FUZZING INTEL GRAPHICS DRIVERS USING SYZKALLER AND KAFL: A COMPARATIVE ANALYSIS

KAINAAT SINGH

Master Thesis

August 2021

Rheinische Friedrich-Wilhelms-Universität Bonn
In collaboration with and sponsored by Intel Corporation

Fuzzing Intel Graphics Drivers
using Syzkaller and kAFL:
A Comparative Analysis
Master Thesis

Submitted by Kainaat Singh
Date of submission: 9. August. 2021

First examiner/Erstgutachter: Prof. Dr. Matthew Smith
Second examiner/Zweitgutachter: Prof. Dr. Michael Meier
Supervisor/Betreuer: Steffen Schulz (Intel Labs), Mischa Meier

Rheinische Friedrich-Wilhelms-Universität Bonn
Institute of Computer Science 4
Methods in Multi-Layer Usable Security Research

In collaboration with and sponsored by Intel Corporation

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available ?updates. See backup for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

©Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

ABSTRACT

With the advent of machine learning and cloud, many applications working with sensitive data are moving their operations to the cloud based Graphics Processing Unit (GPU) to leverage high parallel computing power. The GPU device driver handles and submits the sensitive data to the GPU. It executes in supervisor mode and hence, must be fully trusted. GPU vendors need to ensure proper validation of their GPU device drivers before release. In recent years, coverage-guided fuzzing has become a popular method for automating code validation at a large-scale. Coverage achieved is one of the important indicators of evaluating a coverage-guided fuzzer. While fuzzing a device driver, the interference of the non-deterministic kernel events and the underlying hardware's state conditions in the feedback received can impact the coverage achieved by a fuzzer.

In this thesis, we aim to find a suitable large-scale fuzzing solution that overcomes the challenges of fuzzing i915 graphics driver and can be deployed for large-scale driver validation. The i915 driver communicates with the latest Intel integrated GFX chipsets. Out of the possible fuzzing options, we explore the different Intel device virtualization configurations to fuzz the driver. To choose a suitable fuzzer for our solution, we perform a comparative analysis between the state-of-the-art coverage-guided kernel fuzzer, Syzkaller, and an alternative research proof-of-concept, Kernel AFL (kAFL), by performing fuzzing of the i915 GFX driver with an actual hardware backend.

We first provide a qualitative analysis of the different fuzzing solutions based on defined criteria. For this analysis we create a clever kAFL harness similar to Syzkaller logic. We find that this extra harness intelligence can produce comparable results to Syzkaller with the i915 graphics driver. We find similar numbers of edges covered by both the fuzzers but the coverages are not completely comparable. We present a case study that confirms that the coverages can be considered similar. Hence, we find that kAFL can be a good alternative to Syzkaller for graphics driver fuzzing if the user can provide a clever harness.

CONTENTS

ACRONYMS

OS Operating System

GPUs Graphics Processing Units

kAFL Kernel AFL

GPGPU Graphics Processing Units

AFL American Fuzzy Lop

CPU Central Processing Unit

API Application Programmer Interface

POSIX Portable Operating System Interface

GUI Graphical User Interface

OpenGL Open Graphics Library

DRI Direct Rendering Infrastructure

DDX Device Dependent X

DRM Direct Rendering Infrastructure

iGPU Integrated Graphics Processing Unit

dGPU Discrete Graphics Processing Unit

SVM Shared Virtual Memory

USB Universal Serial Bus

MMIO Memory-mapped I/O

DMA Direct memory access

QEMU Quick Emulator

vGPU Virtual Graphics Processing Units

KVM Kernel-based Virtual Machine

VT-d Virtualization Technology for Directed I/O

I/O Input/Output

SR-IOV Single Root IO Virtualization

VFs Virtual Functions

NICs Network Cards

SSH Secure Shell

RPC Remote Procedure Call

IPC Inter-Process Communication

Progs Programs

TCP Transmission Control Protocol

BLOBs Binary large objects

QEMU-PT QEMU Processor Trace

KVM-PT KVM Processor Trace

PT Processor Trace

KASLR Kernel Address Space Layout Randomization

gce Google Compute Engine

IOMMU I/O Memory Management Unit

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 MOTIVATION

Several classes of vulnerabilities like use-after-free vulnerabilities, buffer overflow and memory corruption are known as prevalent threats for applications in the user-space and the kernel-space [10]. Kernel-space vulnerabilities can lead to privilege escalation or kernel rootkits to gain persistence, and therefore, it is important to keep the kernel-code secure. Many Operating System (OS) vendors have deployed security mechanisms to prevent attacks at user-space and kernel-space. Even then the vulnerabilities in the kernel-space can be difficult to spot.

Device Drivers make up more than 60% of the kernel source code [52]. As most of the drivers are executing in the supervisor mode, they need to be trustworthy. But due to the monolithic architecture of Linux, the user-space applications are forced to trust all the drivers. Therefore, any severe vulnerability in their code can lead to the compromise of the entire system, and hence, the developers need to minimize the vulnerabilities. Google estimates that 85% of the kernel bugs are found in the vendor drivers [47].

Graphics Processing Units (GPUs) have been increasingly gaining popularity in the field of General-Purpose computing on Graphics Processing Units (GPGPU) programming. In GPGPU, the highly parallel nature of the GPUs is leveraged. This massive parallelism is achieved by exploiting thousands of cores, and GPGPU uses this in applications like financial, encryption, big data, and bitcoin mining [33]. GPUs are also made available by cloud computing service providers in a virtualized environment for customers who do not want to buy expensive hardware. This further lead to GPUs being used for cloud gaming, where users could play GPU-intensive games in a virtualized environment. Even though applications are running sensitive data through the GPU, not much effort has been made towards securing the graphics subsystem. Hence, there is a need to have proper validation of the components that handle the sensitive data to and from the GPU.

One interesting problem is validating the OS device drivers as most of them are running with supervisor privileges and are interacting with several untrusted user-space applications. A popular method of for software validation is an automated testing process called fuzzing. Fuzzing the kernel components has often been difficult as the non-determinism of the kernel code and close interactions with stateful hardware poses a problem in getting useful coverage information necessary for the established coverage-guided fuzzers.

1.2 RESEARCH CONTRIBUTION

In this thesis we investigate the challenges to validate and test Intel's i915 graphics driver using coverage-guided fuzzing. To achieve this we aim to overcome the following challenges:

1. Most of the kernel fuzzers available use a virtualized environment for fuzzing the target. The physical GPU must first be attached to the virtualized guest to load the driver.

2. The stateful GPU and kernel add non-determinism to the execution of fuzz inputs against the i915 driver. This can lead to system crashes and ineffective coverage.

In this thesis we aim to find a suitable fuzzing solution that overcomes the above challenges. We first explore the different possible options for the fuzzers to access the Intel GPU such that the i915 driver can be loaded for testing. Next, we investigate the different fuzzing options with two prominent coverage-guided kernel fuzzers, Syzkaller and Kernel AFL (kAFL). Syzkaller is a state-of-the-art coverage-guided kernel fuzzer that is used by the industry for fuzzing the Linux Kernel [15]. kAFL is an alternative research proof-of-concept based on the novel fuzzer American Fuzzy Lop (AFL) [43]. We aim to examine whether Syzkaller and kAFL can deal with the above challenges and effectively fuzz the i915 driver. To summarize, this thesis contributes the following research:

I. ***What are the different options for fuzzing the i915 GFX driver for large-scale driver validation?***

   In this thesis, we aim to explore the different options and define the main research questions that we wish to evaluate for these options in Chapter 3. We aim to answer these questions in Chapter 4 and Chapter 5.

II. ***Present a quantitative analysis of Syzkaller and kAFL with the Intel i915 GFX driver as the target.***

   In this thesis, we introduce the fuzzers and define the main research questions that we want to evaluate for the two fuzzers in Chapter 3. We aim to answer these questions in Chapter 4 and Chapter 5.

1.3 STRUCTURE OF THESIS

The following chapters have been structured as follows:

- Chapter 2 provides a brief background knowledge that is required to understand the motivation behind the thesis and comprehend the upcoming chapters.

- Chapter 3 gives an overview of the challenges in fuzzing the i915 driver, the different fuzzing options for the i915 driver, the technologies and concepts used during this thesis. Defines the research questions for this thesis.
- Chapter 4 describes the evaluation platform and the integration of the fuzzing options with Syzkaller and kAFL. We explain the setup that is used as a benchmark for the evaluation of the fuzzers.
- Chapter 5 presents the experiments conducted and the comparative analysis between the two fuzzers. Answers the research questions that were defined in Chapter 3.
- Chapter 6 presents the conclusion and future work.

2

BACKGROUND

2.1 THE LINUX KERNEL

The Linux Kernel is an important part of many open and closed source projects, including distributions of GNU/Linux operating system. It is for the most part monolithic in nature, running in the kernel-space. That implies that all functionalities like device drivers, dispatcher, scheduling, virtual memory, all inter-process communication,etc run in the kernel-space, as shown in Figure 2.1.

When a user-space program wishes to use any underlying hardware resource, it needs to issue such a request to the operating system. The Linux kernel assesses this request and communicates with the required hardware on behalf of the user-space program. This enforces the hardware protection of system resources [11]. To implement this, modern Central Processing Unit (CPU) architectures provide two modes to the CPU to operate in: a non-privileged *user mode* and a privileged *kernel mode* (also referred to as supervisor mode) and corresponding to these modes, the virtual memory is marked as *user-space* or *kernel-space*.

The OS wishes to ensure that no program is able to exploit another and this is achieved by providing each program with it's own user-space, while the kernel-space is shared across all the programs. In general, the program running in the user space has no rights to read, write, modify or execute the data in the memory marked as the kernel-space. On the other hand, kernel running in the kernel mode from kernel-space has the privileges to perform all those actions on the data that resides in user-space and kernel-space memory. Therefore, we see that these two are separate abstraction layers to provide code isolation and security.

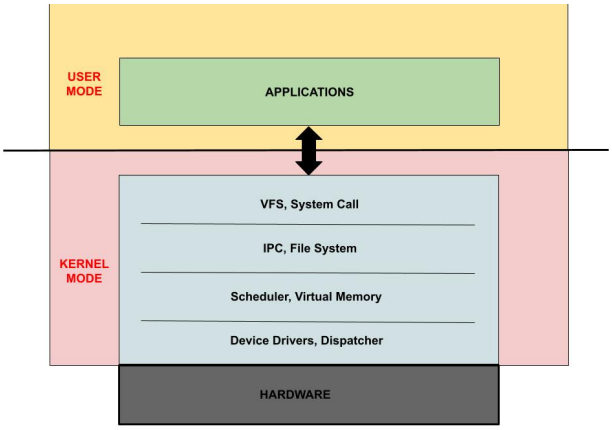

Figure 2.1: Linux Kernel Architecture

2.1.1 *System Call Interface*

OS provides an extra layer of kernel between the user programs and the hardware for the following reasons:

1. As stated in 2.1, the system security is increased due to the kernel assessment of request before execution
2. User programs are made portable to run on every kernel that provides the same interface

In Unix-like systems, this interface is known as *system calls* that help the user-programs communicate the requests for a privileged operation to the kernel. The system calls should not be confused with the Application Programmer Interface (API). An API is a function definition that describes to another program how to obtain a service that your program offers whereas a system call is an explicit request for privileged operation using software interrupts. Every system call has a wrapper routine called *system call handler* that specifies the API for the user-space programs, as shown in Figure 2.2. On the other hand, every API does not correspond to a system call. A detailed difference can be found in [11].

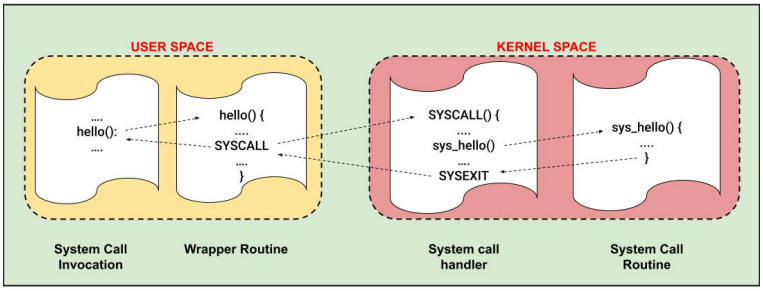

Figure 2.2: Invoking System Calls

There are many possibilities where a user-mode program wants to access the underlying system resources, a privileged operation. For this, the program "signals" the kernel using system call interface that it wants to execute a privileged operation. Once this request is received by the kernel it performs check for sanity, input validation and security [4]. On passing, the CPU privilege mode is changed from user mode to kernel mode and the program starts executing its privileged kernel procedure. When the requested procedure is finished executing, it forces the CPU to transition to the user mode.

"Syscall" instruction is used in x86_64 system to make a system call. The user program passes the parameters for the system call by: 1) pushing and popping off the program stack 2) using the registers in the order RDI, RSI, RDX, R10, R8, and R9 [4] 3) through a data block and the block address is passed to registers as the parameter. The last method is used when there are more parameters than registers. Every system call is identified using an integer which is stored in RAX. Figure 2.3 shows the five types of system calls that are presented by the interface.

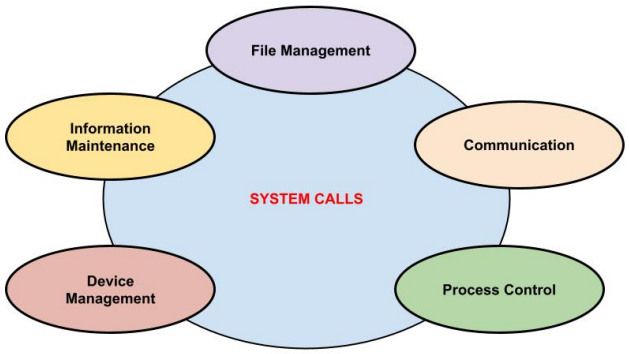

Figure 2.3: Types of System Calls

2.1.2 *Security*

There are several types of vulnerabilities like use-after-free, memory corruption and race conditions that are threats for programs that run in both user-space and kernel-space [41]. Kernel vulnerabilities can be critical because due to the monolithic architecture of the kernel, number of critical user-space applications depend on the kernel, like guest machines or task manager. Therefore, we need kernel security to ensure that the kernel is trustworthy.

Security is a policy that should be handled by only the privileged levels, therefore, the security checks in the OS are enforced by the kernel. The system becomes vulnerable if there are security vulnerabilities in the kernel. In a privilege escalation attack, the attacker can exploit the kernel vulnerabilities to upgrade the privilege level of a process [53]. If the attacker is successful with such attack, he can gain administrative privileges and compromise the entire system. With such privileges, the attacker can steer clear of access control and gain read/write permissions for the entire system data. Hence, the privilege escalation attack constitutes a significant threat to the system security and should be prevented at all costs.

2.1.3 *Device Drivers*

Device Drivers are like "black-boxes" as they are an abstraction to the workings of a hardware by presenting a well-defined internal programming interface for the user-space applications to communicate with the device. The Portable Operating System Interface (POSIX) Standard specifies the communication between the user-space applications and the device drivers. User-space applications use standardized system calls that are independent of a specific driver for hardware operations. The device driver then maps these calls to device-specific operations that are run on actual hardware. These device drivers can be built independent of the kernel and can be "loaded" at runtime when needed. These are known as *loadable kernel modules*.

The device itself is presented to the user-space as a special *device file* on the disk. This echoes the motto of the Linux kernel that "'Everything is a file"'. The user-space application can obtain a handle to the file using the `open()` system call and then interact with the device file in multitude of ways. But the traditional system calls cannot fulfill all the requirements of the different device drivers present out there. For example, how can the user-space application configure memory used by a GPU. The POSIX Standard describes an *`ioctl`* interface that can help device drivers model the functionalities that are typically not possible with the traditional system calls. The C definition of `ioctl` is as follows: `int ioctl(int fd, unsigned long request, ...);`. It takes in an open file descriptor as the first argument, a device-dependent *command identifier* as the second argument and the third is an untyped pointer to the memory. The type and quantity of the third argument is dependent on the driver used and the command identifier.

Privileged code can raise the user-mode privileges to kernel-mode privileges and majority of such privileged code paths exist in the device drivers. The current Linux kernel source tree has over 17 million line of code and device drivers make up the biggest part of the code [35].

As the device drivers are accessible via the `ioctl` interface, the above definition of the interface presents possible vulnerabilities. As the third argument is dependent on the type of command identifier, this means that the user must be aware of how the functionalities implemented in the driver and what type of data do they take as input. Parsing any incorrect data can expose critical vulnerabilities in the kernel-space.

2.2 LINUX GRAPHICS STACK

This section provides a brief introduction to the Linux Graphics Stack to provide a clear vision as to where the i915 driver lies, which components it interacts with and why it might be a lucrative target for an attacker.

To understand the modern graphics stack, it is necessary to have an overview of how it adapted to new technologies over the years.

Modern graphics stack has evolved over the years from the time when *X server* was the only program that could directly communicate to the underlying graphics hardware and perform rendering on the framebuffer. The *framebuffer* is a bitmap in the system memory that holds the pixel data representing a complete display frame. User-space applications could interact with the graphics hardware through the X server, shown in Figure 2.4. X server exposed a library, *Xlib*, that was used by the applications to send rendering commands and after receiving them, the X server would convert them to hardware-specific commands. Hence, the commands sent were hardware-independent and the conversion was done by drivers that were specifically written as modules for the X server. These user-space drivers provided 2D graphics support in the architecture and were called *Device Dependent X (DDX)*.

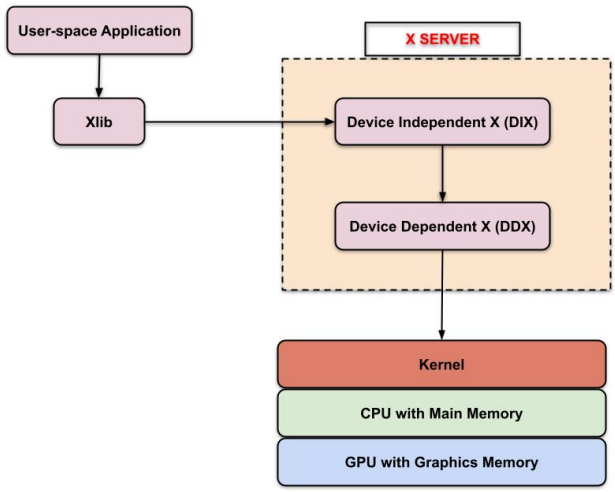

Figure 2.4: X Architecture [50]

With the arrival of the *X windowing system*, 3D graphics were introduced and changed the realm of graphics significantly. X Window system is a type of Graphical User Interface (GUI), where each window needs to handle user interactions, inter-window events and other user-space system-dependent events. The 3D-accelerated graphics hardware came with their own memory inside which a command queue was used to manage the commands directed towards the GPU [49]. The buffers and free space within the memory was required to be managed as well.

Open Graphics Library (OpenGL) is used to implement 3D graphics and it exposes an API via a library called *libGL*. To take advantage of the sophisticated 3D hardware, libGL needed to be hardware-accelerated. As only X server could talk to the graphics hardware directly, OpenGL was implemented such that all the commands would pass via X server, which would then convert them into GPU-specific commands [50]. This is known as *Indirect Rendering*. A Framebuffer application was used to manage the graphics memory. Initially, 3D hardware

had a clear separation from 2D hardware, so having separate drivers made sense. *Utah-GLX* was the first 3D hardware-independent, user-space driver, shown in Figure 2.5. This model had a few drawbacks. First, due to the potential conflicts between all the drivers during context switches, drivers were supposed to take a snapshot of their state before the switch. This was a difficult task for the developers. Second, the unprivileged user-space applications could access the graphics hardware. This was a growing concern for the Linux Security community. It was concluded that this was not an efficient solution for the intensive 3D applications. There was a need to access the graphics hardware directly and securely to get better performance. As there was no way for multiple applications to access the hardware directly without collisions, it lead to a big architectural change in the graphics stack.

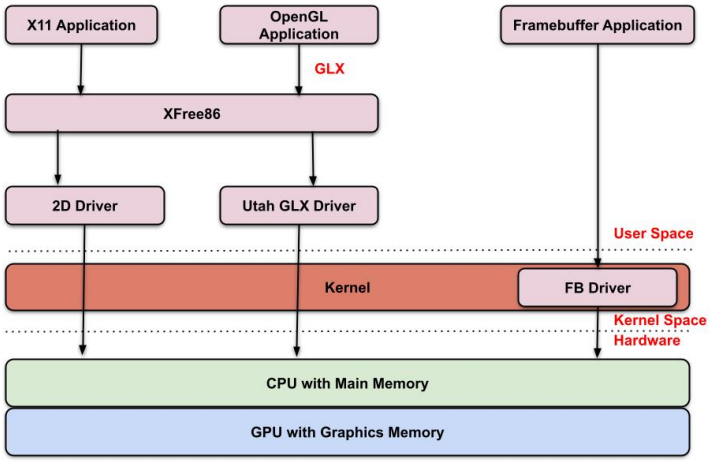

Figure 2.5: Earlier Graphis Stack using Utah-GLX [50]

Direct Rendering Infrastructure (DRI) is a framework that was initially developed to allow the user-space applications direct access to the graphics hardware alongside the X server for efficient 3D rendering. To implement DRI, changes were made to X server, kernel and various client libraries [50].

DRI framework comprises of 3 components and the term DRI refers to the entire framework, it is frequently used for the component that interacts with the user-space applications. The 3 components have distinct characteristics and are as follows:

1. **DRI client:** This component handles the user-space application commands for direct rendering. The application requires a hardware-specific driver to convert the commands to device-specific commands. DRI provides drivers in the form of shared libraries that the application is linked to dynamically[24]. Under this the libGL library of OpenGL is hardware accelerated to get the maximum advantage of the 3D graphics hardware. This library can be a

product form a third party company (*Mesa 3D*) or the hardware vendor itself. In this component, no root privileges are involved and is part of the user-space side of the architecture. This helped remove some of the security concerns.

2. **X server:** The X server exposed the X11 protocol for user-space applications to access the 2D driver DDX for the windowing system. It exposes another protocol called GLX that can be used by the applications for indirect rendering to achieve 3D hardware acceleration. This extension can be helpful for remote applications.

