

# gVisor Nvidia Tooling

Enabling easy driver support and parity

anthonycui@ - 2024-08-13

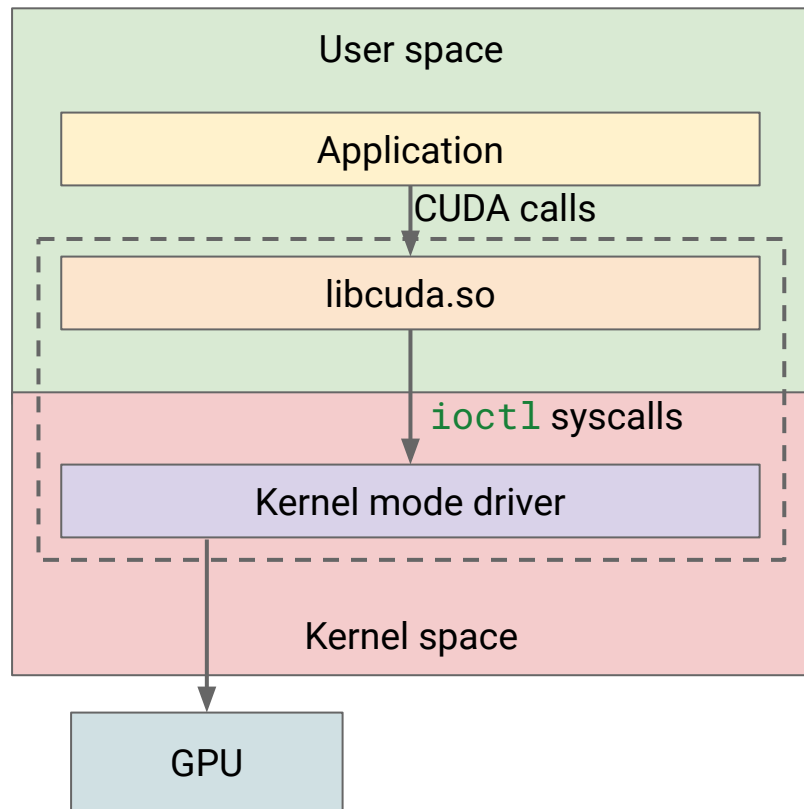
# Outline

1. Interfacing with Nvidia drivers
2. Motivation
3. Tool 1: Sniffing `ioctl` calls
4. Tool 2: Maintaining parity with driver updates
5. Results
6. Future work
7. Q&A

