

FirmUSB:

Vetting USB Device Firmware using Domain Informed Symbolic Execution

Grant Hernandez & Farhaan Fowze, Dave Tian, Tuba Yavuz, Kevin Butler

ACM CCS'17, Dallas, TX
November 2nd, 2017

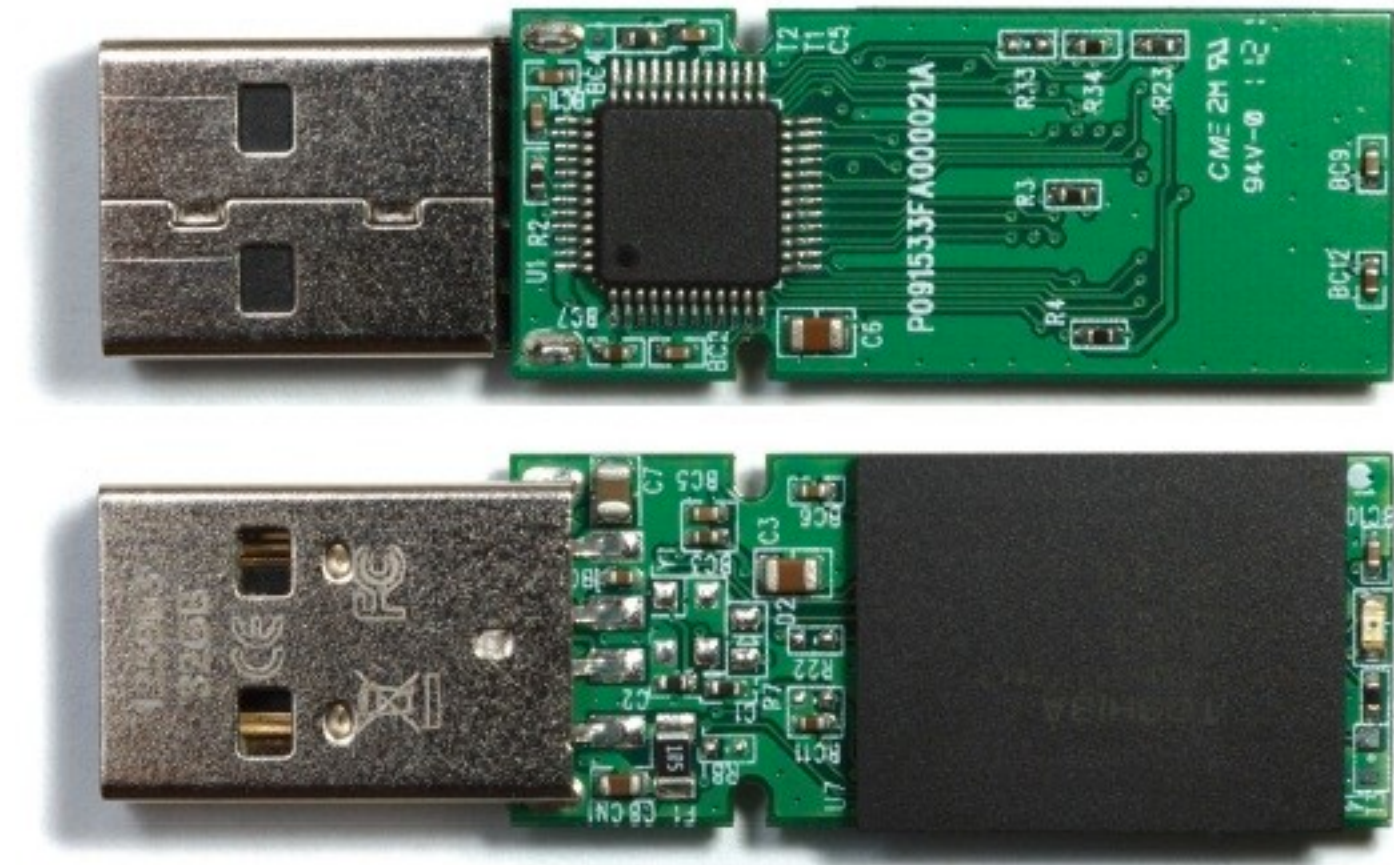
USB is Everywhere



What about USB security?

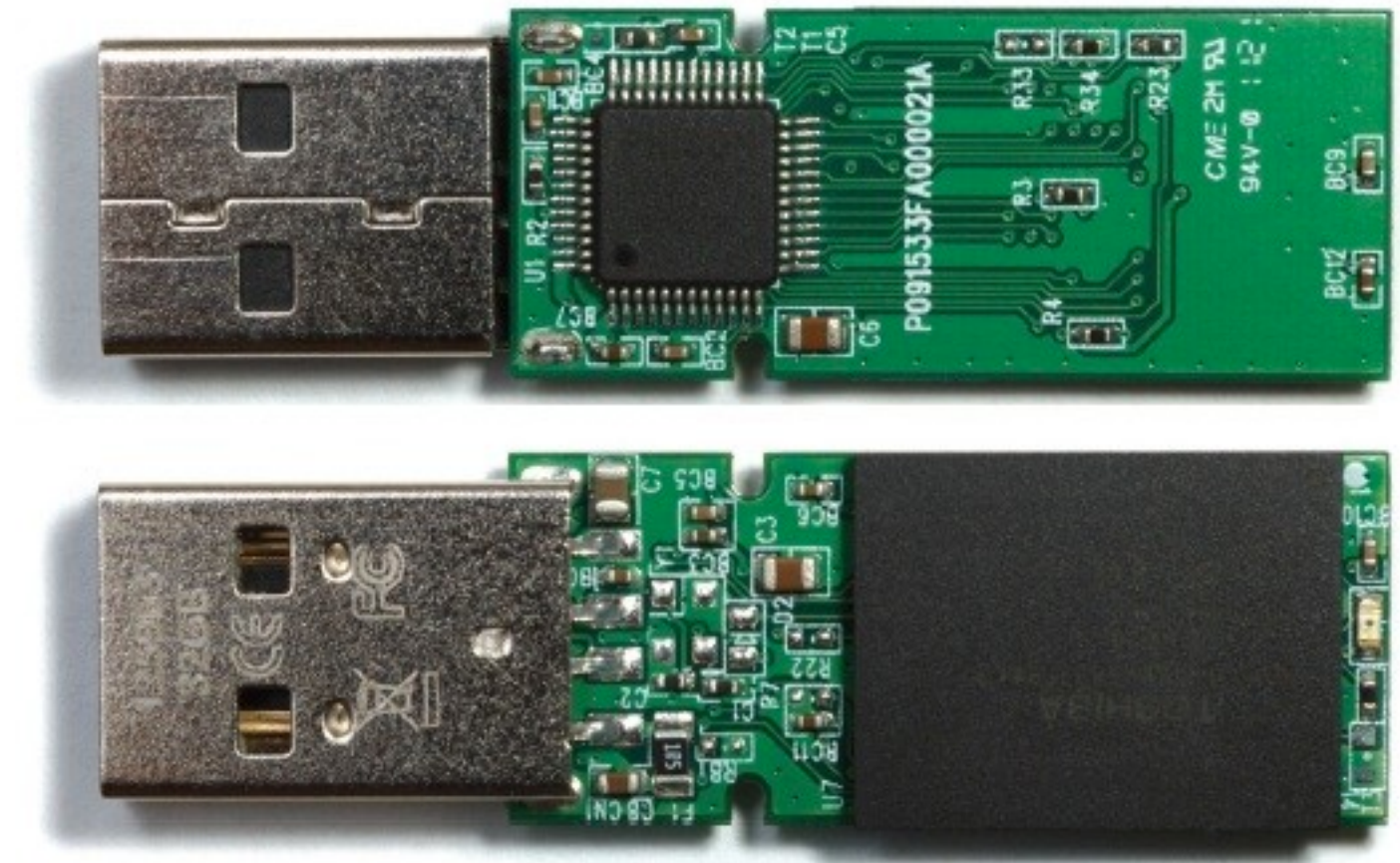
“ “The USB specifications support additional capabilities for security, but **original equipment manufacturers (OEM's) decide whether or not to implement these capabilities in their products.**” ”

— USB Implementers Forum, 2014



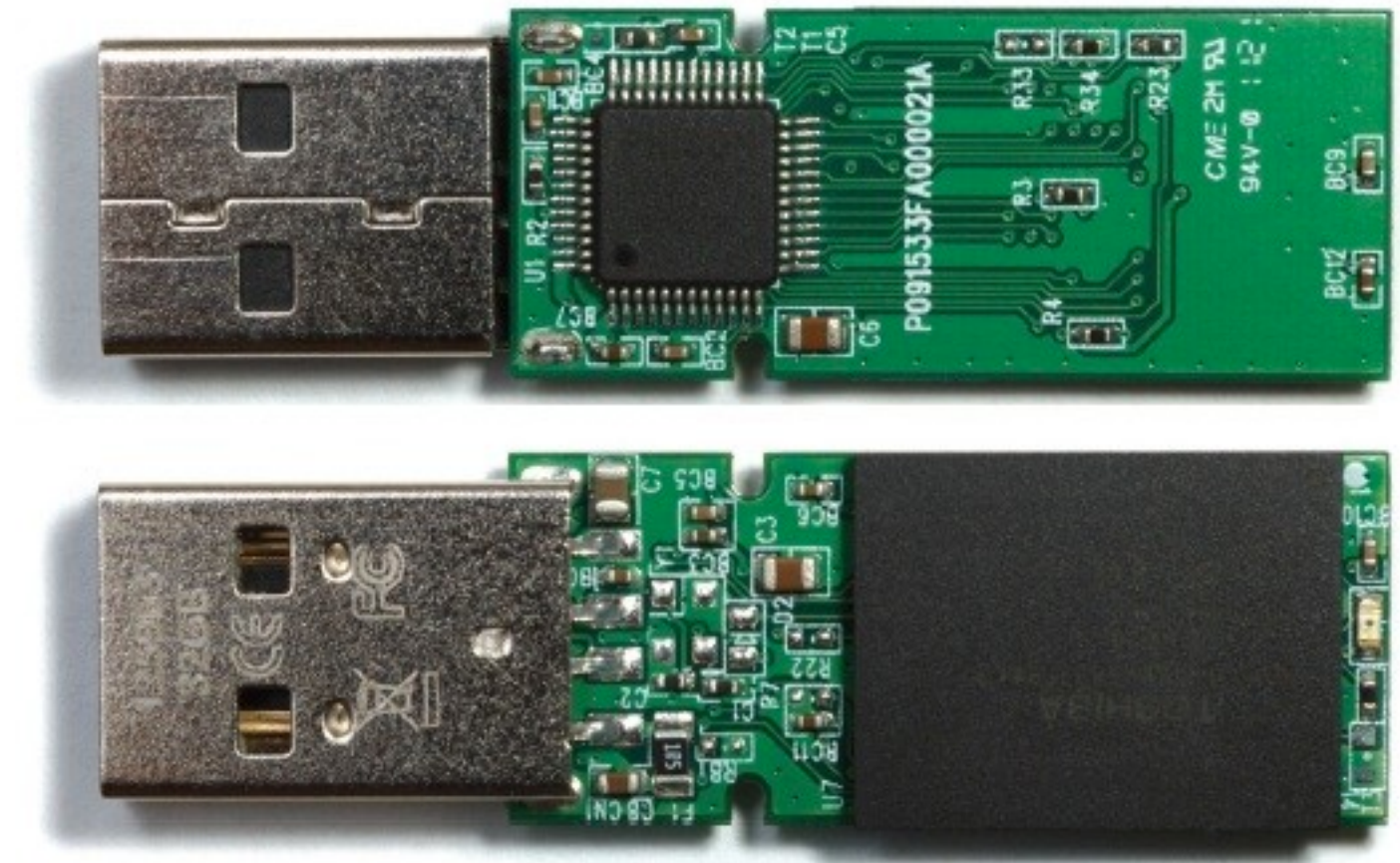
What is BadUSB?

