ENGR-E 399/599: Embedded systems reverse engineering

Lecture 8: Microcontroller emulation

Austin Roach
ahroach@iu.edu

March 3, 2022

Mystery function

```
0x00000000    ff1001e2    and r1, r1, 0xff
0x00000004    0020d0e5    ldrb r2, [r0]
0x00000008    0030a0e1    mov r3, r0
0x0000000c    010052e1    cmp r2, r1
0x00000010    010080e2    add r0, r0, 1
0x00000014    0200000a    beq 0x24
0x00000018    000052e3    cmp r2, 0
0x0000001c    f8ffff1a    bne 4
0x00000020    0230a0e1    mov r3, r2
0x00000024    0300a0e1    mov r0, r3
0x00000028    1eff2fe1    bx lr
```

- How many arguments does the function take?
- What are the types of the arguments?
- What does the function do?
- What does the function return?

Mystery function revealed

```
char *strchr(const char *s, int c)
```

Description

The `strchr()` function returns a pointer to the first occurrence of the character `c` in the string `s`.

Return value

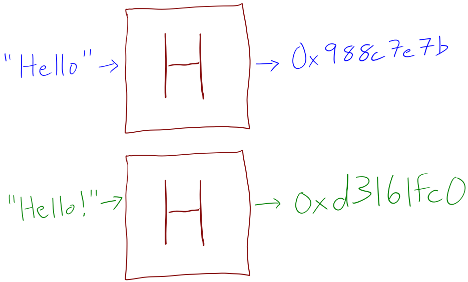
The strchr() function returns a pointer to the matched character or NULL if the character is not found. The terminating null byte is considered part of the string, so that if `c` is specified as `'\0'`, this function returns a pointer to the terminator.

Today's plan

- Discussion of brute-forcing approaches
  - Speeding up UART brute-forcing with firmware patching
- Emulation overview
- Introduction to `simavr`
  - Very relevant to Assignment 03
- Emulation for dynamic analysis
- Popular emulation frameworks

Reminder: brute-forcing hash functions

Assignment 3 contains a cryptographic hash function:



Goal is to determine an input that hashes to a specific value.

To find a match for a good cryptographic hash function, the best you can do is test many possible inputs (brute force).

Brute-forcing bottleneck

Most of our time is spent sending characters back and forth across the UART

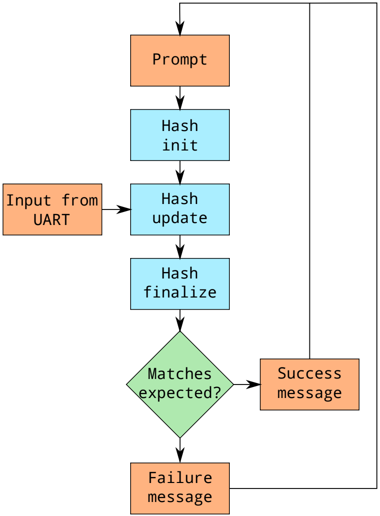
Speeding up brute-forcing over UART

- Speed up UART interface
  - 38400 to 57600 baud; faster would require some platform modifications
- Modify strings to print fewer characters
  - Truncate each message to a single character and newlines

- Start sending next password before result is received
- Run generator and evaluation routine on the microcontroller

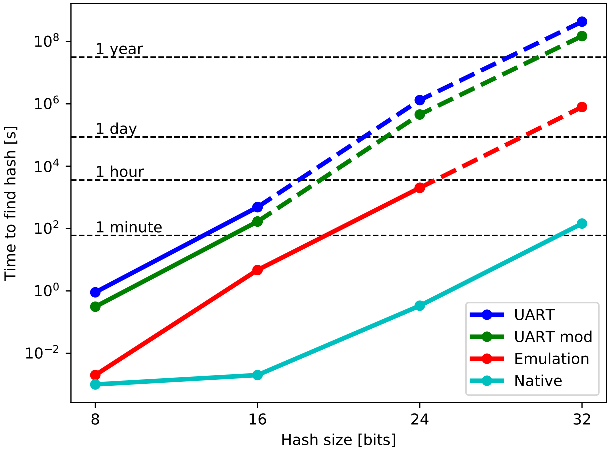
Let's patch some firmware!

Other approaches



- Emulate the code on a faster system
  - Today's topic
- Analyze the hash algorithm, re-write and compile to run on another, faster processor
  - You'll get to do this in Assignment 3

Other approaches compared



Emulation overview

- Emulation allows one system to execute code meant for another system
  - Often cross-architecture
- We have previously emulated using QEMU
  - Linux 'user-mode' emulation of the binary water balloon
  - System-level emulation of embedded Linux systems
- Can also use emulators to execute sequences of instructions
  - Not emulating the full system
  - Carve out the pieces of the code that you care about

Emulation applications

- Dynamic analysis when we don't have debug access on the host system
- Incorporate pieces of cross-architecture code in larger execution framework
- Parallelize searching tasks
- Virtualize I/O or peripheral interaction to investigate system behaviour under various conditions

Emulation frameworks

- Keep track of processor/system state
- Model the effects of instructions on the system state
- Peripheral interaction:
  - Simulate
  - Pass-through to real hardware
  - Provide callbacks for user implementation

Modeling peripheral interaction typically the biggest challenge for embedded systems emulation

