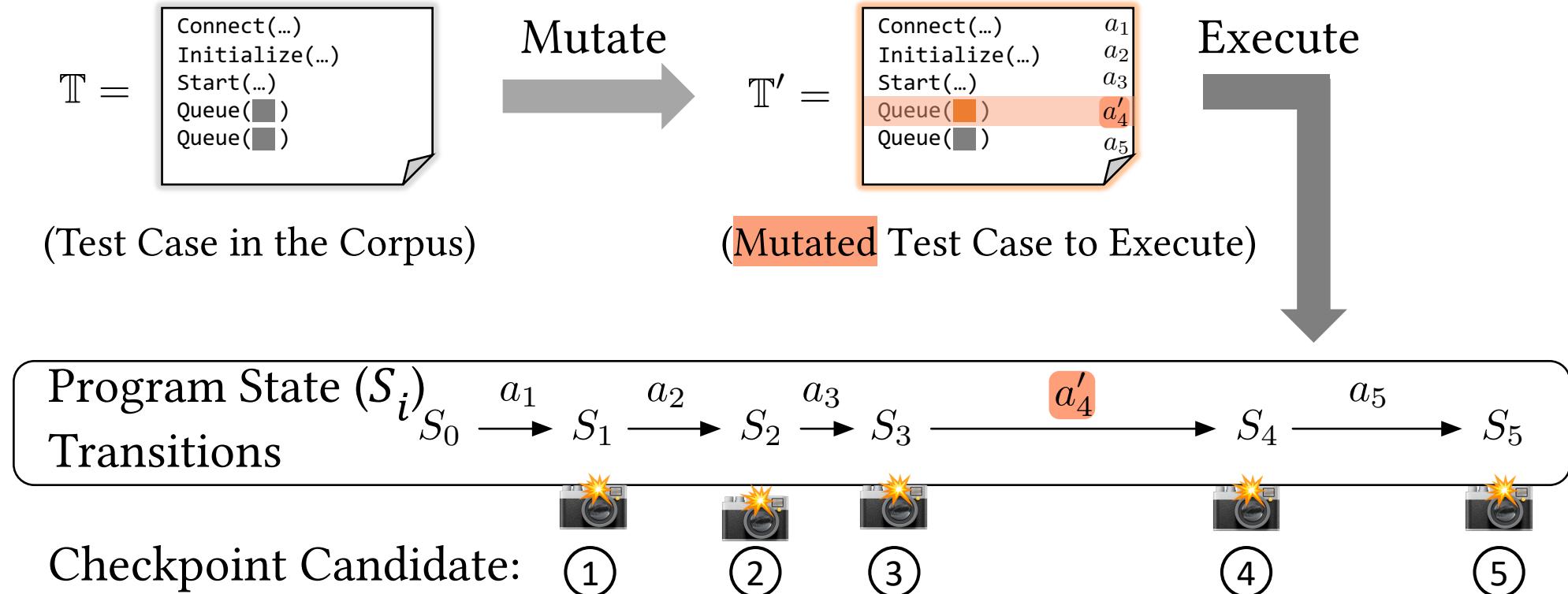
$$\text{Overhead} = \text{[Icon: gray square with diagonal lines]} + \text{[Icon: white square with blue border]} = O(\text{The number of dirty pages})$$