3. **Direct Rendering Infrastructure (DRM):** The aim of this component is to allow direct access of 3D hardware for multiple applications in a synchronized manner. DRM provides the user-space with system calls to send commands which it converts to the hardware-specific commands to carry out the 3D rendering action. This conversion is GPU-specific and as there are many GPU vendors, DRM can contain different drivers. For Intel Integrated Graphics Processing Unit (iGPU), Intel provides i915 GFX driver. Other tasks involve ensuring that 3D resources like framebuffer, memory or command queue do not collide with each other for access. DRM also enforces security policies that ensure that the user-space applications do not access hardware beyond the 3D rendering action [24]. To carry out these tasks successfully, DRM runs with root privileges in the kernel-space and forms the kernel side of the architecture.

   GPU is exposed to the user-space as a device file under /dev filesystem. Multiple GPUs can be detected and can be found as /dev/dri/cardN (N is a sequence number) or /dev/dri/renderD128 which are used by applications to communicate with the GPU. The *DRM API* is presented to the user-space with a library called *libdrm* that provides a wrapper to the DRM API. This prevents the kernel-space API to be directly exposed to the user-space[26]. To interact with the GPU, the applications need to open the device file and use different ioctls to communicate with DRM.

   In the old graphics stack, shown in Figure 2.6 the 2D command stream still was passing through the 2D driver separately which meant that the X server required higher privileges to directly communicate with the graphics hardware. This has lead to the current stack where the X server does not need any root privileges and different drivers do not need to communicate with the single piece of hardware.

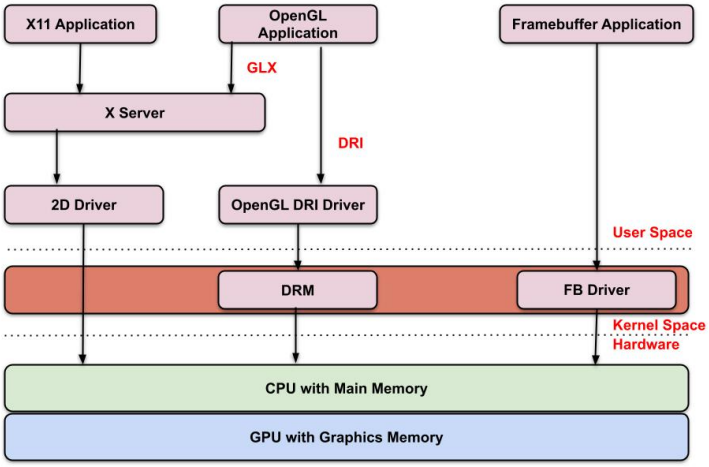

Figure 2.6: Old Graphics Stack [50]

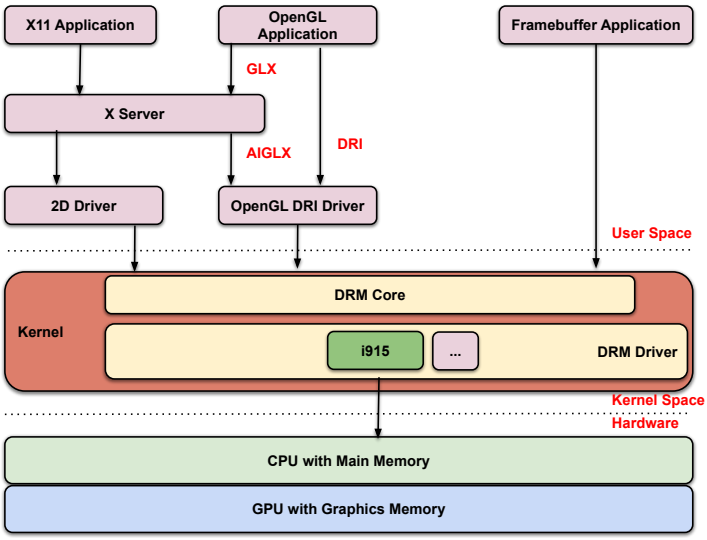

Figure 2.7: Current Graphics Stack [50]

The new DRM architecture shown in Figure 2.7 is mainly comprised of two components:

a) **DRM core:** DRM core can be called as the frozen part of the DRM module. This implies that it is a generic part of the module that exposes the hardware-independent part of the DRM API. The core acts as a kernel abstraction layer by allowing any DRM driver can register with the core.

b) **DRM driver:** The DRM drivers expose all the hardware-dependent part of the DRM API. As DRM drivers can be different in different

systems depending on the GPUs used, they can be called as the hot part of the DRM module. The drivers are responsible for tasks such as conversion of the commands and managing the command buffer, video memory, registers and DMA engines. As stated earlier, i915 is the DRM driver for Intel iGPUs. Therefore, this is the part of the Linux graphics stack that communicates directly with the hardware.

2.3 INTEL INTEGRATED GPU

Intel iGPUs are a common component of today's devices like laptops and desktops. In contrast to the Discrete Graphics Processing Unit (dGPU), iGPUs are included on the same chip as the CPU. iGPU has become an attractive resource for specific tasks like light gaming, media workloads, machine learning applications [13] which carry sensitive data.

Surprisingly, even though researchers have spent decades securing the CPU, there is hardly any clarity on the security of the GPU. This leaves them vulnerable to attacks. The following reasons can be attributed to the lack of research in this department [56]:

1. Understanding how to secure the GPU requires an extensive domain knowledge on the interactions between the device hardware, the software stack, buses and chip-sets.
2. The graphics stack is poorly documented.
3. Many device drivers are closed-source.
4. The drivers have vendor-specific APIs. The security vulnerabilities and prevention measures required can be different for each vendor.

2.3.0.1 *Intel Processor Graphics Model*

This section presents a high-level working of the *Intel GPU Programming Architecture*, shown in figure 2.8. A discussed in section 2.3, Intel iGPUs, shares the system memory with the CPU while the other discrete GPUs might have their own dedicated physical memory.

First, an introduction to the part of the architecture that is common to most GPUs. A GPU has a *render engine* and a *display engine* that communicate with different buffers. The CPU communicates via GPU-specific commands that are submitted via the GPU drivers to the command buffer. APIs like OpenGL see the command buffer with a primary *ring buffer* that helps to chain together the *batch buffers*. The render engine is responsible to fetch the commands from the command buffer and then executes them [48]. The display engine is responsible to fetch these pixels from the frame buffer that is rendered on an external computer display.

Now, we discuss the part specific to the iGPU programming architecture. The *system memory*. Shared Virtual Memory (SVM) is a 2GB *global virtual graphics memory* that is shared between the CPU and the iGPU is mapped via the *global page table*. It serves as the command and frame buffer. The iGPU has 2GB of *local virtual graphics memory* that is mapped via the *local page table* and mainly serves during hardware acceleration.

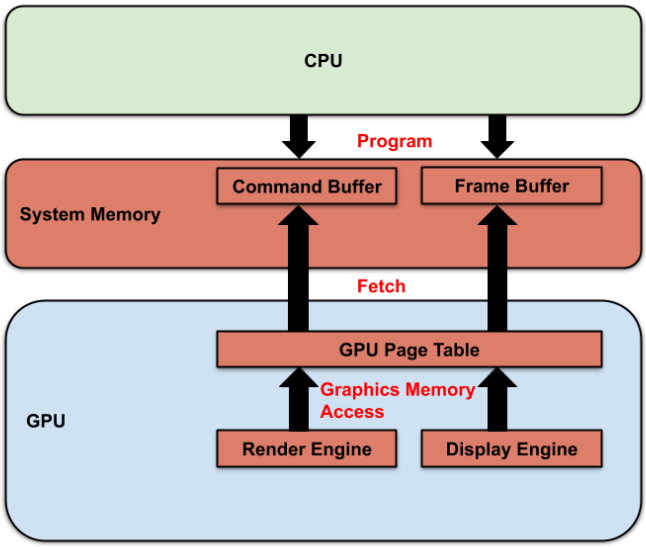

Figure 2.8: Intel Processor Graphics Architecture [1]

2.3.1 *Intel i915 GFX Driver*

One of the DRM device drivers of Intel is *i915 GFX driver*. The i915 driver is presented as a kernel module that exposes an API interface to the user-space applications that can be accessed via the ioctls. The i915 driver provides various functionalities that are divided into following parts:

1. **Display** - This part includes everything related to the display hardware. i915 is the only DRM driver that implements it's own mode setting infrastructure.
2. **GEM** - This part of the driver manages the GPU memory and assists in submitting the commands to the GPU buffer.
3. **Core Driver Infrastructure** - This part is used by both the display and GEM parts of the driver.

The user-space application can access the i915 driver via the device files and can use the system call *open* to request access from the kernel for these files. Once the application is validated, the kernel allocates the resource to the file and provides the application a handle called *file descriptor* to refer to the device file.

The application can then access various ioctls exposed by the i915 driver via the file descriptor. The *close* syscall is used to end the access to the file.

The i915 driver might be a good candidate as a first step at securing the the Intel Linux Graphics Stack because of the following reasons:

1. As stated in section 2.1.3, device drivers operate at a privileged level and talk directly to the hardware. This makes the GPU device driver a potential attack surface.

2. The code for Intel i915 driver is open-source and therefore, any security issues can be easily patched.

3. Intel iGPUs have a market share of 64% [3] which is a huge target surface. This calls for an urgency to investigate the security implications of the iGPU drivers.

2.4 FUZZING

Fuzzing refers to the process of discovering vulnerabilities by repeatedly running fuzz inputs against a target software and observing for unforeseen behaviour. According to [34] *fuzz input* is an input that can provoke an unexpected behaviour from the software under consideration. With the help of fuzzing, various types of potential bugs or a critical software vulnerabilities. Therefore, it has been used by various security researchers to test targets like parsers, network protocols, crypto libraries, compilers, interpreters, browsers, text editors/processors, OS Kernels and many more [4]. On the basis of source code availability and the amount of program analysis involved, the fuzzers can be classified as:

1. **Black-box Fuzzer**: Fuzzer that cannot see the source code of the target software and can only observe the input/output behaviour of the software. With this fuzzer, there is no verification of which parts of the code have been covered.

2. **White-box Fuzzer**: Fuzzers that have access to the source code and hence, have more control over the information retrieved through static analysis and control flow. The fuzzer can channel the fuzz input with the visibility of the internal code being fuzzed to get maximum code coverage.

3. **Grey-box Fuzzer**: Fuzzers that take the middle ground between black-box and white-box. They do not have access to the source code, but they make use of the disassembly or dynamic runtime information to retrieve useful information such as code coverage. This can be achieved by *instrumenting* the source code or using certain tracing approaches.

   The inputs in the above fuzzers can be completely random, mutated or generational.

2.5 COVERAGE-GUIDED FUZZING

In the basic fuzzing algorithms, the output from the fuzz inputs is generally ignored. An extension to these algorithms is to use this output to improve the quality of future fuzz inputs. The *coverage-guided fuzzer* maintains an input corpus containing the interesting fuzz inputs and mutates these inputs using certain procedures. The mutated inputs remain in the corpus only if they provide some "'new coverage"' and hence, receive positive feedback. The rest are discarded. This fuzzing strategy has shown to be very effective in finding real-world bugs and vulnerabilities [15], [37].

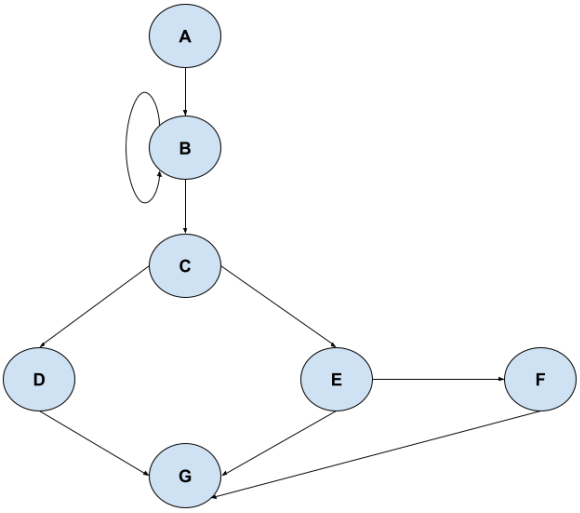

Figure 2.9: Control Flow Graph

Algorithm 1 Generic Coverage-Guided Fuzzing Algorithm

```
1:  Input Corpus: C
2:  while 1 do
3:      Random Input M is chosen from C
4:      M is mutated to M'
5:      Run M' against the target
6:      if crash then
7:          Save to disk
8:      else if new coverage explored then
9:          Add M' to C
10:     else
11:         M' is discarded
```

What really is *coverage*? This would depend on what kind of fuzzer it is and what are it's goals. The most common way to define coverage is by "'how much of the code is covered"'. To determine this, the number of code parts that are

executed are counted. Traditional code-coverage metrics include the *basic blocks*, *edges*, *paths*, *number of lines*, etc. In this case, the fuzzer is known as a coverage-guided fuzzer. When a fuzz input explores a new line, block, edge or path, the code-coverage is incremented. For example, consider a fuzzer uses *edge coverage*. In this case a fuzz input would be considered interesting if it finds a new edge from the control flow graph shown in 2.9. If previously path A->B->C->D->G has been discovered by the fuzzer, then the new fuzz input is considered interesting and kept in the corpus after it finds the path A->B->B->C->E->G as the edges B->B, C->E and E->G have not been explored before. After these two paths have been explored, if a fuzz input explores the path A->B->B->C->D->G, it will not be considered interesting even if it has lead to a unique path.

A generic algorithm for coverage-guided fuzz testing is presented in 1. The algorithm takes a set of fuzz inputs C. The next steps carry out the fuzz testing and are performed inside an infinite loop. A random input M is chosen from the corpus C and mutated it according to some procedure. The mutated input's code-coverage is measured after running it against the target software. If the input causes a crash, then save it to the memory. If it covers new parts of code, then add it to the corpus C. Otherwise, discard the input.

3

FUZZING THE I915 DRIVER

This Chapter provides a brief overview of the different approaches and challenges for fuzzing the i915 driver and gives a brief overview of different technologies and concepts that will be used in this thesis. Section 3.1 presents the challenges a fuzzer needs to overcome to fuzz the i915 driver. Section 3.2 defines the benchmarks on the basis of which a fuzzer setup can be considered reasonable to deploy at large-scale driver validation. Section 3.3 gives a brief overview of the different possible approaches to feed input to fuzz the i915 driver and our chosen approach. The next sections provide an introduction to two prominent kernel fuzzers to perform fuzzing on the i915 driver. Section 3.4.1 and Section 3.4.2 describe the basic idea and concepts of Syzkaller and kAFL respectively.

3.1 CHALLENGES IN FUZZING DEVICE DRIVERS IN KERNEL-SPACE

In this section we present the challenges that can be expected while fuzzing device drivers in kernel-space. We first describe the general challenges that come with fuzzing in kernel-space and then we describe the challenges specific to fuzzing a device driver.

3.1.1 *Kernel-Space Challenges*

The following are the challenges that prevent a user-space fuzzer like AFL to fuzz the kernel-space:

1. It is much harder to collect dynamic branch/edge information during runtime for kernels. Several approaches have been proposed but they lead to poor runtime performance or require the target kernel to be recompiled (`kcov`).

2. The mechanisms of crash detection need to be modified as a kernel bug can lead to termination of fuzzing. Therefore, the fuzzer needs to run external to the target kernel so that it can reset the state of the target kernel. Another possibility could be to run the fuzzer on the host and save the fuzzer state even if the host kernel crashes.

3. The kernel has non-deterministic events like interrupts or trap handlers. These can lead to unnecessary noise in the collected coverage.

3.1.2 *Device Driver Challenges*

Now, we discuss the challenges that can arise while fuzzing device drivers like the i915 driver:

1. **Device Access**: To fuzz a device driver it is necessary for the device to be attached to the system under test. The fuzzer should either have the capability to attach and detach the physical device to the system or be able to simulate it.

2. **Stateful Device**: The added device to the system becomes an added source of fuzzing instability as it contains ***state information*** at runtime and when the device is in a different state, the handling of commands from the device driver and external interrupts may be different. The state of the device is controlled by *configuration registers*. A user process can change the configuration registers, which further changes the working logic of the device. This essentially means that the device has transitioned from the prior state to a new one. The changes according to hardware interrupts also affect the change, which a user cannot control.

   If a fuzz input is able to reach the device, the behavior of the device under this input is dependent on the previous state of the device and the current input itself. The current input can also generate a new state. As hundreds or thousands of random fuzz inputs are sent to the device, the device state is continuously transitioning.

   This is a key problem when fuzzing a target with the involvement of real hardware as it can lead to the following challenges:

   - **Fuzzing termination**: It can possibly cause a device to reach a dirty state and cause a hang that can crash the entire system during the fuzzing process. This could terminate the fuzzing process if the fuzzer is running on the host. This leads to the following problems: (1) The fuzzer looses the data of this fuzzing session; (2) It is not able to contain the crash or record the crash information and (3) The fuzzing process does not remain continuous. In case of a crash, the fuzzer should be able to contain the crash, save the crash information and continue fuzzing.
   - **Input Effectiveness**: Assume the initial state of hardware as $STATE_0$. When the first fuzz input A is executed against the target, it does not reach the device and fails to cause a state transition in the device. The corresponding feedback is provided to the fuzzer. The second input B inherits the $STATE_0$ and is able to cause a state transition in the device to $STATE_1$. The next input C inherits the changed device $STATE_1$ and the corresponding feedback is impacted and provided to the fuzzer. In reality, the input evaluation is done considering that the feedback

received is based on the initial state of the system. This can lead to ineffective inputs being generated.

- **Reproducibility**: It is evident from the previous point, that the state transitions add non-determinism to the input execution. The fuzz input might generate different coverages depending on the device state it inherited. Therefore, if a crash is observed it might not be possible to reproduce the crash without knowing the state information.
- **Non-Deterministic Multi-Core Processor**: This challenge is specific to device drivers for multi-core processors. Most single-threaded programs executing on a uni-core processor exhibit determinism. With the increased parallelism of multi-core processors and multiple fuzz inputs targeted at it, the non-determinism to the fuzzing process increases.

If the state information is not monitored it maybe difficult to get effective inputs and correct coverage information which can impact the effectiveness of using coverage-guiding for the fuzzer and the reproducibility of the results. Therefore, the fuzzer should also be able to monitor state information or reset the device to initial state after every fuzz input. But recording the state information after every the fuzz input can be extreme wastage of resources.

3. **Parallelism**: For a user-space target it is easier for the fuzzer to get maximum efficiency using paralellization. Multiple instances of the the target can be started and fuzzed. But this can be difficult to achieve with only one physical device.

4. **OS-Agnostic Solution**: Device Drivers are *OS-dependent*, therefore the scalability of the fuzzing solution is affected as it is not *OS-agnostic*.

3.2 QUALITATIVE CRITERIA FOR A FUZZING SOLUTION

A good fuzzing solution should deal with all the challenges presented in Section 3.1. One of the goals of this thesis is to find a fuzzing solution that can fuzz the i915 driver. The aim is to find an easy to implement solution that can be used for automated driver validation at a large-scale. We approached this step by first defining what the different criteria are that we look for in an ideally perfect fuzzing setup:

1. **Hardware Access**: The fuzzing setup should have the hardware available in some way or the other for the driver to work. The fuzzer should be able to access the real hardware or simulate the device.

2. **OS-Agnostic**: This is an important criterion for large-scale driver validation process. The solution should be OS-agnostic as it can help with the scalability of the same solution to different OSes wit h little to no effort.

3. **Technical Knowledge**: The amount of technical knowledge required to set up the solution should be minimal.

4. **Crash-Tolerant**: Any fatal kernel or device bug can interrupt the fuzzing process. The fuzzer setup should be able to contain the crashes, store the crash information and continue fuzzing. This will help the developers to inspect the bug.

5. **Cost-effective**: For use in large-scale driver validation, it is important that the fuzzing setup is cost-effective. It should minimize the dependencies on additional expensive hardware. The setup should ideally be able to use one physical device in a parallelized manner to get maximum efficiency and reduced hardware costs. It should additionally prevent manual labor for source code annotation or bug triaging. Also, minimal amount of time should be required to deploy the setup.

6. **Reproducibility**: The fuzzing setup should be able to ensure that the results are reproducible. For this, it should deal with the non-deterministic execution of the fuzz inputs against the target kernel.

3.3 POSSIBLE FUZZING APPROACHES

In this thesis we test the i915 driver by sending random inputs that are generated by a fuzzer. One of the first steps is to decide how we want to feed these generated inputs to the device driver as it requires that the hardware is present. As we have seen in the earlier section, fuzzing with the hardware involved can come with added challenges. Before we introduce our chosen approach, we introduce some of the possible approaches at a high-level that have been used for fuzzing device drivers and could possibly be applied to the i915 driver:

3.3.1 *Device Emulation*

In this approach, the fuzzer uses an *emulated device* to feed the input to the device driver. The emulated device can be integrated to a guest machine with the target kernel loaded. In this case, the native device driver can be loaded on the guest. In [39], the authors have modified the *hypervisor* in the guest such that the user-space application's requests are transferred to the emulated device and not the real hardware. A hypervisor is a software layer that allows one host system to support multiple guest machines by virtually sharing the system resources. Therefore, the fuzzer input can be sent via a user-space application to the device driver code. Under this approach, there might not be any requirement to make changes to the device stack.

Now, we look at the possible advantages of applying this approach to fuzz the i915 driver:

1. **Low hardware costs**: Due to device emulation, the hardware costs are reduced.

2. **Parallelization**: The setup can also be highly scalable as multiple guest instances can be used to fuzz the device driver in parallel. The setup can be scaled up by using the cloud.

3. **Portability**: As this approach works at the hardware level, there is no dependence on the type of OS being used as the target kernel.

Unfortunately, there are certain big challenges with this approach if we want to apply it to the i915 driver:

1. **Technical Knowledge**: As the iGPU is a complex device with a complex graphics stack, as explained in Section 2.2, the user needs a deep understanding of the different components of the stack and the inner workings of the CPU and the iGPU. Thus, the approach is labor intensive and can take a significant amount of time to deploy.

2. **Inaccurate Results**: It is not possible to perfectly emulate the workings of the real hardware and the non-determinism that it adds. Therefore, the results might differ from the results on the real hardware.

3.3.2 *Data Injection in the Device Stack*

In this alternative approach the target kernel is modified to feed the fuzzer generated inputs to the drivers by injecting them at a certain layer of the device's stack. This approach has been followed by Syzkaller for Universal Serial Bus (USB) fuzzing [39] by replacing the driver for the hardware host controller by a software user-space controller that feeds the fuzz inputs into the device's stack (`dummy hcd`). In PeriScope [46] the authors have modified the Memory-mapped I/O (MMIO) and Direct memory access (DMA) interfaces by intercepting the driver's access to the communication channels and fed the fuzzer input to these interfaces.

