I put my testing all over the report with testing/known bugs.

To start off this project, the initial struggle was being able to change the config file correctly. I updated the config file from the one shown in canvas, and after I did so, I made my kernel, powered off my device, and when I tried to run my snapshot again, the kernel broke (stuck on ANDROID x86_64 # screen). I even deleted a few snapshots that just failed. Thankfully, that only happened once, but for an entire day I suffered from lots of rebooting and the "./chompread" command not being able to execute, as it kept showing "usbip err: vhci_attach.c : 356 (attach device) tsp connect" when I did the usbip command. The chompread command said "cannot connect to device." Much later, I realized that using the "ln -T -s name-4.9 name" was able to correctly place the symbolic link relating to the new config file, and I got the usbip command to function properly, as well as chompread's output displaying the current devices' info. After this feat of a scenario, I wrote my code within the chompdrv dir in userspace and had my Makefile and chompdrv.c there.

Moreover, to appropriately connect with libsub in my code, I had to include the <libusb-1.0/libusb.h> header. Also, I originally didn't know that our kernel was a bit older, so that I had to use uinput_user_dev uud;. It took me a lot of compilation errors and stackoverflow forums to figure this small issue out. As an analogy, the usb ip software is "like the client" and so when we run it, it connects to controller, and forwards all the data to usb subsystem (like plugging usb into laptop). Hence, the controller needs to be run first to be able to manage data by ./chompdrv and testing with jstest. When a user presses a button, the raw data is published on the virtual USB port. The next task is to parse the bytes, and when that is done, they are interpreted (to determine the current device state) and that event is published on a device known to the operating system as a joystick (going from USB device to virtual joystick). In other words, the data is sent back to input system of Linux in what it understands is a conventional joystick. It is understood that the USB parlance is no doubt a device that is interrupt-driven with one endpoint (thankfully, ./chompread validly proved this when I ran it – later, so did ./chompdrv). With that said, there is only one communications channel, and so when it occurs that a request is sent to device, it goes through here. The byte of data sent when USB subsystem polls device is implemented in an update function, so that left/right, up/down, and invalid values of X or Y are documented adaptively. I got the configuration, interface, and endpoint attributes of very device, and after changing dev/uinput, wrote the changes to the joystick device. We read from the one-directional endpoint like a pipe.
A big issue I had was that of my driver making event7 instead of js0 (one time it made both). The solution was not just using the uinput functions from the "old interface," but it was in setting the events for the axes (X and Y) correctly. When moving C++ code into C, at first it was difficult & I had boundless compilation errors, but I quickly realized that by changing the cout to printf and declaring objects structs where necessary, the code ran fine. Also, testing the driver through the GatorRaider game through the Android GUI also proved to be quite a challenge, as it had insane lag and slow reflexes, but thankfully the Gator game was a fun little way to demonstrate the efficiency of the driver, and it worked after long periods of time.

I needed to create a variable to store data one byte at a time to read instead of write (instead of writing fake data to device). Furthermore, I knew that I had to construct an endless loop getting the data, and if it changed (a.k.a. the user pressed a button) then that changed data is updated and regarded. It is also important to note that I included a toggling option via EV_ABS so that when I press right, for example, three times, it would toggle from "on" to "off" to "on" instead of stay "on" the entire time (which would prohibit me from turning it off later). After much research, it turns out that ui_set_absbit relates to setting the absolute axes we will use.
Also, I knew that I needed a while loop that always ran as to always poll for updates on the driver and make sure that the joystick (as seen by the OS) functioned properly, and so if buttons were changed, they were represented (in our case, tested in "real-time" by jstest js0 in proper dir). A good way to test jstest was to do jstest –event js0 by just going into /dev/input/. Being that the driver was successfully connected, I saw information about its buttons and axes that updated as I changed the buttons that were pressed.

Going further, we can specify that every USB device is operated with a libusb_device & libusb_device_handle objects in libusb. Even more, it is the case that the libusb API will tie an open device to a specific interface, so that a user will use libusb_claim_interface (before you operate on device) to claim many interfaces on a device and receive a libusb_dev_handle to communicate with each one. Firstly, we can make a session & call libusb_init (init of lib). To get device list that are recognized, libusb_get_device_list gets the job done. Looping through, we see a device by libusb_open() (or with vid/pid), then clear the list via libsub_free_device_list (unreferences all devices in recognized list and then frees list). It is essential to unreference device after the user in done with it, because if done before, it might harm or demolish it. Towards the end, we libsub_release_interface (when done with device), libusb_close (any opened devices), and effectively end the libusb_exit session.
Moreover, we can't claim a device if kernel links to it (libusb_kernel_driver_active returns 1 meaning kernel attached a driver to device), so we use libusb_detach_kernel_driver to take off kernel from device.