Locked & Loaded: An Aggressive Smart Lock

Quinn Johnson
UC Berkeley
204 Cory Hall; Berkeley, CA 94704
quinnhj@berkeley.edu

Harrison Tsai
UC Berkeley
204 Cory Hall; Berkeley, CA 94704
harrison.tsai@berkeley.edu

David Wu
UC Berkeley
204 Cory Hall; Berkeley, CA 94704
wu.david@berkeley.edu

INTRODUCTION

Our EECS C149 class project is a retaliatory smart lock. We model the lock and retaliation attempts as a state machine governed by a combination of sensor inputs, which focus on the security of the owner, as well as strategic offense against intruders. Shown below is a prototype of the system.

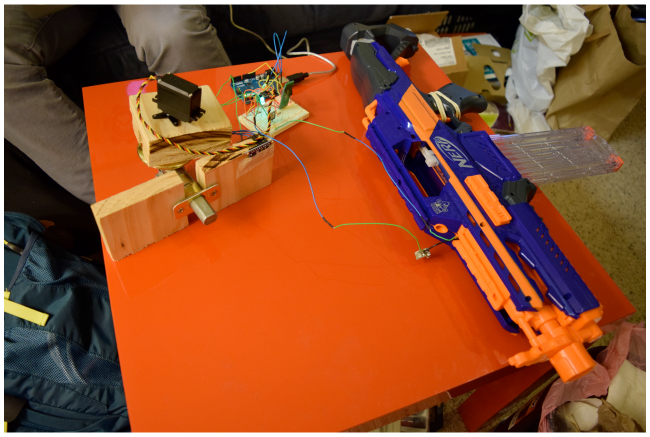

Figure 1. The Locked and Loaded Setup.

This project focused on two key features: First, a convenient keyless entry; Second, intruder retaliation. The first feature uses an android app that sends a password through Bluetooth to the system, and the second feature utilizes an IMU, IR sensor, and Nerf gun.

RESOURCES

- Arduino Uno
- Nexus 7
- BlueSMiRF Bluetooth Module
- Futaba S3003 Servo Motor
- TSOP38238 IR Receiver
- IR204-A IR Transmitter
- Adafruit 9-DOF IMU
- Nerf N-Strike Elite Rapidstrike CS-18 Blaster

Additional resources include a variety of resistors and a TIP120 transitor.

MODEL

Our system is designed as a hierarchical state machine. A high level view of the state machine is shown below. For formatting purposes, we have omitted the guard/action transitions in the diagrams.

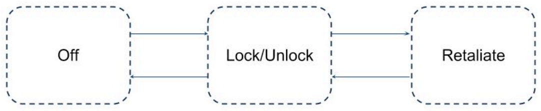

Figure 2. Hierarchical state machine.

As shown above, our finite state machine has three primary states. The "off" state describes a state where the system is unresponsive to any inputs except for a user command to put it into a "Lock" state. Shown below is the "Lock/Unlock" state.

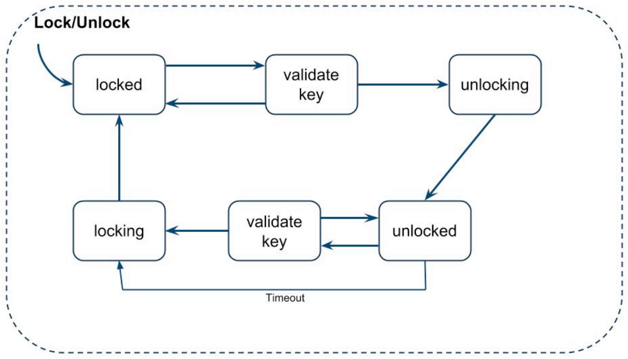

Figure 3. Lock/Unlock FSM.

The "Lock/Unlock" state describes the actions of locking and unlocking the door. We start by always entering into a locked state. In order to unlock the door, we validate a password sent by the Nexus 7 over Bluetooth. Once unlocked, the user may send a lock command to transition the system back to the locked state. If no input is given after a predetermined timeout threshold, the system will automatically transition back into the lock state.

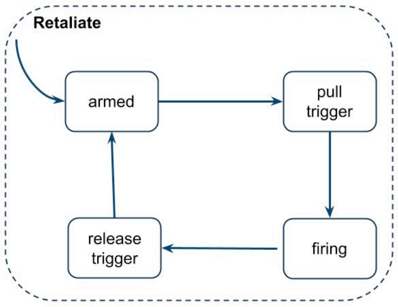

Figure 4. Retaliate FSM.