When applying this approach to the i915 driver, the GPU has to be made available to the virtualized environment. The possible advantages of applying this approach to fuzz i915 driver may be:

1. **Low hardware cost**: As these changes are again at the software level, the hardware costs are reduced.

2. **Parallelization**: The setup can be run in multiple guest machines, making it highly scalable to fuzz the i915 driver in parallel. The setup can also be scaled using the cloud.

The biggest challenges with this approach for fuzzing i915 driver are:

1. **Portability**: This approach is tightly coupled to the specific kernel being modified and probably the kernel version (therefore, the i915 version) as well.

2. **Technical Knowledge**: This approach also requires a deep understanding of the graphics stack, the i915 driver code and the communication channels between them.

3. **Coverage**: The code is not tested end-to-end in this approach as the inputs are injected at some specific stack layer.

3.3.3 *Fuzzing on Bare-Metal*

This next approach is straight forward in the sense that we feed the input to the host device driver directly from the fuzzer running in the user-space on the host itself. The advantages of this approach for i915 driver can be:

1. **Performance**: This option is expected to have the highest performance as no virtualized environment is involved. The native graphics driver and the real hardware are involved in the fuzzing.

2. **Easy Implementation**: The setup is easy to implement as no modification is required at hardware or software level.

3. **Minimal Technical Knowledge**: There is no requirement to understand the graphics stack or the workings of the CPU/iGPU. The knowledge of i915 driver API for a specific OS would be required.

Even if the approach looks tempting, it has certain drawbacks:

1. **High hardware costs**: Hardware costs increase if we want to test the i915 driver on different types of systems.

2. **Parallelization**: As only one system is fuzzed, achieving parallelization might not be that easy.

3. **Crash-Intolerant**: If there is a bug caught in the host kernel, driver or the GPU, there is a high possibility that it causes the system to crash. As the fuzzer is also running on the host, the fuzzing process can be interrupted and cannot recover. The information on the bug is also possibly lost and hence we cannot reproduce the bug.

3.3.4 *Device Virtualization*

In this section we introduce the approach of *device virtualization* that we have chosen for this thesis.

Multiplexing is the ability for multiple virtual machines to share the same physical GPU. Device virtualization multiplexes the real device by presenting each guest with a *virtualized* device and combines the operations of both devices in the hypervisor in a way that all the guests use the real device while preserving the illusion that every guest a device of its own [12].

We choose this approach to fuzz the i915 driver because of the following reasons:

1. **Minimum Technical Knowledge**: The user does not need to know the hardware protocol or the software stack in detail to implement this setup.
2. **Crash-Tolerant**: The crashes are expected to be contained in the virtualized environment.
3. **Cost-effective**: Using certain configurations of this approach it might be possible to achieve parallelism in the fuzzing process using one physical GPU.

Unfortunately, virtualizing a modern GPU is regarded as much more challenging than Input/Output (I/O) devices like disks or Network Cards (NICs) [21] The biggest challenges of virtualizing a GPU device are [55]:

1. The GPU architecture and the graphics stack are complex.
2. The non-standardization of the Graphics Stack due to different vendors. Also, many vendors do not release the source code of the GPU drivers. Even if the source code is reverse-engineered, with every new release of the GPU driver significant changes are made to the code that makes this method unreliable.
3. The GPU architectures involve dramatic changes across generations and their generational cycle is short compared to the CPUs and other devices [12].

Therefore, rather than attempting to model a complete GPU, different vendors have adopted different options for the virtualization of the GPU. In this thesis, we will explore options that can be implemented to virtualize the Intel iGPU to load the i915 driver. In the upcoming Sections, we discuss these different platforms and configurations. Some of these configurations are specific to Intel iGPU. In Section 3.3.5, we discuss which platform and configuration are chosen for this thesis.

API Remoting

The *API Remoting* approach exposes the guest OS with a library that has the same API calls as the GPU API and a frontend driver, shown in Figure 3.1. This driver intercepts the high-level API calls (OpenGL, DirectX, CUDA and OpenCL calls)

from the application before it reaches the GPU driver in the guest OS. These calls are then forwarded to the backend driver in the host OS using shared memory or to a remote host with a GPU. After the computations have been performed, only the results are delivered to the frontend driver through the backend driver. In short, the aim is to provide an emulated wrapper library which passes the actual computation to another machine on the local network.

This approach helps overcoming the vendor dependence and ability to run multiple VMs on one GPU. Due to simple virtualization architecture, it incurs negligible virtualization overhead [21]. As the virtualization is done in user space, hypervisor independence is achieved.

The approach also displays some significant drawbacks. High-level API versions would need to be handled specifically as there might be significant changes to the standards. Maintaining the frontend driver can be difficult as some new API calls might be added to the GPU libraries. [21]. The approach might be a good choice if the guest application is not GPU intensive.

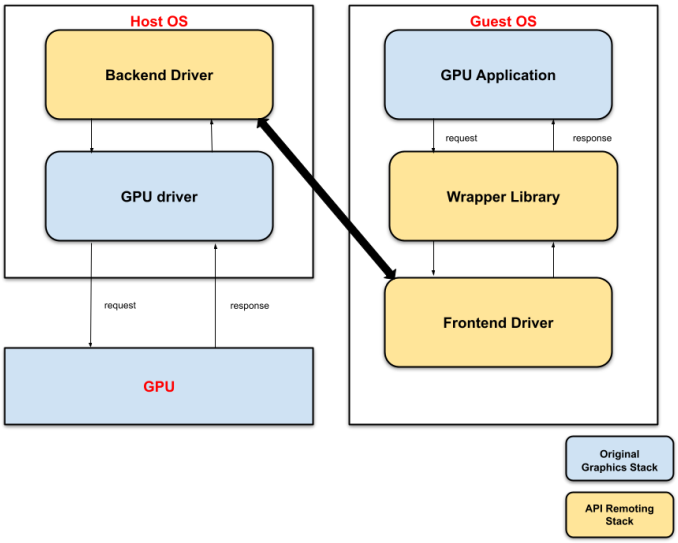

Figure 3.1: API Remoting Architecture [21]

Intel GVT-s

Intel GVT-s has been developed based on this approach, shown in Figure 3.2. It helps to forward the API calls of OpenGL or DirectX applications to the i915 driver on the host OS. It can help to run concurrent guests using only one GPU, abstracting the graphics hardware from the guest applications by performing graphics API forwarding.

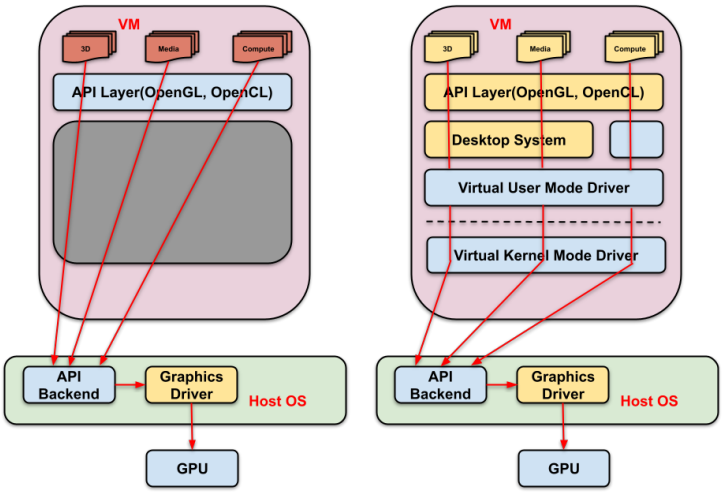

Figure 3.2: GVT-S Software Stack [1]

Full Virtualization

Full Virtualization, shown in Figure 3.3 is another alternative approach. It uses an unmodified GPU driver in the guest OS. The Quick Emulator (QEMU) can emulate a real GPU in such a way that the guest OS regards the emulated device as the real GPU. The GPU calls from the guest applications are intercepted by this emulated device and sent to the host GPU using the shared memory in the hypervisor. The emulated GPU device is known as Virtual Graphics Processing Units (vGPU). The states of all the vGPUs are stored in a *control block* and each vGPU has a queue that schedules the GPU calls. When a GPU call is made, the GPU scheduler picks a vGPU and the corresponding queue for the GPU calls.

As the virtualization is implemented in the lower layers at the driver level, it is possible to reuse the present GPU libraries and handle the latest versions. The biggest drawback of this virtualization method is that its implementation is dependent on which GPU driver is used. This means that this process will rely on whether the code of the driver is available or does it need to be reverse-engineered. If there is a significant change to the GPU microarchitecture, it could prove to be a daunting task to update the implementation.

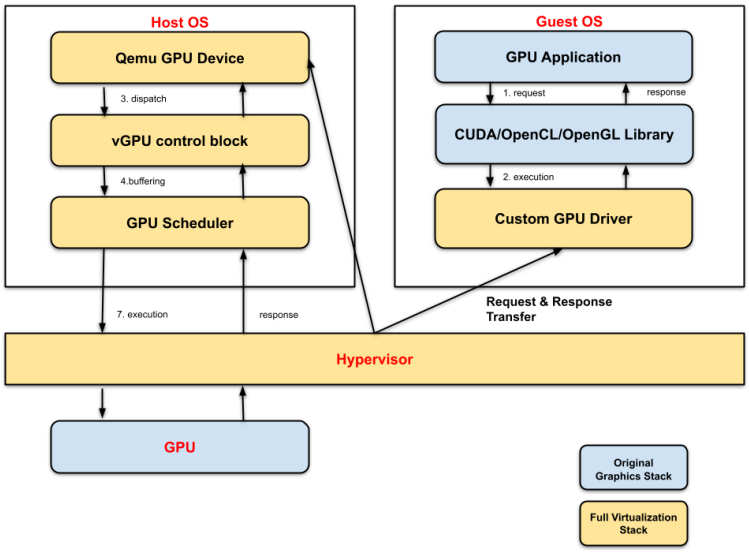

Figure 3.3: Full Virtualization Architecture [21]

Intel GVT-g

Intel GVT-g is an implementation of the Full Virtualization concept for Intel iGPU. Figure 3.4, shows the GVT-g architecture with Kernel-based Virtual Machine (KVM) hypervisor. KVM assists in changing the Linux into a hypervisor.

Intel GVT-g allows the guest to access the real GPU with the native graphics driver in the host. The technique used is called *mediated passthrough* that allows access to performance-critical resources without the hypervisor intervening most of the times by providing the guest applications the power to directly execute the command buffer and the frame buffer. Isolation to the guests is ensured by trapping and forwarding to a mediator driver to emulate the privileged operations. The mediator issues a hypercall to communicate with the real GPU to execute these operations and hence, avoiding the complex task of emulating the render engine. The mediator also provides a GPU scheduler that helps in sharing the real GPU among multiple guests.

The *mediator driver* is implemented as a kernel module and provides emulation of vGPUs. The native graphics driver runs in the guest and accesses the privileged resources via the mediator. The accesses to command buffer and frame buffer are passed through to accelerate the performance-critical operations in the guest. This is achieved by dividing the *global graphics memory* between the different guests. The native graphics driver does not have the knowledge of the partition and expects to be the exclusive owner of that part of the memory. This means that the guest and host will have different views of the *global graphics memory*. To prevent additional overhead of address translations a technique called *address space ballooning* helps each guest in marking the parts of the memory that belong to the other guests as *ballooned*. This makes the global graphics memory view of the guest similar to the host.

While this technique has been used to virtualize Intel GPUs, the authors have claimed that the technique can be used to other GPUs as well.

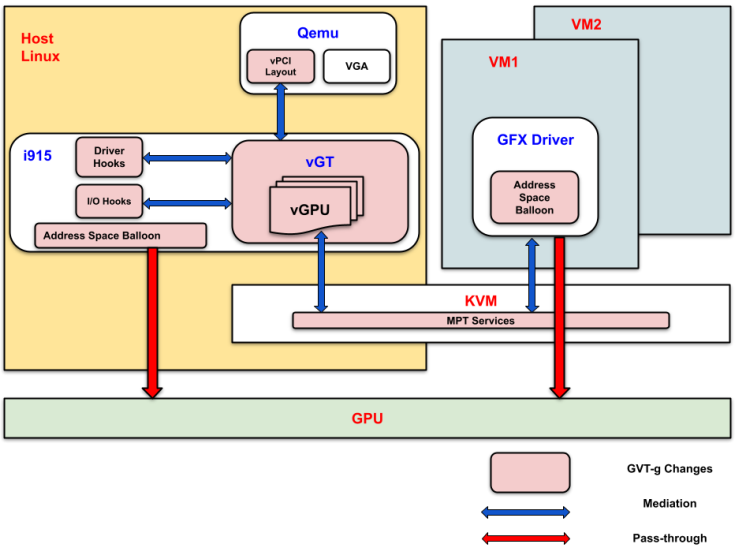

Figure 3.4: GVT-G Architecture with KVM [48]

Hardware-Assisted Virtualization

The *Hardware-Assisted Virtualization* approach involves utilizing the hardware extension features provided by different chip vendors for I/O virtualization. Intel chips provide the feature *Intel Virtualization Technology for Directed I/O (VT-d)*. This hardware feature assists the hypervisor to get the direct access of the devices. It provides isolation and security by restricting device accesses to the owner of the device memory. A technique called *DMA-remapping* is used to restrict the DMA and the interrupts of the device to pre-assigned physical memory regions. This allows the data to flow seamlessly between the guest memory and the pyhsical device without the need of a hypervisor. It also performs access control based on the information provided by the guest OS. If an illegal access is performed, DMA-remapping hardware blocks these calls and reports a fault to the guest OS.

The major drawback of this approach is that only a single guest is supported by Intel VT-d for I/O virtualization.

Intel GVT-d

Intel GVT-d leverages the Intel VT-d technology to provide the guest with full access to the real GPU shown in Figure 3.5. The Linux graphics stack can directly use the real GPU without having to make any modifications to the driver code or GPU libraries. This means that the host will have no access to the GPU during this time and hence, this approach cannot allow the sharing of the GPU with multiple guests.

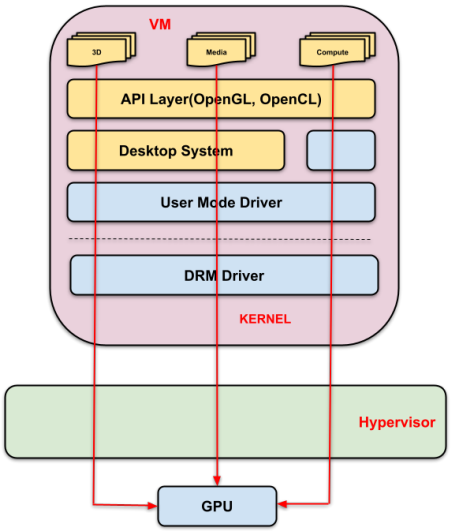

Figure 3.5: GVT-D Software Stack [1]

SR-IOV

This is another variant of hardware virtualization. The goal of this technique is to standardize the way of bypassing the hypervisor's involvement in the movement of data between the guests and the hosts or between the guests. Single Root IO Virtualization (SR-IOV) implements I/O virtualization directly on a device and can provide very fast I/O. Multiple Virtual Functions (VFs) are implemented, with each VF directly assignable to a guest. This technique has been used by some vendors for GPU virtualization. This technique is mainly used for NICs and is currently not supported by Intel iGPU. This approach can overcome the drawback of Intel GVT-d of not supporting multiple guests and also provide the native hardware for fuzzing.

3.3.5 *Choice of Virtualization Method*

In this section we present the reader with the GPU virtualization configurations that we investigate in this thesis. We also discuss their advantages and disadvantages in the fuzzing process.

Intel GVT-s and SR-IOV are not supported for Intel iGPUs at the time of writing this thesis. Therefore, we decide to investigate other available options Intel GVT-g and Intel GVT-d for our fuzzing solution during this thesis.

Both GVT-g and GVT-d configurations can provide certain advantages to our fuzzing solution as follows:

1. The user is not required to possess skilled technical knowledge to implement the configurations. But the user is required to be aware of the GPU pass-through concepts.

2. Their implementation process for machines with similar specs can be automated.

3. GVT-g provides the possibility to increase fuzzer efficiency. It allows the fuzzer to target multiple guests by multiplexing the GPU.

Unfortunately, both the configurations can provide challenges while fuzzing the i915 driver:

1. In GVT-g the guest is presented with a vGPU which does not have any code coverage feedback. Therefore, it might be hard to know the status of the virtualized component to optimize the fuzz inputs for effective testing.

2. GVT-d does not support multiplexing therefore, the efficiency of the fuzzer might be affected.

3.4 FUZZING TOOLS

The next step to building a fuzzing solution is to find fuzzing tools that solve the challenges using the chosen device virtualization techniques. Coverage-guided fuzzing is a well-established technique used by fuzzers for the user-space [14], [44], [17], [40]. But there are only a few that are targeted at the kernel-space. Some proposed coverage-guided kernel fuzzers have been developed keeping the popular AFL design as the base [7], [51], [29]. In this thesis we will look at two prominent solutions: kAFL and Syzkaller.

kAFL is also built upon the fundamental AFL design and benefits from the hardware feature Intel processor trace [43]. It is able to fuzz OS kernels and trace the execution accurately with low overhead. There have been other kernel validation extension that have been published using kAFL [5], [20], [42]. In this thesis we use a version, maintained by Intel, that is modified and integrated with the published extensions. For the purpose of presentation, we call this tool now onwards **Intel kAFL**.

Syzkaller is a widely used coverage-guided fuzzer that exploits the predefined syscall descriptions to generate sequence of syscalls. Many researchers have tried to integrate different kernel validation techniques and optimizations with Syzkaller [38], [22], [54]. It has also been successfully deployed to fuzz the Linux kernel at large-scale.

In the following Sections 3.4.1 and 3.4.2 we introduce both kAFL and Syzkaller in detail.

3.4.1 *Syzkaller*

This section will provide an overview of the technical aspects and the implementation details of Syzkaller.

Syzkaller is a coverage-guided kernel fuzzer that is momentarily the most widely-used fuzzer to fuzz the Linux kernel. It was initially developed as an open-source project by Dmitry Vyukov and a team from Google in 2016 [28]. It has been successful in finding 3000+ bugs to date in the upstream kernel. Syzkaller also supports other OS kernels like Akaros, Darwin/XNU, FreeBSD, Fuchsia, NetBSD, OpenBSD and gVisor [15].

Blind fuzzing for kernel can generate huge amounts of input and only a few of them turn out to be valid inputs for the kernel. It has been observed that most of the bugs that are caught in this manner are input validation bugs[23] and it is difficult to cover the entire kernel code [36]. The best way for the user-space applications to access the kernel is through system calls, therefore, Syzkaller uses the semantic knowledge of these system calls to manipulate fuzzing inputs. Using these descriptions, Syzkaller is able to cover a large part of the kernel and uncover different types of bugs.

For further optimization of the fuzzing process, Syzkaller notes the code-coverage achieved by each input and tries to maximize the coverage [2]. A *corpus* of inputs is created and any input that increases the coverage is mutated to reach more code for testing.

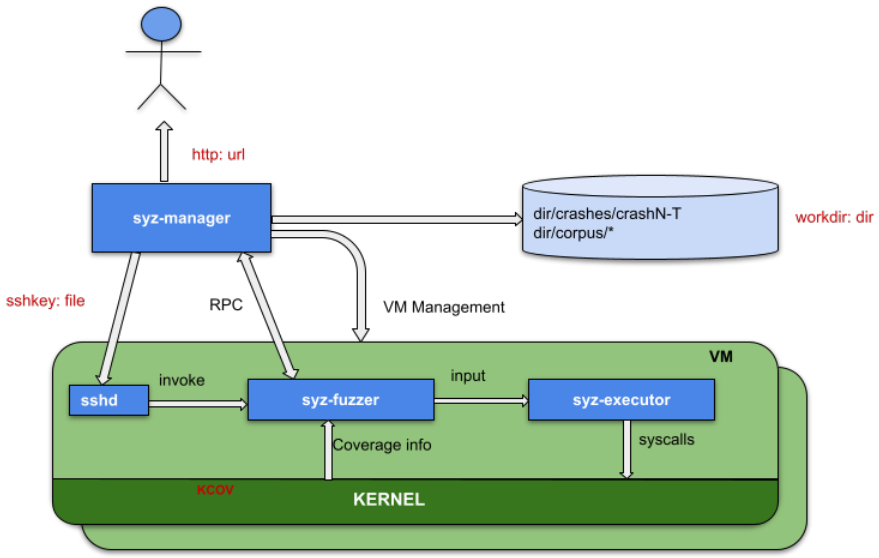

Figure 3.6: Process Structure for Syzkaller [15]

As shown in Figure 3.6, Syzkaller comprises of three main components: syz-manager, syz-fuzzer and syz-executor. *Syz-manager* is the "manager program" that controls the entire fuzzing campaign. Once started, it spawns and monitors multiple guest instances that have the target kernel and the binaries *syz-fuzzer* and *syz-executor* running inside of them. The manager uses the Secure Shell (SSH) protocol to connect to these virtual machines and invoke these binaries. Using

Remote Procedure Call (RPC) communication, the manager remains in contact with the syz-fuzzer to communicate information to and fro the virtual machine. The manager can send the config file and existing inputs to the machines and receive the results via the syz-fuzzer.

On the VM side, syz-fuzzer carries out the fuzzing campaign by generating, mutating and minimizing the test cases. It forwards these inputs to the syz-executor binary, using Inter-Process Communication (IPC), that runs these inputs against the target kernel. This triggers coverage information that is then communicated to the syz-fuzzer. Information about any input that generates new code coverage is provided to the syz-manager. The manager stores the information about crashes and corpus in the work directory provided by the user. The manager also exposes a web-interface for the user to view the details of the fuzzing campaign and the code-coverage information.

3.4.1.1 *Harness Setup*

Syzlang

Syzkaller provides the user with the capability to define the syscall interfaces of the target in a *description file* using a declarative format called *Syzlang*. Following is an example of how the syscalls might be described using Syzlang:

```
open(file filename, flags flags[open_flags], mode flags[open_mode]) fd
read(fd fd, buf buffer[out], count len[buf])
close(fd fd)
open_mode = S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH,
    S_IWOTH, S_IXOTH
```

As seen above, it can support types like file descriptors, flags, etc. It also gives the user the power to define compound data types like struct, union, enum, etc.