# Interfacing with Nvidia drivers

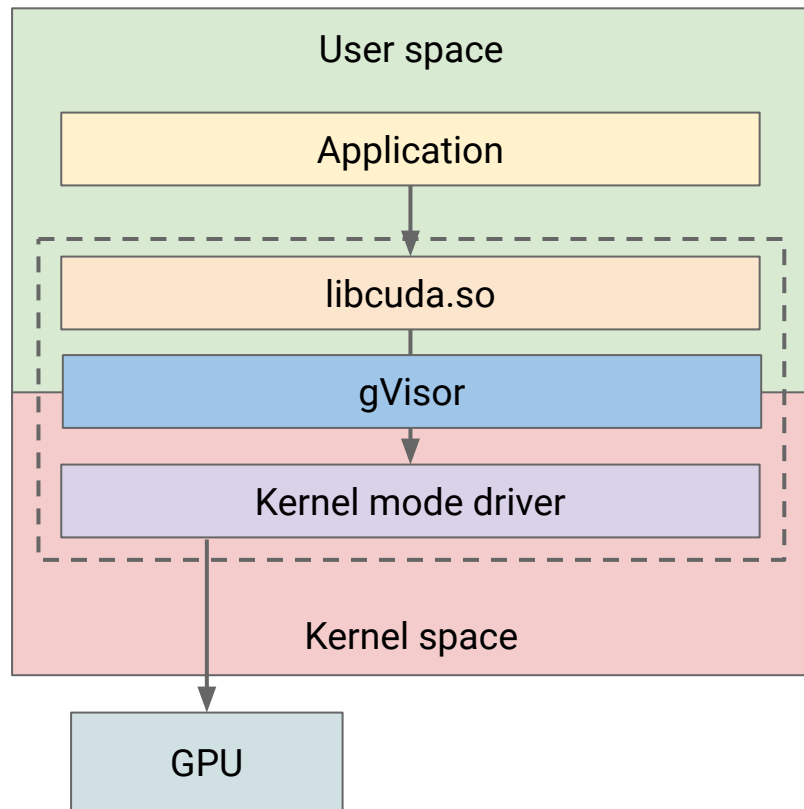
# A basic CUDA call

- What happens when a user space application makes a CUDA call?
- Two parts: libcuda.so and a kernel mode driver.
  - The application makes CUDA calls to libcuda.so.
  - libcuda.so translates CUDA calls into a number of `ioctl` syscalls to the kernel mode driver.



# How does gVisor fit in?

- gVisor inserts itself between libcuda.so and the kernel driver.
- This means we need to **intercept and handle `ioctl` calls**.
- This is all handled by a feature called NVProxy.



# What does NVProxy do?

- Cannot emulate the kernel driver like we do with other Linux syscalls.
- Instead, we filter and proxy the `ioctl` syscalls made to the kernel driver.

`ioctl(fd, op_code, args)`

- `fd` is a file descriptor for a device file. We allow:
  - `/dev/nvidia-uvm`
  - `/dev/nvidiactl`
  - `/dev/nvidia##`
- `op_code` describes what `ioctl` operation is being performed. We have a whitelist of allowed `op_codes`.
- `args` is a pointer to any additional arguments required.

# Simple and complex `ioctl`s

- When proxying `ioctl` syscalls, we may need to translate the arguments.
  - For example, **memory addresses** and **file descriptors** need to be translated.
- This ABI is **not stable!**
  - `ioctl` definitions are meant to be Nvidia-only, so can change at any time.

```
typedef struct nv_ioctl_alloc_os_event
{
    NvHandle hClient;
    NvHandle hDevice;
    NvU32 fd;
    NvU32 Status;
} nv_ioctl_alloc_os_event_t;
```

# Simple and complex `ioctl`s

`ioctl(fd, op_code, args)`



`nvgpu.NV_ESC_CARD_INFO : frontendIoctlSimple`

- Proxied through without translation.
- Can still have arguments.

`nvgpu.NV_ESC_ALLOC_OS_EVENT :`

`frontendIoctlHasFD [nvgpu.IoctlAllocOSEvent]`

- Needs to be translated.
- NVProxy contains the struct definition of `args`.



# Branching within NVProxy

`ioctl(fd, op_code, args)`

Frontend `ioctl`s

- `/dev/nvidiactl`
- `/dev/nvidia##`



## Control commands and Alloc classes

- “Sub-`ioctl`s” of the `NV_ESC_RM_CONTROL` and `NV_ESC_RM_ALLOC` frontend `ioctl`s.
- Command/class is passed in `args`, which also has its own parameters.

UVM `ioctl`s

- `/dev/nvidia-uvm`

```
typedef struct
{
    NvHandle hClient;
    NvHandle hObject;
    NvV32 cmd;
    NvU32 flags;
    NvP64 params;
    NvU32 paramsSize;
    NvV32 status;
} NVOS54_PARAMETERS ;
```

# What are we trying to solve?

- ABI of the kernel mode driver is not stable and always changing.
- Maintaining and supporting new GPU workloads requires constant work.
- My project has been to develop tools to help this in two ways:
  1. Help determine what unsupported `ioctl` a workload depends on
  2. Help adopt `ioctl` changes that may come from driver updates.

