ENGR-E 399/599: Reverse engineering embedded systems
Indiana University — Spring 2021
Prepared by: Austin Roach — Assigned: February 25, 2021

Assignment 3: AVR microcontroller reverse-engineering

Due date: April 2, 2021

For this assignment, you will analyze several programs running on an Atmel ATmega 328p AVR microcontroller. The microcontroller resides on an Arduino Uno development board. Documentation is included for the Arduino Uno, the ATmega 328p microcontroller, and the AVR instruction set in the `resources` directory.

For each of the binaries in this project, there is an input or inputs to the microcontroller that will cause it to turn on an LED and print “SUCCESS!” over the UART interface. Your goal is to find the inputs, while documenting the functionality that you encounter along the way.

You should have received the following hardware for this assignment:

- One Arduino Uno development board
- One AVR Pocket Programmer with ribbon cable
- One USB-A to USB-B cable
- One USB-A to USB-mini-B cable
- One package of hand warmers
- One potentiometer breakout board
- Two male-male jumper wires
- Three male-female jumper wires

0.1 BeagleBone Green

You can interact with the Arduino through the BeagleBone that has been provided to you. Connecting to hardware peripherals is sometimes difficult through a virtual machine. The BeagleBone provides a stable platform for interacting with hardware peripherals. You can connect to the BeagleBone over SSH, and can copy files to and from the BeagleBone using SCP.

Connect the BeagleBone Green (BBG) to your computer using the provided USB-A to USB-micro-B cable. This cable powers the board and also provides the data connection between your computer and the board.

The BBG should present a USB network device when you connect your PC (or VM). The BBG will enumerate both a CDC-NCM an RNDIS USB modem interface. Depending on your choice of operation system, your host computer should preferentially select one interface or the other. The BBG will serve an IP address over DHCP, providing an address of either 192.168.6.1 or 192.168.7.1 to your computer. You can then connect to the BBG on either 192.168.6.2 or 192.168.7.2.

You can SSH to the BBG using the default username of `retools` and default password of `defaultpassword`. On a Linux system, you could accomplish this with the following command to connect to 192.168.6.2:

```
ssh retools@192.168.6.2
```

After logging in, you may choose to create another account or change the password as you wish.

The BBG has a read/write filesystem, and it may be unhappy if you unplug its power source while the board is running. To power down the board safely, momentarily press the 'Power' button above the USB-micro-B connector. The adjacent blue LEDs will stop blinking after a few seconds, at which point it is safe to unplug the board.

0.2 Programming the microcontroller

You can program the ATmega328p using the AVR Pocket Programmer. Connect the AVR Pocket Programmer to your BeagleBone Green using a USB-A to USB-mini-B cable. Use the included ribbon cable to connect the AVR Pocket Programmer to the programming port (labeled ICSP) on the Arduino Uno. The key on the 2x3 pin connector should be facing toward the middle of the Arduino Uno, as depicted in Figure 1. The switch on the AVR Pocket Programmer should be set to 'Power Target' to provide power to the Arduino Uno during programming.

The `avrdude` application installed on your Beagle Bone Green can be used to control the AVR Pocket Programmer. To program a file named `program.bin` onto the Arduino Uno, use the following command. *YOU MUST COMPLETE THE "MEMORY READOUT" PORTION OF PART 1 BEFORE REPROGRAMMING THE DEVICE.*

```
sudo avrdude -v -e -F -V -c usbtiny -p ATMEGA328P \
 -b 115200 -U flash:w:program.bin:r
```

Your board comes preprogrammed with the `uart_intro.bin` program.

0.3 UART

After the device is programmed, unplug the AVR Pocket Programmer from your BeagleBone Green and connect the Arduino Uno directly to the BeagleBone Green using a USB-A to USB-B cable. This cable will power the board and will connect to the UART of the ATmega 328P at a baud rate of 38400. When you plug in the board on a Linux system, a serial