Checkpoint Management Policies

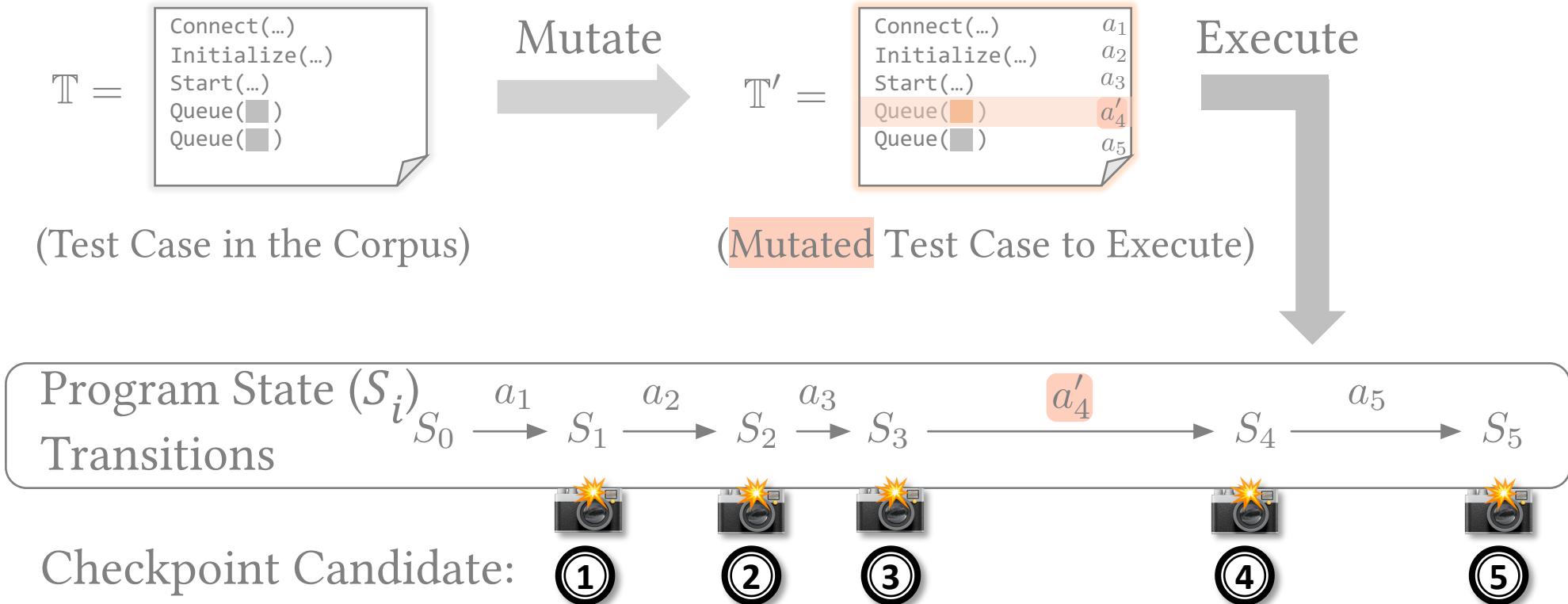
- **Goal:** Increase the Utility of Checkpoints
 - **Constraint #1:** checkpoint creation run-time overhead
 - **Constraint #2:** checkpoint memory overhead
- **High-level Ideas**
 - Control checkpoint creation via **Creation policy**
 - Evict checkpoints via **Eviction policy**