- A reprogrammed USB device with modified functionality



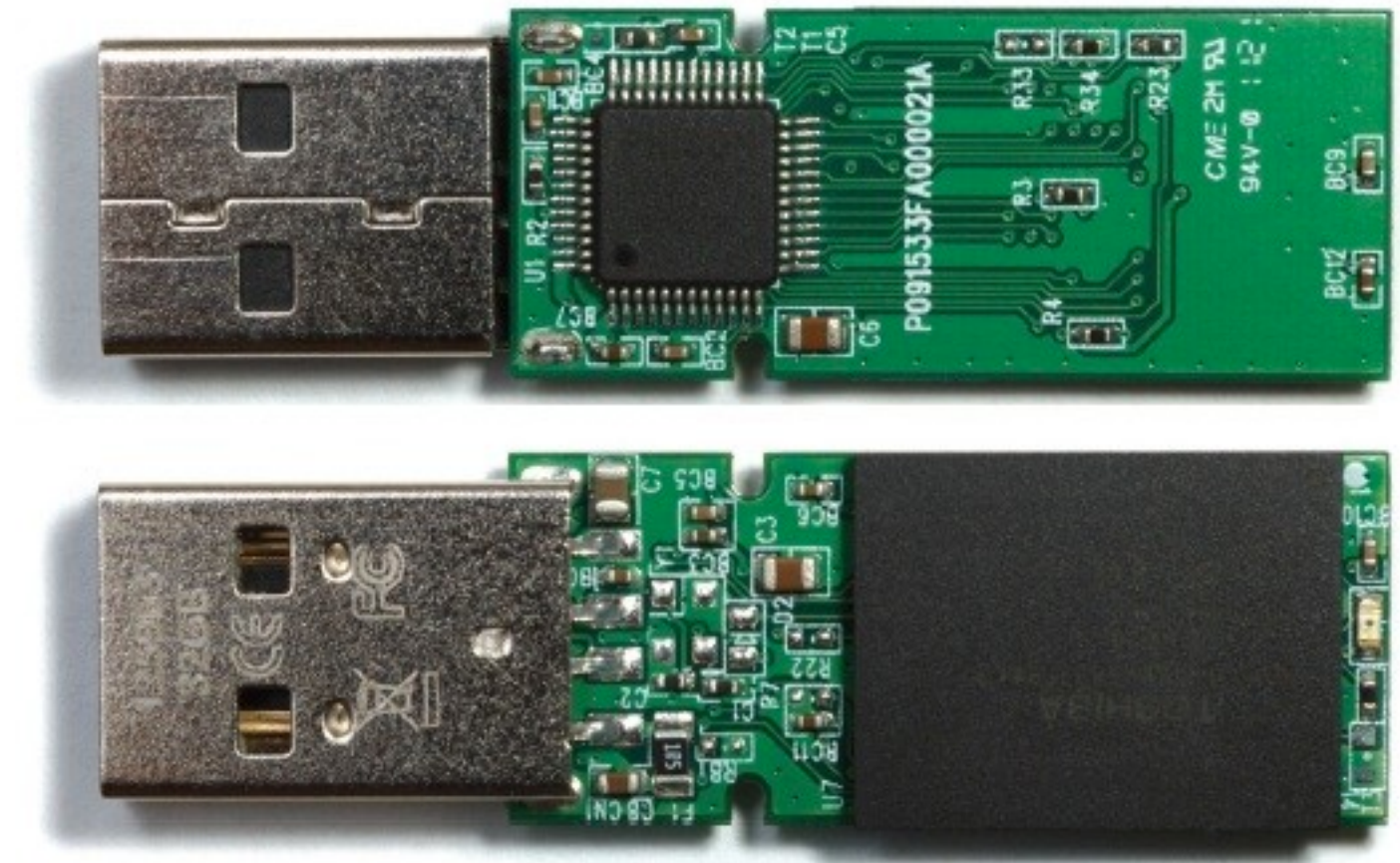
What is BadUSB?

- A reprogrammed USB device with modified functionality
- BadUSB exploits trust in physical device appearance



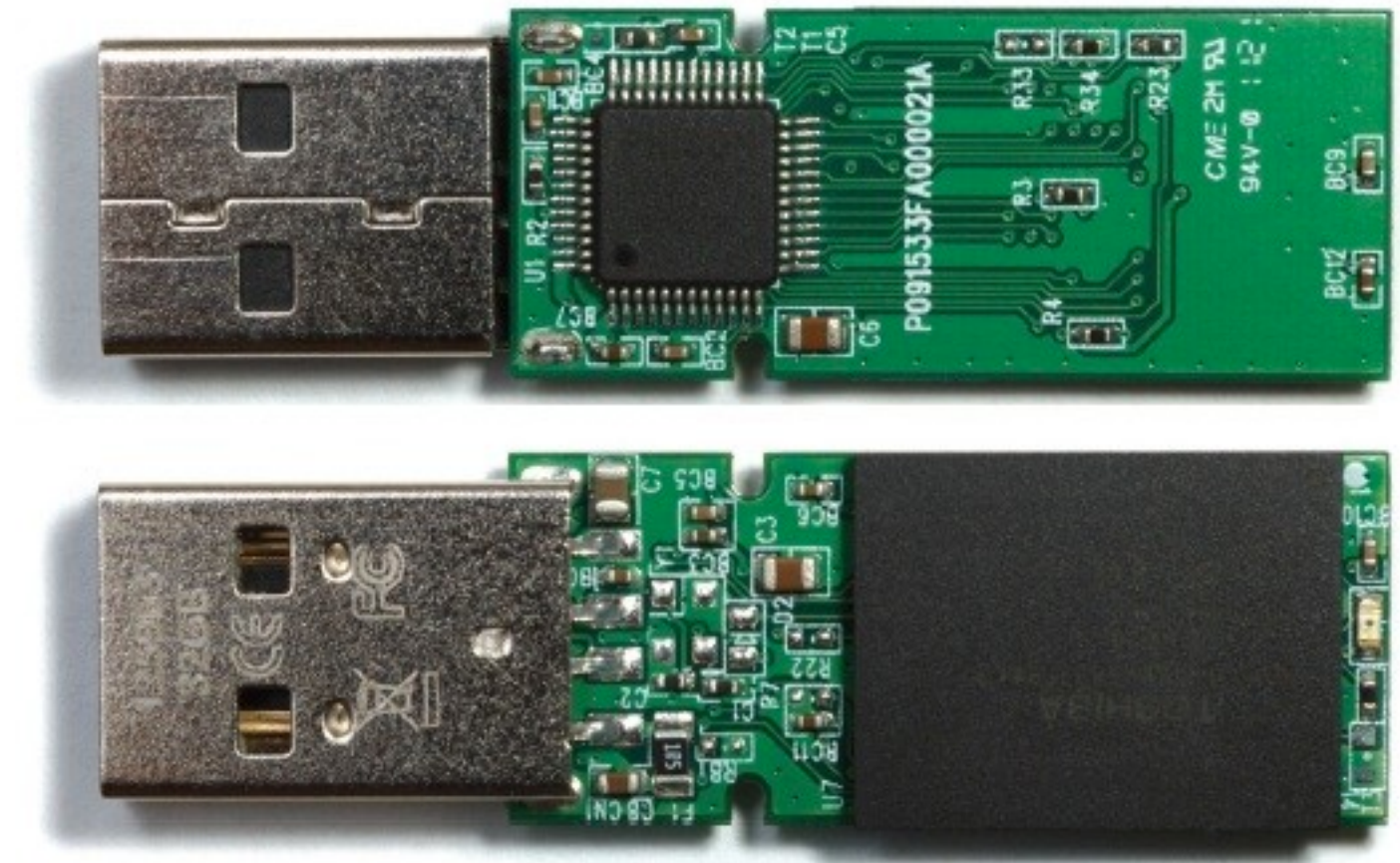
What is BadUSB?

- A reprogrammed USB device with modified functionality
- BadUSB exploits trust in physical device appearance
- **Example:** USB Flash drive reflashed to become a *keyboard* or *network card*
 - Enables keystroke injection to quickly backdoor a system or hijack all network connections



What is BadUSB?

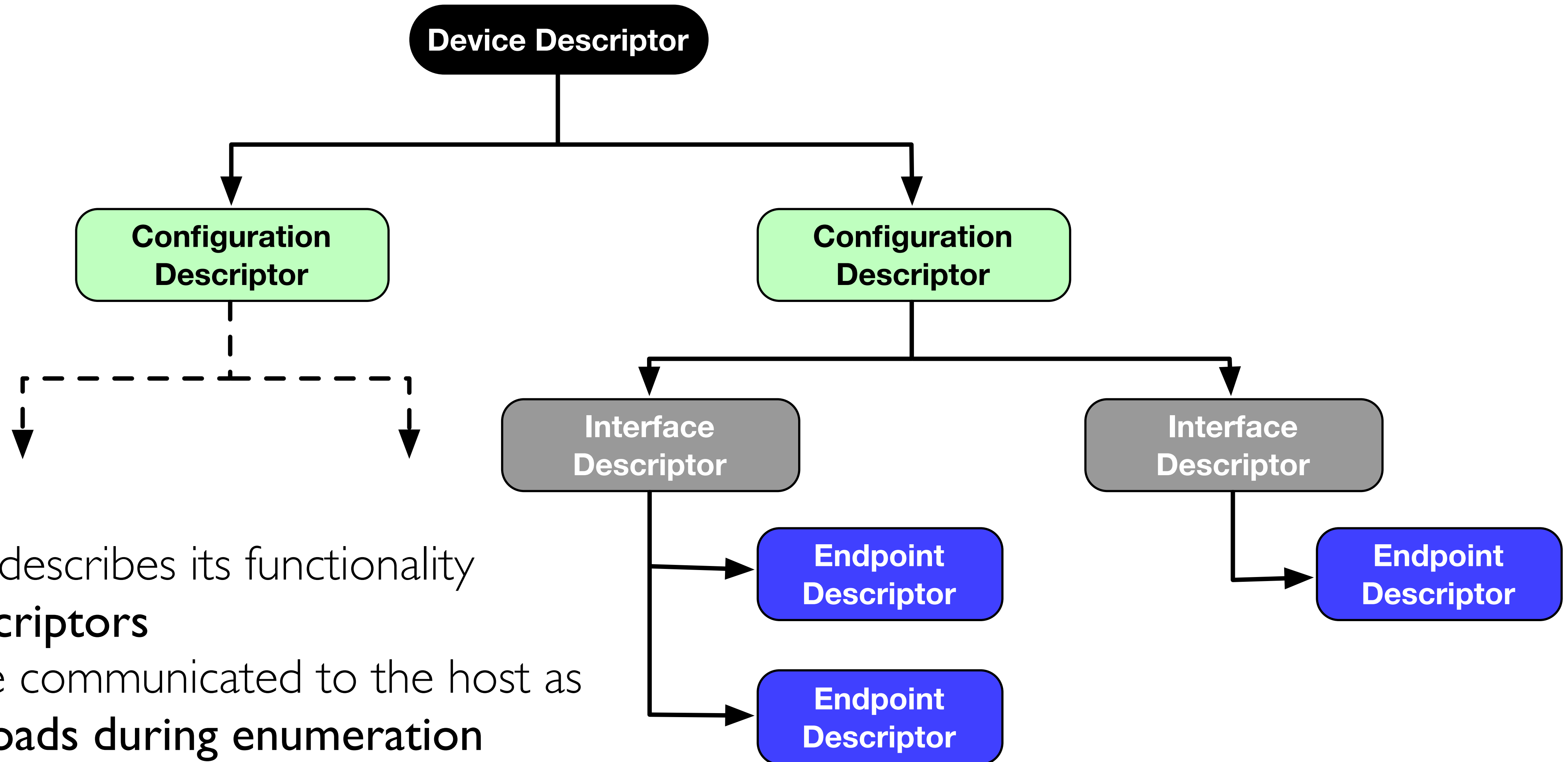
- A reprogrammed USB device with modified functionality
- BadUSB exploits trust in physical device appearance
- **Example:** USB Flash drive reflashed to become a *keyboard* or *network card*
 - Enables keystroke injection to quickly backdoor a system or hijack all network connections



