Side-Channel Defense: Defeating Physical Attacks by Measuring Current Draw

Leah Goggin

1 Background and Motivation

Physical attacks on computers are difficult to defend against. While they are often (and often reasonably) considered unlikely compared to remote attacks, addressing them is critical in situations such as defending valuable air-gapped systems. In particular, USB-sized devices such as the Bash Bunny can spoof almost any other USB device and do anything from opening a shell and running code to sniffing the buffer used by the actual keyboard. A malicious-USB attack can be delivered either by an attacker gaining physical access to a computer or by a legitimate user being tricked into inserting the malicious device.

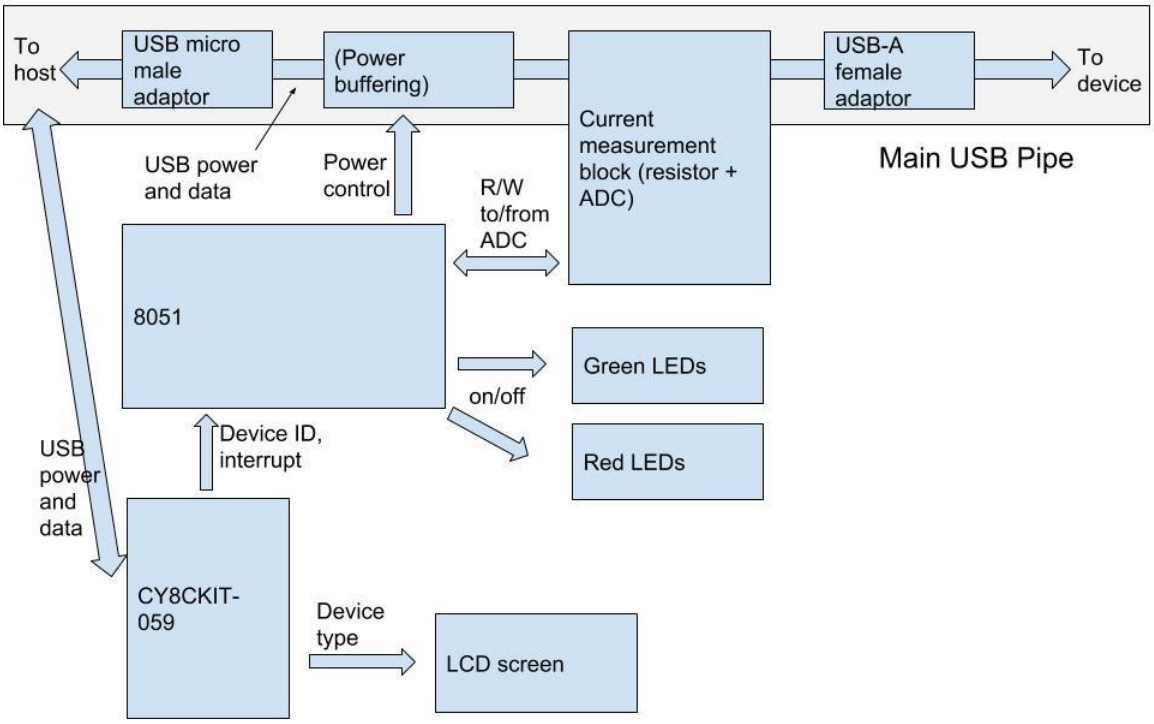
Authenticating all USB devices at the protocol level is impractical because it would require capabilities that are not present in most devices. Rather than attempting this, I used physical measurements of the devices as a way of gathering information about what they are, regardless of what they may claim to be in their token packets. In particular, I used current draw to distinguish Bash Bunnies from devices they might emulate and established proof-of-concept for this approach.

Bash Bunnies contain a quad-core Arm A7, 512 MB DDR3, and 8 GB NAND disk and require about 1.5 amps to operate (considered the "Battery Charging" spec by the USB standard). Because the majority of this current is needed by the processor, it is not possible for the Bash Bunny to spoof this value while it is operating. The USB 2.0 standard allows 500 mA to be drawn by non-charging devices and the 3.0 standard allows 900 mA.