# Tool 1: Sniffing `ioctl` calls

# Overview

- Takes any GPU workload binary, and reports any unsupported **ioctl**s used.
- Runs the binary as a subprocess, intended to be run outside gVisor.

```
> ./run_sniffer nvidia-smi
```

```
Frontend: None
```

```
UVM: None
```

```
Control:
```

```
Control ioctl: request=0xc020462a [nr=NVIDIA_ESC_RM_CONTROL, cmd=0x20810110] => ret=0
```

```
Control ioctl: request=0xc020462a [nr=NVIDIA_ESC_RM_CONTROL, cmd=0x208f1105] => ret=0
```

```
Alloc:
```

```
Alloc ioctl: request=0xc030462b [nr=NVIDIA_ESC_RM_ALLOC, hClass=0xc639] => ret=0
```

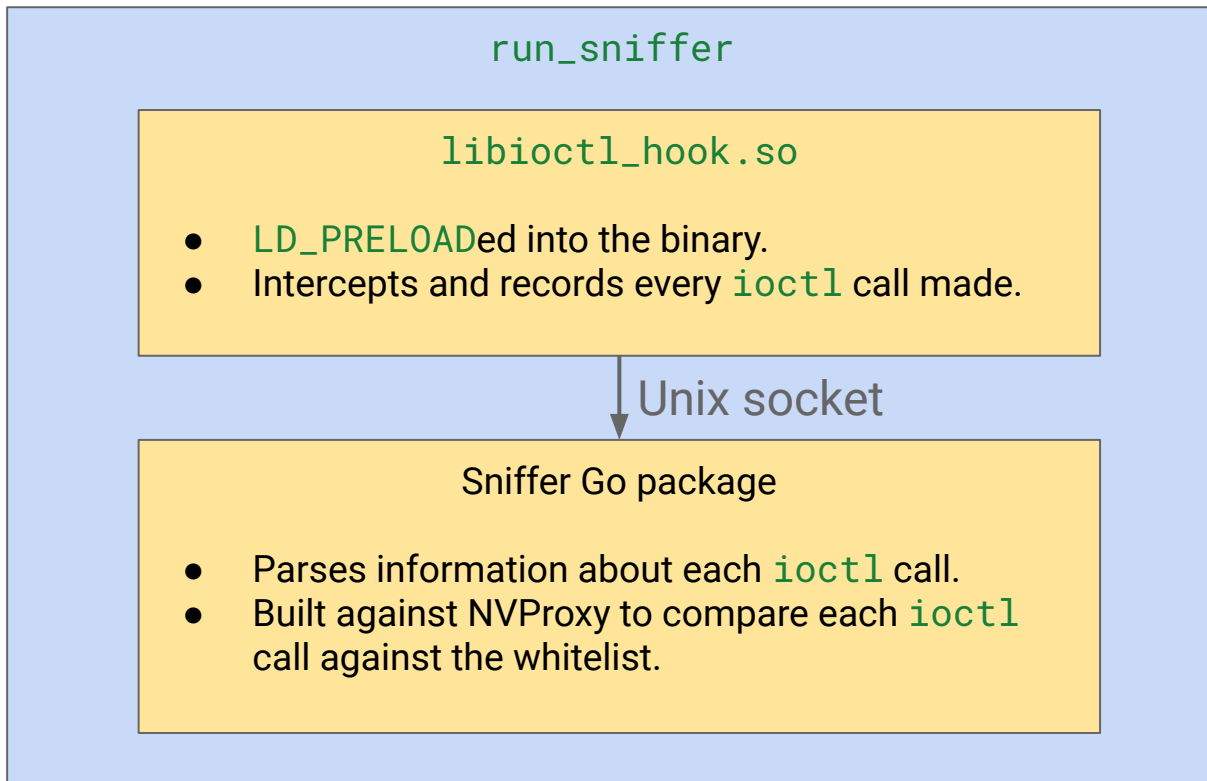
```
Alloc ioctl: request=0xc030462b [nr=NVIDIA_ESC_RM_ALLOC, hClass=0xc640] => ret=0
```

```
Alloc ioctl: request=0xc030462b [nr=NVIDIA_ESC_RM_ALLOC, hClass=0x73] => ret=0
```

```
Alloc ioctl: request=0xc030462b [nr=NVIDIA_ESC_RM_ALLOC, hClass=0x208f] => ret=0
```

```
Unknown: None
```

# How does it work?



## Replacing libc's `ioctl` method

- `LD_PRELOAD` allows us to replace libc's implementation of `ioctl` with our own.
- Our version looks something like this:
  1. Get a handle to libc's `ioctl` method.
  2. Proxy the `ioctl` call to libc without any interference.
  3. Check if this call was to a device file with the name `/dev/nvidia*`.
  4. Send information to the Go package.