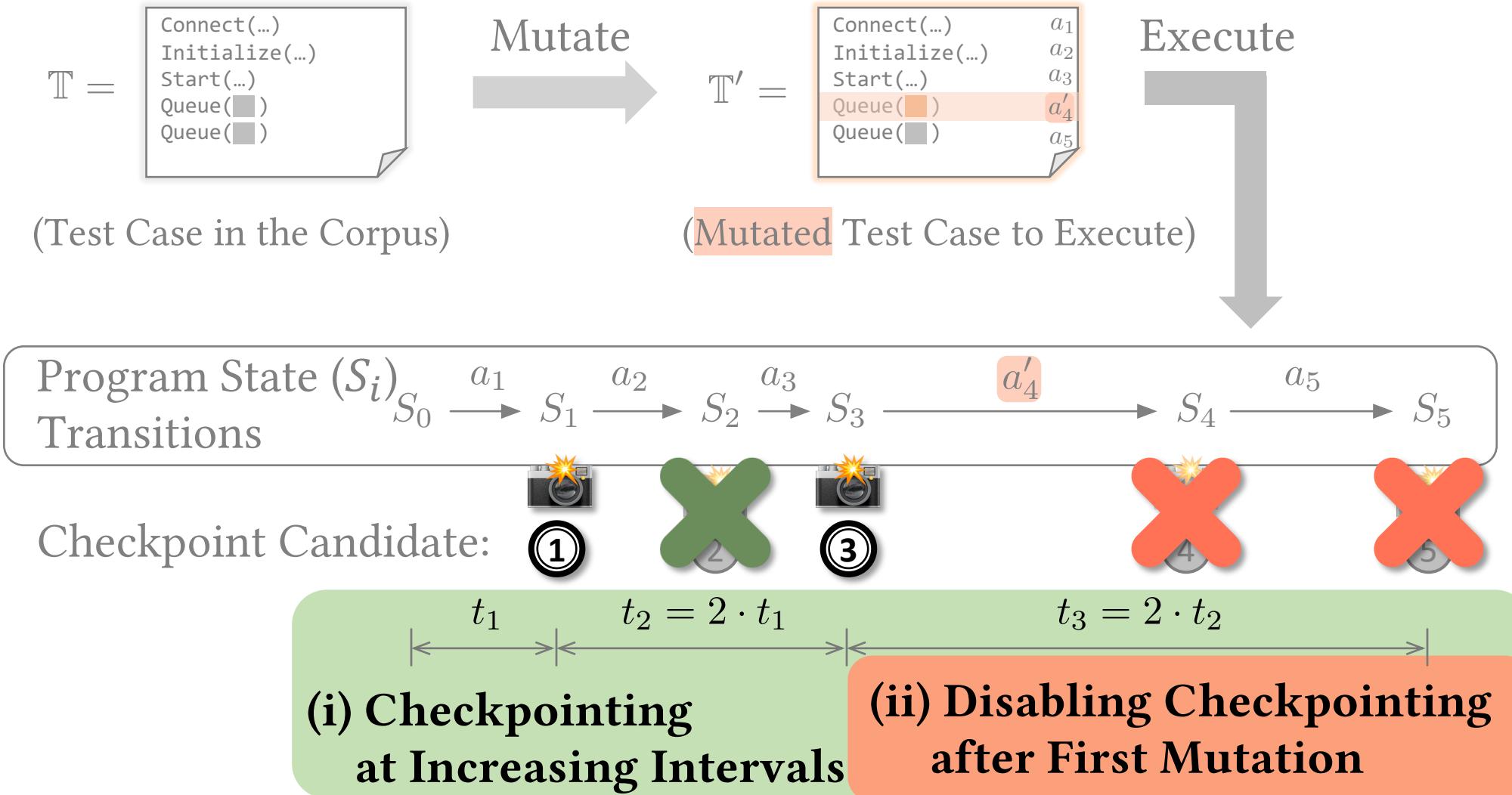
Checkpoint Creation Policy



cf. *Naïve* Checkpoint Creation Policy



Checkpoint Creation Policy





Implementation and Experiments

- Implementation of Agamotto
 - QEMU 4.0.0 with Linux KVM on x86-64
 - Syzkaller for USB fuzzing
 - Our own AFL-based PCI fuzzer for PCI fuzzing
- Experimental Parameters
 - 32 instance parallel fuzzing
 - 12GB checkpoint pool per fuzzing instance
- “Fuzzer–Attack Surface” Configurations
 1. Syzkaller–USB: Tested 8 Linux USB Kernel Drivers
 2. AFL–PCI: Tested 4 Linux PCI Kernel Drivers