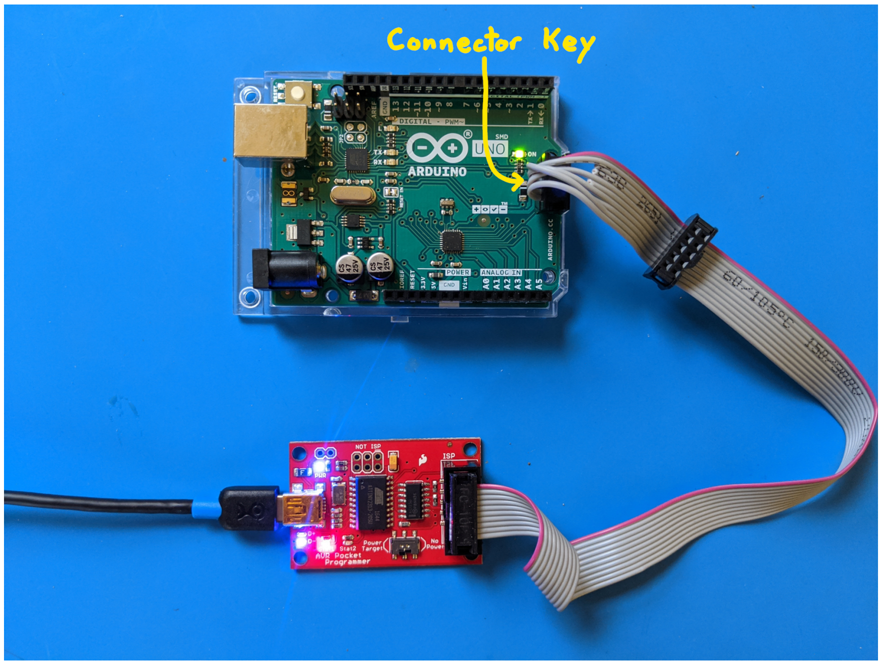

Figure 1: Cable arrangement for programming the ATmega328p microcontroller on the Arduino Uno.

device at `/dev/ttyACM0` should be automatically generated to which you can connect. To use `screen` to connect to this device, run:

```
screen /dev/ttyACM0 38400
```

If you are new to using `screen`, you can search the Internet for a reference guide. If you want to exit screen, type `Ctrl-a k`, and answer `y` at the prompt.

Alternatively, you could connect to a USB A port of your personal computer, using a serial terminal client of your choice.

0.4 Tailoring Ghidra

To make Ghidra aware of the particular details of the processor that we are examining, we will define a new AVR8 processor variant. Copy the included `avr8.ldefs` and `atmega328.pspec` files to the `Ghidra/Processors/Atmel/data/languages/` directory of your Ghidra installation. You may also find it useful to copy `atmel-0856-avr-instruction-set-manual.pdf` to the `Ghidra/Processors/Atmel/data/manuals/` directory. When you import one of the assignment binaries into Ghidra, choose the `Atmega328` processor variant.

You should have seen a demo in class of using the `SkipAVR8Fixup.java` Ghidra script. The conditional jump references added by this script can help produce control flow graphs that more accurately depict the logic of a function. You can use this script after importing and auto-analyzing a program if you would like.

---

1 Arduino introduction

1.1 Fuse states: 1 point

Use `avrdude` to read the microcontroller's fuse states into four files:

```
sudo avrdude -c usbtiny -p ATMEGA328P -b 115200 -U lock:r:lock.fuse:h
sudo avrdude -c usbtiny -p ATMEGA328P -b 115200 -U lfuse:r:lfuse.fuse:h
sudo avrdude -c usbtiny -p ATMEGA328P -b 115200 -U hfuse:r:hfuse.fuse:h
sudo avrdude -c usbtiny -p ATMEGA328P -b 115200 -U efuse:r:efuse.fuse:h
```

List the value of the lock fuse, `lfuse` (Fuse Low Byte), `hfuse` (Fuse High Byte), and `efuse` (Extended Fuse Byte).

Consult the ATmega328p reference manual section 31.2, "Fuse Bits" to answer some questions about the basic device configuration:

- At what address is the Reset Address in the current configuration? (You may also wish to consult section 16.1, "Interrupt Vectors in ATmega328/P".)
- Is SPI programming enabled in this configuration?
- Examine the memory lock bits. Are any memory protection modes enabled for this device? Based on this answer, do you think we will be able to read the program memory out of the device through the programming port?

1.2 Memory readout: 1 point

Verify that you can interact with the `uart_intro.bin` program as we did in class, sending characters to the microcontroller over UART and receiving responses back over the UART.

Attempt to read out the device's flash program memory into a file named `flash.bin`:

```
sudo avrdude -c usbtiny -p ATMEGA328P -b 115200 -U flash:r:flash.bin:r
```

What happens when you try to read the flash?