# Bridging the two components

- `ioctl` information is encoded in a protobuf, which is sent over a Unix socket.
  - Easily extendable in the future.
- Supports concurrent connections on the socket.
  - `libioctl_hook.so` opens a new socket connection for each application thread.

```
message Ioctl {  
    // The path of the file that `fd` is pointing  
    to.  
    string fd_path = 1;  
  
    // The request argument of the ioctl.  
    uint64 request = 2;  
  
    // The return value of the ioctl.  
    int32 ret = 3;  
  
    // The data pointed to by `argp`. For UVM  
    ioctl calls, the argument size is  
    // not easily accessible, so `arg_data` will  
    be empty in this case.  
    bytes arg_data = 4;  
}
```

## Comparing against NVProxy

- The Go component receives `ioctl` information over the socket and compares against NVProxy's whitelist.
- Requires parsing `arg_data` for control commands and alloc classes.



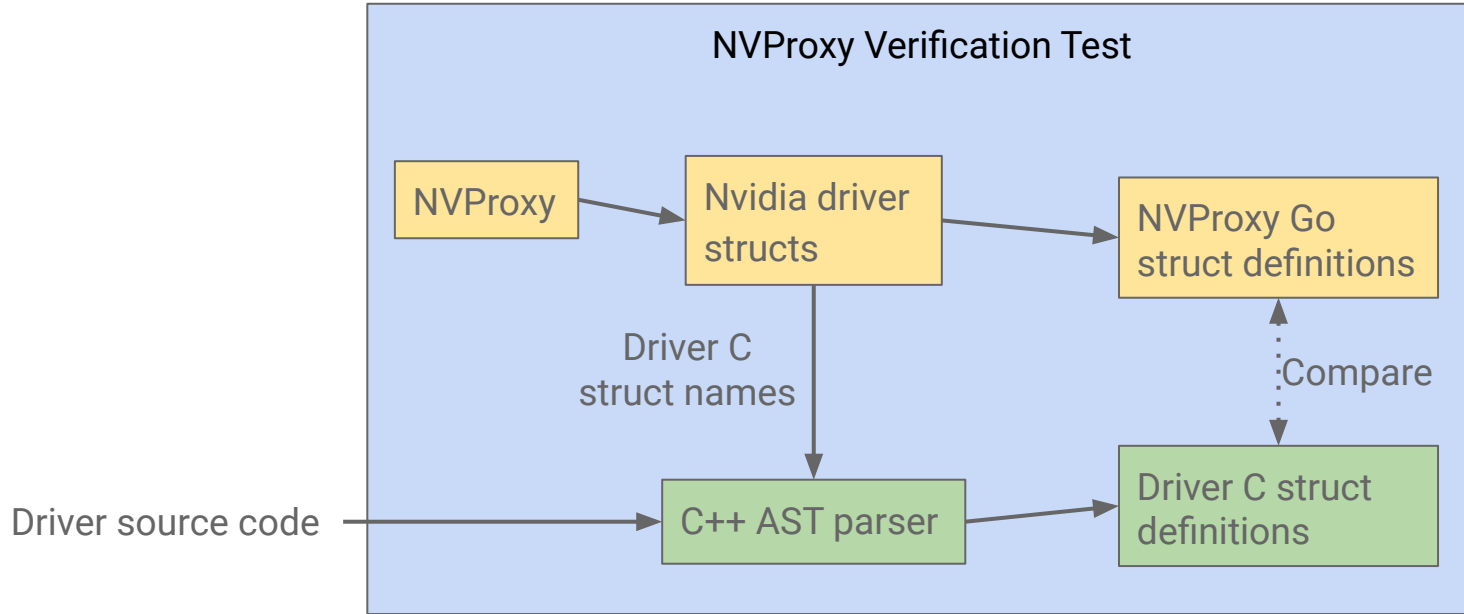
## Tool 2: Maintaining parity with driver updates

# Overview

- Each driver update may carry updates to `ioctl` arguments that we need to investigate.
  - E.g. Adding new fields, changing a field to a pointer, etc.
- This tool parses the Nvidia driver source code to get struct definitions for `ioctl` arguments.
- Can then **verify** NVProxy's definitions, or **compare** definitions between updates.

