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


### NVIDIA GPU DISPLAY DRIVER

<table>
<thead>
<tr>
<th>CVE-ID</th>
<th>Description</th>
<th>Base Score</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2020-5962</td>
<td>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.</td>
<td>7.8</td>
<td>AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H</td>
</tr>
<tr>
<td>CVE-2020-5963</td>
<td>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.</td>
<td>7.8</td>
<td>AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H</td>
</tr>
<tr>
<td>CVE-2020-5964</td>
<td>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.</td>
<td>6.5</td>
<td>AV:L/AC:L/PR:H/UI:R/S:U/C:H/I:H/A:H</td>
</tr>
<tr>
<td>CVE-2020-5965</td>
<td>NVIDIA GPU Display Driver contains a vulnerability in the DirectX 11 user mode driver (nvrgf2um/x.dll), in which a specially crafted shader can cause an out of bounds access, leading to denial of service.</td>
<td>5.5</td>
<td>AV:L/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:H</td>
</tr>
</tbody>
</table>
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

Discover event

Discovered

Initializing

Initialized

Starting

Start failed

Paused

Pausing

Queuing Event

Receive event

Event queue

Dequeued

Processing Event

Checking Queue

New thread started

switch(event.type) {
    case blue: ...
    case red: ...
    case green: ...
    case yellow: ...
}

Driver Execution Flow Example
Problem: Fuzzing Device Drivers is Slow

A fuzzer generated:
- Connect(…)
- Initialize(…)
- Start(…)
- Queue()
- Queue()

```
switch(event.type) {
  case DISCOVERED: …
  case INITIALIZED: …
  case STARTING: …
  case QUEUED: …
  case RECEIVED: …
}
```

Several Seconds

(CPU time given to drivers with a low priority, and frequently interrupted for higher priority tasks.)
Problem: Slowed down further by Crashes

10~30 sec × N (reboot for crash recovery)

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.

\[10\text{~to~}30\text{ sec} \times N\] (reboot for crash recovery)

\(N\) 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

Connect(...) → Root
Initialize(...) → A
Start(...) → B
Queue( )
Queue( )

Create Checkpoint (Root)
Discover event
Discovered
Initialize failed
Initializing
Start
Starting
Start failed
Initialzed
New thread started
Pausing
Discovered
Receive event
Queuing Event
Event queue
Checking Queue
Processing Event

Create Checkpoint (A)

Create Checkpoint (B)

switch(event.type) {
case: ...
case: ...
case: ...
case: ...
}
Our Approach: Dynamic VM Checkpointing

System Initialized → Root

A fuzzer generated

Initialize(…)
Start(…)

Connect(…)
Queue( )

Queue( )

Initializing
Starting
Pausing

Processing Event

switch(event.type) {
  case …
  case …
  case …
}

Dequeued
Checking Queue
Event queue

New thread started

Received event

Event queue

Queuing Event

Disconnected

Discovered

Initialize failed

Start failed

Initialize failed
Our Approach: Dynamic VM Checkpointing

System Initialized → Root

A fuzzer generated

Connect(…)
Initialize(…)
Start(…)
Queue(  )
Skipped

B

Our Approach:
Dynamic VM Checkpointing

1. Initializing
2. Starting
3. Pausing
4. Processing Event

Depending on next test input

Switch(event.type) {
  case   : …
  case   : …
  case   : …
  case   : …
}

Receive event

Dequeued

Checking Queue

Event queue

Queueing Event

Start-failed

Start

Initialized

Discover event

Discovered

Root

Paused

Skipped

Crash

Bug

Depending on next test input
Our Approach: Dynamic VM Checkpointing

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

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)

VM State: L

Checkpoint Node: D

Diagram:

Ref. Chkpt. Tree:
- Root
- Full snapshot
- Incremental Checkpoints

Nodes:
- A
- B
- 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 \( C \)

**VM State**

<table>
<thead>
<tr>
<th>VM State</th>
<th>Checkpoint Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>Root D</td>
</tr>
<tr>
<td>Root</td>
<td>Apply 1</td>
</tr>
<tr>
<td>Root</td>
<td>Apply 2</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

**Checkpoint Node**

- Root D
- A
- C

**Overhead** = \[ \text{Root } + \text{Checkpoint Node} \]
cf. *Naïve* Checkpoint Restoration
- Top-Down

- Restoring VM from a dirty VM state \( D \) To Node \( C \)

VM State

Checkpoint Node

\[
\text{Overhead} = \text{Apply} + \text{Ref. Chkpt. Tree} + \text{Update} = O\left(\text{The number of all + dirty pages}\right)
\]
**Lightweight Checkpoint Restoration**
- **Bottom-up, Delta Restore**

- Restoring VM from a dirty VM state [D] To Node [C]

![Diagram](image)

- Inconsistent Intermediate States

**Overhead =**
**Lightweight Checkpoint Restoration**
- **Bottom-up, Delta Restore**

- Restoring VM from a dirty VM state $\text{D}$ To Node $\text{C}$

**VM State**

- **Checkpoint Node**

**Inconsistent Intermediate States**

Overhead = $\text{Apply} + \text{Checkpoints} = O(\text{The number of dirty pages})$
**Lightweight Checkpoint Restoration**
- **Bottom-up, Delta Restore**

- Restoring VM from a dirty VM state \( \text{D} \) To Node \( \text{C} \)