Now program the device with `uart_intro.bin`:

```
sudo avrdude -v -e -F -V -c usbtiny -p ATMEGA328P \
 -b 115200 -U flash:w:uart_intro.bin:r
```

and try reading out the memory again. Now what results do you get? What has changed?

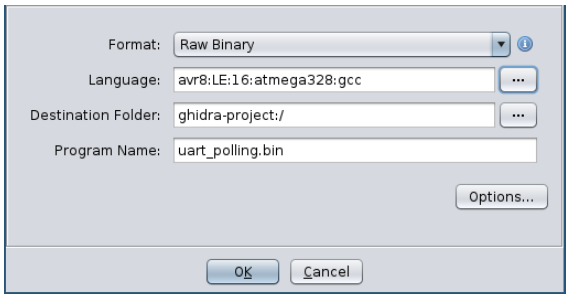
Format: Raw Binary
Language: avr8:LE:16:atmega328:gcc
Destination Folder: ghidra-project:/
Program Name: uart_polling.bin

Figure 2: Ghidra import of raw binary for ATmega328p

1.3 uart_polling: 1 point

Program the device using `uart_polling.bin`. Verify that you can interact with the device through a serial terminal.

Import the file into Ghidra with the language `avr8:LE:16:atmega328:gcc`. The import window should be as shown in Figure 2. Using the `uart_intro.bin` analysis from class as a guide, label some of the common functions.

You may find it useful to copy the constants section from program memory to SRAM as we did in class. Analyze the initialization routine to determine the range that you should copy.

The password for this example isn't particularly well hidden, and you can probably find it with very little code analysis. But the goal of this exercise is to analyze the logic surrounding the password evaluation. Find the main loop of the program and perform some analysis:

- We identified the `uart_getchar()` function together in class. What register holds the character received over the UART when the function returns?
- What characters received over the UART cause the control flow to break out of the normal evaluation loop and head in the direction of the victory() function?
- What values must r16 and r17 have to reach the victory() function?
- How can r16 and r17 be modified with each input character? What information is each register carrying?
- Where in memory is the good password stored?
- What is the password?

1.4 uart_interrupt: 2 points

Program the device using `uart_interrupt.bin`, and import the file into Ghidra as above.

While the evaluation of the password is performed in a similar manner as in `uart_intro.bin`, this program uses an interrupt mechanism to receive data from the UART instead of the polling mechanism used in the previous example. Find the handler for the `USART_RX` interrupt and let's examine how the interrupt handler interacts with the main loop.

- What memory addresses does the interrupt handler load from? What addresses does it store to?
- Which of these memory addresses does the main loop load from? Which of these memory addresses does the main loop store to?
- Describe the data structure that is being used to communicate between the interrupt handler and the main loop. What is the purpose of each of the values stored in memory?
- Is there any circumstance where this mechanism will drop data coming in from the UART? What are the conditions when this would happen?

Export your analyzed programs from Ghidra and submit them with your findings.

2 I/O interaction: 5 points

Program the device using `analog_io.bin`, and import the file into Ghidra as above. This is a significantly more complicated program than the first two that you analyzed. Your goal should be to analyze as little of the program as possible while determining the correct inputs to the device. If you are not certain of what a value may be, use feedback from the device to help guide you.

As part of your analysis, evaluate:

- What I/O configuration actions are performed by the function at code address 0x72?
- What I/O element is used by the function at code address 0xc0? What are the arguments to this function? What does it return?
- What functions call the function at code address 0xc0? What arguments do they pass? What inputs are they interacting with?
- Follow the control flow through the main loop to see how the various I/O values are used. Can you determine a set of inputs that triggers the `victory()` function? Describe your train of thought in determining these inputs.

Export your analyzed program from Ghidra and submit it with your findings.

3 Brute-forcing using peripheral interaction: 5 points

As discussed in class, the `hash_*_bits.bin` files calculate a 32-bit hash value. The 8-bit version compares only the lowest byte of the this hash. The 16-bit version compares the lowest two bytes, etc. For this portion of the assignment, we will interact with the `hash_16_bits.bin` version.

Going backwards from the stored hash value to an input is not a tractable problem for a well constructed hash function. Instead, we will attempt to brute force the hash function to find an input that matches the desired hash. For the 16-bit version, we will simply feed a series of passwords to the Arduino Uno over the serial port until we find one that matches.