**Operates completely within the USB Protocol.
No exploitation required and it is OS independent**

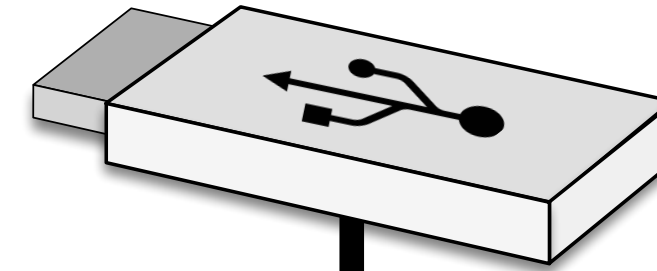


USB Descriptor Hierarchy

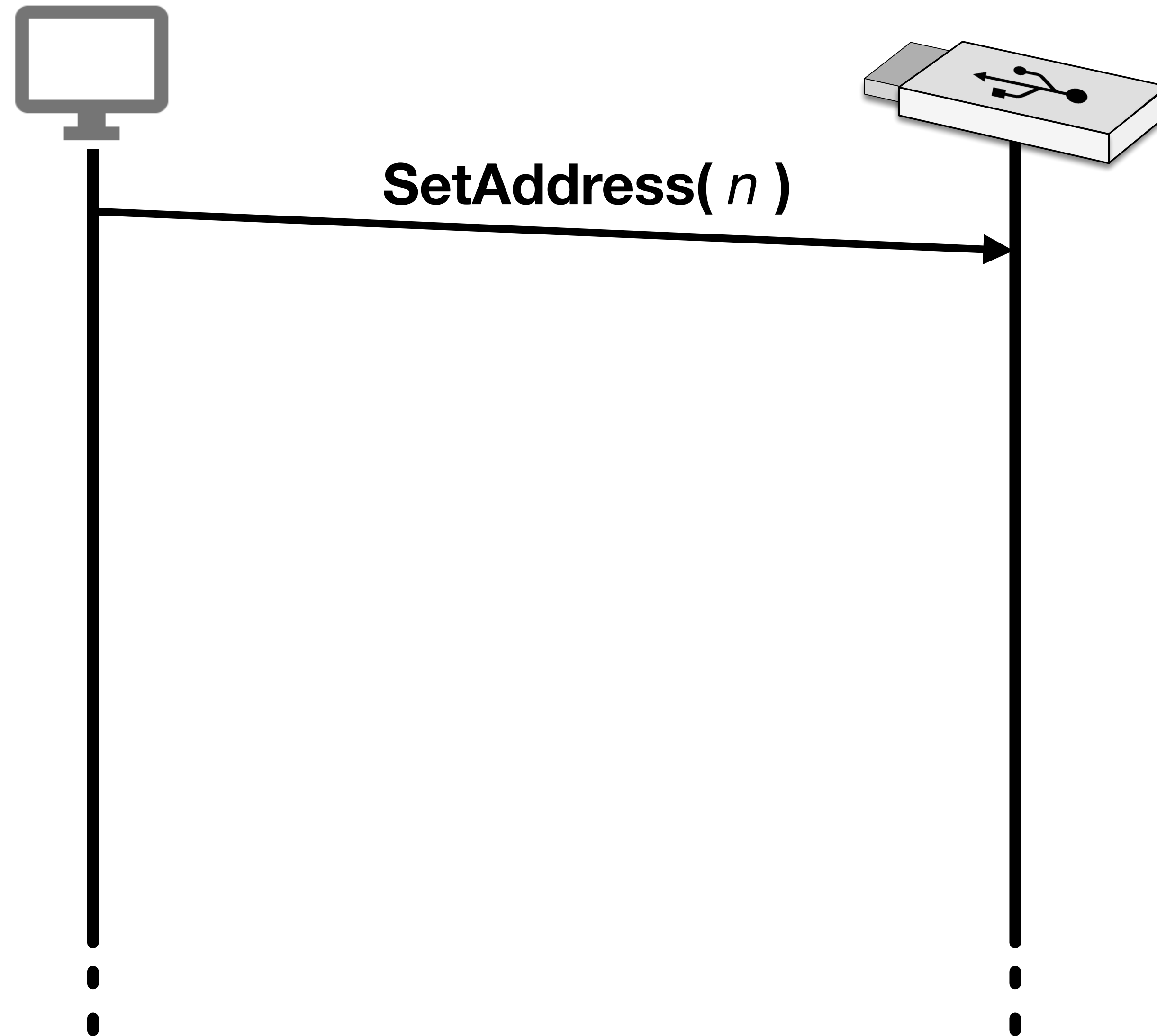


- A device describes its functionality using **descriptors**
- These are communicated to the host as **data payloads during enumeration**