2 Hardware description

The core of the system is an Intel 8051 microcontroller. The USB host and the 8051 are bridged by a Cypress 8CKIT-059 PSoC breakout board, which connects to the host with a USB-to-RS232 cable, receives device descriptors, and forwards necessary information to a pin of the 8051. The breakout board also operates an LCD screen as a UI, telling the user what device is claiming to be connected. The 8051's other input is the voltage across a one-ohm resistor on the connected device's power line, collected by an ADC0804 analog-to-digital converter.

As outputs, the 8051 controls the power to red and green LEDs and (in some configurations) the enable pin of an LM18296 buffer on the connected device's V_{cc} line.



3 Software description

3.1 8051

The 8051 runs x86 assembly responsible for checking current consumption, deciding if it is reasonable, and operating the LEDs and buffer accordingly–that is, turning on a green LED if the device is accepted and a red LED and cutting the device's power if it is judged to be malicious. Since the ADC is measuring voltage drop across a one-ohm resistor, the incoming voltage measurement is equivalent to the current into the connected device. At present, the threshhold which devices identifying themselves as HIDs or mass storage may not exceed is hardcoded into the assembly.

3.2 PSoC

The PSoC receives the connected device class from the host as a single-byte identifier, converts it to a human-friendly string, and displays it on the LCD. (The present system only distinguished between HIDs, mass storage, and "other", but it would be straightforward to add code for all device classes.) It is also responsible for exercising the policy of what classes may draw what currents. It sends an interrupt to the 8051 and operates another pin to signify whether the device may or may not exceed the threshhold. The current implementation allows all devices other than HIDs or storage to do so, but this is mostly meant to allow demonstration that the 8051 isn't blindly

rejecting all devices that exceed the amperage threshhold, i.e. the 8051 and hardware will behave appropriately if something like a cell phone is connected as long as the relevant code is added to the PSoC. The ”other” classification can be considered a stand-in for legitimate charging devices that have high current draws.

3.3 Host

The host uses a Python wrapper of existing C libraries to query the ID of a connected device, then passes this information to the PSoC, which is also connected as a USB device. In practice this code would need to run in the kernel to ensure it happens immediately upon connection of a USB device, before the driver is loaded, but for proof-of-concept it is implemented as a script using a Python-wrapped version of libusb.

4 Results

4.1 Successes

Overall, the project was successful as a proof-of-concept. My system is able to reliably distinguish between Bash Bunnies emulating keyboards/flash drives and real keyboards and flash drives. It can enforce a policy, configurable in simple C code, of what device classes may draw what level of current. It can in principle efficiently disconnect power to a malicious device, with shortfall in that respect described below. It displays a human-friendly string on an LCD, alerting a hypothetical user that what they may have thought was a flash drive is attempting to emulate a keyboard.

4.2 Shortcomings

Current cutoff My system can make and signal (through LEDs) a decision about whether a connected device is malicious. It can also cut off current to a device by manipulating the enable pin of an LM18296 buffer. Unfortunately, this buffer is only rated for up to 1 amp and so it is not possible to run the Bash Bunny with its power line routed through the buffer. Other devices can be powered through it, and I temporarily tweaked the code to establish that the approach does work, i.e. if the current threshhold is lowered to below a keyboard’s current draw and a keyboard is plugged in, the 8051 is able to cut power to the device port upon deciding it’s ”malicious” and restore it on reset. I’m confident that swapping the LM18296 for a similar chip with a higher max amperage would give the full desired functionality.

Another issue with the current cutoff is that the host-side software currently takes the form of a Python script that must be launched by the user from the command line, meaning that huge amounts of USB traffic could be exchanged by the time the 8051 is brought into play. The Python script consists largely of stitched-together wrapped libc functions, so it should be straightforward to achieve the same functionality in C. This code would have to run in the kernel, executing between the initial USB setup protocol and the loading of the device driver.