Processor state in simavr

Represented by `avr_t` struct

- https://github.com/buserror/simavr/blob/master/simavr/sim/sim_avr.h
- Configuration settings (clock frequency, voltages, fuse settings)
- Device information (memory sizes, address size)
- Memory (including register states)
- SREG state
- Handler functions for special events
- Interrupt handler state
- Statistics
  - Cycle counts
- Emulation state (cycle limits, termination address)

Instruction modeling in simavr

Implemented in `avr_run_one()`:

- https://github.com/buserror/simavr/blob/master/simavr/sim/sim_core.c
- Parse machine code representation
  - Determine opcode
  - Determine operands
- Advance program counter
  - Add 2 by default
  - Control flow instructions alter in other ways
- Update memory or register file
- Update status register

simavr: initialization

```c
avr_t *avr;
avr = avr_make_mcu_by_name("atmega328p");
avr_init(avr);
// Define frequency, voltages, etc.
```

simavr: program memory

```
typdef struct avr_t {
    ...
    uint8_t * flash;
    ...
}
```

Initialized in our example with:

```
memcpy(avr->flash, prog_mem, _PROG_MEM_SIZE);
```

simavr: data memory

```
typdef struct avr_t {
    ...
    uint8_t * data;
    ...
}
```

Can be used to access general-purpose registers, I/O control registers, and SRAM.

```
// Set register R0 contents to 0x12
avr->data[0] = 0x12;
// Set memory address 0x200 to 0x30
avr->data[0x200] = 0x30;
```

simavr: control

```c
// Define a starting address
avr->pc = STARTING_ADDRESS;
// Run until ending address
int state = avr->state;
while ((state != cpu_Done) && (state != cpu_Crashed) &&
       (avr->pc != ENDING_ADDRESS)) {
    state = avr_run(avr);
}
```

simavr treats the PC as a *byte* address

simavr: skeleton framework for assignment

simavr: Debug

- Can optionally create a GDB debugger stub
- Emulator will wait for debugger to attach before execution

```
avr->gdb_port = 1234;
avr_gdb_init(avr);
```

Ubuntu packages

`libsimavr2`, `libsimavr-dev`, `gdb-avr`

simavr: Executing code with debug

```
$ ./emulate uart_intro.bin
Read 380 bytes from uart_intro.bin.
avr_gdb_init listening on port 1234
gdb_network_handler connection opened
```

simavr: Connecting with GDB

```
$ avr-gdb
...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x00000000 in ?? ()
```

simavr: Setting breakpoints in GDB

```
(gdb) b *0x13a
Breakpoint 1 at 0x13a
(gdb) c
Continuing.
Note: automatically using hardware breakpoints for read-only addresses.

Breakpoint 1, 0x0000013a in ?? ()
(gdb)
```

- Code addresses are *byte* addresses in this context
- Some versions of avr-gdb require a crazy cast to set breakpoints at program memory addresses:
  `b *(void (*)())0x13a`

simavr: Inspecting system state in GDB

```
(gdb) info reg
r0             0x0                 0
r1             0x0                 0
r2             0x0                 0
r3             0x0                 0
...
(gdb) x/2s 0x800100
0x800100: "Please enter the passkey:\r\n"
0x80011c: "SUCCESS!\r\n"
...
(gdb) x/1xb 0x800024
0x800024: 0x20
```

GDB adds 0x800000 to SRAM addresses to separate Harvard architecture address spaces.

simavr: Examining instruction effects in GDB

```
(gdb) info reg
...
r24            0x18                24
r25            0x0                 0
...
(gdb) si
0x0000013c in ?? ()
(gdb) info reg
...
r24            0x0                 0
r25            0x0                 0
...
(gdb) si
0x0000013e in ?? ()
(gdb) info reg
```

simavr: Dumping memory state with GDB

```
(gdb) dump memory /tmp/mem.bin 0x800000 0x800900

$ xxd mem.bin
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0001 0000 0000 0000 0001 2801 ff08 7c01  ..........(...|.
00000020: 0000 0000 2000 0000 0000 0000 0000 0000  .... ...........
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
...
```

simavr: Importing memory into Ghidra

- Delete SRAM block in Memory Map
- Use File→Add to Program...
- Map 0x800 bytes from offset 0x100 in file to mem:0x100

QEMU

Quick EMUlator `https://www.qemu.org/`

- User-mode emulation
- System-mode emulation
- Supports (at least): `Aarch64`, `Alpha`, `ARM`, `AVR`, `CRIS`, `HPPA`, `i386`, `m68k`, `Microblaze`, `MIPS`, `MIPS64`, `OpenRISC`, `PowerPC`, `RISC-V`, `s390x`, `SH4`, `SPARC`, `SPARC64`, `Tricore`, `unicore32`, `x86-64`, `xtensa`, `TILE-Gx`

Unicorn engine

https://www.unicorn-engine.org/

- Uses QEMU for instruction emulation
- Emulate raw binary code (not an ELF or disk image)
- Bindings for multiple programming languages
- Register all sorts of customized handlers
- Falling behind QEMU a bit...

Translation

Emulation is still not the fastest way to calculate these hashes:

- 32-bit arithmetic with 8-bit instructions
- Instruction translation slower than native instruction execution

Translation to your PC's native ISA is significantly faster:

- Understand the hash function algorithm
- Write it in C (or another language)
- Compile for the processor on your PC's processor (probably x86)
- Profit!

To do this successfully requires an exact understanding of the hash algorithm