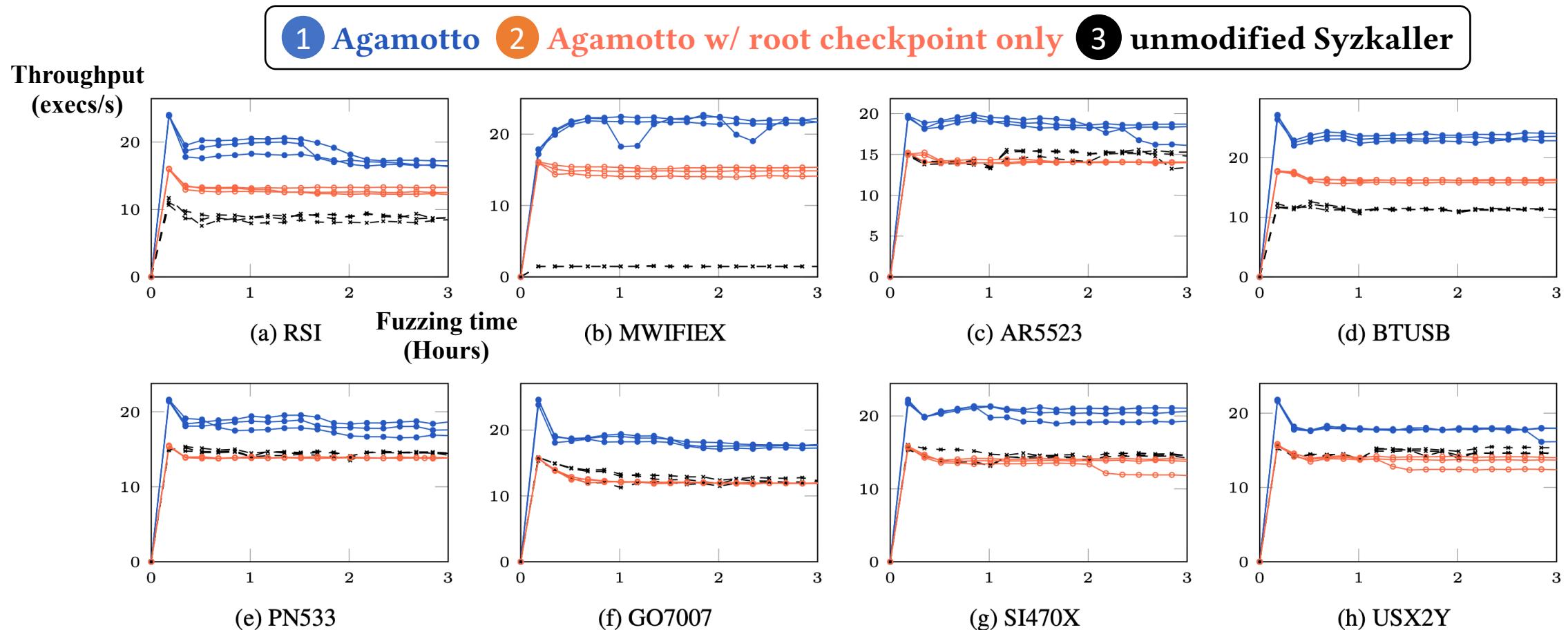
Implementation and Experiments

- Implementation of Agamotto
 - QEMU 4.0.0 with Linux KVM on x86-64
 - Syzkaller for USB fuzzing
 - ~~Our own AFL-based PCI fuzzer for PCI fuzzing~~

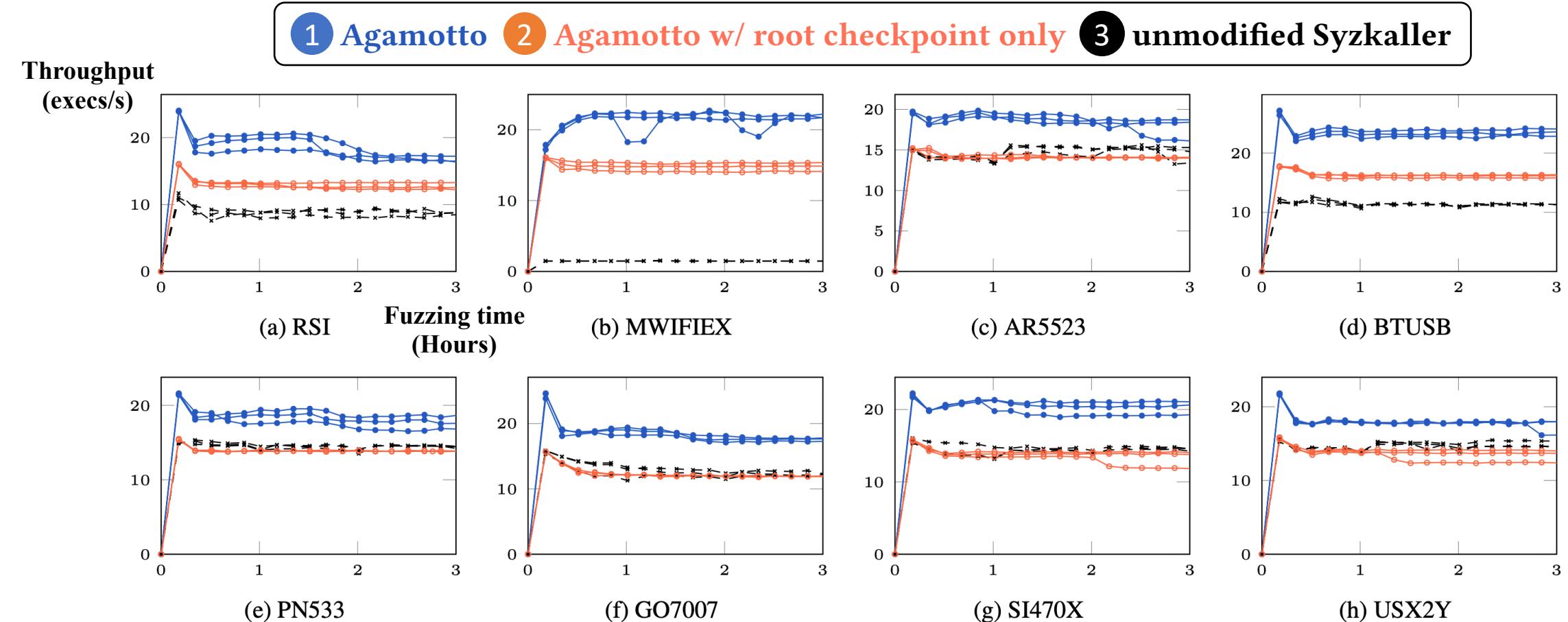
The fuzzing algorithms of Syzkaller/AFL
were NOT modified.