Input Generation

The description file is translated into an executable code by syz-executor. The descriptions are stored as a *map* data structure. If the user wants intends to perform targeted fuzzing, they need to enable the target syscalls in the *config* file. If none has been defined, the fuzzer uses all the syscalls defined in the target kernel folder. If certain syscalls have been enabled, then the fuzzer marks them enabled in the *map*.

Syzkaller implements a mechanism to create Programs (Progs) such that new coverage can be triggered. Using the map, the fuzzer builds a *ChoiceTable* which contains weighted probabilities of all the enabled syscalls depending on the previously executed syscalls. For given syscalls X and Y, the probability of Y is a best guess whether a fuzz input already containing the syscall X would give new coverage. If not, then syscall Y is added.

The probability is calculated using the *static priority* and the *dynamic priority*. The static priority is calculated by examining the argument types of both the syscalls. The dynamic priority is calculated by examining the frequency of occurrence of the pair of X and Y syscalls in the current corpus.

The ChoiceTable is then used to generate a list of programs `Prog` that are to be executed. Each program is a list of sequential syscalls with concrete values for arguments. Following is an example of program generated by Syzkaller:

```
r0 = openat$i915(0xffffffffffffff9c, &(0x7f0000002840)='/dev/i915\x00', 0x0,
    0x0)
ioctl$DRM_IOCTL_I915_QUERY(r0, 0x4602, 0x0)
```

3.4.1.2 *Fuzzing Strategy*

Syzkaller aims to prioritize the test cases that can lead to interesting areas. It employs the following techniques to achieve this:

Mutations

After a Prog has been generated and run against the target kernel, Syzkaller mutates the current test case using smart mutations. The following mutations are used according to their likelihood calculated using certain heuristics:

1. Syzkaller can remove a syscall from the Prog test case at random.
2. Syzkaller can change the value of a particular argument. This becomes possible because it already has the knowledge of the types of the syscall arguments. For example, it can resize the arrays/buffers, change union options, flags, len/bytesize, filename or pointers.
3. Syzkaller can insert a syscall to the Prog test case.
4. Syzkaller can also splice a Prog with another one from the corpus. This is based on resources. First, it helps in deciding which programs are to be spliced. For example, if the Prog X uses a Transmission Control Protocol (TCP) socket then Syzkaller finds another Prog Y that also uses a TCP socket and splices the two test cases. Second, it helps in deciding which syscalls to merge. For example, if a syscall A uses the TCP socket, Syzkaller would include the syscall that creates a TCP socket.
5. Binary large objects (BLOBs) are a bunch of binary data stored in a single data type. These objects are used to store binary data for audio, images or executable code. If such an argument exists, then Syzkaller uses traditional mutation strategies like flipping bits, inserting/removing bytes, arithmetic, etc.

Coverage Collection

The first part of coverage collection involves setting up *compile-time instrumentation* which is supported both by GCC and Clang/LLVM. GCC and Clang/LLVM compilers insert a function call as instrumentation into every basic black and edge.

```
_sanitizer_cov_trace_pc();
while(...){
        _sanitizer_cov_trace_pc();
        ....
}
_sanitizer_cov_trace_pc();
```

The next part of coverage collection takes place during runtime. Syzkaller uses kernel module **KCOV** which provides the coverage data of the target kernel via the *kcov* debugfs file. KCOV can collect precise coverage of a single syscall as it is enabled on a thread basis. It is to be noted that interrupts cannot be traced with KCOV and certain kernel files like scheduler and memory allocator are not instrumented. There is a shared buffer between kernel-space and user-space that contains the coverage information of the target kernel. This buffer can be read by Syzkaller from the user-space.

Syzkaller maintains a *max coverage* and a *min coverage*. If the latest received coverage falls between both these coverages, the fuzz input is immediately discarded. But if the latest coverage is some new coverage and also the new *max coverage*, this fuzz input would be considered interesting.

Syzkaller runs the new interesting fuzz input multiple times to make sure that the coverage received is not flaky. If in all the runs the input produces new coverage, the input is added to the corpus otherwise, discarded.

3.4.1.3 *Why Syzkaller?*

There are many kernel fuzzers that have been proposed that promise attractive performances. we choose Syzkaller as it is a state-of-the-art fuzzer and widely used in the industry for fuzzing. Following reasons make it widely accepted in the industry and a good fuzzer for comparison to:

1. **Accessibility**: The Syzkaller source code has been released as open-source with Apache License 2.0 on Github. This favours the industry and acamedic research like this thesis into adopting the fuzzer as the costs of kernel fuzzing are reduced.

2. **Fuzzing Stateful Driver Interface**: The device driver interface is stateful. The fuzz input of Syzkaller includes a sequence of syscalls and the resources created by an earlier syscall is used by subsequent syscalls. In fact, syscalls are only fuzzed if their prerequisite resources have been created. This can result in higher code coverage of i915 driver because fuzzing would be

performed with valid syscall structures and there is a good chance that the interesting parts of the code will be reached.

3. **Support**: Syzkaller is under active maintenance with regular updates. There is an active Google group that provides technical guidance and troubleshooting support for the general users and academics. As there is active academic research in extensions for Syzkaller that are merged with the code, the industry can reduce their costs of maintaining the modified code.

3.4.2 *kAFL*

AFL's smart fuzzing design has seen overwhelming success over the last few years for user-space applications. kAFL is an *AFL-like fuzzer* that adds the layer of kernel fuzzing. In this section, we provide a technical overview of kAFL, how we set it up to fuzz the i915 driver and the results.

As kAFL's fuzzing logic is from the AFL family, it is important to first have a quick overview of AFL working.

3.4.2.1 *AFL*

AFL employs an *input queue* to store all the inputs that are either provided by the user or generated while fuzzing. The fuzzer picks an input and performs different *mutations* on it for a while and runs each mutated input against the target application. AFL leverages the compile-time instrumentation of the target applications to trace the coverage information and write it into a *bitmap*. If the coverage includes a new path, the mutated input is put in the queue, otherwise discarded. The *bitmap* is a shared memory buffer between the target application and the fuzzer's virtual memory and is 64KB in size by default. Each basic block and edge of the target application is assigned a random identifier. All the new paths are stored as a hashed tuple of two edge identifiers. These identifiers are added by compilers like `gcc` and `clang` after any x86-64 conditional and unconditional `jmp` instruction.

```
/* AFL INSTRUMENTATION */
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;

while(...){
        /* AFL INSTRUMENTATION */
        cur_location = <COMPILE_TIME_RANDOM>;
        shared_mem[cur_location ^ prev_location]++;
        prev_location = cur_location >> 1;

        ....
}
```

The mutating strategies are executed sequentially. First, comes the *deterministic stage* that includes sequential flipping of bits, sequential addition/subtraction of small integers and sequential addition of known interesting integers [16]. The purpose is to produce fuzz inputs in such a way that there is not much difference between crashing and non-crashing inputs. This stage is executed first for all the inputs that are currently in the queue. Second, is the non-deterministic stage that includes bit flips, additons, subtractions, arithmetics and splicing of fuzz inputs. These mutations can be perfomed at multiple random locations.

3.4.2.2 *General*

kAFL is an *OS-independent*, *AFL-inspired* and *hardware assisted* kernel fuzzer prototype. The 3 components of the fuzzer are (1) Fuzzing logic; (2) Guest infrastructure of QEMU Processor Trace (QEMU-PT) and KVM Processor Trace (KVM-PT); and (3) User-space agent. The fuzzing logic acts as the command and control of the fuzzer by managing the input queue, analyzing coverage traces, mutating inputs, scheduling them for testing and presenting results to the user. kAFL makes use of the hardware-accelerated feature of Intel's Processor Trace (PT) to gather coverage information of the guest process. kAFL uses a modified version of KVM and QEMU (called KVM-PT and QEMU-PT). KVM-PT running in kernel-space is responsible to activate or deactivate the Intel PT trace for the guest process. QEMU-PT running in user-space is responsible to communicate with the user-space agent, decodes the trace data into *AFL-compatible bitmaps* and passes it on to the fuzzing logic.

The guest infrastructure facilitates data sharing and communication using custom *hypercalls* and by providing direct memory access to the guest's memory[5]. KVM-PT has been patched to pass the custom hypercalls to the fuzzing logic.

The *user-space agent* runs inside the guest in user-space and simply gathers inputs sent by the fuzzing logic using the hypercalls and runs them against the target kernel in the guest. This agent is divided into two parts: (1) loader; (2) and

user-mode agent. The *loader* has the main responsibility to get the user-mode agent to the guest and execute it there. There are two reasons to divide the agent in two parts:

1. If the user-mode agent crashes, the loader can restart it.
2. We can pass any binary harness to the guest and multiple target harnesses while reusing the same guest snapshots to fuzz different components of the kernel

The biggest deviation from the design of AFL is the extensive use of parallelism and multiprocessing[43].

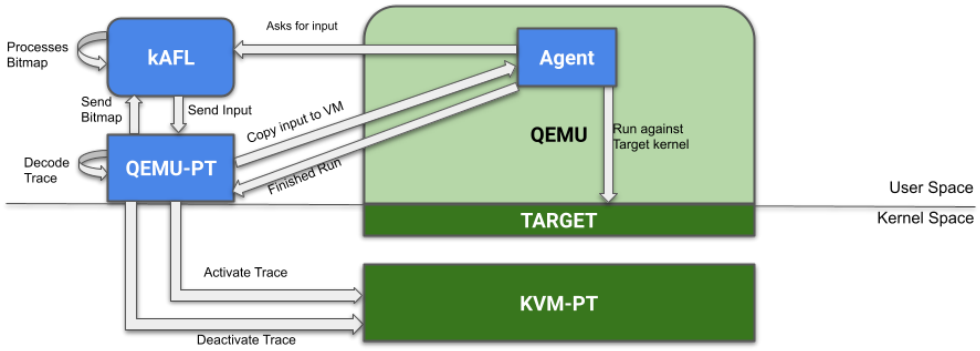

Figure 3.7: Overview of kAFL Structure [43]

3.4.2.3 *Harness Setup*

The user-space agent can execute a binary harness that is used to target fuzzing for a specific component of the kernel. It can submit the address range of the target component to the QEMU-PT decoder for filtering the traces received. It can also submit the PANIC and KASAN handler function of the guest to the QEMU-PT. If this handler function is called during the fuzzing process, code in listing 3.4.2.3 gets executed. This code is used to inform the host of a crash and then halts the vCPU.

```
void kafl_panic(void){
 asm volatile("cli\n\t");
 writel(MMIO_REG_PANIC, kafl_guest_dev.regs+status_reg);
 asm volatile("hlt\n\t");
}
```

The binary harness needs to communicate with the fuzzing logic to get inputs. It does so via pre-defined custom hypercalls which is the interface to communicate with the hypervisor. Following are some important hypercalls that need to be included in the harness:

1. *HYPERCALL_KAFL_GET_PAYLOAD*: This is used by the harness to feed the host the payload buffer address to write the payload.
2. *HYPERCALL_KAFL_NEXT_PAYLOAD*: This is used to request the next payload after the snapshot has been restored. The hypercall blocks the harness until the fuzzing logic copies the payload to the buffer.
3. *HYPERCALL_KAFL_ACQUIRE*: This hypercall starts the tracing of the execution.
4. *HYPERCALL_KAFL_RELEASE*: This hypercall stops the tracing of the execution and reloads the guest snapshot to the point where next payload is to be requested.

The harness can then use this payload to inject it in a fuzzing routine or certain syscalls.

3.4.2.4 *Fuzzing Strategy*

The fuzzing logic is based on the AFL fuzzing logic with added *radamsa* [20] and *redqueen*[5].

One of the differences from AFL in coverage collection is that kAFL uses the basic block addresses rather than a randomly assigned identifier to calculate the hashed tuple. The major difference is that kAFL is a hardware-assisted fuzzer that collects coverage using Intel PT.

3.5 RESEARCH QUESTIONS

Having described in brief the challenges in fuzzing the i915 driver, the possible fuzzing options for the i915 driver and the two prominent coverage-guided kernel fuzzers, we finally turn our focus towards organizing this landscape to enable us to design some solutions to overcome the challenges with fuzzing the i915 driver described in 3.1.

According to the research contributions discussed in Section 1.2, we now define the main research questions for this thesis.

(I) *What are the different options for fuzzing the i915 GFX driver for large-scale driver validation?*

RQ1: ***Which of the two fuzzing tools, Syzkaller or kAFL, is better suited for large-scale driver validation?***

RQ2: ***Which of the different fuzzing solutions explored is better suited for large-scale driver validation?***

(II) *Present a quantitative analysis of Syzkaller and kAFL with the Intel i915 GFX driver as the target.*

RQ3: ***To compensate for the lack of knowledge of the syscall structures in kAFL, can we write a more clever harness and get similar coverage for the i915 graphics driver?***

RQ4: ***Can we use kAFL as an alternative to Syzkaller?***

4

EXPERIMENT SETUP

This Chapter describes in detail the target platform and configurations that we use for our evaluation, the process to set them up and fuzz, the challenges faced and the solutions proposed.

The chapter is structured as follows - In Section 4.1 we present the process of setting up the evaluation platform. We first describe the experimental environment and then the setup of each configuration, GVT-g and GVT-d. In Section 4.2 we present the different fuzzing solutions that are tested with syzkaller and kAFL, the limitations and challenges faced in each setup and fuzzing process. We also present the fuzzing results for each solution. In Section 4.3 we discuss why and how to setup comparable fuzzing solutions for syzkaller and kAFL for the performance evaluation of the fuzzers.

4.1 EVALUATION PLATFORM

Initially, the objective of this thesis was to provide a performance analysis for both the device virtualization techniques, Intel GVT-g and Intel GVT-d with both the fuzzers. Unfortunately, we are only able to compare the fuzzers using Intel GVT-g configuration. The following sections describe the evaluation platform used to carry out experiments. In Section 4.1.0.1 we detail the hardware and the software requirements and in Sections 4.1.0.2 and 4.1.0.3 we present the steps to reproduce the evaluation configurations, GVT-g and GVT-d respectively.

4.1.0.1 *Environment*

To be able to evaluate the impact of our design choices, we setup the fuzzing solutions on Intel-based x86_64 machine, Intel NUC 9 (`NUC9i9QNX`). The machine is equipped with 2.40GHz Intel Xeon E3-1200 v5/E3-1500 CPUs (each with 8 cores), Intel UHD Graphics 630 and 32GB RAM running Ubuntu 20.04. The BIOS version used was the latest QXCFL579.0044.2020.0617.1537. It is mandatory that the machine supports virtualization and I/O Memory Management Unit (IOMMU) groups. To account for possible variations in the results, all the fuzzing solutions are run on this single machine.

We use the Linux kernel `v5.11.16` as the host kernel as it is supported by both the fuzzers. The QEMU version we use is `v5.0.0`. We choose the Linux kernel `v5.10.52` as the target kernel version for i915 driver because it was the latest long term support (LTS) at the time of conducting the experiments. According to the kAFL developers, the number of edges covered reported by kAFL were more accurate than the basic blocks covered at the time of writing this thesis. According

to Syzkaller, it can report the number of edges covered if the Linux kernel is compiled via Clang/LLVM . Therefore, to get comparable coverage metrics, the target kernel for both the fuzzers was compiled with Clang/LLVM version 11. We use the minimal Debian Stretch image as the target OS for Syzkaller and Ubuntu 20.04 for kAFL

4.1.0.2 *Intel GVT-g*

Based on our practical evaluation and assessment in Section 3.3.5 we choose the GVT-g virtualization option. This section will give a brief overview of the technical concepts and the process of setting up this configuration.

We successfully tested the Linux kernel versions `v5.4.0-59`, `v5.8.0-66`, `v5.10.52` and `v5.11.16` (latest stable) and the Qemu versions `v4.2.1` and `v5.0.0` this configuration.

Creating the virtual GPU

The first step is to create a virtual GPU (vGPU) on the host and then assign it to the guest machine. The guest will see this vGPU as the real GPU and is able to access it even if it does not have the particular device driver. By adding the kernel parameters `intel_iommu=on` and `i915.enable_gvt=1` the IOMMU support and Intel GVT-g is enabled on the host respectively. By loading or compiling the kernel modules `kvmgt`, `vfio_mdev` and `vfio_iommu_type1`, vGPU configurations are created. Finally, a universally unique identifier (UUID) is assigned to create a vGPU.

Add vGPU to Guest

The following QEMU arguments are added to attach the vGPU device to the guest machine.

```
-vga none -device vfio-pci,sysfsdev=/sys/bus/pci/devices/0000:00:02.0/<UUID>
```

To check if the addition has been successful, run `lspci` on the guest machine to see if your GPU is visible. Also, the GPU device files `/dev/dri/card0` and `/dev/dri/renderD128` should be available on the guest machine.

4.1.0.3 *Intel GVT-d*

The next option we choose is the GVT-d virtualization option, according to Section 3.3.5 . This section gives a brief look into the steps taken to set up this configuration.

We successfully tested the Linux versions `v4.19.0`, `v5.4.0-59`, `v5.8.0-66`, `v5.10.52` and `v5.11.16` (latest stable) and Qemu versions `v4.2.1` and `v5.0.0` for this configuration.

Host Settings

Devices that are part of IOMMU groups can be passed to a guest machine. But these devices are isolated and therefore, not available to the host anymore. Also, at one time only the devices in one IOMMU group can be isolated at the same time. It is important in our case to isolate only the iGPU. The IOMMU feature is enabled by adding the kernel parameter `intel_iommu=on`. It is important that the host does not load the i915 driver on boot, else it is not possible to isolate the iGPU. To prevent the host from loading the i915 driver on boot, we add the GPU to the `vfio-pci` module which reserves the GPU for the passthrough.

GPU Passthrough

The following QEMU arguments are used to pass the iGPU to the guest machine:

```
-vga none -device vfio-pci,host=00:02.0,x-vga=on
```

Similar to GVT-g, check if the addition has been successful by running `lspci` on the guest machine to see if the GPU is visible. Also, the GPU device files `/dev/dri/card0` and `/dev/dri/renderD128` should be available on the guest machine.

4.2 FUZZING SETUPS

Setting up Syzkaller comprises of 3 phases: (1) Syzkaller Setup; (2) Target Setup in Guest; and (3)Fuzzing. [45]. In the following sections, we discuss these phases in detail for 3 configurations: (1) GVT-g; (2) GVT-d; and (3) Isolated Target. where we describe what challenges/setbacks were faced at every step and what solutions were tried.

4.2.1 *Syzkaller Setup*

The first phase is common to all the configurations and involved preparing the following elements required for fuzzing with Syzkaller:

1. Syzkaller requires that the kernel should be compiled with static instrumentation. For the compilation of the target kernel, it was essential to have a recent version of a C compiler with coverage support.
2. The kernel also needs coverage support. For this, the Linux kernel needed to be compiled with coverage (**kcov**) additions.
3. Syzkaller requires a virtualization tools like the guest kernel image, KVM, QEMU, etc.
4. The Go toolchain is required to be setup to build Syzkaller.

Syzkaller supports different kernel architectures and guest machine types. For this thesis, two of these setups seemed interesting. The first one that we chose is the **Setup: Ubuntu host, QEMU vm, x86-64 kernel** because we had tested the virtualization configurations using QEMU and it is also the virtualization software used by kAFL and hence, it would provide us a good basis of comparison. The second setup chosen is the **Setup: Linux isolated host** as it could give us the opportunity to fuzz the i915 driver on bare metal without the need of virtualization and to compare the bare-metal approach and our chosen approach of device virtualization.

For the first step we pick the default gcc version 9.3.0 that comes with Ubuntu 20.04. We compile the Linux kernel using default compiler and with additions to the configuration file like coverage support and different debugging options. The compilation process generates the kernel bZImage.

For the guest Linux image, we pick the minimal Debian Stretch Linux image provided by Syzkaller. For the Isolated Host setup we use Ubuntu 20.04 on the target machine.

4.2.1.1 *Fuzzing Setup*

To fuzz the i915 driver, the next step is to setup the device virtualization configurations Intel GVT-g and Intel GVT-d, as discussed in Section 3.3.4. We discuss below how we incorporate these GPU virtualization configurations to fuzz the i915 driver with Syzkaller. We also provide details of the Isolated Host setup.

For the configuration setups, the target kernel bZImage and the Debian Linux image are loaded into a QEMU instance. For the Isolated setup the target kernel bZImage and Ubuntu image are loaded into the target machine.

In the following sections, we present the modified Syzkaller configurations, challenges/setbacks that we faced and the solutions applied for all three approaches. We present the findings on running Syzkaller with each configuration. It is to be noted that minimizing the inputs of the Syzkaller bug report, reproducing the bugs and finding vulnerability or exploitability of the bugs is out of scope of this thesis. The findings were simply reported to the relevant development teams for further analysis.

Intel GVT-g

Guest Setup

The following Syzkaller configuration file is used to enable GVT-g configuration in the QEMU guest for Syzkaller.

```
{
        "target": "linux/amd64",
        "http": "127.0.0.1:56741",
        "workdir": "~/gopath/src/github.com/google/syzkaller/workdir",
        "kernel_obj": "~/linux/",
        "image": "~/image/stretch.img",
        "sshkey": "~/image/stretch.id_rsa",
        "syzkaller": "~/gopath/src/github.com/google/syzkaller",
        "procs": 8,
        "reproduce": false,
        "type": "qemu",
        "enable_syscalls":[
                "openat$i915",
                "ioctl$DRM_IOCTL_I915_*"
        ],
        "vm": {
                "count": 1,
                "qemu_args": "-enable-kvm -cpu host,migratable=off -vga none
                    -device vfio-pci,sysfsdev=/sys/bus/pci/devices
                   /0000:00:02.0/<UUID> -append i915.reset=1,i915.verbose_
                   state_checks=1,drm.debug=0x01,drm.debug=0x02",
                "kernel": "/~/linux/arch/x86/boot/bzImage",
                "cpu": 4,
                "mem": 2048
        }


}
```

Limitations of the Setup:

Limitation 1: **Fuzzing solution with one guest.** Even though Syzkaller allows parallelization with multiple guests for fuzzing, it currently only supports configuring the same QEMU configuration for all the guests. But we would have had to attach a different vGPU for every guest that needed to be started as we can only assign one vGPU to one VM.