Adherence to USB standard As discussed in the background, the Bash Bunny's spec calls for a 1.5A current source, well above the .9A maximum allowed by the USB standard. In practice, the Bash Bunny may draw less than .9A for periods of time. While this is in some sense not a problem because the real keyboards and flash drives draw so little current that it is still easy to distinguish them, it does imply that a system built on the current-draw principle can never guarantee detecting the Bash Bunny with no false positives against legitimate USB devices. Using an average rather than instantaneous current measurement could probably give negligible false negatives and positives.

Requirement of host-side software My hope was to locate my system completely external to the host, such that a more streamlined version could function as a transparent sleeve over a USB port. Unfortunately, it is very difficult to determine a USB device's device class just by watching the electrical signals it sends across a breadboard. The options were either to use a surface-mount chip that carries on a full USB protocol with the device and the host (impossible for this project because breakout boards with accessible pins were prohibitively expensive) or to work around the problem by asking the host for the device class. This is not an ideal solution because 1. it forces a user to modify their kernel, as mentioned above, and 2. it requires 2 host USB ports to connect a single device. A more developed version may either use more complex chips or be located entirely within the host to avoid this.

4.3 Security characteristics

As mentioned above, the system is currently only aware of HIDs, mass storage, and "other" USB devices. For a complete system, it would be necessary to add code for each device class and determine a policy defining which classes may draw how much current.

There is currently no functionality allowing the host to make sure that my system is connected to its port, meaning an attacker with physical access could unplug it amd plug their own device in normally. This functionality could be implemented by some sort of handshake conducted before new device connections are accepted. This would require the module capable of USB communication with the host (currently the PSoC) and the module that carries connected-device traffic (the Main USB Pipeline in the diagram) to be collapsed to a single port (which is obviously desirable in a more developed version anyway).

My system assumes that the device under study is being powered by the host. It is possible for an attacker to externally power a device and not connect to the host power line at all, in which case the current measurement will always be zero. A better version of my device could prevent this in at least two ways. One would be to prevent the attacker from accessing the GND pin, since the connected device must share a common ground with the host to communicate with it (and to avoid damaging itself or the host). Another would be to note that no operating device draws *no* current, and anything doing so should be considered suspicious.

This rule was not possible to implement since the available analog-to-digital converter is not sensitive enough to reliably distinguish between true zero and a keyboard's current draw, but like the buffer, this should be fixable just by getting higher-end chips. Some range vs. precision difficulties

counld be introduced, but this could be fixed by, for example, including one ADC that covers the 0 to 2 volt range and one that covers something like 0 to 30 mV.

5 Appendix A: Host-side script

```
import serial.tools.list_ports
import usb.core

ports = [comport.device for comport in
   serial.tools.list_ports.comports()]
name = ports[0] # making assumption of exactly 1 serial port
port = serial.Serial(name)

#stash everything already plugged in
devs = usb.core.find(find_all=True) # generator
stash = []
for dev in devs:
  stash.append((dev.idVendor, dev.bDeviceClass, dev.bus, dev.address))

junk = input("please plug in device and press Enter")

def dev_eq(dev1, dev2):
  for i in range(4):
    if dev1[i] != dev2[i]:
      return False
  return True

plugged_dev = None
# now see what's new
new_devs = usb.core.find(find_all=True)
for new_dev in new_devs:
  found_match = False
  new_dev_tuple = (new_dev.idVendor, new_dev.bDeviceClass,
     new_dev.bus, new_dev.address)
  for old_dev_tuple in stash:
    if dev_eq(old_dev_tuple, new_dev_tuple):
      found_match = True
  if not found_match:
    plugged_dev = new_dev
    break # assume user only added 1 device

config = plugged_dev[0]
intf = usb.util.find_descriptor(config)
dev_class = intf.bInterfaceClass
print (dev_class)

# keyboard: 413c:2107
```

```
# psoc: 04b4:f131
# flash drive: 0781:5575

message = str(dev_class)
port.write(message.encode('utf-8'))
port.close()
```