- 12GB checkpoint pool per fuzzing instance
- “Fuzzer–Attack Surface” Configurations
 1. Syzkaller–USB: Tested 8 Linux USB Kernel Drivers
 2. AFL–PCI: Tested 4 Linux PCI Kernel Drivers

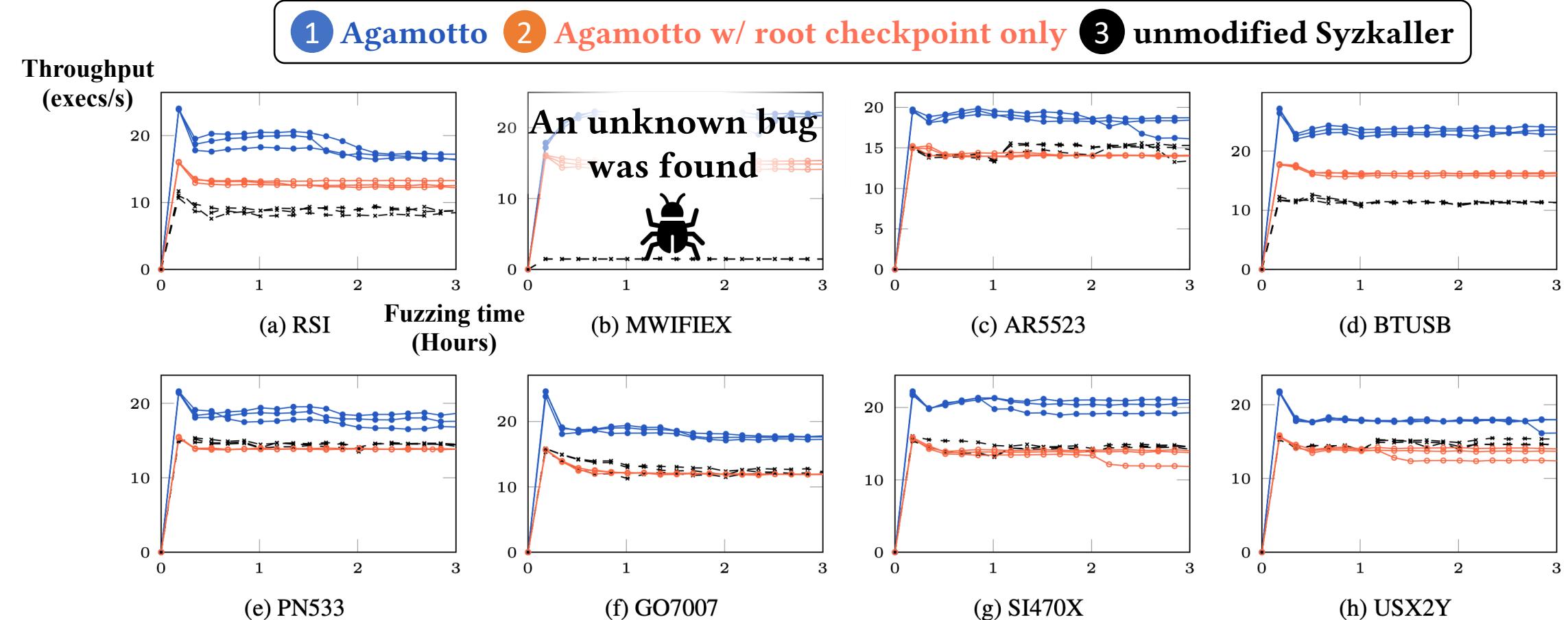
Syzkaller-USB Throughput



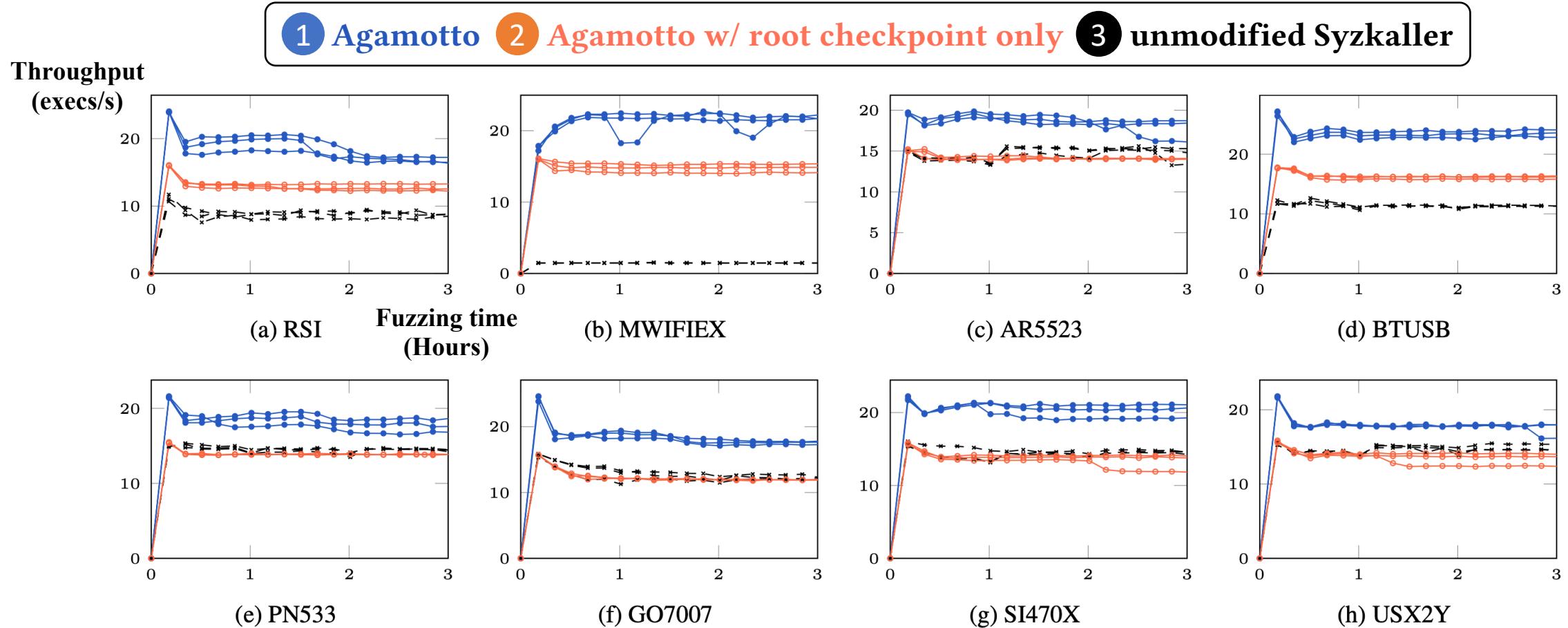
Snapshot v. No Snapshot: Comparing ①② and ③