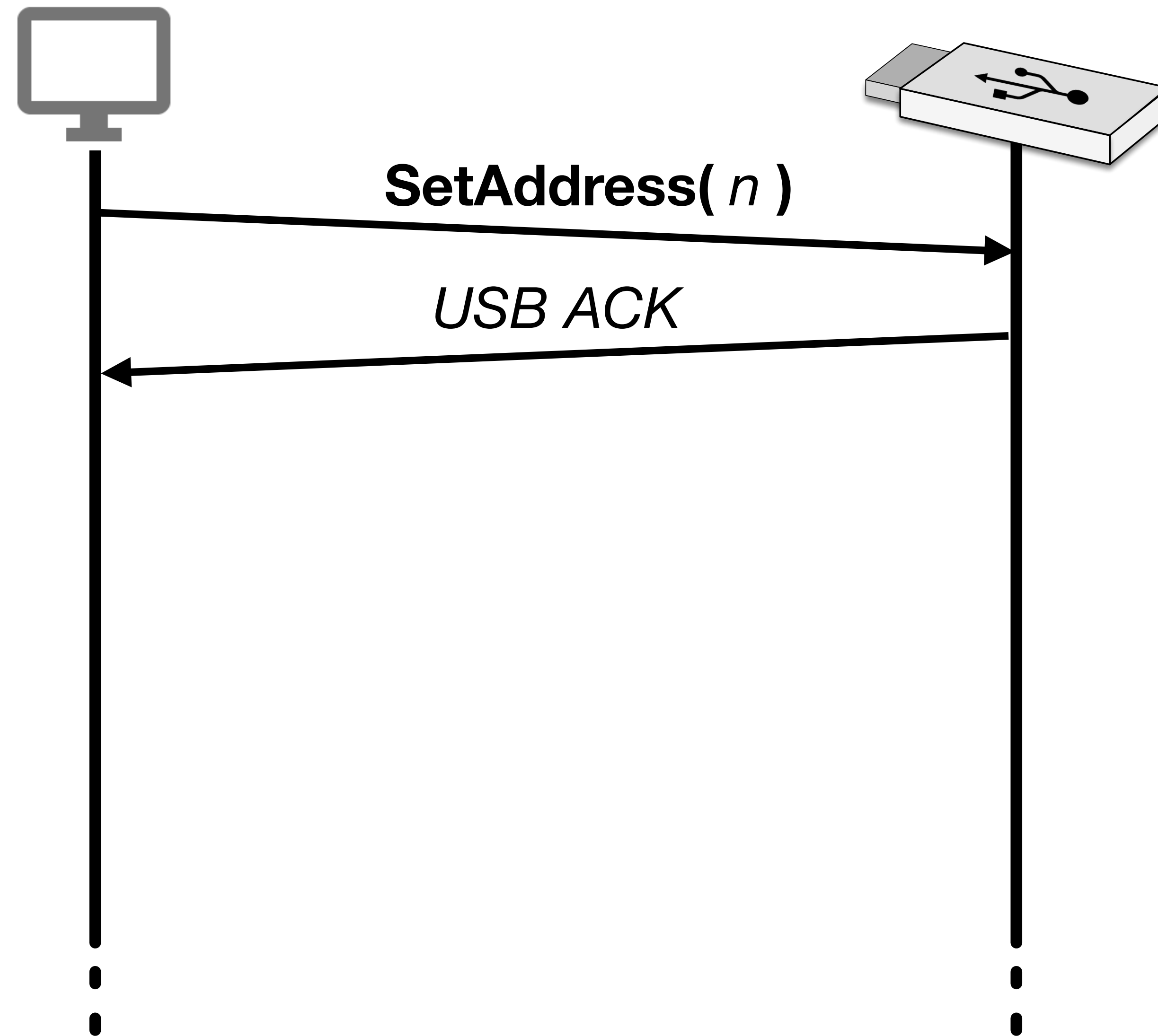
USB Enumeration



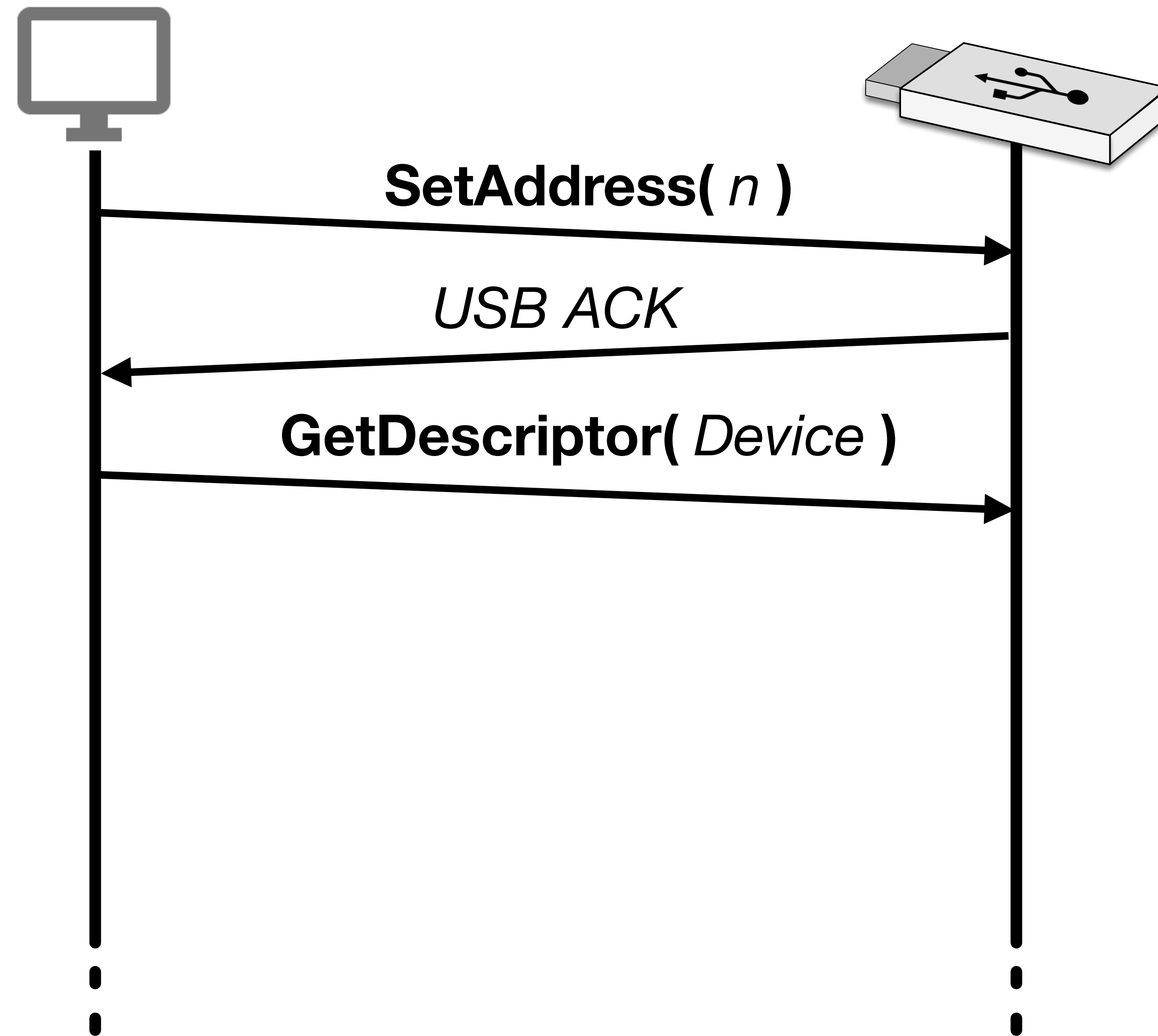
USB Enumeration



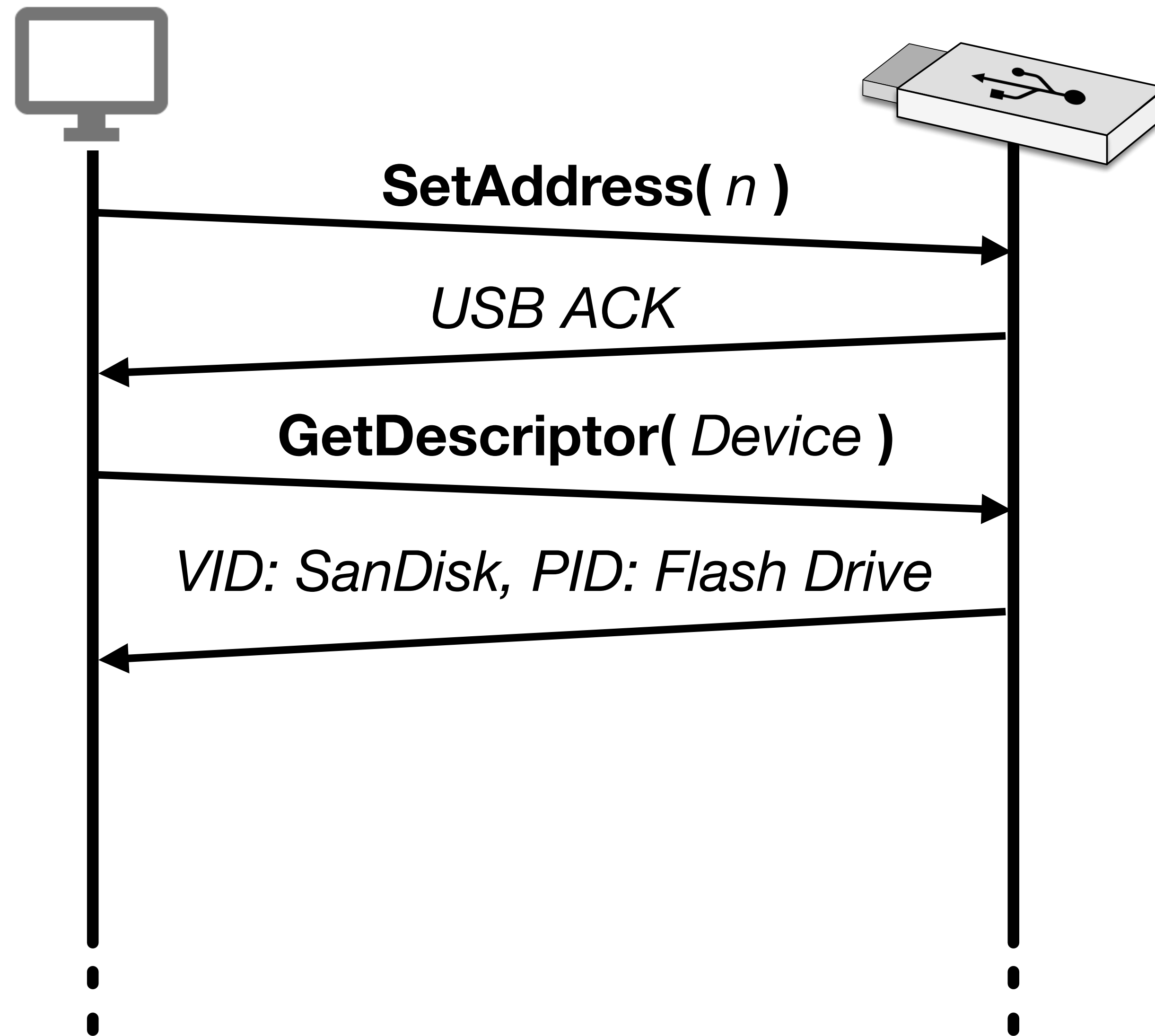
USB Enumeration



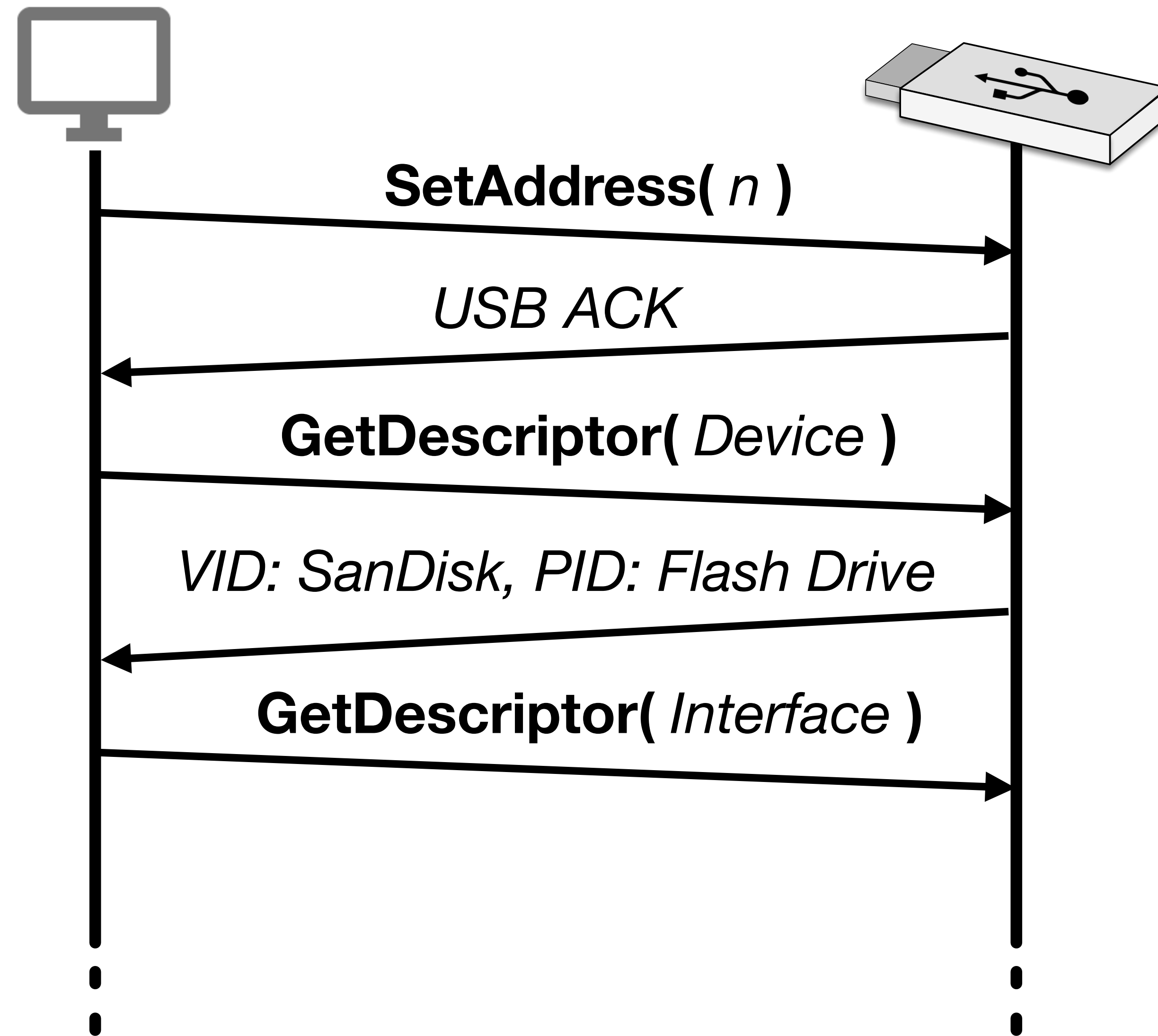
USB Enumeration



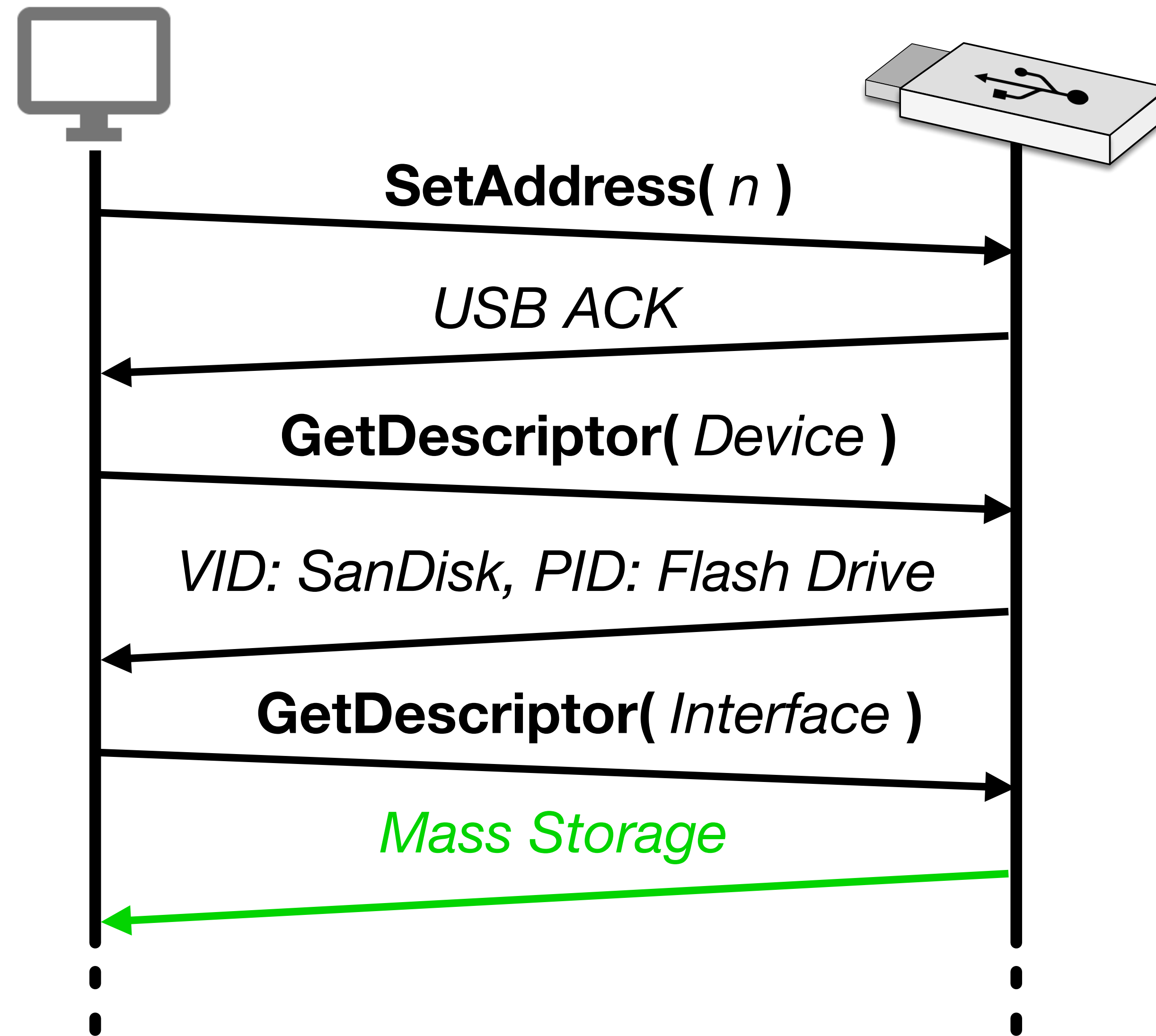
USB Enumeration



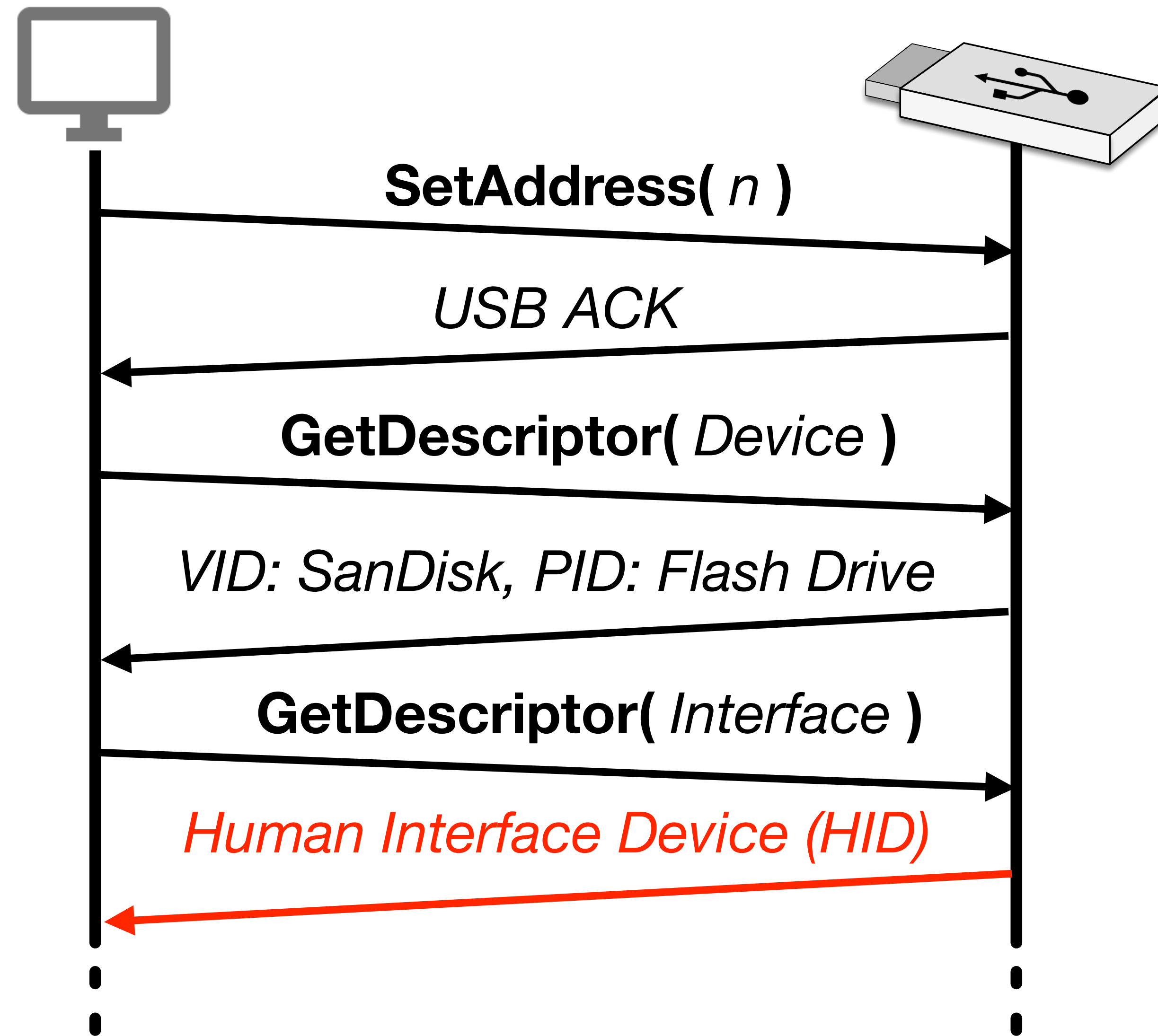
USB Enumeration



USB Enumeration



USB Enumeration



**Intel
8051**



- Analyze USB firmware to determine intent using static and symbolic analysis

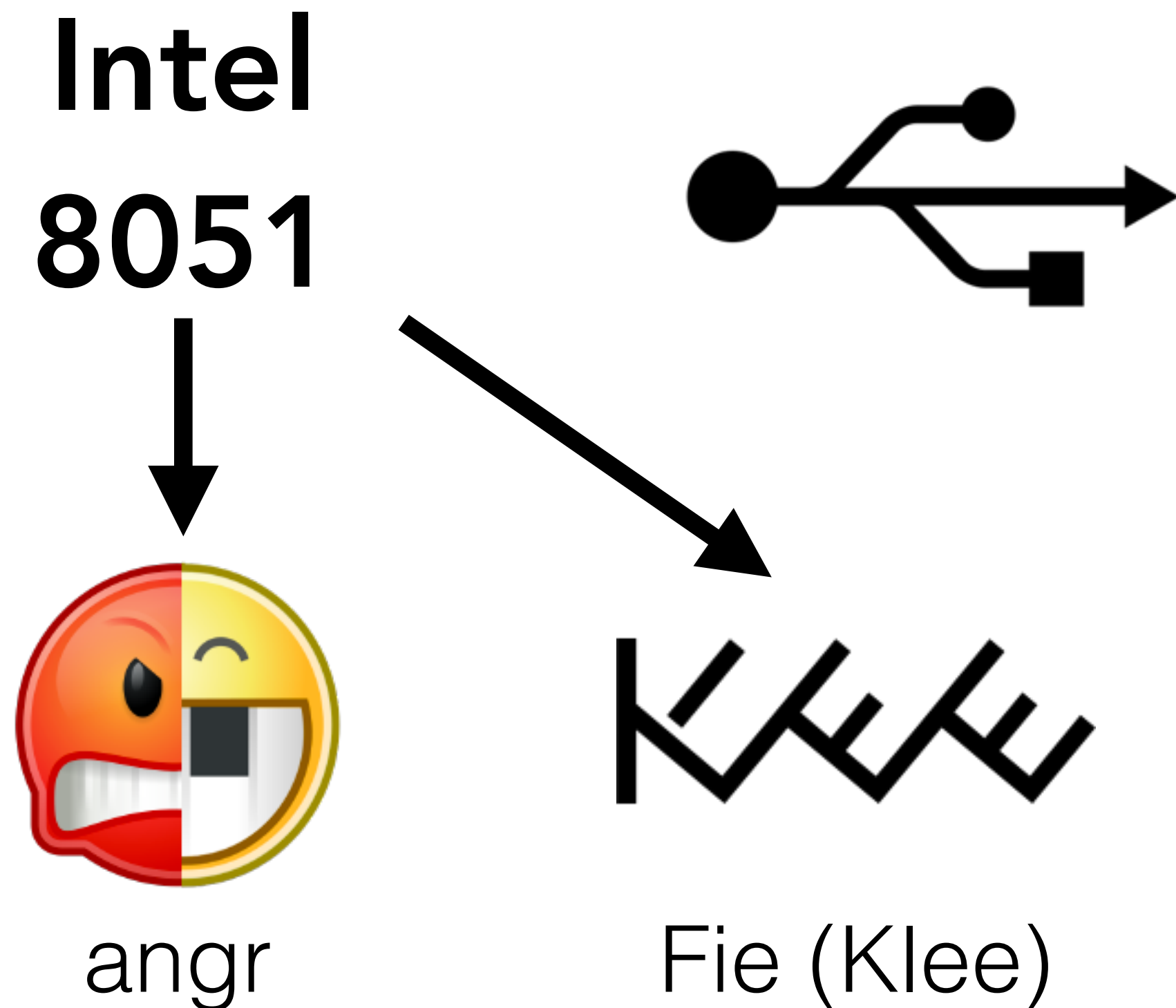


angr



Fie (Klee)

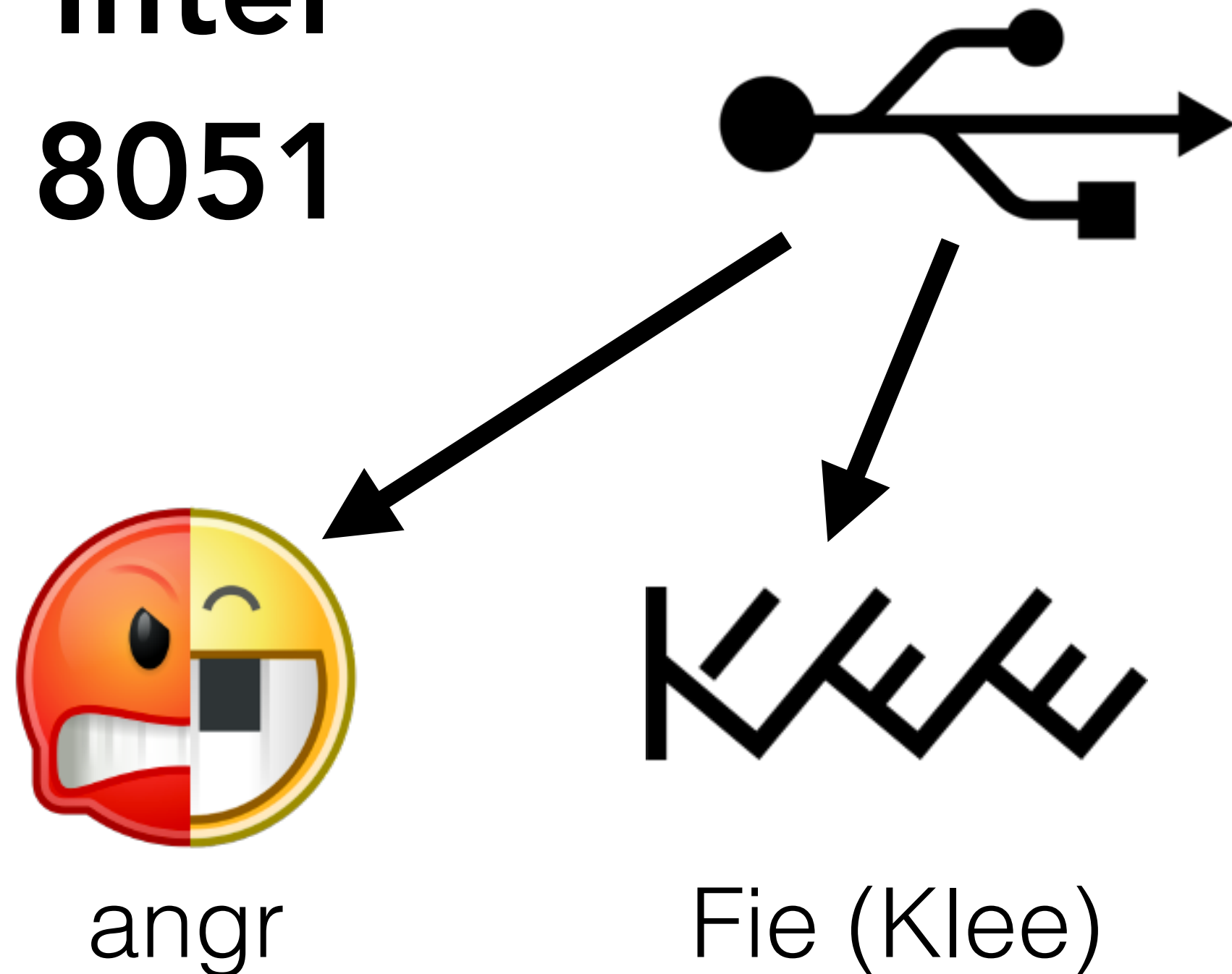
**Symbolic Execution
Engines**



- Analyze USB firmware to determine intent using static and symbolic analysis
- Extend existing symbolic execution support (Fie and angr) to the 8051 CPU architecture