6 Appendix B: 8051 code

```
; used: R0, R1, R2, R3, R4

; P3.2 is interrupt from ADC (unused)
; P3.3 is interrupt from PSoC
; P3.4 is can_use_high bool from PSoC
; P3.5 is LM18293 (current control to device)

.org 00h
ljmp start

.org 013h
ljmp EX1_ISR

start:
; init R1 to store most recent ADC value
mov R1, #0h

; init R3 to store our ADC result cutoff
mov R3, #06h

; start ADC
mov dph, #0feh
mov dpl, #10h
mov a, #0h
movx @dptr, a

; SETUP 8255
; all ports output
mov dph, #0feh
mov dpl, #0bh
mov a, #80h
movx @dptr, a
lcall waste_time

;set LCD disp for 8bit, 5x7 char set
mov dpl, #0ah  ; port C
mov a, #0h
movx @dptr, a  ; lower E line
lcall waste_time
mov dpl, #09h  ; port B
mov a, #38h
movx @dptr, a  ; do stuff? this is probably wrong
```

```
lcall waste_time
mov dpl, #0ah  ; port C
mov a, #04h
movx @dptr, a  ; raise E
lcall waste_time
mov a, #0h
movx @dptr, a  ; lower E
lcall waste_time

; turn disp on, hide cursor
mov dpl, #09h
mov a, #0ch
movx @dptr, a
lcall waste_time
mov dpl, #0ah  ; port C
mov a, #04h
movx @dptr, a  ; raise E
lcall waste_time
mov a, #0h
movx @dptr, a  ; lower E
lcall waste_time

lcall clear
lcall RAM0

; enable interrupts
mov ie, #84h   ; enable EX1

; default allow current to device port
setb P3.5

; default to being a voltmeter
mainloop:
  mov dpl, #10h
  movx a, @dptr ; fetch ADC value
  movx @dptr, a ; start next conversion
  mov R1, a     ; store newest voltage measure
  mov 20h, a    ; need to bit-address into voltage
  mov a, 0h
  jnb 7h, loop6
  add a, #25d
  loop6: jnb 6h, loop5
  add a, #13d
  loop5: jnb 5h, loop4
  add a, #6d
```

```
  loop4: jnb 4h, loop3
  add a, #3d
  loop3: jnb 3h, loop2
  add a, #2d
  loop2: jnb 2h, loopend
  add a, #1d
  loopend:      ; a holds tenths of volts
    push acc
    lcall clear
    pop acc
    mov b, #10d
    div ab      ; quotient in a, remainder in b
    anl a, #0fh ; clear high nibble
    orl a, #30h ; set high nibble to 0011
    mov R2, a   ; quotient in R2
    lcall display ; print quotient
    mov R2, #2eh ; '.' in R2
    lcall display ; print '.'
    mov a, b    ; remainder in a
    anl a, #0fh ; clear high nibble
    orl a, #30h ; set high nibble to 0011
    mov R2, a   ; remainder in R2
    lcall display ; print remainder
ljmp mainloop

EX1_ISR:        ; do nothing if R1 < R3. if more, do something.
   mov R4, a    ; store whatever main had in acc
   mov P1, P3   ; debug
   jb P3.4, greenlight ; don't bother testing current if it's a known
      high-i device

   current_test:
   mov a, R1
   subb a, R3  ; this will be a carry if measured V is less than cutoff
   jnc high_current
greenlight:    ; not malicious
   mov dpl, #0ah  ; port C
   mov a, #40h ; set C6 (green light)
   movx @dptr, a
   hang: sjmp hang ; wait for reset
   mov a, R4
   reti

high_current:  ; malicious
   clr P3.5    ; disconnect device power
```

```
    mov dpl, #0ah  ; port C
    mov a, #80h ; set C7 (red light)
    movx @dptr, a
    hang2: sjmp hang2 ; wait for reset
    mov a, R4
    reti

display:        ; print contents of R2
   mov dpl, #0ah
   mov a, #01h
   movx @dptr, a ; lower e
  lcall waste_time ; give LCD time to balance its chakras
   mov dpl, #09h  ; port B
   mov a, R2
   movx @dptr, a  ; write
  lcall waste_time
   mov dpl, #0ah  ; back to C
   mov a, #05h
   movx @dptr, a
  lcall waste_time
   mov a, #01h
   movx @dptr, a
  lcall waste_time
   ret

waste_time:
   mov R0, #0ffh
   inner:
     djnz R0, inner
   mov R0, #0ffh
   inner2:
     djnz R0, inner2
   mov R0, #0ffh
   inner3:
     djnz R0, inner3
   ret

clear:
   mov dpl, #09h
   mov a, #01h
   movx @dptr, a
   lcall waste_time
   mov dpl, #0ah  ; port C
   mov a, #04h
   movx @dptr, a  ; raise E
```

```
  lcall waste_time
  mov a, #0h
  movx @dptr, a  ; lower E
  lcall waste_time
  ret

RAM0:
  mov dpl, #0ah  ; port C
  mov a, #00h
  movx @dptr, a  ; raise E
  lcall waste_time
  mov dpl, #09h
  mov a, #80h
  movx @dptr, a
  lcall waste_time
  mov dpl, #0ah  ; port C
  mov a, #04h
  movx @dptr, a  ; raise E
  lcall waste_time
  mov a, #0h
  movx @dptr, a  ; lower E
  lcall waste_time
  ret
```