Due to lack of time we were only able to explore Syzkaller fuzzing with one VM due to this challenge. The probable solution would be to use the *syz-hub*. Under this we can start multiple *syz-manager* instances, each provided with one vGPU for the QEMU instance. Then *syz-hub* would connect all these instances in such a way that they can exchange their corpus information, test case information and reproducers. This is not tested during this thesis.

Limitation 2: **Fuzzing solution to fuzz only one device file at a time.** We reuse the description file *dev_i915.txt* that was already created and added to Syzkaller. It

defines all the ioctls for the i915 driver. But as we reuse the file, the target device file is declared as */dev/i915*. But the actual i915 device file names are */dev/cardo* and */dev/renderD128*.

The *create-image.sh* is a helper script to create a minimal Debian Stretch Linux image. In this script, we add a symlink for */dev/cardo* or */dev/renderD128* to */dev/i915*. This means we can fuzz only one device file out of the two at a time.

Fuzzing

Using the configuration file in 4.2.1.1, we start the fuzzing process to target the i915 driver and come across the following challenges:

Challenge 1: After a few hours of the run, fuzzing the i915 driver causes certain denial-of-service bugs that hang the GPU and the host system. This leads to the halting of the fuzzing process as the *sys-manager* is running on the host and rebooting is the only way to recover.

Solution 1: We investigate the bugs and find that they are potential denial-of-service bugs in GVT-g. This prompts to explore the possibility of doing the following after every fuzz input: (1) restarting the virtualization platform; (2) resetting the state of the GPU or vGPU; (3) and reloading the host i915 driver. Restarting of the virtualization platform is ruled out as it would consume a lot of time. Therefore, we first try to reset the vGPU by providing the `i915.reset=1` to QEMU args in the Syzkaller config file. This does not seem to stop the host hangs. Next, we investigate the possibility of resetting the GPU and the host i915 driver.

GPU Reset

The following design choices are made to prepare a script to reset the GPU and the host i915 driver:

- The kernel does not allow to unload the i915 driver even when the module dependent on it (kvmgt) has been removed and would show the driver *in use*. We notice that i915 driver is attached to the GPU and still running the X server. Therefore, we stop the X server and detach the GPU before unloading the i915 driver. The reloading of the i915 driver registers the GPU automatically.

- Even after unbinding the driver from the GPU, the kernel shows that the i915 driver is in use. We notice that the unbinding of the driver takes a significant amount of time. Therefore, we add a wait before the removal of the i915 driver. Different times are tested and the minimum time it needed to unbind is chosen as the wait time.

- If all the commands are run instantaneously, the script fails to reload the driver or reset the GPU. We observe that since every step is dependent on the successful completion of the previous step, the script fails if the previous

step is not yet finished. We add a minimum wait after each step to ensure that the script runs smoothly.

Following is the script that is used:

```
#!/bin/bash

echo 1 > /sys/bus/pci/devices/0000:00:02.0/<UUID>/remove                //
    remove the vGPU
sleep 2
rmmod kvmgt
systemctl stop gdm
sleep 2
echo 1 > /sys/kernel/debug/dri/0/i915_wedged
                                    //reset the GPU state
sleep 2
sh -c "echo -n 0000:00:02.0 > /sys/bus/pci/drivers/i915/unbind"
            //unbind the driver from GPU
sleep 30
rmmod i915
sleep 10
modprobe i915
sleep 2
systemctl start gdm
sleep 2
modprobe kvmgt
sleep 2
sh -c "echo <UUID> > /sys/bus/pci/devices/0000:00:02.0/mdev_supported_types/
    i915-GVTg_V5_4/create"              //recreate the vGPU
sleep 4
```

Next, we decide when Syzkaller should execute this script. We decide that the best time to execute the script would be before every time the Qemu instance is started. By default, a Qemu instance is restarted every one hour assuming that there have been no crashes. Syzkaller does not provide a documented way to hook a script before a Qemu instance is launched. Therefore, we modify the Syzkaller code to call our script before it launches a Qemu instance.

We are able to execute this script successfully in the newer kernel versions *5.11.x* and *5.12.x*. Therefore, we decide to use *5.11.16* as the host kernel as it is the same as being used by kAFL on the host.

Challenge 3: As the host X server is being stopped before unloading the i915 driver, there are no graphics on the screen and it never recovers back.

Solution 3: It is observed that Syzkaller cannot be a child process of a graphical terminal. Therefore, we either need to run it from text-mode or from a remote-

shell. We decide to run it from a **remote-shell** and are able to get the web interface to view statistics using port forwarding.

Challenge 4: We run into permission issues when we try to run Syzkaller with the GPU resetting scripts as Syzkaller is running as a user process. QEMU in Syzkaller is also unable to access the vGPU device due to permission issues.

Solution 4: We run Syzkaller fuzzing campaigns with root access so that Syzkaller can execute the GPU resetting script that contains privileged commands and QEMU can attach the vGPU device to the guest.

After resolving the problems, we are able to successfully run 2 fuzzing campaigns of continuous 48 hours using the same configuration file.

Intel GVT-d

Guest Setup

The following Syzkaller configuration file is used to enable GVT-d configuration in the QEMU guest for Syzkaller. The QEMU args *device* allows for the direct passthrough of the GPU to the QEMU guest as shown in in the Listing 4.2.1.1.

```
{
        "target": "linux/amd64",
        "http": "127.0.0.1:56741",
        "workdir": "~/gopath/src/github.com/google/syzkaller/workdir",
        "kernel_obj": "~/linux/",
        "image": "~/image/stretch.img",
        "sshkey": "~/image/stretch.id_rsa",
        "syzkaller": "~/gopath/src/github.com/google/syzkaller",
        "procs": 8,
        "reproduce": false,
        "type": "qemu",
        "enable_syscalls":[
                "openat$i915",
                "ioctl$DRM_IOCTL_I915_*"
        ],
        "vm": {
                "count": 1,
                "qemu_args": "-enable-kvm -cpu host,migratable=off -vga none
                     -device vfio-pci,host=00:02.0,x-vga=on -append i915.
                    reset=1,i915.verbose_state_checks=1,drm.debug=0x01,drm.
                    debug=0x02,log_buf_len=1M",
                "kernel": "~/linux/arch/x86/boot/bzImage",
                "cpu": 4,
                "mem": 2048
        }}
```

Setbacks

Setback 1: **Fuzzing via remote-shell**. As the entire GPU is passed through to the guest, the graphics on the host are disabled. The challenge is to find the best way to run Syzkaller on the host.

As the graphics are disabled, the text-mode is not supported. Therefore, our solution is to run the fuzzing campaign via remote-shell and make the web-interface of Syzkaller available via port-forwarding.

Fuzzing

Using the configuration file in 4.2.1.1, the fuzzing of i915 driver is started and the following challenges are encountered:

Challenge 1: As the guest now has direct access of the real GPU, we are fuzzing the native graphics driver. Therefore, chances of causing a hang on the GPU and host are high, similar to Intel GVT-g configuration. We again observe the hangs during the fuzzing campaign using GVT-d.

Solution 1: We reuse the solution of GPU resetting discussed in Section 4.2.1.1. We are able to stabilize the fuzzing campaign using this solution. We observe similar challenges regarding permission issues for Syzkaller to execute the GPU resetting script and for QEMU to attach the vGPU to the device. Similar to the case before, we run the fuzzing campaigns with root to provide necessary privileges to Syzkaller and QEMU.

We are able to test this fuzzing setup with 2 fuzzing campaigns of continuous of 48 hours using the same configuration file. This file covers most of the ioctl interface of i915 driver.

Isolated Host

Guest Setup

For this setup, we require two machines to run Syzkaller, the source machine `X` and the isolated target machine `Y`. Machine `X` is equipped with Intel(R) Core(TM) i5-8250U CPU at 1.60GHz. The machine described in Section 4.1.0.1 is used as machine `Y`. The `syz-manager` running on machine `X` uses ssh to launch a fuzzing session in machine `Y` and monitor the process. The following Syzkaller configuration file is used on machine X:

```
{
        "target": "linux/amd64",
        "http": "127.0.0.1:56741",
        "rpc": "127.0.0.1:0",
        "workdir": "~/gopath/src/github.com/google/syzkaller/workdir",
        "kernel_obj": "~/linux",
        "syzkaller": "~/gopath/src/github.com/google/syzkaller",
        "sandbox": "setuid",
        "type": "isolated",
        "enable_syscalls":[
                "openat$i915",
                "ioctl$DRM_IOCTL_I915_*"
        ],
        "vm": {
                "targets" : [ "192.168.0.110" ],
                "pstore": true,
                "target_dir" : "~/isolated",
                "target_reboot" : false
        }
}
```

Fuzzing

We are able to start the fuzzing of i915 driver on bare-metal but quickly run into the following challenge:

Challenge 1: The fuzzing process is able to crash the target kernel and the entire isolated system which leads to the disconnection of SSH between the `syz-manager` on machine X and `syz-fuzzer` on machine Y. Once the manager loses connection to the machine it is unable to reboot the machine as the system had crashed.

Tested Solution 1: We test the **kdump** feature of Linux that captures the kernel memory image at the time of a crash and then boots another Linux kernel to preserve the system consistency. But once the kernel crashes the Syzkaller binaries stop execution and on new kernel boot everything is reset. The `syz-manager` is able to reinstate the SSH connection but is left in a waiting state as there is no `syz-fuzzer` binary to communicate.

For this setup to work we need to find a way to not loose connection with the isolated machine in case of a crash, store the crash information and continue fuzzing. We are not able to research further into this setup during this thesis due to time constraints.

4.2.1.2 *Fuzzing Results*

In this Section we present and discuss the findings from fuzzing with Intel GVT-g and GVT-d configurations. Table 4.1 and Table 4.2 present the findings that are

discovered during the fuzzing of all i915 ioctls with Syzkaller using the superuser for GVT-g and GVT-d respectively.

Table 4.1: Findings with GVT-g Configuration

Type of Bug	Total Found
Task Hung	2
Memory Leak	2
Softlockup	1

Table 4.2: Findings with GVT-d Configuration

Type of Bug	Total Found
Kernel Bug	24
General Protection Fault	23
Memory Leak	11
WARNING	6
Task Hung	2
Lost connection to test machine	1
Stack Segment Fault	1
KASAN	1

The findings are reported to the relevant i915 development teams and according to their feedback these findings are deemed irrelevant as they are found with the superuser with higher privileges. According to the feedback from the i915 development team, it is more relevant if the fuzzing is performed with a user that has restricted access to the i915 graphics driver interface. This means that we need to perform fuzzing with a user that does not have the access to all the privileged ioctls in the driver interface.

We also investigate the bugs on our own in Table 4.1 and Table 4.2 by using the Syzkaller tool, `syz-repro`, to automatically minimize the crashing input from the bug reports and reproduce the bugs. We try to reproduce several bugs but are unsuccessful in reproducing them after running the tool multiple times. Each run fails to reproduce after running for 3 hours and times out. We try to reproduce each bug for 3 days and are unsuccessful. This probably means that the bugs in Table 4.1 and Table 4.2 are false positives.

Fuzzing with Lesser User Privileges

Based on the feedback received from the i915 development teams, we decide to modify the Syzkaller setup to fuzz with a lesser privileged user. Such a

setup is not documented by Syzkaller. We find that Syzkaller provides a *sandbox mode* under which fuzzing can be performed with a *nobody* user which is an unprivileged user.

We now perform fuzzing for all i915 ioctls using Syzkaller with the *nobody* user. In this case the fuzz inputs are sent to the i915 driver with the lower privileged user. We fuzz both the configurations of Intel GVT-g and Intel GVT-d for 2 trials. Each trial is of continuous 48 hours with the configuration file modified to fuzz with a lower privileged user. We observe that fuzzing with the *nobody* user covers a lower number of ioctls than with the root user. No bugs are reported during these trials.

4.2.2 *kAFL Setup*

As kAFL is a proof-of-concept fuzzer and even though there is good documentation for user-space fuzzing, there is sparse documentation to setup the fuzzer for Linux kernel fuzzing. Due to this we run into a multitude of undocumented behaviors. Also, since it is a prototype there were constant updates to the kAFL versions during this thesis. We try setting up kAFL with the open-source version first but are unsuccessful due to certain bugs present for the Linux kernel fuzzing. We then use an unpublished version and with the help of the kAFL developer (Steffen Schulz) we setup a working kAFL version.

In this section, we discuss the successful setup. Similar to Syzkaller, the kAFL setup involves 3 phases: (1) kAFL setup; (2) Guest Setup; and (3)Fuzzing.

In the first phase, we prepare the components required for kAFL:

1. *KVM-PT*: The KVM-PT module is to be installed on the host kernel.
2. *QEMU-PT*: The Intel-PT decoder component in QEMU-PT requires the following libraries:
    - *libXDC*: It is an Intel-PT decoding library that is built for binary-only fuzzing purposes. This library assists in speeding up the process of repeatedly decoding similar traces against the same binary with the help of a fast runtime cache.
    - *capstone v4*: The libXDX library depends on the multi-architecture disassembly framework, capstone v4. This version is not included in many distributions and therefore, needs to be installed.
3. *Target Kernel*: We compile the target Linux kernel using the default compiler, gcc v9.3.0 for Ubuntu 20.04 and add the same debugging options as the Syzkaller target kernel.

The working version of kAFL currently supports host Linux kernel v4.19.0 in the open-source version and v5.11.16 in the unpublished version.

4.2.2.1 *Fuzzing Setup*

The second task involves setting up the QEMU guest with virtualization configurations to fuzz the i915 driver and creating a simple harness on the guest for fuzzing. In Section 4.2.2.1 we first describe setting up the guest with the 2 configurations of Intel GVT-g and Intel GVT-d that are possible with kAFL. The isolated host setup is currently not supported by kAFL. In Sections 4.2.2.1 and 4.2.2.1 we discuss the fuzzing challenges and results faced with each of the configurations respectively.

Guest Setup

We first install Ubuntu 20.04 as the guest OS, same as the host OS. On the guest kernel, Kernel Address Space Layout Randomization (KASLR) and spectre mitigations are turned off. KASLR can prevent the tracing of a particular device driver module and spectre mitigations can lower the performance significantly.

We modify the kAFL source code to enable GVT-d and GVT-g configuration for QEMU. We provide a commandline argument for the user to enable either of the configuration and kAFL can then use the corresponding QEMU args that were defined.

This setup is successful on both supported host kernel versions for *GVT-d* configuration. The GVT-g configuration is successful on kernel `v5.11.16`.

Harness

To target the fuzzing specifically towards i915 driver we build a simple kAFL harness and agent. We follow the stated approach:

- Gather knowledge on the structure of inputs required by an IOCTL and implement a C program that calls this IOCTL with a structured input with constant values.
- Run this against the host and trace the execution. Ensure that the required functions are being reached.
- Transfer the logic for kAFL harness with necessary hypercalls.
- Create an agent script that extracts the address range of the i915 driver and passes it to the harness where the necessary hypercalls help to register this range to filter the trace received via Intel PT.

Using this approach, we implement a simple harness that can open the device file before the snapshot is taken and call the i915 Query ioctl with a structured input.

Intel GVT-d Fuzzing

We are unable to start fuzzing the i915 driver using GVT-d configuration as we run into kAFL bugs. We do not explore this option further during this thesis.

Intel GVT-g Fuzzing

We are able to successfully start fuzzing the i915 driver with the simple harness described in 4.2.2.1 and get some coverage on the i915 driver. There are no interesting findings during this fuzzing campaign.

Limitations of the setup

Limitation 1: **Fuzzing solution with one guest.** Similar to Syzkaller, kAFL allows parallelization with multiple guests for fuzzing, but it also currently only supports configuring the same QEMU configuration for all the guests.

4.3 COMPARABLE EXPERIMENTAL SETUPS

Syzkaller is a stateful fuzzer that incorporates significant amount of technical knowledge of the target by describing the valid target's syscall interface. In theory, this approach can definitely lead to more code coverage even without any feedback than kAFL that has no idea how to generate a valid syscall and has to learn it overtime while sending unstructured random data. This prevents kAFL from triggering interesting paths in the earlier stages of fuzzing and a lot of time is expected to be used on learning the structure of a valid syscall.

One of the goals of this thesis is to provide a performance comparison between Syzkaller and kAFL in fuzzing the i915 driver. The above mentioned complication prevents us from providing a fair performance comparison between the two fuzzers. To tackle this, we leverage the freedom of the kAFL harness by creating a clever harness that has the knowledge of the valid syscall structure.

We design our kAFL harness to implement the syscall fuzzing logic similar to Syzkaller. The harness is modeled on the basis of the i915 description file of Syzkaller. This aims to eliminate the advantage of Syzkaller's prior knowledge of the valid syscall structure and bring the fuzzers on the same level.

In the upcoming sections, we first describe how a Syzkaller program is generated for the i915 device driver. Then we describe the different versions of harnesses implemented. We also present an overview of the similarities and differences between the Syzkaller and new kAFL setup.

4.3.1 *Syzkaller i915 programs*

To create a syscall logic similar to Syzkaller for the i915 driver, we analyze the i915 description file of Syzkaller and the inputs that are generated in a 1 hour

time period while fuzzing the i915 driver with the same description file. This gives us an idea as to what kind of inputs we want the kAFL harness to generate.

After anaylzing the Syzkaller inputs and the description file, we finalize the mechanism that would be required for kAFL harness and following are the features that need to implemented:

1. The ioctl definitions, data structures and flags need to be defined in the harness similar to the ones defined in the description file. This will provide the valid ioctl structure to kAFL.

2. According to Syzkaller, the `openat` ioctl needs to be enabled by the user as it provides the necessary resource for the following syscalls. We also observe that in the description file openat is defined with 3 constant arguments and one variable. Therefore, there should be atleast one `openat` ioctl with the similar arguments.

3. The program can start with an `openat` ioctl or with some other enabled ioctl. In such a case:
   - The other ioctl will get a special value `0xffffffffffffffff (-1)` for "'no file descriptor"'
   - With high probability the program should start with a valid openat as described above.

4. The harness should keep a list of all the file descriptors (fd) that have been returned by every openat in a program. The syscalls will be provided a fd which exists in this list at that point in time.

Following are two sample fuzz inputs that encompass the above mentioned features:

Listing 4.1: program start with openat ioctl

```
openat$i915(0xffffffffffffff9c, 0x20000300, 0x0, 0x0)
openat$i915=0x3 errno=14 cover=1345
openat$i915(0xffffffffffffff9c, 0x20002d40, 0x0, 0x0)
openat$i915=0x4 errno=14 cover=1367
ioctl$DRM_IOCTL_I915_GEM_EXECBUFFER2(0x4, 0x40406469, 0x20002e00)
ioctl$DRM_IOCTL_I915_GEM_EXECBUFFER2=0xffffffffffffffff errno=25 cover=130
ioctl$DRM_IOCTL_I915_GEM_EXECBUFFER2(0x4, 0x40406469, 0x20000240)
ioctl$DRM_IOCTL_I915_GEM_EXECBUFFER2=0xffffffffffffffff errno=25 cover=130
openat$i915(0xffffffffffffff9c, 0x0, 0x0, 0x0)
openat$i915=0xffffffffffffffff errno=14 cover=617
ioctl$DRM_IOCTL_I915_GEM_EXECBUFFER2(0x3, 0x4611, 0x0)
ioctl$DRM_IOCTL_I915_GEM_EXECBUFFER2=0x0 errno=14 cover=14855
```

Listing 4.2: program start with another enabled ioctl

```
ioctl$DRM_IOCTL_I915_GEM_GET_CACHING(0xffffffffffffffff, 0xc0086470, 0
    x20000000)
ioctl$DRM_IOCTL_I915_GEM_GET_CACHING=0xffffffffffffffff errno=9 cover=55
openat$i915(0xffffffffffffff9c, 0x20000040, 0x1, 0x0)
openat$i915=0x3 errno=14 cover=1362
ioctl$DRM_IOCTL_I915_GEM_CONTEXT_CREATE(0x3, 0xc008646d, 0x20000080)
ioctl$DRM_IOCTL_I915_GEM_CONTEXT_CREATE=0xffffffffffffffff errno=25 cover
    =130
openat$i915(0xffffffffffffff9c, 0x200000c0, 0x80242, 0x0)
openat$i915=0x4 errno=14 cover=1579
ioctl$DRM_IOCTL_I915_PERF_ADD_CONFIG(0xffffffffffffffff, 0x40486477, 0
    x20000100)
ioctl$DRM_IOCTL_I915_PERF_ADD_CONFIG=0xffffffffffffffff errno=9 cover=55
ioctl$DRM_IOCTL_I915_GEM_CONTEXT_GETPARAM(0x3, 0xc0186474, 0x20000180)
ioctl$DRM_IOCTL_I915_GEM_CONTEXT_GETPARAM=0xffffffffffffffff errno=25 cover
    =130
ioctl$DRM_IOCTL_I915_PERF_REMOVE_CONFIG(0x3, 0x40086478, 0x200001c0)
ioctl$DRM_IOCTL_I915_PERF_REMOVE_CONFIG=0xffffffffffffffff errno=25 cover
    =130
openat$i915(0xffffffffffffff9c, 0x20000200, 0x60040, 0x0)
openat$i915=0xffffffffffffffff errno=40 cover=1235
ioctl$DRM_IOCTL_I915_PERF_ADD_CONFIG(0xffffffffffffffff, 0x40486477, 0
    x20000240)
ioctl$DRM_IOCTL_I915_PERF_ADD_CONFIG=0xffffffffffffffff errno=9 cover=55
ioctl$DRM_IOCTL_I915_GEM_THROTTLE(0x4, 0x6458, 0x0)
ioctl$DRM_IOCTL_I915_GEM_THROTTLE=0xffffffffffffffff errno=25 cover=130
openat$i915(0xffffffffffffff9c, 0x20000380, 0x80, 0x0)
openat$i915=0x5 errno=14 cover=1335
```

4.3.2 *kAFL Harness Implementation*

During this thesis we are able to implement 3 ioctls in the harness due to time constraints. The following ioctls are implemented:

1. **OPENAT**: This syscall is important as it is required to open the device file and target the i915 driver. As stated earlier, Syzkaller also requires this ioctl to be enabled and since we wanted to make the logic similar to Syzkaller, this ioctl is included.