**Symbolic Execution
Engines**

**Intel
8051**



**Symbolic Execution
Engines**

- Analyze USB firmware to determine intent using static and symbolic analysis
- Extend existing symbolic execution support (Fie and angr) to the 8051 CPU architecture
- Specialize and tailor symbolic engines via USB domain knowledge

**Intel
8051**



- Analyze USB firmware to determine intent using static and symbolic analysis
- Extend existing symbolic execution support (Fie and angr) to the 8051 CPU architecture
- Specialize and tailor symbolic engines via USB domain knowledge
- Develop a USB firmware semantic query engine which enables high-level analysis of firmware images



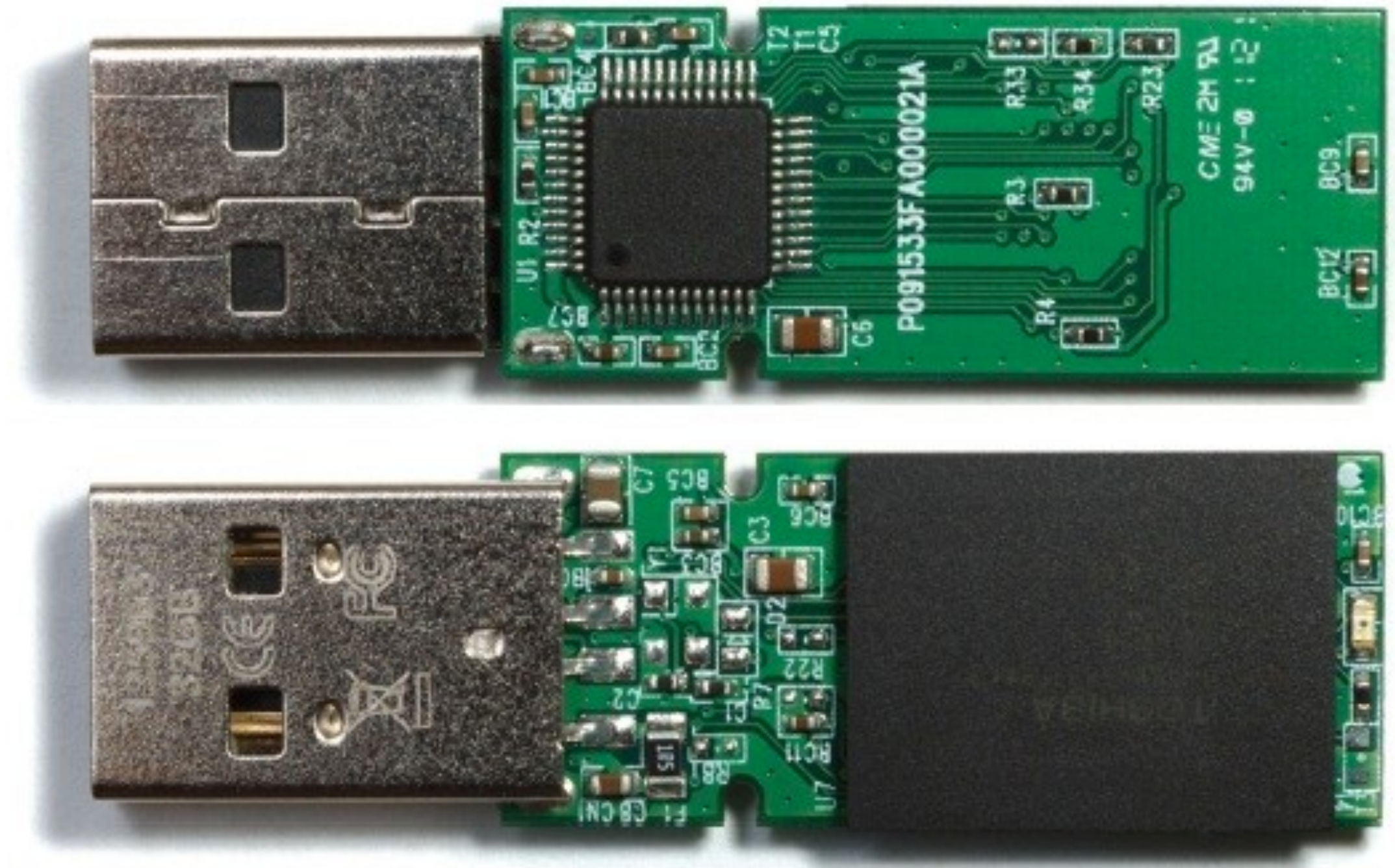
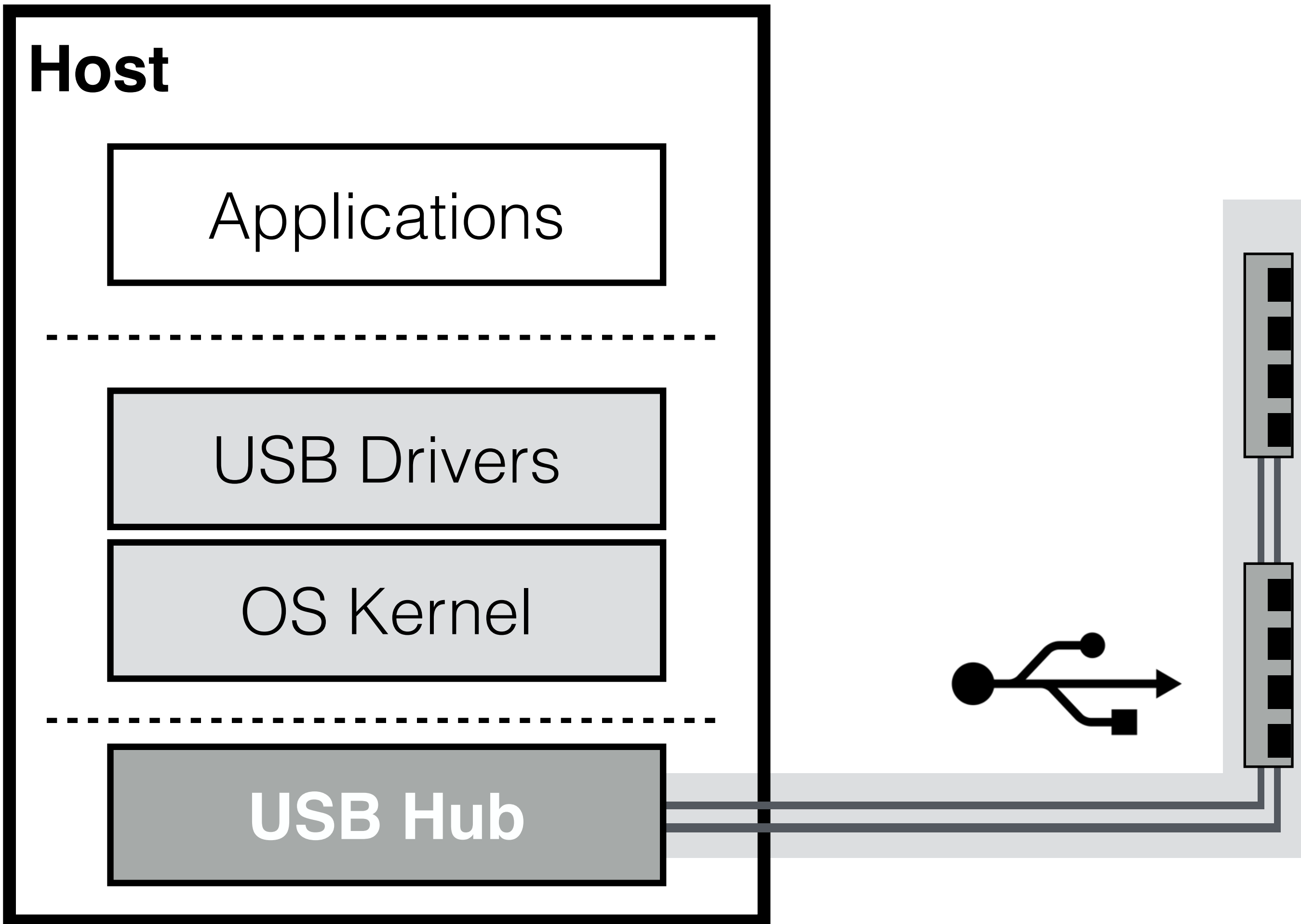
angr