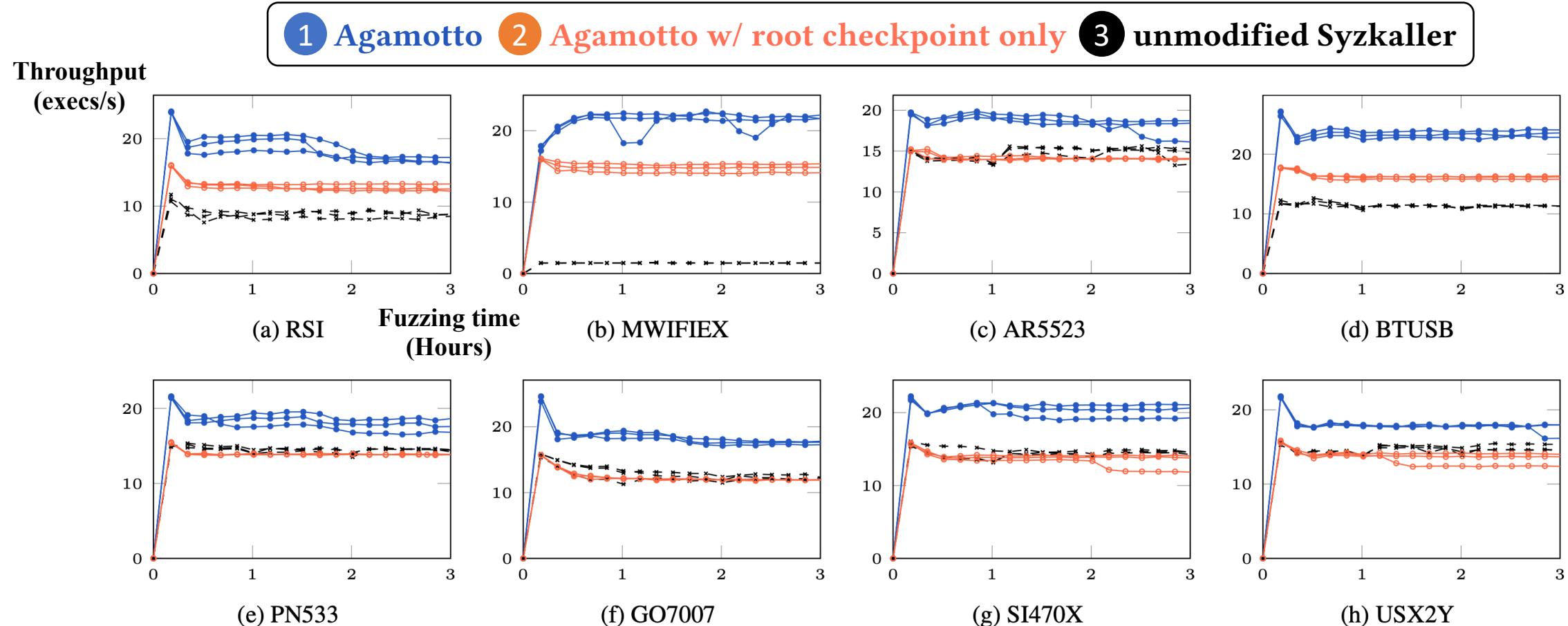
Snapshot v. No Snapshot: Comparing ①② and ③

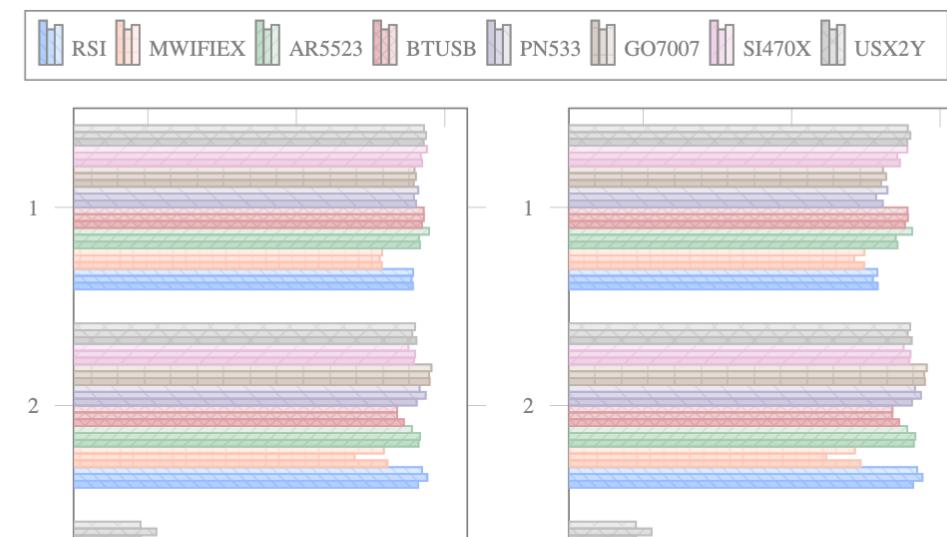
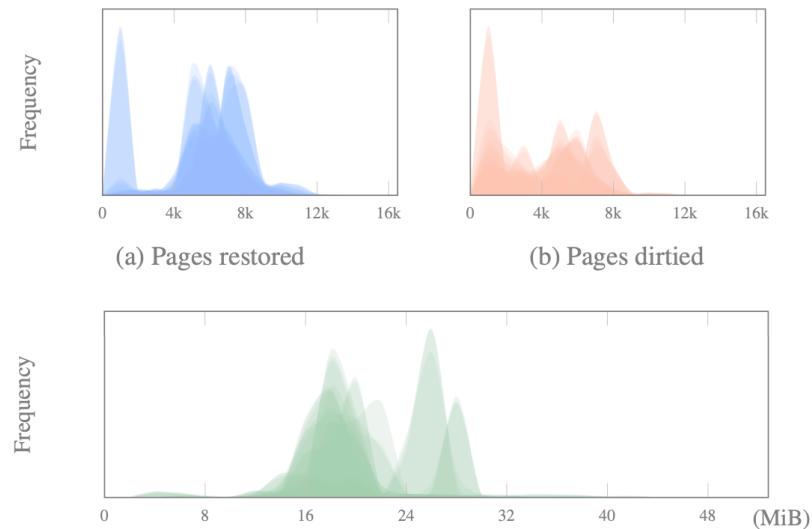


