AR.Pwn: Hacking the Parrot AR.Drone (Part 2)
Jeremy Rand
Team SNARC (Sooners / Norman Advanced Robotics Coalition)
jeremy.rand@ou.edu

AR.Pwn: Hacking the Parrot AR.Drone (Part 2)

11 Welcome to Part 2

Welcome back. Part 2 continues where Part 1 left off, so if you haven't read Part 1 yet, you should do so before continuing. And now, let's continue!

12 Color Spaces

The AR.Drone's cameras return an image in YUV color space, which separates brightness data (Y) from chroma data (UV). Y corresponds to the perceived brightness of a pixel, while the UV components correspond to how that brightness is spread between red, green, and blue. YUV is useful in image processing because both humans and color trackers treat brightness of pixels very differently from how they treat chroma of pixels.

In humans' case, the human eye has a very high resolution for the Y components, but very low resolution for the UV components. For this reason, most color televisions and cameras compress the UV resolution heavily to make room for a higher Y resolution. The AR.Drone's cameras feature 4 Y subpixels (2×2) for each pair of UV subpixels.

Color trackers also benefit from YUV. In RGB color space, all of the components are affected by lighting, but in YUV space, a red object will have a high V component and a low U component, independently of the brightness of the lighting (which will in ideal conditions only affect the Y component). In fact, YUV color tracking was used in Botball in 2003 and 2004 (the CMUCam). YUV tracking was scrapped from Botball in 2005 with the release of the XBCCam, which used HSV color space instead. HSV, which is still used in Botball to this day, is somewhat more robust and intuitive than YUV, but also requires CPU power to convert since cameras generally do not produce HSV-formatted data.

Michael Kohn has created an excellent web-based tool for converting between RGB and YUV color spaces [5].

We had to choose between doing color tracking in YUV versus doing it in HSV. We initially planned to convert to HSV for each frame, but halfway through implementing this, we decided to try directly using YUV. The AR.Drone's CPU is heavily loaded by the flight controller, so any potential CPU usage optimizations would be highly desirable. We ended up sticking with YUV.

13 Vision Processing

We decided to use Matthew Thompson's mb_vision library for blob tracking, because it is simple to build (just a few C++ files, no dependencies), it works reasonably well, it has relatively low CPU usage, and we were already familiar with it. We seriously considered OpenCV, but decided against it because it is more difficult to build, we hadn't used it as much, and we didn't need any of the advanced features which it offered.

mb_vision, however, was designed for RGB images, which it converted in real-time to HSV. Since we had decided to attempt YUV color tracking, this needed some tweaking. We were able to modify mb_vision to handle YUV tracking quite easily (the hardest part was handling different resolutions for the brightness and chroma data; we ended up using the lower chroma resolution, since color tracking mainly cares about chroma).

After making these modifications, we were able to run mb_vision pretty much verbatim and do blob tracking on both the front and vertical cameras. Unfortunately, as of this writing, changing the color models requires compiling and downloading new code to the AR.Drone and rerunning the vision app (luckily, rebooting `program.elf` is not necessary). We are hoping to improve this, by using variable files in `/tmp/` which control the vision parameters. These could be set remotely using telnet commands, without downloading new code.

14 Data Retrieval

We initially tried to use `printf` over a terminal to output the color tracking data. Unfortunately, terminals are slow, TCP is slow, and printf is slow. This used an obscene amount of CPU time, and prevented us from getting an acceptable framerate. We eventually decided to use a UDP/IP protocol which transmits an array of blob data structs each frame. We attempted to send the data to the broadcast IP of `255.255.255.255`, but the data didn't show up in our receiver. (Maybe we did something wrong?) We instead sent data to `192.168.1.2`, which is the IP assigned by the Drone to the device controlling it (e.g. a CBC, Link, or PC), and this worked acceptably.