The only way to enter the "Retaliate" state is to trigger an "intrusion". We define an intrusion to be when someone attempts to "break down" your door. We measure this with the accelerometer and an infrared sensor. There must be sufficient movement on the door, measured by taking the magnitude of the accelerometer, and a break in the IR sensor for an intrusion. The "Retaliate" state is straightforward – we simply fire the Nerf gun for a predetermined amount of time, then move back into a ready "armed" state. This armed state simply checks if the system should fire again, or if it should move back into the safe "locked" state.

DESIGN

Hardware

We chose the Arduino platform for its simplicity and ease to quickly develop prototypes. All of the hardware, excluding the Nerf gun, we select integrates with the Arduino platform natively. To integrate the Nerf gun, we re-wired the gun so we could electronically control its fire. Additionally, we selected the servo motor based on the torque required to turn our home and model lock. We utilized a torque wrench to measure the torque required to turn these locks, and found that a 4.1 kg-cm servo motor was more than sufficient for the requirement. The hardware configuration can be seen in the image below:

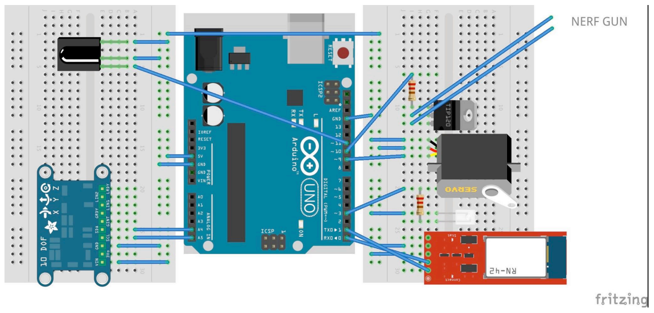

Figure 5. The schematic diagram of the project.

In addition, a 9-DOF is mounted on the model of a door frame to detect intrusion. The serial clock line (SCL) and (SDA) pins on the IMU are part of the I2C protocol for components to communicate with each other. SCL and SDA are connected to two analog pins. In this project, pin A4 on the Arduino is connected with SDA and A5 is connected with SCL. The readings are initially read as raw data, but with the help of libraries, it can be transformed to standard units for acceleration: m/s2. For this project, we are using the magnitude of the acceleration recorded.

$$\|a\| = \sqrt{{a_x}^2 + {a_y}^2 + {a_z}^2}$$

Figure 6. The magnitude of acceleration.

The infrared transmitter is an LED that emits light at 940nm wavelength, invisible to the human eye.

Software – Arduino

The Arduino code followed a similar pattern as the Hill Climb C code done during lab. We defined each state using if-else blocks in the general Arduino loop method. A snippet of these blocks is shown below:

```
if (lockState == locked &&
Serial.available()) {
    validated = false;
    lockState = validateUnlock;
} else if (lockState == unlocking &&
lockServo.read() == SERVO_UNLOCK) {
    lockState = unlocked;
    unlockTime = millis();
```

Figure 7. A snippet of the transitions in the Arduino code.

Our system enables any Bluetooth device to connect to it, and requires password validation to interact with the system. This emphasizes the real time behavior of our system since a command can be passed into the system at any time. An example of the Bluetooth code is shown below:

```
while (Serial.available()) {
    char received = Serial.read();
    if (received == ',') {
        validateKey(dataFromBT, "unlocked") ?
lockState = locking : lockState = unlocked;
        dataFromBT = "";
        break;
    }
    dataFromBT.concat(String(received));
}
```

Figure 8. A snippet of the Bluetooth processing in the Arduino code.

The bulk of the logic resides in the Arduino code since it controls the various defined states. The code itself is relatively straightforward since it simply follows the state machine; however, the majority of the guards are very sensitive to thresholds. As such, much more time was spent testing the various thresholds to calibrate the system accordingly. Since most of the sensors had fine thresholds, we included delays throughout the design in order to prevent chattering. The design for the IR sensor best exemplifies this. By enforcing a minimum time in an armed state, it avoided busy waiting swapping in and out of an armed state when the IR sensor is just on the border of its threshold.