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

\[
\text{Overhead} = \frac{\text{C}}{\text{Node}} + \frac{\text{A}}{\text{VM State}} = O\left( \text{The number of dirty pages} \right)
\]
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

\[ T = \begin{array}{l}
\text{Connect}(\cdots) \\
\text{Initialize}(\cdots) \\
\text{Start}(\cdots) \\
\text{Queue}(\square) \\
\text{Queue}(\square)
\end{array} \]

\( T' = \begin{array}{l}
\text{Connect}(\cdots) \quad a_1 \\
\text{Initialize}(\cdots) \quad a_2 \\
\text{Start}(\cdots) \quad a_3 \\
\text{Queue}(\square) \quad a'_4 \\
\text{Queue}(\square) \quad a_5
\end{array} \)

Program State (\( S_i \))

Transitions

\[ S_0 \overset{a_1}{\rightarrow} S_1 \overset{a_2}{\rightarrow} S_2 \overset{a_3}{\rightarrow} S_3 \overset{a'_4}{\rightarrow} S_4 \overset{a_5}{\rightarrow} S_5 \]

Checkpoint Candidate: \( \text{\color{red}{1}} \), \( \text{\color{red}{2}} \), \( \text{\color{red}{3}} \), \( \text{\color{red}{4}} \), \( \text{\color{red}{5}} \)
cf. *Naïve* Checkpoint Creation Policy

(Test Case in the Corpus) $T =$

[Diagram showing transitions and checkpoint candidates]

(Mutated Test Case to Execute) $T' =$

Program State ($S_i$) Transitions

$S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} S_2 \xrightarrow{a_3} S_3 \xrightarrow{a'_4} S_4 \xrightarrow{a_5} S_5$

Checkpoint Candidate: ① ② ③ ④ ⑤

Naïvely Checkpointing after Every Action in the Test Case
Checkpoint Creation Policy

$$T = \begin{array}{c}
\text{Connect(…)} \\
\text{Initialize(…)} \\
\text{Start(…)} \\
\text{Queue(□)} \\
\text{Queue(□)}
\end{array}$$

(Test Case in the Corpus)

Mutate

$$T' = \begin{array}{c}
\text{Connect(…)} \\
\text{Initialize(…)} \\
\text{Start(…)} \\
\text{Queue(□)} \\
\text{Queue(□)}
\end{array}$$

(Mutated Test Case to Execute)

Execute

Program State ($S_i$)

\[
\begin{align*}
S_0 & \xrightarrow{a_1} S_1 \\
S_1 & \xrightarrow{a_2} S_2 \\
S_2 & \xrightarrow{a_3} S_3 \\
S_3 & \xrightarrow{a_4} S_4 \\
S_4 & \xrightarrow{a_5} S_5
\end{align*}
\]

Checkpoint Candidate:

- ①
- ②
- ③
- ④
- ⑤

(i) Checkpointing at Increasing Intervals

(ii) Disabling Checkpointing after First Mutation
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

<table>
<thead>
<tr>
<th>Throughput (execs/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) RSI</td>
</tr>
<tr>
<td>(b) MWIFIEX</td>
</tr>
<tr>
<td>(c) AR5523</td>
</tr>
<tr>
<td>(d) BTUSB</td>
</tr>
<tr>
<td>(e) PN533</td>
</tr>
<tr>
<td>(f) GO7007</td>
</tr>
<tr>
<td>(g) SI470X</td>
</tr>
<tr>
<td>(h) USX2Y</td>
</tr>
</tbody>
</table>

- **Agamotto**
- **Agamotto w/ root checkpoint only**
- **unmodified Syzkaller**
Snapshot v. No Snapshot: Comparing ① ② and ③

<table>
<thead>
<tr>
<th>Throughput (execs/s)</th>
<th>Fuzzing time (Hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>① Agamotto</td>
<td></td>
</tr>
<tr>
<td>② Agamotto w/ root checkpoint only</td>
<td></td>
</tr>
<tr>
<td>③ unmodified Syzkaller</td>
<td></td>
</tr>
</tbody>
</table>

(a) RSI  
(b) MWIFIEX  
(c) AR5523  
(d) BTUSB  
(e) PN533  
(f) GO7007  
(g) SI470X  
(h) USX2Y
Snapshot v. No Snapshot: Comparing ① ② and ③

Throughput (execs/s)

An unknown bug was found

(a) RSI  (b) MWIFIEX  (c) AR5523  (d) BTUSB

(e) PN533  (f) GO7007  (g) SI470X  (h) USX2Y
Dynamic Checkpointing v. Single Snapshot: Comparing ① and ②

Throughput (execs/s)

Agamotto ① Agamotto w/ root checkpoint only ③ unmodified Syzkaller

(a) RSI  Fuzzing time (Hours)  (b) MWIFIEX  (c) AR5523  (d) BTUSB

(e) PN533  (f) GO7007  (g) SI470X  (h) USX2Y
Dynamic Checkpointing v. No Snapshot: Comparing 1 and 3

1 Agamotto 2 Agamotto w/ root checkpoint only 3 unmodified Syzkaller

Throughput (execs/s)

(a) RSI  (b) MWIFIEX  (c) AR5523  (d) BTUSB

(e) PN533  (f) GO7007  (g) SI470X  (h) USX2Y
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/securesystemslab/agamotto