For our test, we used a color model for lime green, and pointed the Drone's front camera at a green target from the 2011 AAV Contest, and immediately the PC running the UDP receive client showed an object. Unfortunately, we were only getting about 2fps. We then tried using the `-O3` flag (which fully optimizes code for speed) while compiling the Drone vision code, but got a dynamic linking error (apparently the standard C library in our CodeSourcery install wasn't the same version as the one on the Drone, which caused Bad Things). The same error showed up with the `-O2` flag (which optimizes a bit less), but `-O1` (which optimizes a little bit) worked. The use of `-O1` bumped our framerate to 4fps. Just on a whim, we tried compiling with `-O3` but also with static linking. The rationale was that since statically linked programs don't care about standard C library versions, we might be able to bypass the linking issue. And sure enough, that worked, bringing AR.Pwn's framerate to 9fps, a benchmark result that continued to hold true while accessing the data from a CBC.

The latency of the data was not quantitatively measured, but appeared to be less than 250ms. As of this writing, the official CBC libraries are getting about 10 seconds of latency, and the Link libraries are getting about 1.5 seconds of latency. The difference between AR.Pwn and the official KIPR code may make the difference between being able to quickly adjust to camera data versus having to rely on accelerometer data for quick maneuvers.

15 Modifying the Video Data

Once we've performed some processing on the video data, it may be useful to write a modified version back to program.elf's buffers. For example, replacing all pixels which match a color model with a single color would make color model adjustment much easier, since the color model's effectiveness could be inspected on any device which can receive video from the AR.Drone (e.g. an Android phone).

Doing this is relatively easy. Right after the AR.Pwn hook program finishes its implementation of `DQBUF`, we can write to the buffer in addition to reading from it, and `program.elf` will be fooled into accepting the modified image.

We have made modifications to mb_vision for this purpose: when enabled to do so, it will replace all matched pixels in the original YUV image with a bright blue color. The AR.Pwn vision app will then write the modified image to a file, where the AR.Pwn hook program can send it to the AR.Drone's flight controller. It should be noted that doing this **will** cause decreased performance, dropped frames, and/or unstable flight, so do not use this feature during flight. It should only be used for initial calibration of color models.

16 Using the Code

Our code will be released at GCER 2013. Note that this code is considered beta code, and should not be considered complete. There's almost no documentation outside of this paper, and you'll need to edit source code at times. (But since you're reading a hacking paper, you probably expected this.)

The AR.Pwn hook code is located in `hook.c`. To build it, run the `build_hook.sh` script on a PC which has an ARM-Linux compiler [6] installed. We've also provided a pre-built version, `libhook.so`.

The vision code is in the "AR.Pwn Telnet Interface" project folder. This should build with Code::Blocks if you have an ARM-Linux compiler installed. It has two targets of note: Release and ReleaseBlueMarker. These will produce files called `TelnetInterface` and TelnetInterfaceBlueMarker, respectively.

To install the code, transfer `libhook.so`, `TelnetInterface`, `TelnetInterfaceBlueMarker`, and `run_hook.sh` to the AR.Drone using FTP. We placed it in a new subfolder of the FTP root, which we called "`snarc`", because we weren't sure if the AR.Drone would be happy about having files sitting where firmware updates typically go. Then log in via Telnet and copy `/bin/program.elf` to `/update/snarc/program_backup.elf`. Lastly, as a safety measure, run the `sync` command. (Readers unfamiliar with FTP or Telnet are advised to consult Google. Readers who are unable to figure it out via Google are advised that hacking may not be for them.)

Now that the code is installed, to run the code, login via Telnet, and navigate to `/update/snarc/`. To hook the flight controller, run the following command:

```
./run_hook.sh
```

The LED's will turn red, and then should turn green after a short time. Once the LED's are green, you'll need to open a 2nd Telnet session, and set up the vision parameters using one or more of these commands:

For the front camera's color models 0 through 3:

```
touch /tmp/video0_enable_0
touch /tmp/video0_enable_1
touch /tmp/video0_enable_2
touch /tmp/video0_enable_3
```

And for the vertical camera:

```
touch /tmp/video1_enable_0
touch /tmp/video1_enable_1
touch /tmp/video1_enable_2
touch /tmp/video1_enable_3
```

And the last step:

```
./TelnetInterface &
```

The vision algorithm will now be performing blob tracking on the color models whose enable flags you `touch`ed. Remember, right now you'll need to recompile `TelnetInterface` if you want to change the color models (`telnet_interface.cpp`); we're working on improving this. Also, you will need to re-run the process starting with `./run_hook.sh` after you reboot the AR.Drone. While it is theoretically possible to make these steps happen automatically when the AR.Drone is booted, we considered this too potentially dangerous to attempt.