Software – Android

We developed an android app to interact with the system. The application can connect to the Arduino system at any time. A screenshot of the app can be seen below:

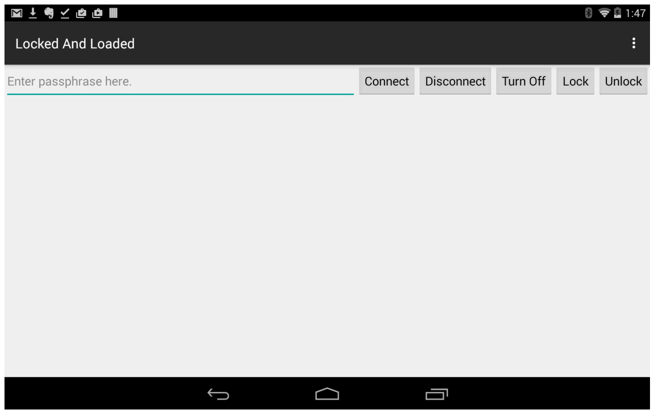

Figure 9. The Locked and Loaded Android application.

As shown in the screenshot, the application supports the ability to connect and disconnect to the system at any time, turn the system on and off at any time, and lock and unlock the system. We use password validation to verify if the user is “safe” and is allowed to interact with the system. For the purposes of the prototype, we set up the android app to directly interact with our system.

In a real application, there would be a set up phase where we would pair the application with the smart lock. For instance, the software could run silently in the background, attempting to pair with open Bluetooth devices that match our smart lock. If the devices are paired somehow, either by a passphrase or RSA system, the app could be used to automatically unlock doors when in detectable range.

ANALYSIS

As an initial step in the project, we designed and planned our project around certain behaviors and modeled them as fundamental linear temporal logic equations (LTLs). We formed the five fundamental LTLs listed below:

G(Bluetooth ∧ Locked => X(Validate Key))

In any state that we receive a Bluetooth signal while locked, we will next validate the key. This is a defining characteristic of our project because it provides fundamental security by restricting users. It ensures that only valid users are able to pass messages and interact with our system. This was implemented in the Arduino section of the code where every read message was validated before taking any action. Our validation is done through password checking. The user must submit a valid password with each command they send.

G(IR Sensor ∧ Locked => X(Armed))

This LTL defines the requirement that an intruder must be present in order for the system to fire. In a safety critical application such as our project, eliminating false positives is a necessity. Our solution was to have a the IR sensor act as a “tripwire” to ensure the Nerf gun would be shooting at a target instead of blank space. Additionally, this LTL states the obvious by claiming that an intruder can only be breaking in if the door is locked. This was simply implemented in the Arduino code as a guard before moving into the retaliate state.

G(Unlocked => F(Locked))

Since safety is a key value in our system, a requirement is to always move from the unlocked state to the locked state. This is a fundamental component in our system because we always assume a locked state is the safest state. In our system, there are two possible ways to move back to the locked state from the unlocked state. First, we can validate user input to relock the door. This follows the core idea that the user can relock their door. Second, the door will relock via timeout. Since we specify a timeout in our system, the door will always relock itself after a given amount of time.

G(¬Locked => ¬X(Armed))

This LTL exemplifies the requirement that non-locked states cannot move into the armed state. This primarily addresses the unlocked and off states, showing that both states cannot move into retaliation. This reinforces the second LTL presented earlier by preventing wrong retaliations. As mentioned earlier, eliminating misfires is a primary concern in this system, so this LTL is a necessary, fundamental component of our model. This LTL was implemented in the Arduino code by ensuring no non-locked states could move into the retaliation state.

G(Locked ∧ Validated => F(Unlocked))

This LTL represents the requirement that users must be validated and the system must be in a locked state before moving to the unlocked state. This is inherently to our system since validation essentially acts as our “key” into the system. We implemented this LTL in the Arduino code by using the password validation within the locked state as users try to move to the unlocked state.