Dynamic Checkpointing v. Single Snapshot: Comparing ① and ②

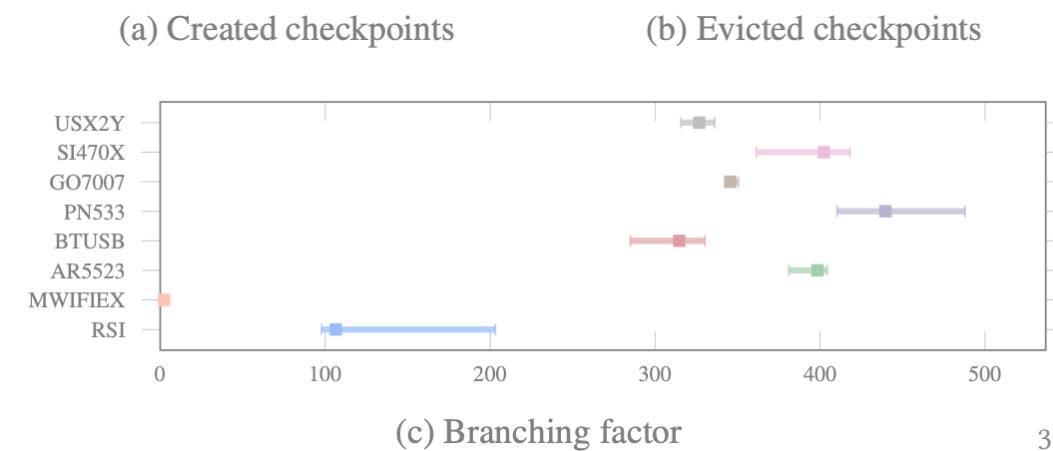
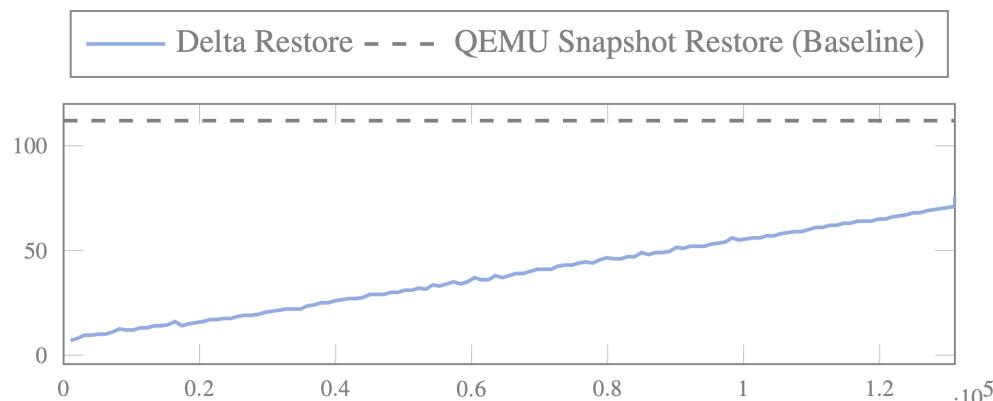


Dynamic Checkpointing v. No Snapshot: Comparing ① and ③





More evaluation results available in the paper.
(micro benchmarks, checkpoint statistics, etc.)



Conclusion and Future Work

- State-of-the-art fuzzing algorithms produce similar test cases in a short timeframe, which is another dimension to accelerate fuzzing.
- Lightweight VM checkpointing with dynamic checkpoint management policies can automatically accelerate kernel driver fuzzing.
- Changes to the fuzzing algorithm can be explored, e.g., optimizing it together with the checkpoint management policies.

Thank you!

Contact: Dokyung Song



dokyungs@uci.edu



<https://www.ics.uci.edu/~dokyungs>

Artifact: <https://github.com/securerootslab/agamotto>