# Tool 2.1: Verifying NVProxy

# How does it work?



# Getting struct dependencies in NVProxy

- Augment existing system for representing the driver's ABI
  - Keep the structure and definitions alike, so it is simpler to understand and maintain
- To save memory, we only generate when **getStructNames** is called.

```
type driverABI struct {  
    frontendIoctl  map[uint32]frontendIoctlHandler  
    uvmIoctl       map[uint32]uvmIoctlHandler  
    controlCmd     map[uint32]controlCmdHandler  
    allocationClass map[nvgpu.ClassID]allocationClassHandler  
    getStructNames driverStructNamesFunc  
}
```

```
type DriverStruct struct {  
    Name DriverStructName  
    Type reflect.Type  
}
```

```
func() *driverStructNames {  
    return &driverStructNames{  
        frontendNames: map[uint32][]DriverStruct{  
            nvgpu.NV_ESC_CARD_INFO:      simpleIoctl("nv_ioctl_card_info_t"),  
            nvgpu.NV_ESC_CHECK_VERSION_STR: getStructName(nvgpu.RMAPIVersion{}),  
            nvgpu.NV_ESC_ATTACH_GPUS_TO_FB: nil, // NvU32 array containing GPU IDs  
            // ...  
        },  
        // ...  
    }  
}
```

# Getting struct dependencies in NVProxy

- NVProxy struct names don't always match the driver names.
- Add tags to NVProxy structs with their driver names.
- Use this design for a few reasons:
  - Cannot directly use NVProxy struct name, may differ due to versioning.
  - Struct comments like `// +marshal` require compile-time code generation.
  - Can read this tag at runtime using `reflect`.

```
type UVM_SET_PREFERRED_LOCATION_PARAMS struct {  
    RequestedBase    uint64 `nvproxy:"same"`  
    Length           uint64  
    PreferredLocation NvUUID  
    RMStatus         uint32  
    Pad0             [4]byte  
}
```

```
type UVM_SET_PREFERRED_LOCATION_PARAMS_V550 struct {  
    RequestedBase    uint64  
    `nvproxy:"UVM_SET_PREFERRED_LOCATION_PARAMS"`  
    Length           uint64  
    PreferredLocation NvUUID  
    PreferredCPUNumaNode int32  
    RMStatus         uint32  
}
```

# C++ AST Parser

- Parse the driver source code using Clang's AST Matcher API.
- Takes a list of struct names as input, outputs their definitions.

```
typedef int OtherInt;

typedef struct TestStruct {
    int a;
    int b;
    struct {
        OtherInt c;
        OtherInt d;
    } e[4];
    TestUnion f;
} TestStruct;
```

Input:

```
{
  "structs": ["TestStruct"]
}
```

Output:

```
{
  "records": {
    "TestStruct": {
      "fields": [
        {"name": "a", "type": "int", "offset": 0},
        {"name": "b", "type": "int", "offset": 4},
        {"name": "e", "type": "TestStruct::e_t[4]", "offset": 8},
        {"name": "f", "type": "TestUnion", "offset": 40},
      ],
      "size": 44,
      "is_union": false,
      "source": "test_struct.cc:25:16"
    },
    ...
  },
  "aliases": {
    "OtherInt": {"type": "int", "size": 4}
  }
}
```