2. **QUERY**: This ioctl is implemented first as it was an easier target to understand the working of the i915 driver. It is used to query the following information:

   - Topology Info
   - GPU Engine Info
   - Perf Config

3. **EXECBUFFER2**: This ioctl is chosen after observing the findings via fuzzing the i915 driver with Syzkaller. Also, it is a complex ioctl that manages the user command execution on the physical GPU. This ioctl performs the following tasks to handle user command execution:

   - *Validates* the user-space input like pointers handles and flags.
   - *Reserves* GPU address space for buffer objects that contain the user commands
   - *Relocates* the buffer objects in the address space, if required.
   - *Serializes* the user request according to its dependencies
   - *Constructs* a request to execute a batchbuffer which is a chain of buffer objects.
   - *Submits* the batchbuffer for the GPU to execute. This means all the user commands are submitted to the GPU buffer which the GPU is reading.

   More details into the working of the ioctl are out of scope of this thesis. Please refer the source code of the i915 driver for more details [30].

4.3.2.1 *First implementation*

Now, we describe the first implementation of the kAFL harness. This implementation includes **all** the features that are discussed in Section 4.3.2.

Algorithm 2 First Harness Implementation

```
1:  function MAIN
2:      kAFL Handshake using KAFL_ACQUIRE and KAFL_RELEASE
3:      while true do
4:          payload ← KAFL_GET_PAYLOAD()
5:          KAFL_ACQUIRE()
6:          RANDOMIZE_SYSCALLS(fd, payload)
7:          KAFL_RELEASE()

8:  function RANDOMIZE_SYSCALLS(payload)
9:      Program: P
10:     filedescriptors ← 0
11:     Choose ioctls to be included
12:     repetitions ← Get number for each included ioctl from the payload
13:     if P begins with openat ioctl then
14:         P ← add openat
15:         P ← Add next ioctls according to repetitions
16:         Choose order of run of ioctls after openat
17:     else
18:         P ← Add query or execbuffer2 according to repetitions
19:         P ← add openat
20:         P ← Add next ioctls according to repetitions
21:         Choose order of run of ioctls after openat
22:     Execute P
```

We first create a *test harness* that calls each of the 3 ioctls with custom values, execute it on the host and ensure that we are able to reach the required ioctl functions. We then transfer the logic of these test harnesses to the kAFL harness.

After the fuzzer submits a payload buffer we are able to call our program with a sequence of ioctls. We use the byte and bit values of the payload buffer to make decisions required to generate an input program. We will refer to our decision making mechanism as *payload mechanism* from now onwards.

Following is the incremental approach we take to build the harness that is presented in Algorithm 2:

- **Step 1:** After the payload buffer is generated, we pass the payload directly to the data structures of the ioctl and then we call the ioctl. This is initially done in 3 iterations, one for each of the 3 ioctls to ensure we are getting the right coverage.
- **Step 2:** A service *randomize_syscalls* is created that chooses the ioctls as part of a program and calls them in an order. Following are the decisions taken using the payload mechanism:
  - Whether the ioctl has to be added to the program or not.

  - The number of times the chosen ioctl is to be added in a program.
  - The order in which the ioctls are to be called.

  With this service we are able to create programs with a sequence of ioctls and differing orders of execution.

- **Step 3:** In this step, we implement the service that decides whether the program starts off with an openat ioctl or another enabled ioctl. We give 90% probability to openat as we observe high probability of this case in Syzkaller input generation.

- **Step 4:** The last feature addition is to keep track of all the unique file descriptors that are returned when multiple openat ioctls are executed in a program.

Results and Challenges: We are able to start the fuzzing campaign with the GVT-g configuration but we run into a potential memory corruption bug on the host. This bug is found to be an unresolved known bug. We responsibly disclose it to the Intel Security Team. Without resolving this bug, this harness cannot be tested further.

On inspection, it is a potential memory corruption bug in GVT-g that interrupts the fuzzing process and is generated on enabling the `openat` ioctl in the harness. The bug is not encountered immediately when the fuzzing process is started instead it builds up over a few minutes. We observe a `gvt:guest page write error` in the host syslog for every `openat` ioctl executed before it leads to a `gvt: vgpu 1: fail` error on the host. Unfortunately, the Linux team is not able to fix the bug during the ongoing of the thesis so we decide to modify the harness to work around the bug.

4.3.2.2 *Second implementation*

In this section we present our second try at implementing the kAFL harness. As the bug mentioned in Section 4.3.2.1 is being triggered due to openat ioctls, we first try to reduce the number of openat in the fuzz inputs. We try this approach in two different ways:

Approach 1: We reduce the number of openat to maximum of 2 times in one input program. The other parts of the harness remain unmodified.

Results and Challenges: We run into the exact same bug experienced with the first implementation as discussed in Section 4.3.2.1.

Approach 2: We then try to reduce the number of openat by calling the first openat with all valid arguments before the snapshot is taken. This means when a snapshot is restored, the i915 device file already has one valid file descriptor. The

second openat ioctl is called after the snapshot is restored. The other parts of the harness remain unmodified.

Results and Challenges: We run multiple fuzzing campaigns and every time we run into same bug as discussed in Section 4.3.2.1 after an average run of 6 hours. This means that we are able to reduce the load on the Intel GVT-g. Using this approach means that the harness cannot generate inputs that can start with query or execbuffer2. Also, the openat ioctl at the start of each program will have all valid arguments.

4.3.2.3 *Third implementation*

Unfortunately, the above result means we have to try and remove the second openat ioctl after the snapshot is restored. This means we now only have one valid openat ioctl before the snapshot is taken. This means we will only have a valid file descriptor to fuzz the other ioctls. To overcome this we make a constant addition to the list of file descriptors with `(-1)` so that with low probability, the other ioctls can also be fuzzed with this file descriptor. The other parts of the harness remain unmodified. A simple algorithm of the final harness is presented in Algorithm 3

Results and Challenges: We are able to run stable fuzzing campaigns with this harness and do not run into any bugs. We use this setup for the quantitative evaluation of the fuzzers.

Algorithm 3 Final Harness Implementation

```
1:  function MAIN
2:      kAFL Handshake using KAFL_ACQUIRE and KAFL_RELEASE
3:      while true do
4:          payload ← KAFL_GET_PAYLOAD()
5:          fd ← OPENAT()
6:          KAFL_ACQUIRE()
7:          RANDOMIZE_SYSCALLS(fd, payload)
8:          KAFL_RELEASE()

9:  function RANDOMIZE_SYSCALLS(fd, payload)
10:     Program: P
11:     filedescriptors ← [fd,-1]
12:     Choose ioctls to be included
13:     repetitions ← Get number for each included ioctl from the payload
14:     Add ioctls according to repetitions
15:     Choose order of run of ioctls after openat
16:     Execute P
```

4.3.3 *Syzkaller Harness*

In this Section, we present the configuration file that is used to restrict the Syzkaller fuzzing for the 3 ioctls that we implemented in the kAFL harness. The file is listed in Listing .

```
{
        "target": "linux/amd64",
        "http": "127.0.0.1:56741",
        "workdir": "~/gopath/src/github.com/google/syzkaller/
            workdir_final_second",
        "kernel_obj": "~/kernel_sources/linux-5.10.52/",
        "image": "~/image/stretch.img",
        "sshkey": "~/image/stretch.id_rsa",
        "syzkaller": "~/gopath/src/github.com/google/syzkaller",
        "procs": 8,
        "type": "qemu",
        "reproduce": false,
        "enable_syscalls":[
                "openat$i915",
                "ioctl$DRM_IOCTL_I915_GEM_EXECBUFFER2",
                "ioctl\$DRM_IOCTL_I915_QUERY"
        ],
        "vm": {
                "count": 1,
                "qemu_args": "-enable-kvm -cpu host,migratable=off -vga none
                    -device vfio-pci,sysfsdev=/sys/bus/pci/devices
                    /0000:00:02.0/<UUID> -append i915.reset=1,i915.
                    verbose_state_checks=1,drm.debug=0x01,drm.debug=0x02",
                "kernel": "~/kernel_sources/linux-5.10.52/arch/x86/boot/
                    bzImage",
                "cpu": 4,
                "mem": 2048
        },
        "cover_filter": {
                "files": ["^drivers/gpu/drm/i915"]
        }

}
```

5

EVALUATION

With different potential choices of fuzzing solutions available, it is often difficult for potential users to identify which platform and configuration is best suited according to their requirements. Therefore, based on the comparable setup as described in Section 4.3 and our findings while setting up the fuzzing solution, we present a qualitative and a quantitative (performance) analysis of the fuzzers and the fuzzing options in this Chapter.

This chapter is organized as follows - We start with the qualitative analysis presented in Section 5.1. Then, in Section 5.2 we present the performance analysis of the comparable setups that help us evaluate the efficiency and effectiveness of both the fuzzers with i915 driver as the target.

5.1 QUALITATIVE ANALYSIS

We would like to recall the first two research questions listed in Section 5.3 and aim to answer them as follows:

RQ1: ***Which of the two fuzzing tools, Syzkaller or kAFL, is better suited for large-scale driver validation?***

In this Section, we aim to answer this question by performing a qualitative analysis of the *usability* and *versatility* of the two fuzzers.

RQ2: ***Which of the different fuzzing solutions explored is better suited for large-scale driver validation?***

In this Section, we aim to answer this question by performing a qualitative analysis of the *crash-tolerance* and *cost-effectiveness* and the *effort required* for the fuzzing solutions.

Table 5.1 presents the summary of the qualitative analysis that is performed during this thesis. The analysis is covered into two parts: (1) Qualitative Analysis of the Fuzzing Tools and (2) Qualitative Analysis of the Fuzzing Solutions.

In evaluating the fuzzers for *usability*, we find that kAFL does not require the user to compile the target kernel whereas Syzkaller requires it and it can be time consuming. But the technical knowledge required to implement the kAFL harness is high whereas for Syzkaller it is moderate. The user-interface of Syzkaller for coverage achieved is user-friendly to use whereas for kAFL the user needs to be skilled at binary analysis to use the provided Ghidra plugin to view the coverage achieved. Under *versatility*, we find that kAFL can provide an

OS-agnostic solution which is a desirable feature for large-scale driver validation. On the other hand, Syzkaller supports a bare-metal fuzzing option unlike kAFL. In our evaluation of the fuzzing solutions, for both the fuzzers we are able to set up and test the GVT-g configuration. This is a desirable feature with maximum cost-effectiveness out of the other solutions.

		Syzkaller			**kAFL**	
		GVT-g	GVT-d	Isolated	GVT-g	GVT-d
	Fuzzing Tools					
Usability	Target Kernel	Linux with compile-time instrumentation			Off-the-Shelf kernels supported	
	Technical Knowledge	medium			high	
	User-Interface	Web-Interface, annotated source			Real-time GUI, Ghidra Plugin	
Versatility	OS-Agnostic	No			yes	
	Hardware Access	Bare-metal or virtualized target system			Virtualized target systems	
	Fuzzing Solution					
	Crash-Tolerant	yes	yes	no	yes*	Not tested
	Cost-Effective	maximum	moderate	moderate	maximum	moderate
	Technical Knowledge	moderate	moderate	minimum	moderate	moderate

Figure 5.1: Summary of Qualitative Anaylsis

In the upcoming Sections, we present a detailed analysis. The analysis is covered in two sections, Section 5.1.1 discusses the qualitative analysis of the fuzzing tools and Section 5.1.2 discusses the qualitative analysis of the fuzzing options.

5.1.1 *Qualitative Analysis of the Fuzzing Tools*

To effectively analyze the two fuzzing tools, Syzkaller and kAFL, for deployment in large-scale i915 driver validation, we identified two metrics that we would like to evaluate them for: *usability* and *versatility*. For each metric we raise certain research questions that we aim to answer from the analysis as follows:

Usability

RQ1.1: How much effort is required to set up the fuzzer to fuzz the i915 graphics driver?

RQ1.2: How well does the fuzzer present the results for human consumption?

Versatility

RQ1.3: What are the constraints introduced by the fuzzer to fuzz the i915 graphics driver?

Now, we answer these questions with the qualitative analysis of the fuzzing tools, Syzkaller and kAFL.

RQ1.1: How much effort is required to set up the fuzzer to fuzz the i915 graphics driver?

Target Kernel: Syzkaller requires you to compile the Linux kernel before fuzzing because it uses compile-time instrumentation, *kcov*, which is not enabled by default. Compiling the kernel requires technical skills and is time consuming.

For kAFL on the other hand, it is not necessary to compile the kernel as it uses hardware-assisted coverage and does not rely on OS modules for coverage. The user can directly fuzz an off-the-shelf kernel image and hence, in such a case no effort is required to prepare the target kernel.

Technical Knowledge: The suitable fuzzer should require minimal technical knowledge to deploy. Both fuzzers have an easy-to-follow installation process documented. The difficulty lies in setting up the harness. In Syzkaller the user needs to define a description file using the Syzkaller grammar, *syzlang*. This file is used to define the interface of the target. If the description file is not already defined to the user's requirements, the user needs to write their own description file. To write this file the user needs to have the knowledge of the target interface and the required data arguments. But if the description file has been defined in the Syzkaller code, the user does not need to create a new description. The Syzkaller code needs to be rebuilt every time a change is done to the description file or a new file is added. This can be a cumbersome process. During this thesis we do not set up a harness for Syzkaller as we reuse the i915 description file that is already defined.

For kAFL, the user needs to write a harness for the fuzz target. The kAFL harness is highly customizable and can for example, define the driver interface, create fuzz inputs and call the required ioctls. But writing a harness for the entire target interface is time consuming. To create an effective harness the user requires an in-depth knowledge of how the target interface is working and the dependency of the target ioctls on other ioctls for system resources. We spent a significant amount of time on understanding the driver interface and creating the harness for three ioctls during this thesis.

RQ1.2: How well does the fuzzer present the results for human consumption?

User Interface: Syzkaller provides a web interface that displays the statistics of the fuzzing run, as shown in Figure 5.2. The coverage is displayed in another web interface as shown in Figure 5.4 and the user is able to see detailed coverage

per file in the Linux kernel. The edges covered by Syzkaller are annotated on the source code files that are displayed. The only drawback is that both the web interfaces need to be refreshed manually to update the results.

kAFL provides a real-time tracking graphical user interface (GUI) to keep track of the statistics shown in Figure 5.3 that requires no refreshing. To display the coverage kAFL provides a Ghidra plugin which requires the user to first import the Linux kernel binary. Ghidra is an open-source reverse engineering tool. Importing the kernel image for the first time to ghidra can be time consuming as the software analyzes the binary for all the symbols. Next, the user needs to provide the list of edges to the plugin script before executing it. After running the script, Ghidra annotates the edges covered or uncovered on the disassembled binary code, as shown in Figure 5.5. Ghidra provides a rough decompilation of the binary and the user can map the decompiled code to the disassembly. This means that the user needs to be be skilled at binary analysis to understand the results. We observe that more manual labor is required to analyze the kAFL coverage

Stats:

revision	6eba14c3
config	
uptime	4m10s
fuzzing	7m0s
corpus	2
triage queue	0
cover	579
signal	567
syscalls	1564
crash types	1 (16/hour)
crashes	1 (16/hour)
exec candidate	0 (0/hour)
exec fuzz	0 (0/hour)
exec gen	23 (376/hour)
exec hints	0 (0/hour)
exec minimize	483 (131/min)
exec seeds	0 (0/hour)
exec smash	0 (0/hour)
exec total	560 (152/min)
exec triage	54 (14/min)
executor restarts	23 (376/hour)
new inputs	3 (49/hour)
rotated inputs	0 (0/hour)
suppressed	0 (0/hour)
vm restarts	3 (49/hour)

Crashes:

Description	Count	Last Time	Report

Log:

```
2020/03/09 12:00:49 VMs 1, executed 44, cover 0, crashes 1, repro 0
2020/03/09 12:00:59 VMs 1, executed 44, cover 0, crashes 1, repro 0
2020/03/09 12:01:04 fuzzer vm-1 connected
2020/03/09 12:01:09 VMs 2, executed 60, cover 0, crashes 1, repro 0
2020/03/09 12:01:19 VMs 2, executed 79, cover 0, crashes 1, repro 0
2020/03/09 12:01:29 VMs 2, executed 115, cover 0, crashes 1, repro 0
2020/03/09 12:01:39 VMs 2, executed 141, cover 0, crashes 1, repro 0
2020/03/09 12:01:49 VMs 2, executed 178, cover 0, crashes 1, repro 0
2020/03/09 12:01:59 VMs 2, executed 196, cover 0, crashes 1, repro 0
2020/03/09 12:02:09 VMs 2, executed 232, cover 0, crashes 1, repro 0
2020/03/09 12:02:19 VMs 2, executed 257, cover 0, crashes 1, repro 0
2020/03/09 12:02:29 VMs 2, executed 287, cover 3, crashes 1, repro 0
2020/03/09 12:02:39 VMs 2, executed 320, cover 3, crashes 1, repro 0
2020/03/09 12:02:49 VMs 2, executed 352, cover 3, crashes 1, repro 0
2020/03/09 12:02:59 VMs 2, executed 389, cover 3, crashes 1, repro 0
2020/03/09 12:03:09 VMs 2, executed 421, cover 6, crashes 1, repro 0
2020/03/09 12:03:19 VMs 2, executed 467, cover 6, crashes 1, repro 0
2020/03/09 12:03:29 VMs 2, executed 497, cover 6, crashes 1, repro 0
2020/03/09 12:03:39 VMs 2, executed 528, cover 6, crashes 1, repro 0
2020/03/09 12:03:49 VMs 2, executed 560, cover 567, crashes 1, repro 0
```

Figure 5.2: Syzkaller Statistics Interface

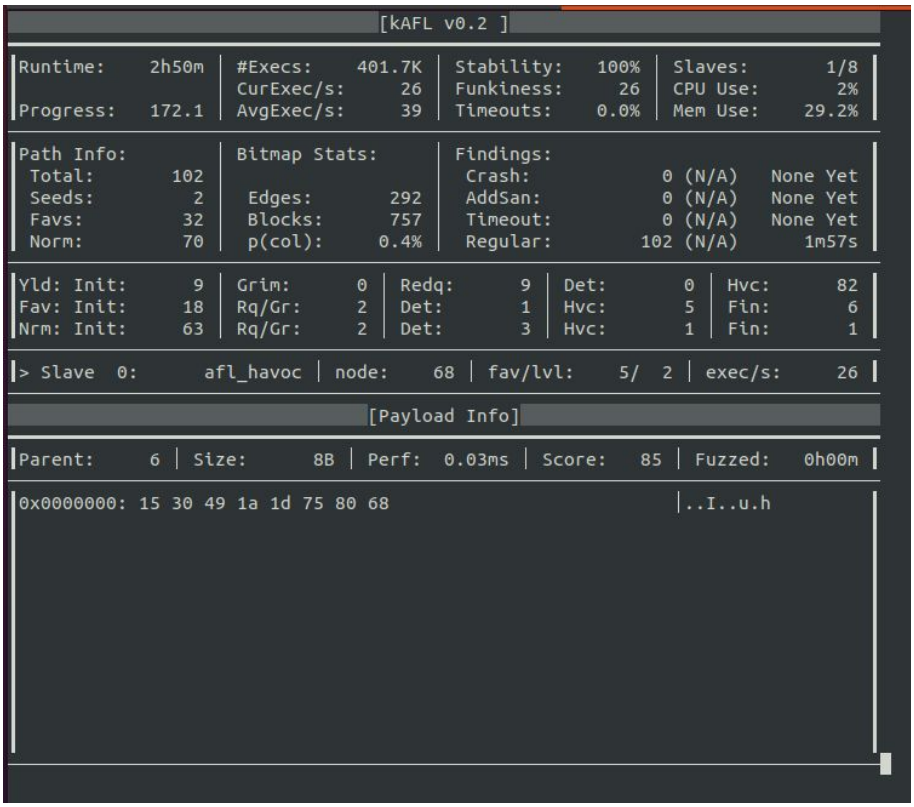

Figure 5.3: kAFL GUI Interface

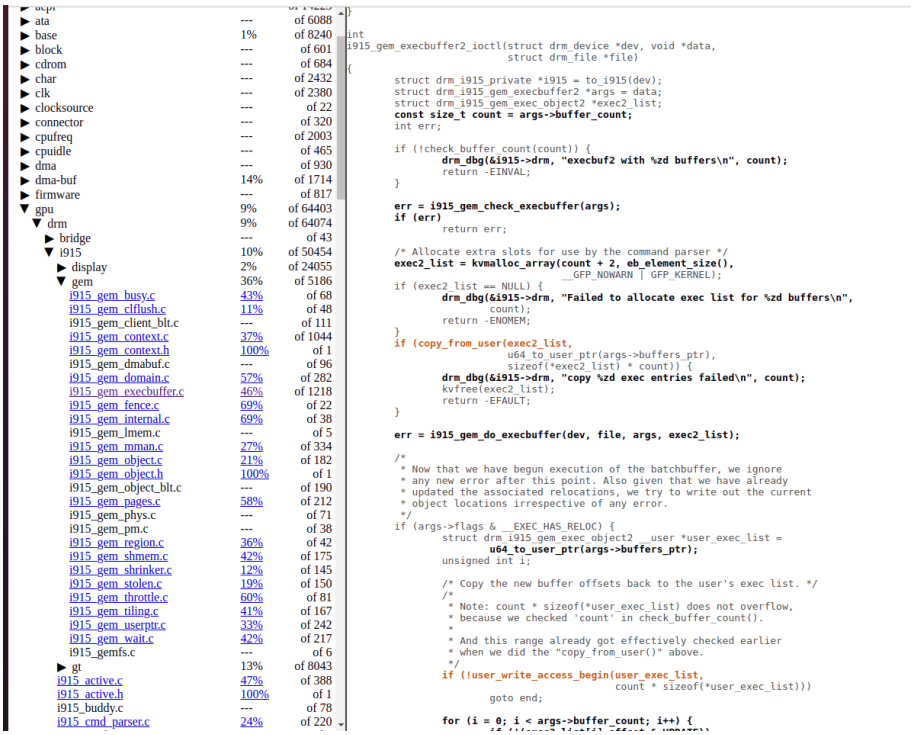

Figure 5.4: Syzkaller Coverage Interface

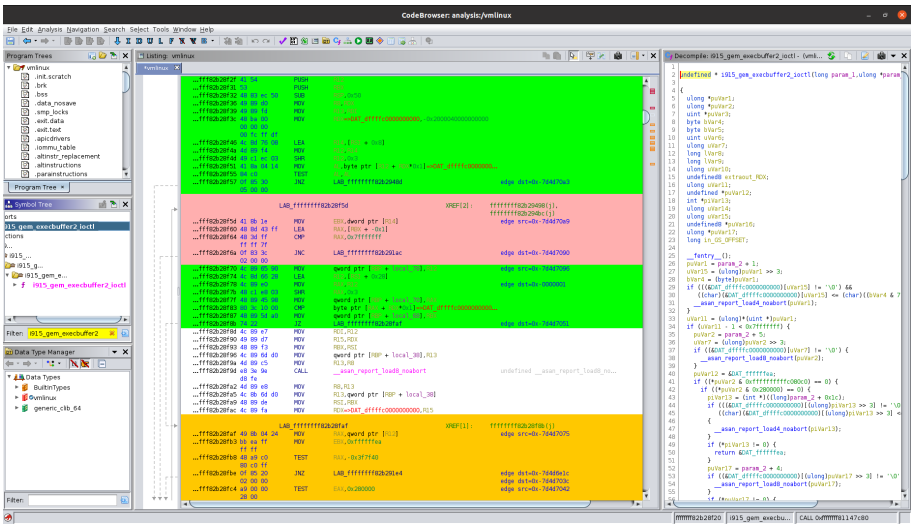

Figure 5.5: Ghidra as kAFL Coverage Interface

RQ1. 3: What are the constraints introduced by the fuzzer to fuzz the i915 graphics driver?