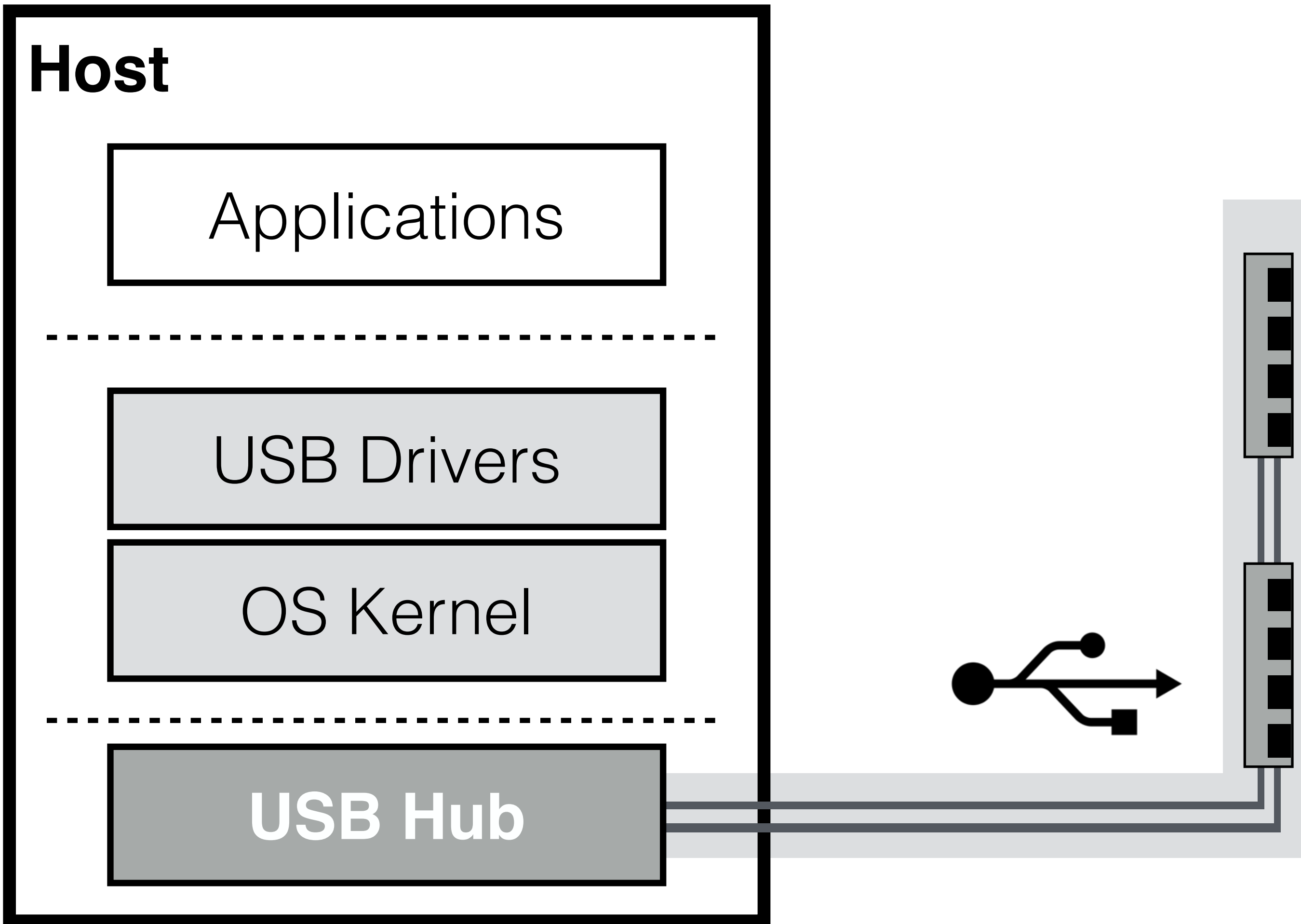
Fie (Klee)

**Symbolic Execution
Engines**

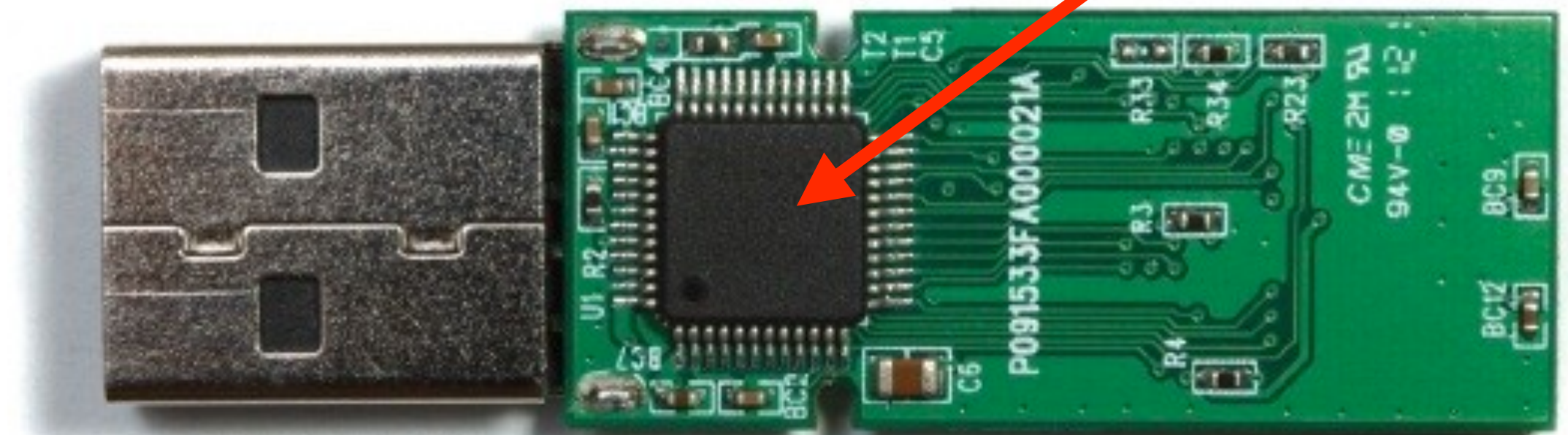
Where does FirmUSB fit?



Where does FirmUSB fit?