If you use `TelnetInterfaceBlueMarker` instead of `TelnetInterface`, any video streamed by the AR.Drone will show matched pixels as blue (you can change this color in the source code, it's in the `getSegments` method of `BlobImageProcessorYUV.cpp`). Because the cameras supply frames faster than the blob tracker, only some of the frames will have this post-processing enabled (so the blue pixels will appear to flash). We've only tested this feature with the Android app; we're not certain how well it will work with the CBC or Link libraries.

To access blob tracking data from a CBC, look at the example code in the CBC folder. A similar example is included for the Link (in the Link folder). There's also an example Cygwin app for reading it from a PC (compilable with Code::Blocks, in the "Ar.Pwn Reader" folder).

Since it is not advisable to have to enter a bunch of terminal commands when setting up your robots, the Link example will automatically use the Telnet client on your Link to hook the firmware and begin blob tracking. The CBC does not include a Telnet client, so we did not include any such code for the CBC. (A competent hacker might be able to make it work on the CBC... any takers?)

17 SOCKS Proxies and Security

The AR.Drone supports a MAC pairing feature to prevent unauthorized access. Failure to activate this feature will permit any device to connect to the AR.Drone's network, including the Telnet daemon, and do all sorts of evil things, ranging from hijacking your Drone, to making it fall out of the sky, or even wiping the filesystem. It should go without saying that **this is a Very Bad Thing**, and as a result you should **never** operate your AR.Drone at a competition venue without the MAC pairing feature activated. The problem is that the AR.Drone can only pair with one device at a time, which would normally be a CBC or Link. In the event that you need to access your AR.Drone's FTP or Telnet services from a PC for emergency hacking purposes while at a tournament venue, **do not** unpair the AR.Drone to do this, as you will be opening the AR.Drone to malicious traffic.

There is a better way. You can still connect the PC to the AR.Drone's network, as long as you give the PC a static IP address rather than using DHCP assignment. The AR.Drone will refuse connections from your PC, since it can see that the PC is not paired, but the PC can still talk to the CBC or Link. What you can do is use the SSH daemon on the CBC or Link to setup a SOCKS tunnel, which will allow the PC to proxy its Telnet and FTP connections through the CBC/Link. The AR.Drone will see the connection as coming from the CBC/Link, so you will be able to connect. Obviously, you should make sure that the SSH connection is secured. By default, the CBC and Link have no password for SSH, so you should set one yourself (preferably as a user other than root). Even better, use public key authentication rather than a password.

The networking concepts discussed above are out of the scope of an AR.Drone hacking paper. Readers unfamiliar with these concepts are advised to use Google. Readers who are unable to find this information with Google are advised that perhaps hacking is not for them.

18 Conclusion

AR.Drone hacking is new to the Botball hacking scene, so there are a lot of things we weren't able to cover in this paper (mainly because we're still figuring them out). If you'd like to get involved and hack some cool stuff, drop by the Botball Community [7] and say hello. (The Botball hacking scene is really small these days; it'd be awesome to get some new hackers involved!) If you spot any bugs in our code, the Botball Community is also the preferred way to reach us. (However, remember the disclaimer: there's no warranty implied with our code.)

Happy Hacking!

References

[5] M. Kohn. YUV RGB Javascript Converter. http://www.mikekohn.net/file_formats/yuv_rgb_converter.php , Retrieved 2013.
[6] Mentor Graphics. Sourcery CodeBench Lite Edition for ARM GNU/Linux. http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/arm-gnu-linux , 2013.
[7] Botball Youth Advisory Council. Botball Community. http://community.botball.org , 2013.