7 Appendix C: PSoC code

```
/* ========================================
 *
 * Copyright MIT 6.115, 2013
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF MIT 6.115.
 *
 * This file is necessary for your project to build.
 * Please do not delete it.
 *
 * ========================================
*/

#include <device.h>

const uint8 HID = '3';
const uint8 STORAGE = '8';

// unused since rearranged so PSoC decides bool of whether device can
   have high current
void data_Write(uint8 data) {
   data_0_Write((data & (1 << 0)) ? 1 : 0);
   data_1_Write((data & (1 << 1)) ? 1 : 0);
   data_2_Write((data & (1 << 2)) ? 1 : 0);
   data_3_Write((data & (1 << 3)) ? 1 : 0);
   data_4_Write(0);
   data_5_Write(0);
   data_6_Write(0);
   data_7_Write(0);
}

CY_ISR(RX_INT)
{
   uint8 incoming = UART_ReadRxData();
   //data_Write(incoming) ; // send to output pins to 8051
   high_current_bool_Write(((incoming == HID) || (incoming ==
      STORAGE)) ? 0 : 1);
   int_8051_Write(0);              // send interrupt to 8051
   CyDelayUs(8);
   int_8051_Write(1);              // unsend interrupt to 8051
```

```
    if (incoming == HID) {
        LCD_ClearDisplay();
        LCD_PutChar('K');
        LCD_PutChar('E');
        LCD_PutChar('Y');
        LCD_PutChar('B');
        LCD_PutChar('O');
        LCD_PutChar('A');
        LCD_PutChar('R');
        LCD_PutChar('D');
    } else if (incoming == STORAGE) {
        LCD_ClearDisplay();
        LCD_PutChar('S');
        LCD_PutChar('T');
        LCD_PutChar('O');
        LCD_PutChar('R');
        LCD_PutChar('A');
        LCD_PutChar('G');
        LCD_PutChar('E');
    } else {
        LCD_ClearDisplay();
        LCD_PutChar('O');
        LCD_PutChar('T');
        LCD_PutChar('H');
        LCD_PutChar('E');
        LCD_PutChar('R');
    }
}

void main()
{
    int_8051_Write(1);
   LCD_Start();           // initialize lcd
   LCD_ClearDisplay();

    CyGlobalIntEnable;
    rx_int_StartEx(RX_INT);    // start RX interrupt (look for CY_ISR
        with RX_INT address)
                               // for code that writes received bytes
                                  to LCD.

    UART_Start();              // initialize UART
    UART_ClearRxBuffer();
```

```
    for(;;)
    {}
}

/* [] END OF FILE */
```