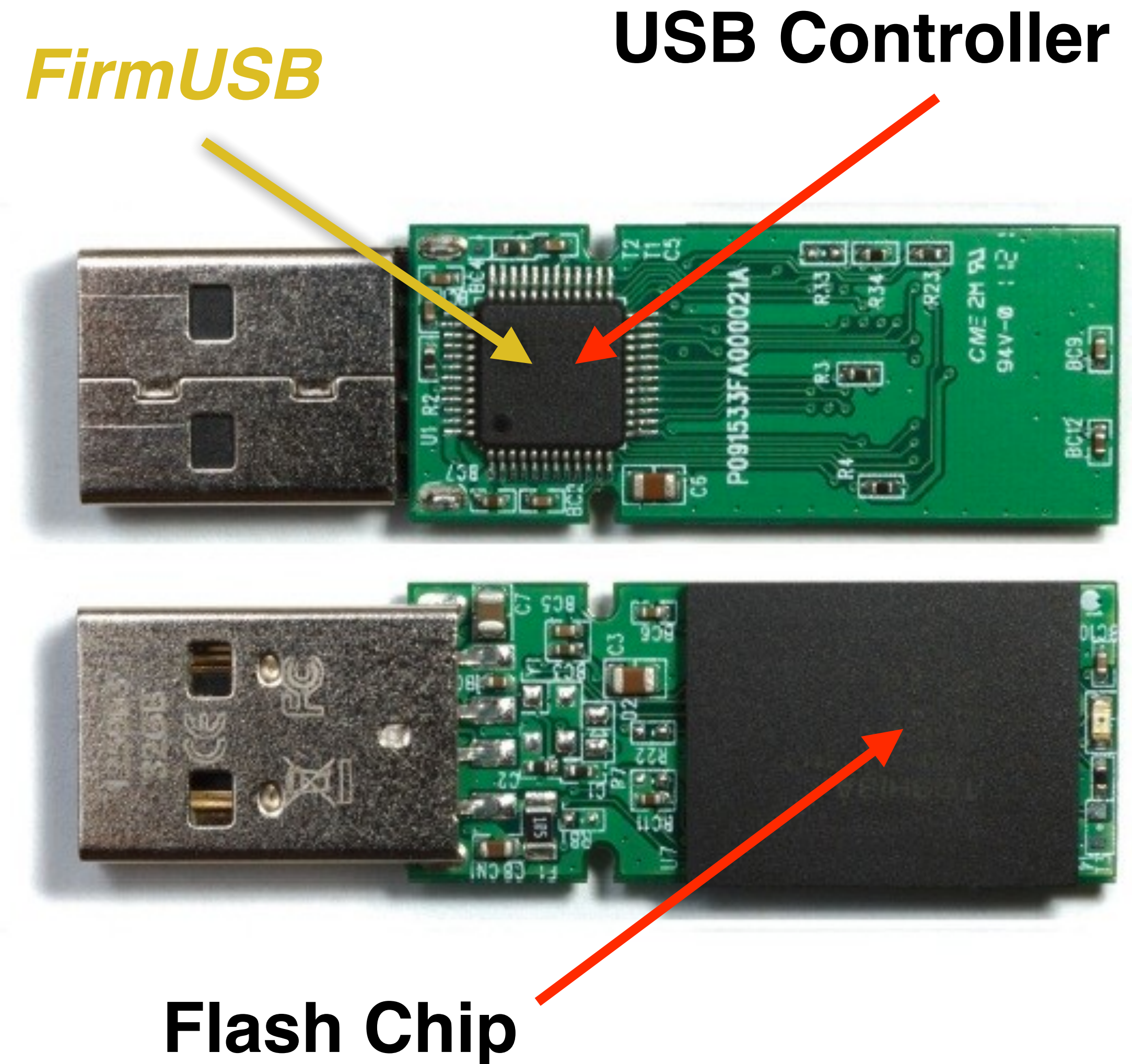
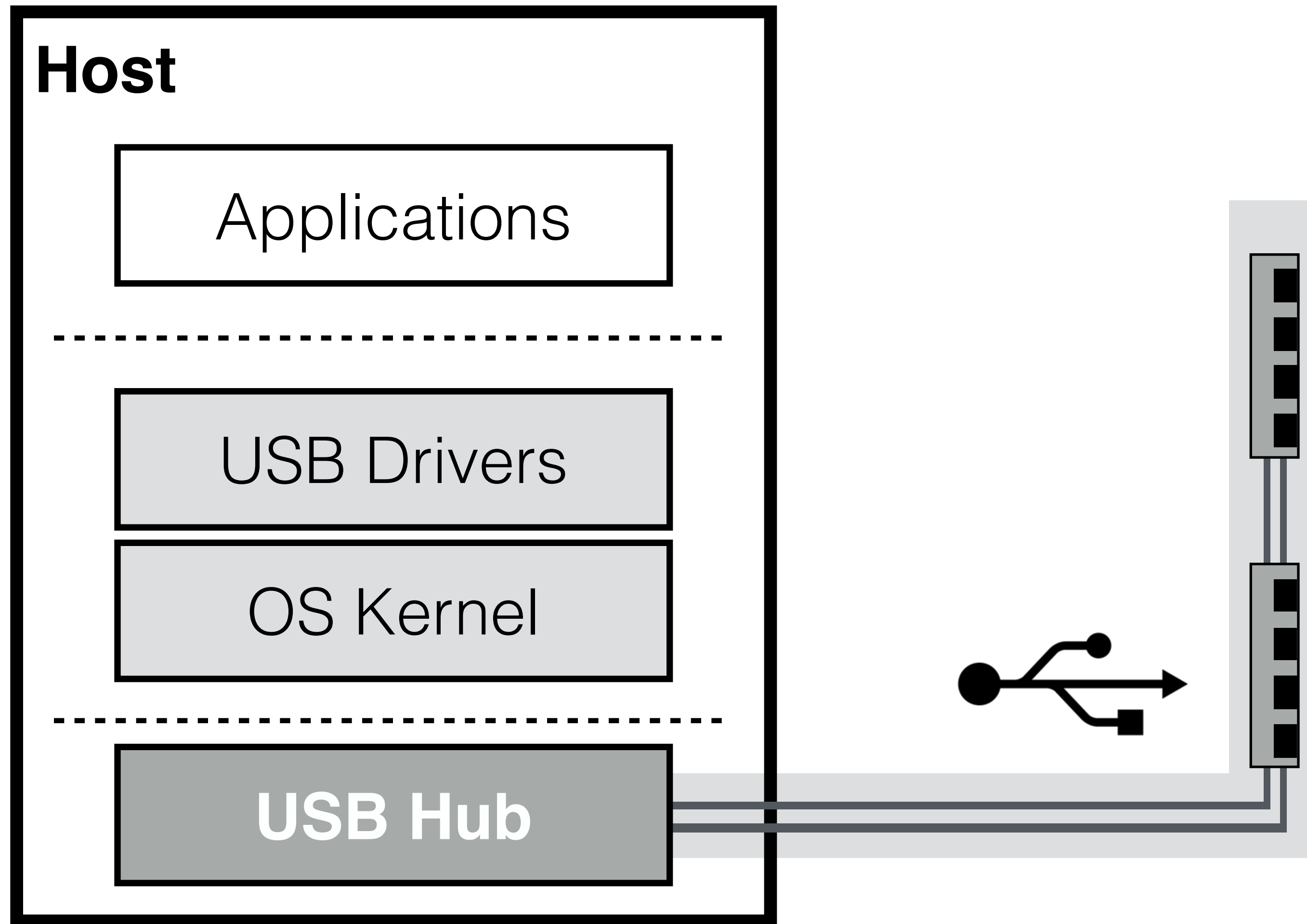


USB Controller



Flash Chip

Where does FirmUSB fit?



```
unsigned short pincode = 0x????;
```

Which pin codes are valid?

```
if(pincode % 10 == 0) {  
    if(pincode > 1000) {  
        if((pincode >> 2) & 0x43) {  
            printf("Correct!\n");  
        }  
    }  
}
```

```
unsigned short pincode = 0x????;
```

```
if(pincode % 10 == 0) {  
    if(pincode > 1000) {  
        if((pincode >> 2) & 0x43) {  
            printf("Correct!\n");  
        }  
    }  
}
```

Which pin codes are valid?

Use symbolic execution to find a program path that reaches 'Correct'


```
unsigned short pincode = 0x????;
```

```
if(pincode % 10 == 0) {  
    if(pincode > 1000) {  
        if((pincode >> 2) & 0x43) {  
            printf("Correct!\n");  
        }  
    }  
}
```

Which pin codes are valid?

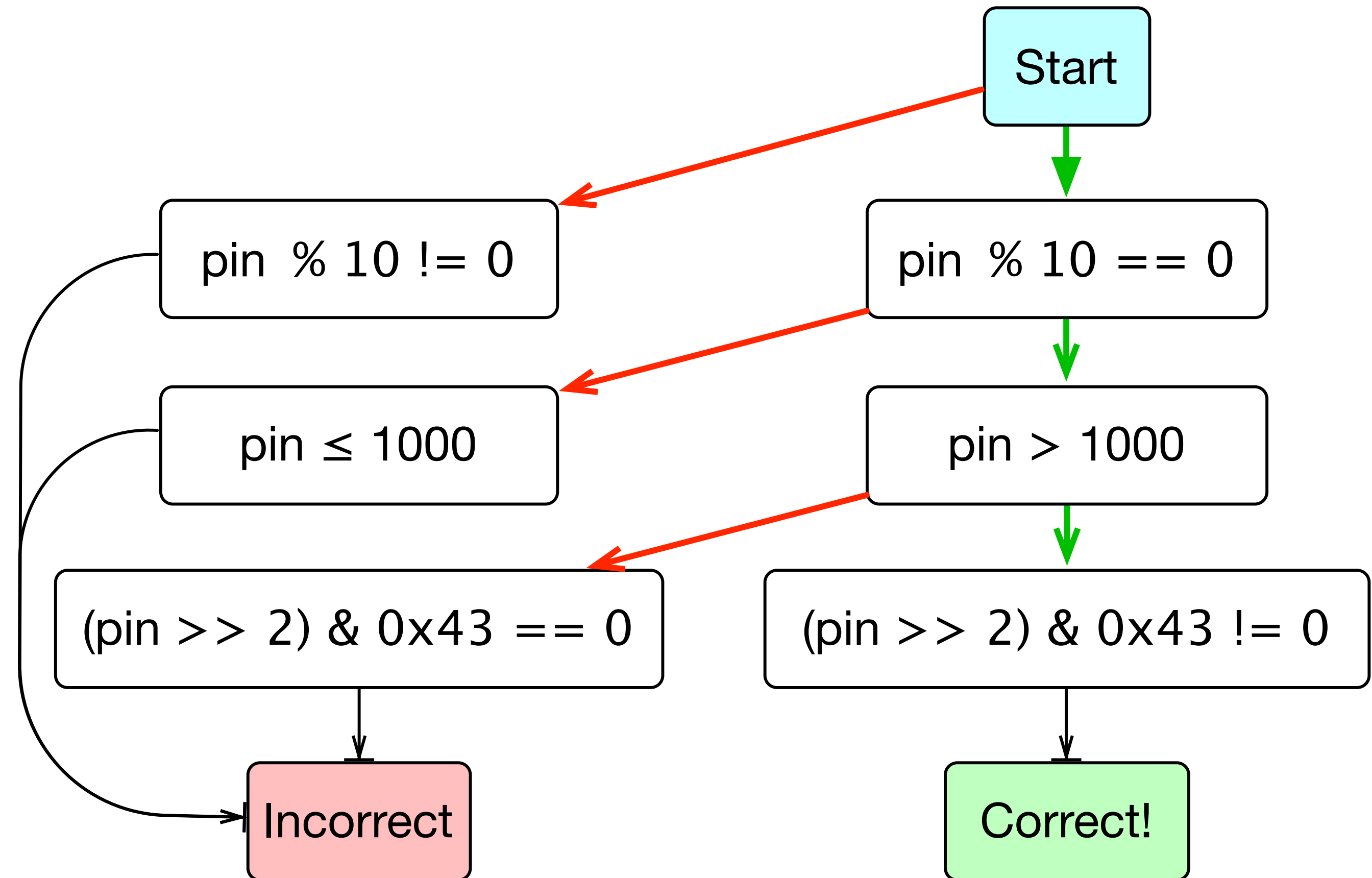
Use symbolic execution to find a program path that reaches 'Correct'

Solve for pin using constraint solver to find valid pins

Symbolic Execution (Cont.)

```
unsigned short pincode = 0x????;
```

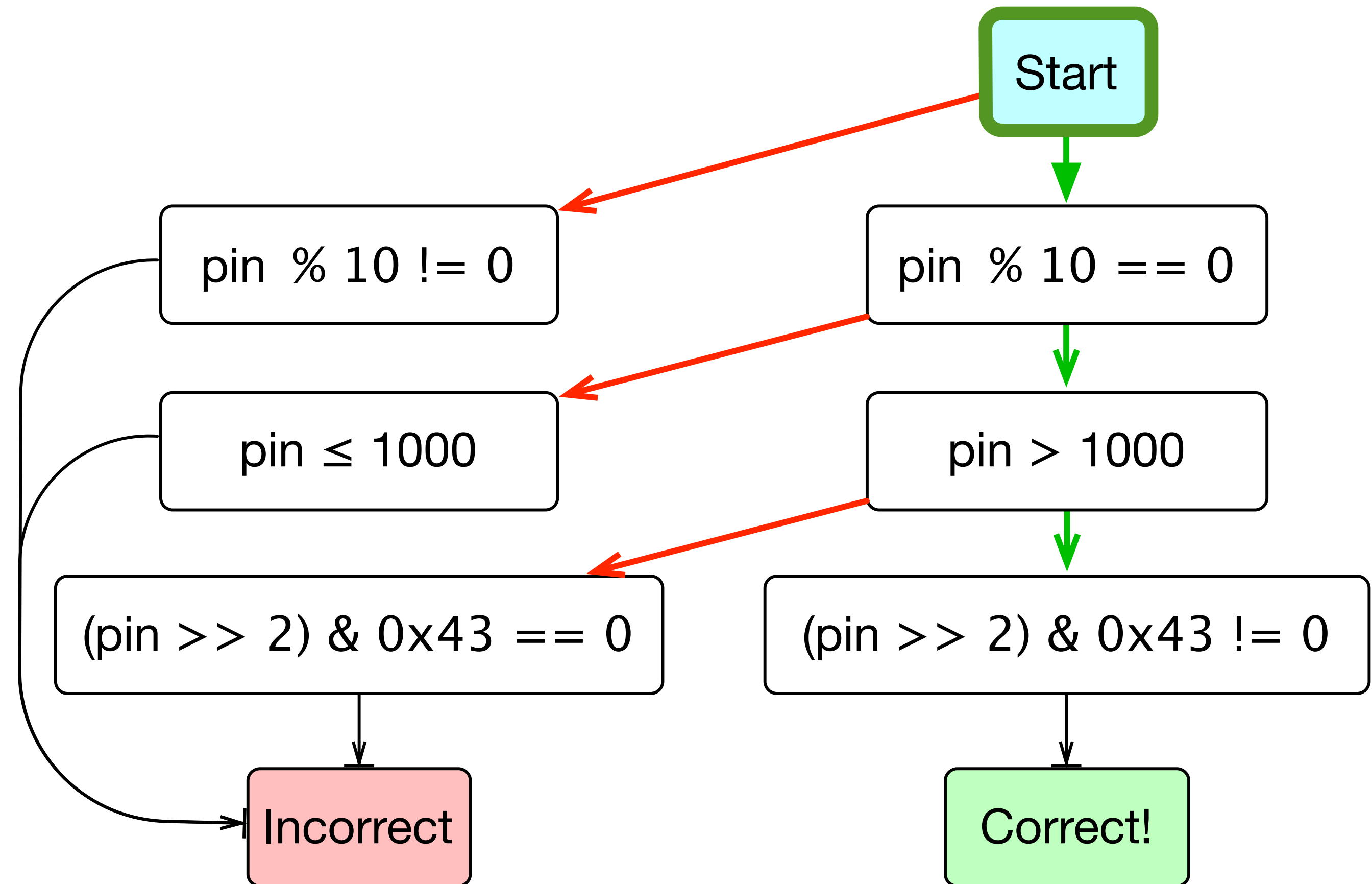
```
if(pincode % 10 == 0) {  
    if(pincode > 1000) {  
        if((pincode >> 2) & 0x43) {  
            printf("Correct!\n");  
        }  
    }  
}
```