OS-Agnostic: While setting up a fuzzing solution for the i915 driver, it is beneficial if the solution is scalable to different OSes. This helps in the acceptance of one solution for large-scale driver validation across all OSes instead of using OS-specific solutions.

Syzkaller relies on the kernel *kcov* for coverage feedback, therefore, it is unable to support other OSes that are not Unix-like systems like Windows. The current Windows support from Syzkaller is at early stages and is only supported on Google Compute Engine (gce) guests at the time of writing this thesis. This is a huge drawback for deployment in a large-scale driver validation.

On the other hand, kAFL uses hardware-assisted feedback mechanism, that makes the fuzzer OS-independent. Therefore, it is possible for kAFL to provide an OS-agnostic solution which is a favorable feature.

Hardware Access: To fuzz the i915 device driver, the fuzzer needs to be able to access the GPU. Syzkaller and kAFL allow different fuzzing options to access the hardware. The options allowed by each fuzzer are summarized in Table 5.1

Table 5.1: Hardware Access Configurations

	Syzkaller	kAFL
GVT-g	✓	✓
GVT-d	✓	✓
Isolated	✓	✗

We are able to integrate GVT-g and GVT-d configurations for the guest machine in Syzkaller. We can successfully run fuzzing campaigns for atleast 48 hours

for each configuration. We are able to retrieve i915 coverage and find bugs in the driver. Syzkaller also supports an Isolated Host Setup that seemed like a promising setup to fuzz the i915 driver on the bare-metal. We are able to start the fuzzing process with this setup but it turned out problematic in case of system crashes.

With kAFL as well, we are able to integrate GVT-g and GVT-d configurations for the guest machine. With the GVT-g configuration, we can successfully run fuzzing campaigns for atleast 48 hours against the i915 driver and retrieve coverage. With GVT-d we currently run into kAFL bugs that have been reported during this thesis but not yet fixed. Once fixed, this configuration can be tested in the future as a potential choice for fuzzing the i915 driver. kAFL does not currently support the fuzzing option to fuzz on the bare-metal.

In this analysis, we can conclude that each fuzzer has atleast one fuzzing option to successfully fuzz the i915 driver at the time of writing this thesis. In the future it might also be interesting to explore SR-IOV, discussed in Section 3.3.4, as another driver fuzzing option once it becomes available.

5.1.2 *Qualitative Analysis of the Fuzzing Solutions*

In this analysis we aim to analyze the different fuzzing options with the two fuzzers for the i915 graphics driver. For this analysis we have identified three metrics that we would like to evaluate: *crash-tolerance*, *cost-effectiveness* and the *effort* required. We raise a research question for each metric as follows:

Crash-Tolerance

RQ2.1: Is the fuzzing option able to handle the consequences of the non-deterministic kernel and the stateful hardware?

Cost-Effectiveness

RQ2.2: Is the fuzzing option cost-effective for deployment at large-scale i915 driver validation?

Effort Required

RQ2.3: What are the constraints introduced by the fuzzer to fuzz the i915 graphics driver?

Now, we answer these questions with a qualitative analysis of the fuzzing solutions.

RQ2.1: Is the fuzzing option able to handle the consequences of the non-deterministic kernel and the stateful hardware?

One of the major challenges in fuzzing the device driver is dealing with the impacts of non-deterministic kernel and stateful hardware. One possible impact is that a kernel bug or a device hang can cause the entire system to crash during the fuzzing process, as discussed in Section 3.1.2. Therefore, it is important to us that the fuzzing solution can handle this challenge. We now document our findings on how the fuzzers overcome the challenges for each of the configurations:

1. *Syzkaller and GVT-g*: We face system crashes as expected while fuzzing the i915 driver with this configuration using a single guest machine. We are able to resolve this by resetting the GPU and reloading the host driver to clear their states before Syzkaller launches a Qemu instance. We are successful to get Syzkaller with the GVT-g configuration to run continuously without crashing the system. With this fix, in the future this setup can potentially be scaled up to fuzz the i915 driver in parallel using multiple guest instances because GVT-g supports multiplexing of the GPU.

2. *Syzkaller and GVT-d*: We face system crashes as expected while fuzzing the driver in one guest and are able to again resolve this using the same solution as used for GVT-g. With our fix, this setup can now be used to fuzz the i915 driver over extended periods of time. In the future, with this modified version of Syzkaller, this setup can be tested for fuzzing the i915 driver in multiple guests using the *syz-hub* tool provided by Syzkaller.

3. *Syzkaller and Isolated*: While fuzzing the i915 driver in this configuration, system crashes are observed but we are unable to find a solution to recover or even store the state of the crash. In the future it would be interesting to find a solution to somehow retain the crash information, recover from the crash while maintaining connection with the `syz-manager` running on the remote machine. This can be a promising setup to fuzz the i915 driver because it does not introduce an extra virtualization layer and thus, most closely resembles real world scenarios.

4. *kAFL and GVT-g*: We are able to run this setup to fuzz the i915 driver with one guest instance. But with certain targets we run into system crashes for which we are unable to find a solution during this thesis, as discussed in Section 4.3.2.1. In kAFL, snapshot reloading sets the guest back to an initial state but that does not help. As the harness is executing the inputs on the guest, we are not able to use our previous solution of resetting the host driver and GPU as we are unsuccessful to switch back to the host from

inside the harness running in the guest. We are still able to run a successful and stable fuzzing campaign by avoiding certain syscalls as a fuzz target.

5. *kAFL and GVT-d*: We are unable to test this setup for fuzzing because of existing bugs in kAFL which have been disclosed to the kAFL developers. Once these bugs are fixed it would be interesting to evaluate this options.

After exploring the different fuzzing options, we find GVT-g to be a good solution to run a crash-tolerant fuzzing campaign on both the fuzzers as it can potentially be upscaled to fuzz the i915 driver in parallel using multiple guests in the future.

RQ2.2: Is the fuzzing option cost-effective for deployment at large-scale i915 driver validation?

The fuzzing configurations that we test have different costs associated to them. Intel GVT-g has the potential to have the lowest parallelization costs. It can be used to parallelize the fuzzing process by sharing the hardware resource across the multiple guests. This is possible if the fuzzers can provide the capability to have separate qemu configurations for each guest.

Intel GVT-d and Isolated have direct access of the hardware and hence, do not support parallelization. But it can be possible on the fuzzer side to parallelize the fuzzing process across multiple machines. But this increases the additional hardware costs for a large-scale driver validation. Syzkaller's *syz-hub* is not tested during this thesis which can be a possible solution.

RQ2.3: How much effort is required to set up the fuzzing option?

In this section we discuss the technical knowledge required for setting up the different fuzzing options. The user is required to have some knowledge of the GPU pass-through techniques to setup Intel GVT-g and Intel GVT-d. The steps to setup can differ depending on the type of hardware used and the host OS used. But once the steps are clear, the process can be easily automated on machines with similar specs. On the other hand, isolated setup requires the user to only know how to setup the configuration file for Syzkaller.

5.2 QUANTITATIVE ANALYSIS

In Section 5.1, we conclude that both Syzkaller and kAFL are capable of fuzzing the i915 graphics driver with GVT-g configuration. We also note that Syzkaller only supports Unix-like OSes, which excludes the Windows OS. We also see that kAFL provides an OS-agnostic solution. Therefore, kAFL is a possible candidate to fuzz the Windows graphics drivers. Besides that, kAFL also supports a potentially much faster guest snapshot feature than Syzkaller's fuzzing which relies on full guest reboots.

In Section 4.3, we discussed that kAFL is lacking the knowledge of the the driver interface unlike Syzkaller. This means Syzkaller will achieve higher coverage in lesser time than kAFL. This leads us to the following research questions listed in Section 5.3:

RQ3: ***To compensate for the lack of knowledge of the syscall structures in kAFL, can we write a more clever harness and get similar coverage for the i915 graphics driver?***

RQ4: ***Can we use kAFL as an alternative to Syzkaller for graphics driver fuzzing?***

To answer these questions, we first create a clever harness that implements the input generation logic similar to syzkaller as described in Section 4.3. In this Section, we now describe the experiments we run using the comparable harnesses with the GVT-g configuration against the i915 graphics driver as the target. We then perform a quantitative analysis of the *coverage achieved over time* for the two fuzzers. This analysis will give us insights for the above research questions.

This section is structured as follows: In Section 5.2.1 we describe the experiments that were executed. In Section 5.2.2 we inform the reader about the requirements for the validity of the results and comparison. In Section 5.2.3 we present the results and discuss the inferences. In Section 5.2.4 we present two case studies that were carried out to get a closer look into the feedback mechanisms of both the fuzzers.

5.2.1 *Experiments*

All the experiments with both the fuzzers are performed on the same machine to account for any possible system variations. The machine used is as described in Section 4.1.0.1. The harnesses that we use for Syzkaller and kAFL are described in Section 4.3.3 and 4.3.2.1 respectively. As concluded in Section 5.1, we are able to run GVT-g configurations on both the fuzzers. Therefore, we use GVT-g configuration for maximum comparability.

We first test both the fuzzing setups for one run of 24 hours and 12 hours each. We observe no change in the coverage after approximately 4 hours mark for both Syzkaller and kAFL. Therefore, we decide to run each of the fuzzer trial for 6 hours only. A total of 4 trials per fuzzer are measured. 3 different seed files with manually generated random input data are provided to kAFL including a seed file with null bytes. All the trials are run with root access to the device file. Both the fuzzing setups are started with only one guest machine due to Intel GVT-g configuration as explained in Section 4.2.1.1 and Section 4.2.2.1.

5.2.2 *Requirements*

The coverage collection method for Syzkaller, *KCOV*, is specific to Linux and cannot be applied to any closed-source target. Therefore, we would like to note that the comparison in this thesis would be fair if considered only for the open-source target i915 driver for the Linux Graphics Stack.

5.2.3 *Results and Discussion*

In this section we present and discuss the results from the experiments. In each figure presented, the solid line represents the average number of edges found every 5 seconds over the course of 4 fuzz trials. The average values are rounded off. The other data lines are the original results observed from each trial.

It is to be noted that all the values in this section are reported according to the results reported on the user-interfaces of both the fuzzers.

5.2.3.1 *Coverage Over Time*

The results presented in Figures 5.6 and 5.7 for kAFL, represent the cumulative number of edges covered (on the Y axis) over logarithmic time and linear time (on the X axis) respectively. Figures 5.8 and 5.9 for Syzkaller, also represent the cumulative number of edges covered (on the Y axis) over logarithmic time and linear time (on the X axis). In Figure 5.10 we map the kAFL and Syzkaller average number of edges covered over logarithmic time for better visualization.

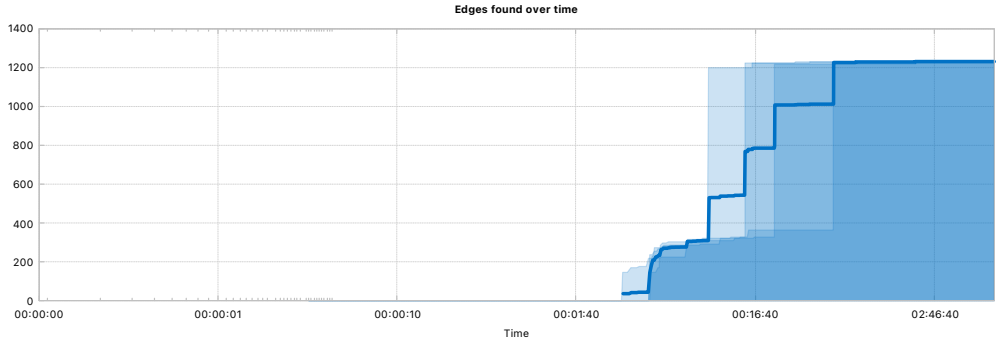

Figure 5.6: kAFL Edges Found Over Time (Logarithmic Scale)

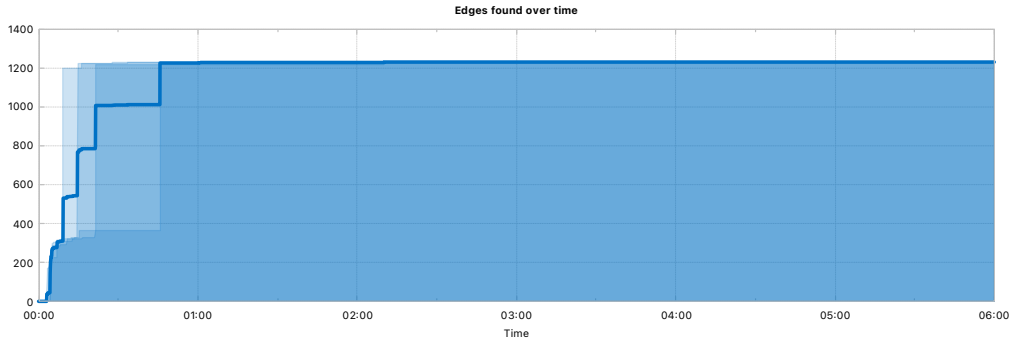

Figure 5.7: kAFL Edges Found Over Time

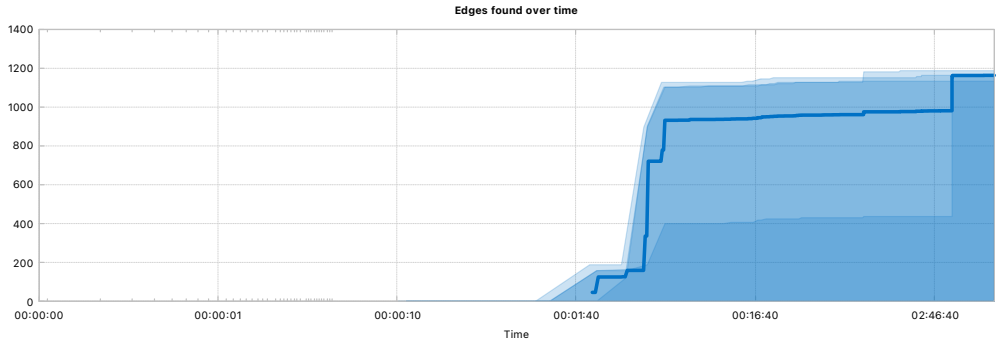

Figure 5.8: Syzkaller Edges Found Over Time (Logarithmic Scale)

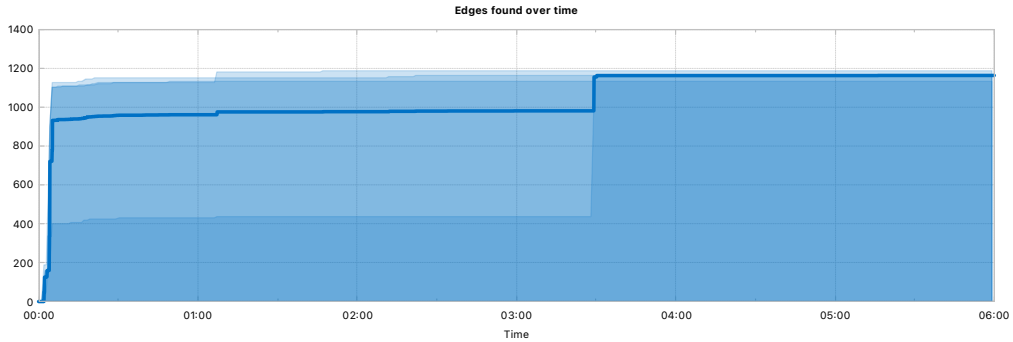

Figure 5.9: Syzkaller Edges Found Over Time

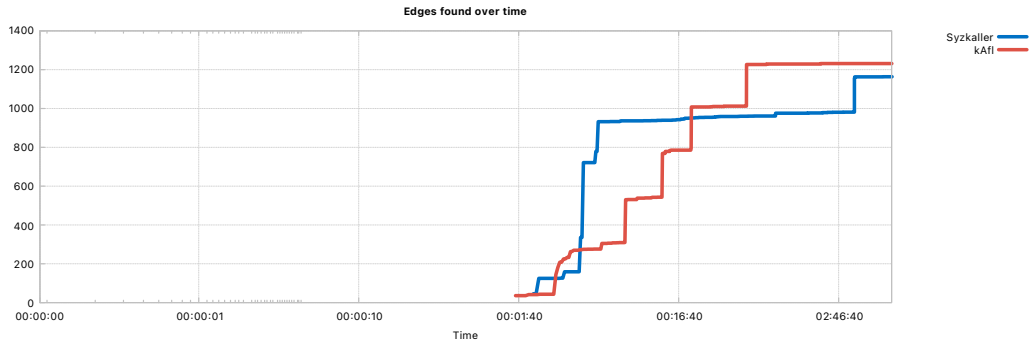

Figure 5.10: Syzkaller vs kAFL Edges Found Over Time

We now state our observations from these graphs as follows

1. **Observation 1**: We observe from the figures that considering only one run can lead to wrong conclusion. If we consider Figure 5.6, the lightest colored data plot reaches close to the total edges covered quickly than other two trials. If we consider the darkest colored data plot in the same figure, it can lead us to incorrectly believe that kAFL took way longer than Syzkaller to reach the total edges covered. Also, if we consider Figure 5.8, we see that the darkest data plot took a significant amount to time to jump closer to the total number of edges covered. This plot would have led us to conclude that Syzkaller has a slower increase in coverage.

   Observation 2: We compare the number of edges found. We see that the number of edges found in all the trials for both the fuzzers are similar. For Syzkaller, the average topped at 1160 and for kAFL it topped at 1229.

   Observation 3: We observe in the linear scale plots of Syzkaller and kAFL that coverage stops growing after a certain time. For kAFL we observe that the coverage does not increase on an average after approximately 3 hours and for Syzkaller the coverage does not increase after approximately 3.5 hours.

   Observation 4: We display the logarithmic scale because the linear scale does not give us any information about the first 20 minutes or so of the fuzzing campaign. We see that there was no interesting activity upto a minute after the start of the fuzzing campaigns. In Figure 5.10, we see that kAFL is able to find the first edge faster than Syzkaller. But Syzkaller attains a higher coverage quickly up until 20 minutes while kAFL is rising slowly and steadily. After this point, kAFL looks faster in achieving higher coverage.

   Observation 5: The number of trials we carried out for both fuzzers are not enough to carry out a statistical analysis. But we can still infer certain things by looking at the averages of the trials in Figure 5.10. First, we can see that the variance between the edges found over time between Syzkaller and kAFL is high. This is because both fuzzers can take completely different

paths and some of these can take a longer time to process, for example, loops.

5.2.3.2 *Coverage Over Executions*

The results presented in Figures 5.11 and Figure 5.12 represent the cumulative number of edges covered (on the Y axis) over the number of executions (on the X axis) for kAFL and Syzkaller respectively. In Figure 5.10 we map the kAFL and Syzkaller average number of edges covered over logarithmic time for better visualizations.

Observation 6: We can see that the number of executions per second for kAFL were significantly higher than Syzkaller. This is potentially due to the faster performance of the snapshot reload after every input.

Observation 7: We observe that even with lower number of executions Syzkaller is able to get almost the similar number of coverage achieved as kAFL.