# Clang's AST Matcher: Finding struct definitions

- API sets up a C++ tool that takes in a source file, generates an AST, and searches that AST against a matcher expression.
- We can note that every struct is named via a **typedef**, use a matcher expression for **typedefs** to a given name.

```
typedef struct NV00FD_CTRL_GET_INFO_PARAMS {  
    NvU64 alignment;  
    NvU64 allocSize;  
    NvU32 pageSize;  
    NvU32 numMaxGpus;  
    NvU32 numAttachedGpus;  
} NV00FD_CTRL_GET_INFO_PARAMS;
```



# Clang's AST Matcher: Finding struct definitions

- Expression gives us a **RecordDecl** node that represents the struct definition.
- Can iterate through its **FieldDecls** to get the name, type, and offset of each field.

```
CXXRecordDecl 0x604d2c866628 <./550.54.14/src/common/sdk/nvidia/inc/ctrl/ctrl00fd.h:79:9, line:85:1>
line:79:16 struct NV00FD_CTRL_GET_INFO_PARAMS definition
|-FieldDecl 0x604d2c866808 <line:80:24, col:30> col:30 alignment 'NvU64': 'unsigned long long'
|-FieldDecl 0x604d2c866910 <line:81:24, col:30> col:30 allocSize 'NvU64': 'unsigned long long'
|-FieldDecl 0x604d2c8669f8 <line:82:5, col:11> col:11 pageSize 'NvU32': 'unsigned int'
|-FieldDecl 0x604d2c866a58 <line:83:5, col:11> col:11 numMaxGpus 'NvU32': 'unsigned int'
`-FieldDecl 0x604d2c866ab8 <line:84:5, col:11> col:11 numAttachedGpus 'NvU32': 'unsigned int'
```

# Clang's AST Matcher: Nested records

- Arguments can have other structs or unions nested as fields.
  - Structs or unions are collectively called “records”
- Recurse on the type of each **FieldDecl**.
- Even if the type is not a record, it is likely an alias of some base type. We also note these aliases that we find, in case they change.
  - E.g. **NvV32** aliases **unsigned int**.

```
typedef struct
{
    NVOS02_PARAMETERS params;
    int fd;
} nv_ioctl_nvos02_parameters_with_fd;
```

```
typedef struct
{
    NvHandle    hRoot;
    NvHandle    hObjectParent;
    NvHandle    hObjectNew;
    NvV32       hClass;
    NvV32       flags;
    NvP64       pMemory;
    NvU64       limit;
    NvV32       status;
} NVOS02_PARAMETERS;
```

# Clang's AST Matcher: Anonymous records

- Sometimes, we may have an anonymous nested record.
- Clang's auto-generated name includes the absolute path of the file, which we cannot use.
- We generate our own name instead, of the form **PARENT\_RECORD::FIELD\_t**.

```
typedef struct NV2080_CTRL_GPU_GET_NAME_STRING_PARAMS {  
    NvU32 gpuNameStringFlags;  
    union {  
        NvU8  ascii[NV2080_GPU_MAX_NAME_STRING_LENGTH];  
        NvU16 unicode[NV2080_GPU_MAX_NAME_STRING_LENGTH];  
    } gpuNameString;  
} NV2080_CTRL_GPU_GET_NAME_STRING_PARAMS;
```



**NV2080\_CTRL\_GPU\_GET\_NAME\_STRING\_PARAMS::gpuNameString\_t**

# Clang's AST Matcher: Array types

- We add `[ARRAY_LEN]` to the end of the base type name, and use that as the field's type.
- Recurse on the base type of the array (which could be an anonymous record).

```
typedef struct
{
    NvU64                                base;
    NvU64                                length;
    UvmGpuMappingAttributes perGpuAttributes[UVM_MAX_GPUS];
    NvU64                                gpuAttributesCount;
    NV_STATUS                            rmStatus;
} UVM_ALLOC_SEMAPHORE_POOL_PARAMS;
```



`UvmGpuMappingAttributes[256] perGpuAttributes`

# Clang's AST Matcher: Top-level **typedefs**

- Some structs are defined as a **typedef** of a pre-existing struct.
- We copy the definition of the **typedef**'d type.

```
typedef NV906F_CTRL_GET_CLASS_ENGINEID_PARAMS NVC36F_CTRL_GET_CLASS_ENGINEID_PARAMS;
```