Before starting, let's think about what we expect from this interaction:

- How does the program on the microcontroller communicate that it is ready to receive a password?
- How does the program on the microcontroller communicate that a password is incorrect?
- How does the program on the microcontroller communicate that a password is correct?
- What is your estimate of the rate that you can test passwords over the serial port?
- What is your estimate of the amount of time that will be required to brute-force a 16-bit hash over the serial port?

Write a program to interact with the serial port in an automated way. You can use Python's `serial` module,[1] or any other means that you prefer. In addition to automating the interaction with the serial port, you will also need a way to generate passwords to test. Document how you plan to do this.

Suggestion: If can be helpful to test your program on a simpler version of the hash. If you load `hash_8_bits.bin` onto the Arduino Uno, you should find a solution in only a few hundred attempts. Once you know that your program works, load the Arduino Uno with `hash_16_bits.bin` and run your program again to find the solution.

What password(s) did you find? How many attempts were required to find a password? How much time did it take?

Submit the source code for your program along with your write-up.

4 Brute-forcing using emulation: 5 points

Now let's work on the `hash_24_bits.bin` program. How much time do you estimate would be required if you brute forced this 24-bit hash over the UART?

Let's take a different approach. Even without fully understanding the hashing algorithm, we can use an emulator to execute the code on a faster processor while avoiding the slow serial

[1] Install python-serial or python3-serial modules on Ubuntu. The serial module is already installed on the BeagleBone Green.

communication interface.

The emulator that we will use is `simavr`[2]. On Ubuntu 20.04, you can install the necessary libraries with:

```
sudo apt-get install simavr libsimavr-dev build-essential
```

Details about the emulation engine that we covered in class should be sufficient for you to complete the assignment, but additional documentation is available at the `simavr` Github repository.

A skeleton emulation program is provided for you in `resources/emulation-skeleton.c`. This provides some structure for the program, but you will need to fill in some details from your analysis of `hash_24_bits.bin`

4.1 Define the initial processor state

To avoid interaction with I/O, which complicates the emulation, choose a starting address for emulation after the password prompt has been printed to the UART. You will need to define the initial processor state at this point:

- Define the program counter value. *Don't forget that for simavr this is a byte address rather than a 16-bit code address.*
- Define a stack pointer value. (Note that you can choose any reasonable address for the bottom of the stack, since all stack accesses are relative to the stack pointer.)
- Define any register values that have been set in `main()` up to this point.
- Define any values in memory or on the stack as appropriate.

Note that since we are using the interrupt mechanism for reading UART values in this example, you can arrange the contents of memory as if your password has already been read in completely over the UART interface and stored in memory.

4.2 Define the endpoint of the program

Define the address in the program at which emulation will end. *Again, don't forget that this is a byte address rather than a 16-bit code address.* Determine how you will extract the computed hash value from the processor state for comparison to the good hash.

4.3 Generate test passwords and check hashes in controlling loop

Provide a means to generate passwords to test, and document the algorithm that you will use. In your program, return the computed hash to your controlling loop, and test each of the generated passwords. How many iterations are required to find a match?

[2]https://github.com/buserror/simavr

Compile your program with `-lsimavr`, for example:

```
gcc -o emulation_skeleton emulation_skeleton.c -lsimavr
```

You may find it helpful to test your program on a simpler version of the hash. You can test only the lowest byte or lowest two bytes of the calculated hash, then verify that you found valid passwords by loading `hash_8_bits.bin` or `hash_16_bits.bin` into the microcontroller and testing the passwords. Once you are confident that your program works, run it while testing the lowest three bytes to find an acceptable password for this portion of the assignment, testing with `hash_24_bits.bin` loaded into the microcontroller.

Submit the source code that you used for the emulation, as well as a list of any valid passwords that you found. How many iterations and how much computation time were required to find a matching password?

5 Brute-forcing using translation: 5 points

Emulation of the AVR code is still not performant enough to brute-force the 32-bit password hash in reasonable time. For this portion of the assignment, reverse-engineer the algorithm used to generate the hash. Implement this algorithm in C and compile it for the native architecture of your operating system (probably x86), then provide candidate passwords to your hash calculator until you find a match.

Submit the source code that you developed, as well as a list of any valid passwords that you found. How many iterations and how much computation time were required to find a matching password?