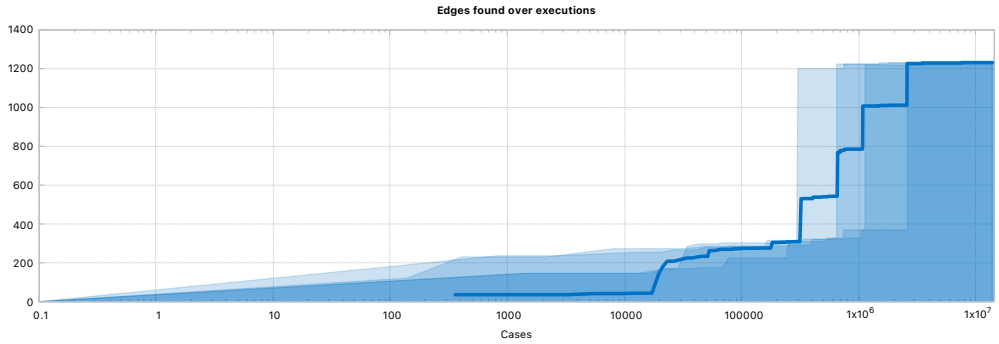

Figure 5.11: kAFL Edges Found Over Executions (Logarithmic Scale)

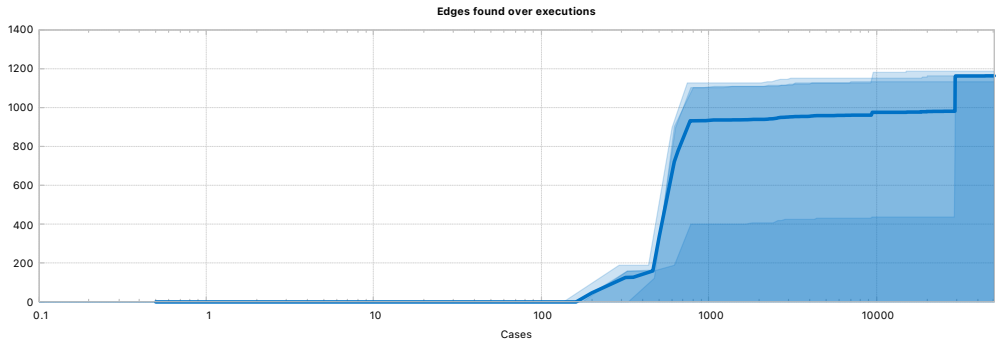

Figure 5.12: Syzkaller Edges Found Over Executions (Logarithmic Scale)

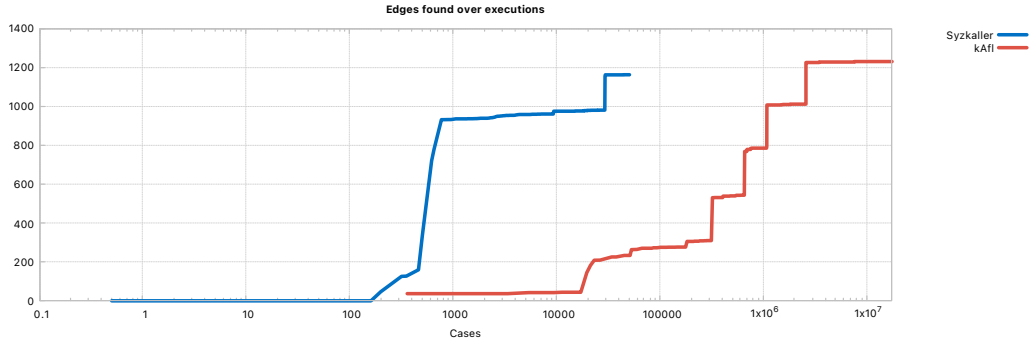

Figure 5.13: Syzkaller vs kAFL Edges Found Over Executions

5.2.4 *Comparing Coverages*

It is to be noted that the coverage results from both the fuzzers are not completely comparable when retrieved from the user-interface. Firstly, Syzkaller coverage includes edges covered with the *openat* ioctl whereas kAFL coverage does not. Secondly, it differs due to the difference in the nature of feedback mechanisms used. The feedback mechanisms of both the fuzzers deploy different kinds of transformation and optimization techniques. As a result, coverage received may poorly map to the source code. For example, depending on the C compiler and kernel build options used to compile the target kernel for Syzkaller, data structures can contain different alignment of structures or functions can be included in different ways (inline or not). While anaylzing Syzkaller coverage, it is not uncommon to see a covered edge after a non-covered edge or you may not see an edge where you expected one. Even then, assessing the coverage can be useful and gives us an idea of how the fuzzing processes are working for both the fuzzers. We see that even with a significantly lower number of executions per second, Syzkaller is able to get similar number of edges as kAFL. It also reaches close to the maximum coverage faster than kAFL in all the trials.

Since the results are not completely comparable, we decided to manually investigate the coverage achieved by both the fuzzers to understand the difference in the fuzzing results. We used the coverage analysis tools provided by each fuzzer. We now discuss some significant findings via these methods in the form of a case study.

Case Study

Syzkaller provides a final list of edges that every individual ioctl covered. We are able to extract the coverage per ioctl for `i915_GEM_EXECBUFFER2_IOCTL` and `i915_QUERY_IOCTL`. The average number of edges covered by Syzkaller across 4 trials is 344. We are able to map the kafl coverage to source code via a script *kafl_cov.py* provided by kAFL. We are able to filter the coverage of only the i915 module from this mapping. The average number of edges discovered across

all the 4 trials is 2191. This number is significantly different from the number reported by the kAFL user-interface. We decide to discard this number and use the Ghidra plugin provided by kAFL to manually analyze the coverage for further investigation.

We now compare the coverage information received by the Syzkaller web interface and the Ghidra plugin. For Syzkaller, we are able to see the edges that are instrumented in the source code. We investigate one of the interesting files, *i915_gem_execbuffer.c* which contains the execbuffer2 ioctl that we fuzzed. We find certain functions like *reloc_cache_init* in the file that are not instrumented for Syzkaller. But this function has edges that are covered by kAFL in the edge mappings to source code results. This is because Clang/LLVM uses optimization techniques to not instrument redundant code and inline functions and Syzkaller relies on this instrumentation to collect coverage.

But we can see that the source code of most major functions like textiti915_gem_execbuffer2_ioctl, textiti915_gem_do_execbuffer textiti915_gem_check_execbuffer is covered almost equally. This means apart from functions that are not covered by Syzkaller instrumentation, we can consider Syzkaller and kAFL coverage almost comparable.

5.3 DISCUSSION

In this Section, we answer the research questions defined in Section by using the analyses from the above Sections as follows:

RQ1: ***Which of the two fuzzing tools, Syzkaller or kAFL, is better suited for large-scale driver validation?***

Syzkaller is a state-of-the-art fuzzer and is able to fuzz the i915 driver on Linux. It provides user-friendly interface to view the statistics and coverage. And the harness requires moderate technical knowledge about the Syzkaller grammar *syz-lang* and the driver interface. But Syzkaller comes with two major drawbacks for large-scale driver validation: It needs a custom compiled target kernel and it can currently only fuzz Unix-like OSes.

kAFL is a fuzzer prototype that can achieve the similar coverage to Syzkaller when fuzzing the i915 graphics driver but only if the user provides kAFL a clever harness. kAFL has a good real-time GUI for statistics but to view the coverage requires third party tools and knowledge of binary analysis. The major advantage of kAFL for large-scale driver validation is that it provides an OS-agnostic fuzzing solution which means it can also potentially fuzz the Windows graphics driver.

Both the fuzzers can be good options for large-scale driver validation depending on the individual use case.

RQ2: ***Which of the different fuzzing solutions explored is better suited for large-scale driver validation?***

After exploring the different fuzzing solutions, we find GVT-g to be a good option to run a crash-tolerant fuzzing campaign on both the fuzzers as it can potentially be upscaled to fuzz the i915 driver in parallel using multiple guests in the future. This will be a cost-effective solution for large-scale driver validation as it reduces the additional hardware costs.

GVT-d and Isolated can be tested with Syzkaller's *syz-hub* tool to fuzz multiple machines at a time as a future work. But this will come with higher hardware costs as each target machine needs its own GPU.

RQ3: ***To compensate for the lack of knowledge of the syscall structures in kAFL, can we write a more clever harness and get similar coverage for the i915 graphics driver?***

We are able to create a clever harness that implements a similar input generation logic for three ioctls as Syzkaller. But we run into a bug with this harness, as discussed in Section 4.3.2.1. This leads us to modify our kAFL harness to fuzz two ioctls. The only difference lies in fuzzing the `openat` ioctl which Syzkaller is able to fuzz while the kAFL harness is not. By running the comparable harnesses, we are able to get similar coverage in a similar time frame with both the fuzzers. But due to differences in the harnesses and the feedback mechanisms of the fuzzers, the coverage is not 100% comparable. Therefore, we manually investigate the coverage using the tools provided by the fuzzers and find that they are almost comparable. Therefore, we can conclude that our clever kAFL harness was able to get similar coverage for the i915 driver as Syzkaller in the same time frame. Thus, compensating for the initial lack of knowledge of the syscall structure in kAFL.

RQ4: ***Can we use kAFL as an alternative to Syzkaller for graphics driver fuzzing?***

From our qualitative analysis we know that kAFL is able to fuzz the i915 driver continuously with the GVT-g configuration. Our qualitative analysis tells us that kAFL also performs very similar to Syzkaller regarding coverage achieved over time with a clever kAFL harness. Therefore, it is safe to conclude that kAFL is a good alternative to Syzkaller in graphics driver fuzzing if the user is willing to put in the time and effort to create such a clever harness.

In case the user wants to fuzz the graphics driver on the Windows OS, kAFL presents itself as a good alternative to Syzkaller that does not support Windows fuzzing.

6

CONCLUSION AND FUTURE WORK

In this thesis, we aim to find a fuzzing solution that can overcome the challenges of fuzzing the i915 driver such that it can be deployed for i915 driver validation at large-scale. We define our two major research contributions and research them as follows:

I. ***What are the different options for fuzzing the i915 GFX driver for large-scale driver validation?***

   For our solution, we choose Intel's device virtualization configurations to fuzz the i915 driver with both Syzkaller and kAFL. We face system crashes as expected while fuzzing the i915 driver with all the configurations. We overcome this challenge by resetting the GPU and reloading the i915 driver on the host system. We successfully setup Syzkaller with GVT-g and GVT-d and kAFL with GVT-g. With the experiences made during the solution setup and fuzzing of the i915 driver with both the fuzzers, we present a qualitative analysis of both the fuzzers and the fuzzing solutions for large-scale driver validation.

II. ***Present a quantitative analysis of Syzkaller and kAFL with the Intel i915 GFX driver as the target.***

   We compare the performances of the fuzzers based on speed and coverage achieved to investigate which fuzzer is more effective at fuzzing the i915 driver. To make the fuzzing setups comparable, we create a clever kAFL harness that implements a structure aware harness and input generation logic similar to Syzkaller. We fuzz the i915 driver using both the fuzzers with GVT-g configurations and we are able to get a similar number of edges covered. But the coverage is not completely comparable due to certain differences in harness implementation and the feedback mechanisms used by the fuzzer. We investigate manually to understand the results in detail. We present a case study where we explore the coverage achieved manually. We find that the major functions of the fuzzed *ioctls* were equally covered by both the fuzzers and hence, conclude that the coverages can be considered comparable. This means that kAFL with a clever harness can achieve comparable results to Syzkaller

   Finally, we would like to conclude that both fuzzers have certain advantages and certain disadvantages. Syzkaller is state-of-the-art fuzzer with many user-friendly options and hence, can be a good option for fuzzing but only works with Unix-like OSes. With kAFL writing a clever harness and maintaining it can be labor intensive. But if the user wants an OS-agnostic

solution for large-scale driver validation as it can be cost-effective, kAFL is a good alternative option.

Our comparison was limited on results by fuzzing only two ioctls for kAFL and three for Syzkaller. Therefore, the future work to write a kAFL harness for the entire i915 driver interface can provide more findings. In the future, Syzkaller and kAFL can be modified to accept different QEMU arguments for each guest to setup parallelization for Intel GVT-g. Also, Syzkaller's *syz-hub* can be tested on all the configurations to parallelize the fuzzing process to validate the graphics driver on multiple machines.

A

APPENDIX

A.1 INTEL GVT-G SETUP

The intent of this section is to document the setting up of the Intel iGPU virtualization methods Intel GVT-g and Intel GVT-d on the host kernel. The following steps were followed to enable Intel GVT-g.

A.1.0.1 *Software Versions Tested*

The following versions were tested for Setup. The results with these **Linux Kernels**

a) 4.19.0

b) 5.4.0-59

c) 5.8.0-66

d) 5.11.6 (latest stable)

Qemu

a) 4.2.1

b) 5.0.0

A.1.0.2 *Creating the virtual GPU*

The first step is to create a virtual GPU (vGPU) on the host and then assign it to the guest machine. The guest will see this vGPU as the real GPU and is able to access it even if it does not have the particular device driver. This is achieved by using PCIe Passthrough. Following steps need to be followed to create a vGPU:

a) **Kernel Parameters**: The following kernel parameters are required to be enabled in `/etc/default/grub`. The grub file used has been added to the appendix.

- IOMMU support can be enabled on the host OS by adding `intel_iommu=on` to the kernel parameters
- Enable support for Intel GVT-g virtualization by adding `i915.enable_gvt=1` to the kernel parameters

- For Gen11+ hardware and since Linux 5.4, the GuC/HuC firmware has been enabled by default. This might lead to issues on certain systems and can be disabled by adding `i915.enable_guc=0` to the kernel parameters.
- Some additional parameters can be used for debugging like `drm.debug=0x01` and `drm.debug=0x02`

b) **Kernel Modules**: The following kernel modules need to be loaded. It is to be noted that depending on the version of Linux kernel these modules might be present either as loadable modules or might be compiled directly to the kernel. The loadable modules can be loaded on boot by adding them to `/etc/modules-load.d/modules.conf`

- **kvmgt**: This module activates the Intel GVT-g for KVM.
- **vfio_mdev**: The VFIO driver framework provides many APIs that can be used to expose direct access to the device from the user space. This framework is also used for mediated devices, therefore this can assist GVT-g which is based on mediated passthrough. The generic operations exposed are:
  i. Create and destroy a device
  ii. Add or remove the device to the mediator driver
  iii. Add or remove a device from the IOMMU group

  The mediator driver discussed in section 3.3.4 is provided by the module vfio-mdev.
- **vfio_iommu_type1**: This module provides a x86 IOMMU API for Intel VT-d.

c) Reboot the system and check if the vGPU types are available. This can be done by ensuring the folder `mdev_supported_types` has been created. This folder should contain the different types of vGPU that you can create. The types differ on the amount of VRAM they can provide.

```
# ls /sys/bus/pci/devices/0000:00:02.0 | grep
    mdev_supported_types
mdev_supported_types
```

```
# ls /sys/bus/pci/devices/0000:00:02.0/mdev_supported_types |
    grep i915
i915-GVTg_V5_4  // Video memory: <128MB, 512MB>, resolution: up
    to 1920x1200
i915-GVTg_V5_8  // Video memory: <64MB, 384MB>, resolution: up to
     1024x768
```

d) The last step is to create the vGPU. First, a UUID is required that can be generated using the command `uuidgen`. This UUID needs to be assigned to a vGPU type using the following command:

```
# echo <UUID> > /sys/bus/pci/devices/000supported_types/i915-GVTg
    _V5_4/create
```

If the previous command was successful, a folder with the UUID as the folder name in the GPU's /sys folder should have been created

e) To make the vGPU always available on boot, a systemd service can be created as `/etc/systemd/system/setup-gvt.service` and the following code can be used:

```
[Unit]
        Description=Setup GVT

[Service]
        Type=oneshot
        ExecStart=/usr/bin/bash -c 'echo ef8e2751-94bf-4b71-9c1d
            -432dca83a9ec' > /sys/bus/pci/devices/0000:00:02.0/
            mdev_supported_types/i915-GVTg_V5_4/create'

[Install]
        WantedBy=multi-user.target
```

A.1.0.3 *Add vGPU to Guest*

The following QEMU arguments need to be added to add the vGPU device to the guest machine.

```
-vga none -device vfio-pci,sysfsdev=/sys/bus/pci/devices/0000:00:02.0/<
    UUID>
```

To check if the addition has been successful, run `lspci` on the guest machine to see if your GPU is visible. Also, the GPU device files `/dev/dri/card0` and `/dev/dri/renderD128` should be available on the guest machine.

A.2 KAFL HARNESS ALGORITHM

Algorithm 4 Harness Implementation

```
1: function MAIN
2:     kAFL Handshake using KAFL_ACQUIRE and KAFL_RELEASE
3:     while true do
4:         fd ← OPENAT()
5:         payload ← KAFL_GET_PAYLOAD()
6:         KAFL_ACQUIRE()
7:         RANDOMIZE_SYSCALLS(fd, payload)
8:         KAFL_RELEASE()

9: function RANDOMIZE_SYSCALLS(fd, payload)
10:    filedescriptors ← [fd,-1]
11:    repetitions ← Get 2 numbers from the payload
12:    if sum of repetitions = 0 then
13:        CALL_I915_EXECBUFFER(fd, payload)
14:    else
15:        func_names ← repetitions[0] ∗ "o" + repetitions[1] ∗ "e"
16:        shuffle func_names according to the buffer
17:        for all char in func_names do
18:            fd_index ← index for file descriptors from payload
19:            fd′ ← filedescriptors[fd_index]
20:            if char = "o" then
21:                CALL_I915_EXECBUFFER2(fd′, buffer)
22:            else
23:                CALL_I915_QUERY(fd′, buffer)

24: function CALL_I915_QUERY(fd, buffer)
25:    query ← input for DRM_IOCTL_I915_QUERY from payload data or constant value
26:    Set special flags
27:    ioctl(fd, DRM_IOCTL_I915_QUERY, query)

28: function CALL_I915_EXECBUFFER2(fd, buffer)
29:    query ← input for DRM_IOCTL_I915_GEM_EXECBUFFER2 from payload data or
       constant value
30:    ioctl(fd, DRM_IOCTL_I915_GEM_EXECBUFFER2, query)
```

BIBLIOGRAPHY

[1] Project ACRN. Project ACRN. Technical report.

[2] Andre Almeida. Using syzkaller: Fuzzing the Linux kernel, 2020.

[3] Thomas Alsop. PC GPU market share worldwide by vendor 2020 | Statista, 2020.

[4] Rao Arvind Mallari. Linux Kernel System Call Fuzzing. Technical report.

[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. 01 2019.

[6] Patrick Bitterling. Operating System Kernels. Technical report.

[7] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1967–1983, 2019.

[8] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 73â88, New York, NY, USA, 2001. Association for Computing Machinery.

[9] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. *SIGOPS Oper. Syst. Rev.*, 35(5):73â88, October 2001.

[10] Catalin Cimpanu. Chrome: 70Technical report, 2020.

[11] Marco Cesati Daniel P. Bovet. Understanding the LINUX KERNEL. Technical report.

[12] Micah Dowty and Jeremy Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *SIGOPS Oper. Syst. Rev.*, 43(3):73â82, July 2009.

[13] Prasun Gera, Hyojong Kim, Hyesoon Kim, Sunpyo Hong, Vinod George, and Chi-Keung Luk. Performance characterisation and simulation of intel's integrated gpu architecture. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 139–148, 2018.

[14] Google. Afl.

[15] Google. GitHub - google/syzkaller: syzkaller is an unsupervised coverage-guided kernel fuzzer.

[16] Google. Technical "whitepaper" for afl-fuzz. Technical report.

[17] Google. Honggfuzz, 2016.

[18] Wenjian He, Wei Zhang, Sharad Sinha, and Sanjeev Das. IGPU Leak: An Information Leakage Vulnerability on Intel Integrated GPU. *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, 2020-Janua:56–61, 2020.

[19] Wenjian HE, Wei Zhang, Sharad Sinha, and Sanjeev Das. iGPU Leak: An Information Leakage Vulnerability on Intel Integrated GPU. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 56–61, 2020.

[20] Aki Helin. Radamsa.

[21] Cheol-Ho Hong, Ivor Spence, and Dimitrios S Nikolopoulos. Gpu virtualization and scheduling methods: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 50(3):1–37, 2017.

[22] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2019.

[23] Mark Johnson. FreeBSD Bay Area Vendor Summit, 2019.

[24] Jens Owen Kevin E. Martin, Rickard E. Faith and Allen Akin. Direct Rendering Infrastructure, Low-Level Design Document, 1999.

[25] Taehun Kim, Jaehan Kim, and Youngjoo Shin. Constructing Covert Channel on Intel CPU-iGPU platform. *International Conference on Information Networking*, 2021-Janua:39–42, 2021.

[26] Simon Kitching. Mine of Information - DRM and KMS kernel module implementations, 2012.

[27] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. Technical report.

[28] Dan Li and Hua Chen. FastSyzkaller: Improving Fuzz Efficiency for Linux Kernel Fuzzing. *Journal of Physics: Conference Series*, 1176(2), 2019.

[29] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 809–814, 2018.

[30] Linux. Intel GFX Driver Documentation. Technical report.

[31] Valentin J M Manès, Man' Manès, Hyungseok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. Technical report, 2019.

[32] Stéphane Marchesin. *Linux Graphics Drivers: an Introduction*. 2012.

[33] Clementine Maurice, Christoph Neumann, Olivier Heen, and Aurelien Francillon. Confidentiality issues on a gpu in a virtualized environment. In Springer, editor, *FC 2014, 18th International Conference on Financial Cryptography and Data Security, 3-7 March 2014, Barbados*, Barbados, 2014. Â© Springer. Personal use of this material is permitted. The definitive version of this paper was published in FC 2014, 18th International Conference on Financial Cryptography and Data Security, 3-7 March 2014, Barbados and is available at : http://dx.doi.org/10.1007/978-3-662-45472-5_9.

[34] Pedram Amini Michael Sutton, Adam Greene. Fuzzing: Brute Force Vulnerability Discovery, 2007.

[35] Serguei A. Mokhov, Marc André Laverdière, and Djamel Benredjem. Taxonomy of Linux kernel vulnerability solutions. In *Innovative Techniques in Instruction Technology, E-Learning, E-Assessment, and Education*, pages 485–493. Kluwer Academic Publishers, 2008.

[36] Yoav Alon Netanel Ben-Simon. Bugs on the Windshield: Fuzzing the Windows Kernel - Check Point Research, 2020.

[37] Charlie Osborne. Google's Fuzz bot exposes over 1,000 open-source bugs. Technical report, 2017.

[38] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 729–743, 2018.

[39] Hui Peng and M. Payer. Usbfuzz: A framework for fuzzing usb drivers by device emulation. In *USENIX Security Symposium*, 2020.

[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[41] Sergej Schumilo. Design and Implementation of a Hardware Accelerated, General Purpose and Coverage-Guided Operating System Fuzzer, 2016.

[42] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[43] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. Kafl: Hardware-assisted feedback fuzzing for os kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 167â182, USA, 2017. USENIX Association.

[44] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE, 2016.

[45] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jia-Guang Sun. Industry practice of coverage-guided enterprise linux kernel fuzzing. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[46] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[47] Jeffrey Vander Stoep. Android: protecting the kernel. in linux security summit. linux foundation, 2016.

[48] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 121â132, USA, 2014. USENIX Association.

[49] Tom. How DRI and DRM Work - Forbidden Projects, 2011.

[50] Iago Toral. A brief introduction to the Linux graphics stack | Developer Log, 2019.

[51] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 61–64, 2018.

[52] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Sirer, and Fred Schneider. Device driver safety through a reference validation mechanism. pages 241–254, 01 2008.

[53] Toshihiro Yamauchi, Yohei Akao, Ryota Yoshitani, Yuichi Nakamura, and Masaki Hashimoto. Additional kernel observer: privilege escalation attack prevention mechanism focusing on system call privilege changes. *International Journal of Information Security*, pages 1–13, jun 2020.

[54] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154, 2017.

[55] Andrew J Younge, John Paul Walters, Stephen Crago, and Geoffrey C Fox. Evaluating gpu passthrough in xen for high performance cloud computing. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 852–859. IEEE, 2014.

[56] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. Understanding the security of discrete GPUs. *Proceedings of the General Purpose GPUs, GPGPU-10 2017*, 11:1–11, 2017.