# Comparing definitions

- First want to flatten every struct definition out, simplifying any nested structs.

```
typedef struct
{
    NVOS02_PARAMETERS params;
    int fd;
} nv_ioctl_nvos02_parameters_with_fd;
```

```
typedef struct
{
    NvHandle    hRoot;
    NvHandle    hObjectParent;
    NvHandle    hObjectNew;
    NvV32       hClass;
    NvV32       flags;
    NvP64       pMemory;
    NvU64       limit;
    NvV32       status;
} NVOS02_PARAMETERS;
```



```
typedef struct
{
    NvHandle    hRoot;
    NvHandle    hObjectParent;
    NvHandle    hObjectNew;
    NvV32       hClass;
    NvV32       flags;
    NvP64       pMemory;
    NvU64       limit;
    NvV32       status;
    int fd;
} nv_ioctl_nvos02_parameters_with_fd;
```

# Comparing definitions: simple `ioctl`s

- Case 1: an `ioctl` is treated as simple by NVProxy
  - Want to verify that it is a simple `ioctl`.
  - Do this by checking if:
    - A field has type `NvP64`.
    - A field name ends in `fd`.

```
type NV0000_CTRL_GPU_GET_ID_INFO_PARAMS struct {  
    GpuID          uint32 `nvproxy:"same"`  
    GpuFlags       uint32  
    DeviceInstance uint32  
    SubDeviceInstance uint32  
    SzName         P64  
    SliStatus      uint32  
    BoardID        uint32  
    GpuInstance    uint32  
    NumaID         int32  
}
```

# Comparing definitions: aliases

- Case 2: NVProxy has a definition, but the driver uses an alias.
  - This is a edge case that only applies to `NvHandle`.
  - We currently compare the size of the two definitions.

```
type Handle struct {  
    Val uint32 `nvproxy:"NvHandle"`  
}
```

```
#ifdef NV_TYPESAFE_HANDLES  
typedef struct  
{  
    NvU32 val;  
} NvHandle;  
#else  
typedef NvU32 NvHandle;  
#endif // NV_TYPESAFE_HANDLES
```



# Comparing definitions: matching struct definitions

- Case 3: Both NVProxy and the driver have struct definitions.
  - NVProxy may have padding fields; need to remove these first.
  - Look for matching offsets when comparing fields.
  - Want to compare the types of each matching field.
    - Base type -> base type
    - Enums -> `uint32`
    - Unions -> `[n]byte`
    - Arrays -> match length and base type

```
type NV0041_CTRL_GET_SURFACE_INFO_PARAMS struct {  
0:      SurfaceInfoListSize  uint32 `nvproxy:"same"`  
4:      Pad                  [4]byte  
8:      SurfaceInfoList      P64  
}
```