These LTLs fundamentally shaped our state machines and approach towards the project. By creating these LTLs we were able to simplify the project to a few finite states, and quickly implement each section. In addition, by defining these LTLs we were able to better redesign the system to account for flaws (violations in the LTLs during verification). One key example is how we shifted from a proximity based approach to a password verification approach. Given our third LTL, we wanted to ensure the unlocked state would eventually move to the locked state; however, the "finally" clause does not specify time. Our initial design of proximity sensing could potentially stay in an unlocked state forever (where a user never leaves a specified radius around the door). Given the final LTL in combination with the previous concerns, we decided the proximity detection was insufficient due to the possibility of disconnects and the inconsistent signal strengths emitted from Android's Bluetooth as it was very dependent on battery strength in addition to distance. This led to the combination of a password verification and timeouts as our new system where the system is forced to relock itself given a maximum time for user input. This was able to enforce the third LTL very clearly as the timeout assured the system would relock, and the last LTL as users must validate themselves to unlock the door. By enforcing these LTLs through verification techniques, we were able devise the design presented previously with faith that our model will behave as intended.

One key concept that our system involves is concurrency. The Bluetooth integrated into our system is inherently concurrent since we allow any number of messages and any number of devices to connect to our system at any time; however we deal with this in a serial manner. Unfortunately, there were a number of issues with the Bluetooth that we had to deal with. First, the Arduino platform utilizes a limited buffer to process data transmitted via Bluetooth. Additionally, the serial port wrapper on Android has issues with receiving rates during reads. To resolve these issues we limited the speed of sending from Android such that the Arduino would always be able to read faster than Android could write.

In addition, we wanted to build a reliable real time system. Since Bluetooth can disconnect, we had to account for disconnections from either Android or Arduino. In either case, we assumed an incomplete message was invalid, and rejected the user. Upon disconnect, the Android application would notify the user of disconnection, and require the user to reconnect. Since our Arduino is always listening for Bluetooth connections, we simply have the Android device reconnect and resend the incomplete messages. Due to our limited number of Android devices, we were not able to fully stress test this part of the project and the system is possibly prone to a DoS attack; however, in the case that Bluetooth is overwhelmed, the system will simply stay in the locked state because of the timeout feature in the lock.

CONCLUSION

In this project, the majority of the time was spent in our formal analysis. As we designed and implemented each component of our machine, hardware limitations and violated LTLs caused constant redesign. The prime example is with the Bluetooth proximity explained earlier. The inconsistent strengths across devices and various power levels caused a redesign in how we decided to approach verification. While the final state machine is not very complex, this system reinforces the importance of formal verification. Letting the LTLs represent our system requirements, we were able to use them to guide our design process and truly create a system that models the intended behavior.

As an initial prototype, this project does a fairly good job demonstrating the benefits and negative consequences of a retaliatory smart lock; however, further steps ought to be taken. Our system was heavily designed around our personal use case. The thresholds are not very representative of a typical door since there was a very limited amount of testing. Since our servo simply rotated a fixed degree each time, it may not work on other deadbolts if those locks require a different amount of rotation and/or a different torque. A better system would incorporate a stepper motor to take advantage of its positional control via fractional increments. Sacrificing the high speed and torque of a servo for positional accuracy is appropriate as we want to prevent the lock from breaking, and accurately support a variety of locks.

REFERENCES

1. Arduino - Reference. http://arduino.cc/en/Reference/HomePage

2. Ken Shirriff's blog: A Multi-Protocol Infrared Remote Library for the Arduino. http://www.righto.com/2009/08/multi-protocol-infrared-remote-library.html

3. Ken Shirriff's blog: Detecting an IR Beam Break with the Arduino IR Library. http://www.righto.com/2010/03/detecting-ir-beam-break-with-arduino-ir.html

4. Connecting It Up | Adafruit 9-DOF IMU Breakout | Adafruit Learning System. https://learn.adafruit.com/adafruit-9-dof-imu-breakout/connecting-it-up

5. DC Motor: Actuation Assignment 1. https://courses.ischool.berkeley.edu/i262/f14/9

6. Servo Motor: Actuation Assignment II. https://courses.ischool.berkeley.edu/i262/f14/10

7. Using the BlueSMiRF. https://learn.sparkfun.com/tutorials/using-the-bluesmirf

8. TR-27 GRYPHON - CR-18 Rapidstrike Mod – NerfHaven. http://nerfhaven.com/forums/index.php?showtopic=25012

9. [TUTORIAL] How to Modify the Nerf Rapidstrike CS-18 - Modification Guide. https://www.youtube.com/watch?v=sphHIFf8-Mc

10. Tutorial: Nerf RapidStrike Modification Tutorial. https://www.youtube.com/watch?v=C7enXgH357E