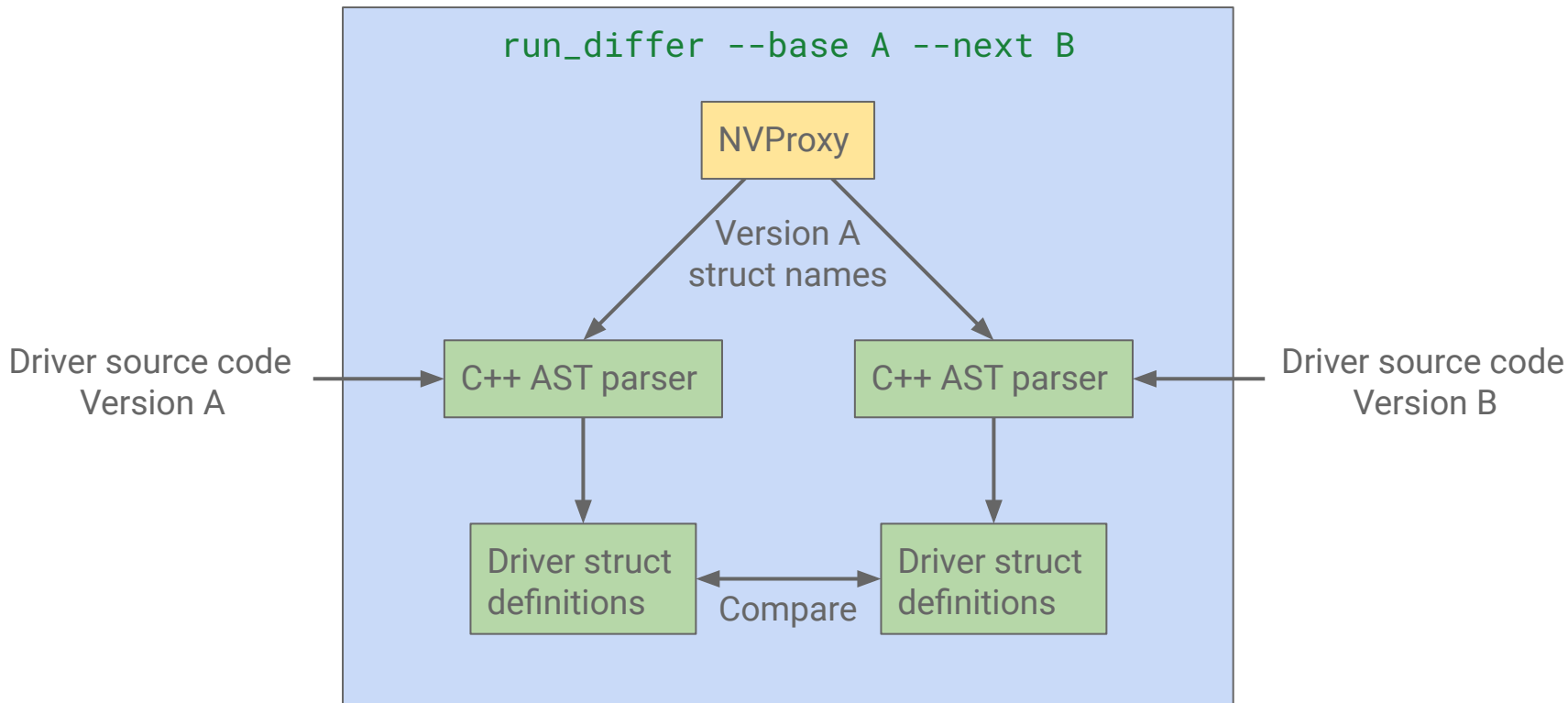
```
typedef struct NV0041_CTRL_GET_SURFACE_INFO_PARAMS  
{  
0:      NvU32 surfaceInfoListSize;  
8:      NvP64 surfaceInfoList;  
} NV0041_CTRL_GET_SURFACE_INFO_PARAMS;
```

# Putting it all together

1. Get list of struct names
  - Saved to a temporary JSON file.
2. Clone driver source code
  - `git clone` to a temporary directory
3. Create source files for Clang
  - Use `#include` for every file that includes argument structs.
  - Paths are hard-coded for now.
4. Create `compile_commands.json`
  - Gives Clang the include directories.
5. Run the C++ AST parser on the source files.
6. Compare driver definitions against NVProxy definitions

## Tool 2.2: Diffing driver versions

# How does it work?



# Results

# Results

- Used sniffer to create stronger regression tests that fail if **any** unsupported `ioctl` call is made.
  - Currently just in smoke tests, but planning to integrate into some more tests before the end.
- Running the differ between every driver version in NVProxy confirms that we have captured every update.
- Verification test caught some discrepancies between NVProxy and Nvidia driver definitions.

# Future Work

# Future Work

- Incorporate the sniffer into all GPU regression tests.
  - Held back by weird seg-fault bug when running Xvfb through the sniffer.
- Use driver differ to enable support for driver version **ranges**.
  - Etienne's proposal: <https://github.com/google/gvisor/issues/10628>
- Combine the C++ parser with the sniffer to identify unsupported **ioctl**s.
- Automatic code generation for NVProxy using the C++ parser.



# Thank You