Symbolic Execution (Cont.)

```
unsigned short pincode = 0x????;
```

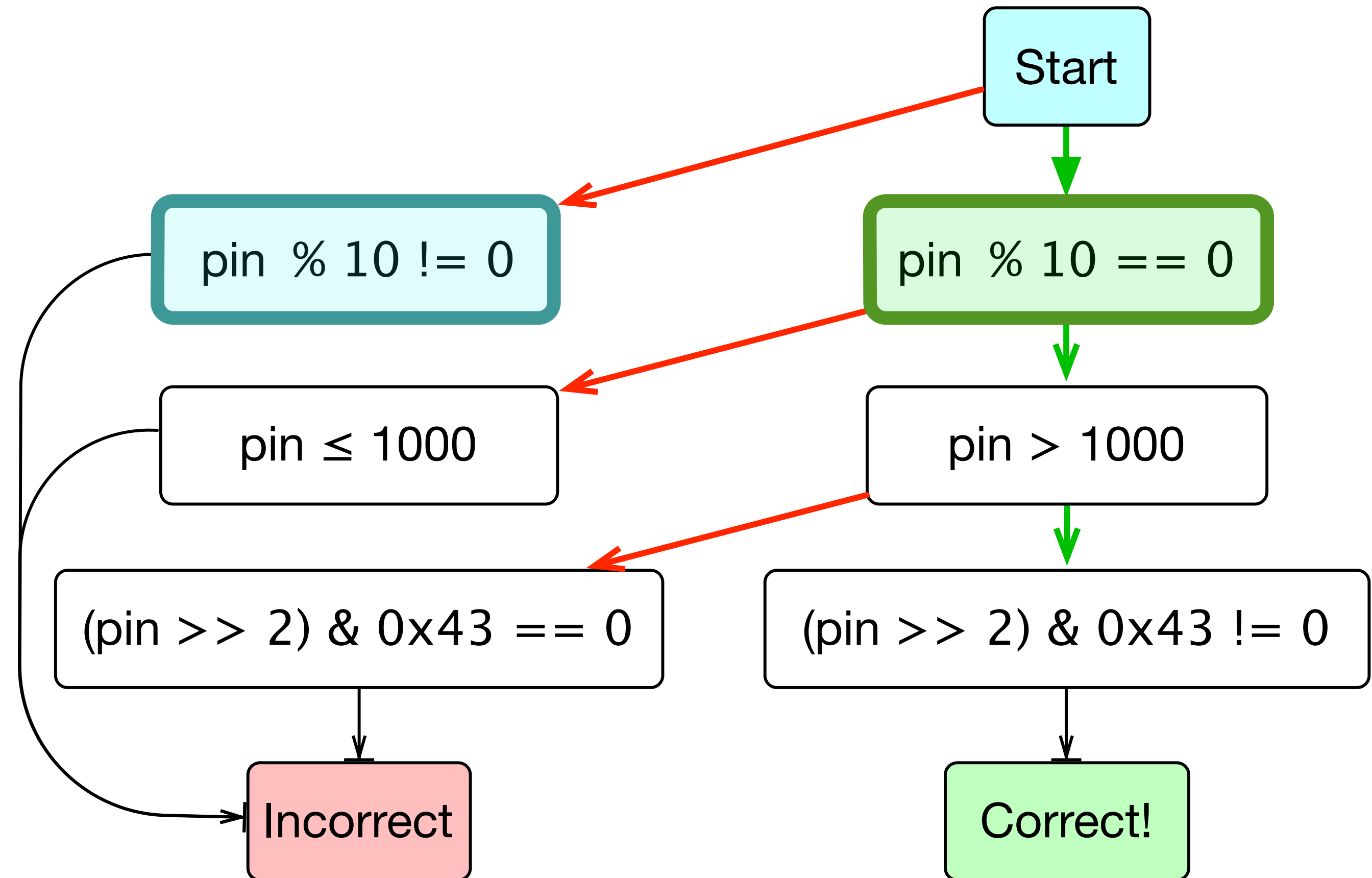
```
if(pincode % 10 == 0) {  
    if(pincode > 1000) {  
        if((pincode >> 2) & 0x43) {  
            printf("Correct!\n");  
        }  
    }  
}
```



Symbolic Execution (Cont.)

```
unsigned short pincode = 0x????;
```

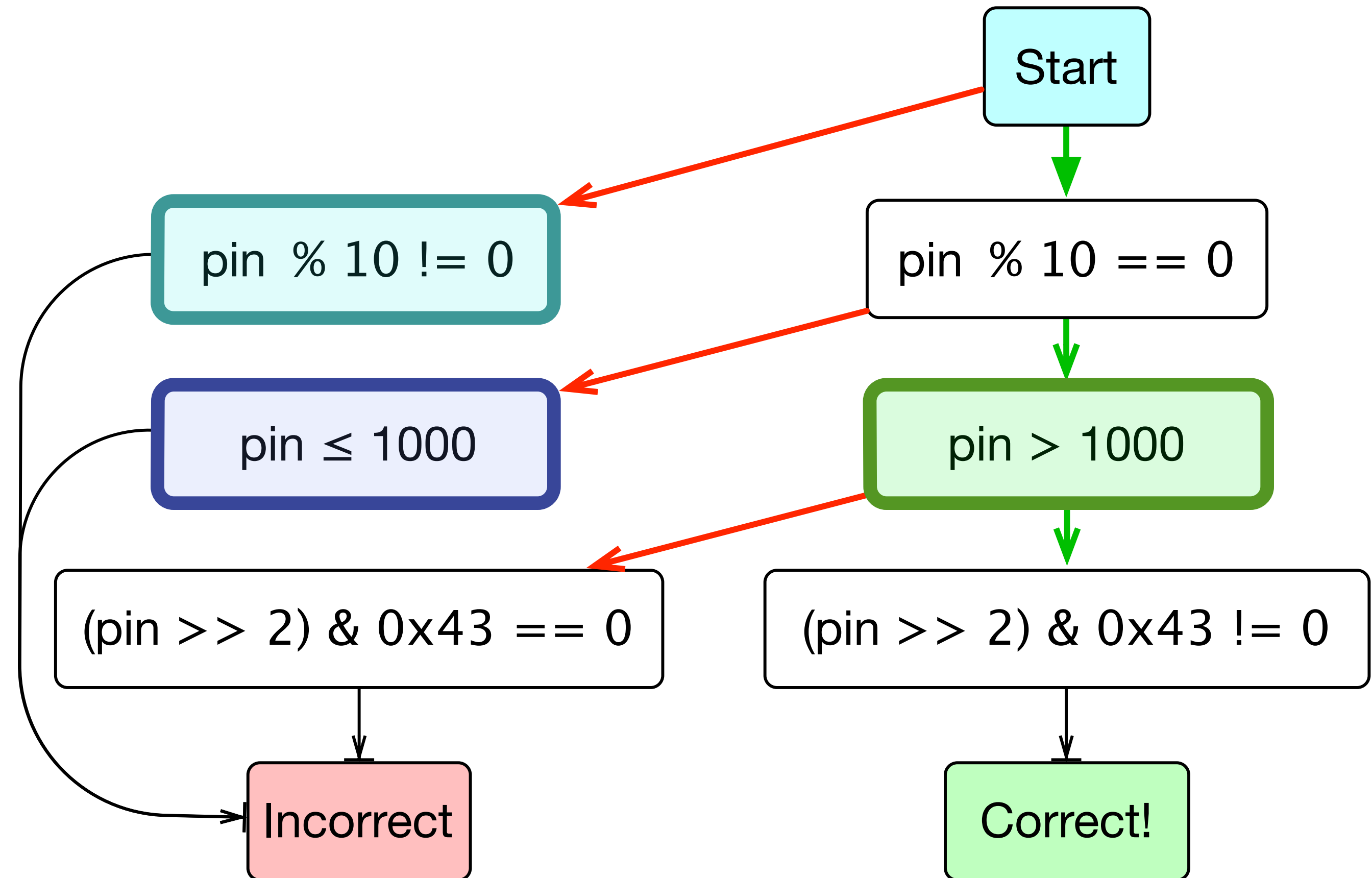
```
if(pincode % 10 == 0) {  
    if(pincode > 1000) {  
        if((pincode >> 2) & 0x43) {  
            printf("Correct!\n");  
        }  
    }  
}
```



Symbolic Execution (Cont.)

```
unsigned short pincode = 0x????;
```

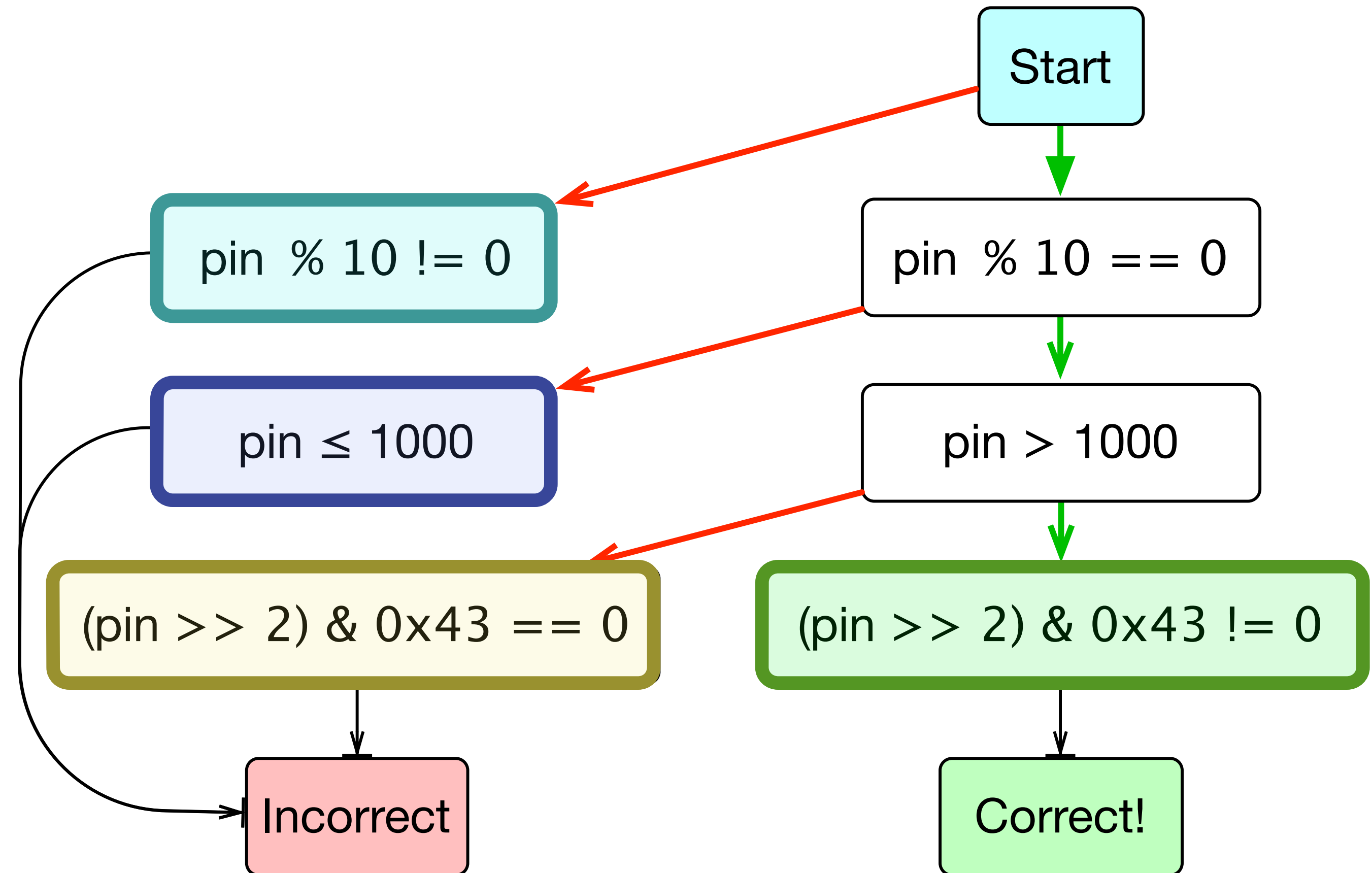
```
if(pincode % 10 == 0) {  
    if(pincode > 1000) {  
        if((pincode >> 2) & 0x43) {  
            printf("Correct!\n");  
        }  
    }  
}
```



Symbolic Execution (Cont.)

```
unsigned short pincode = 0x????;
```

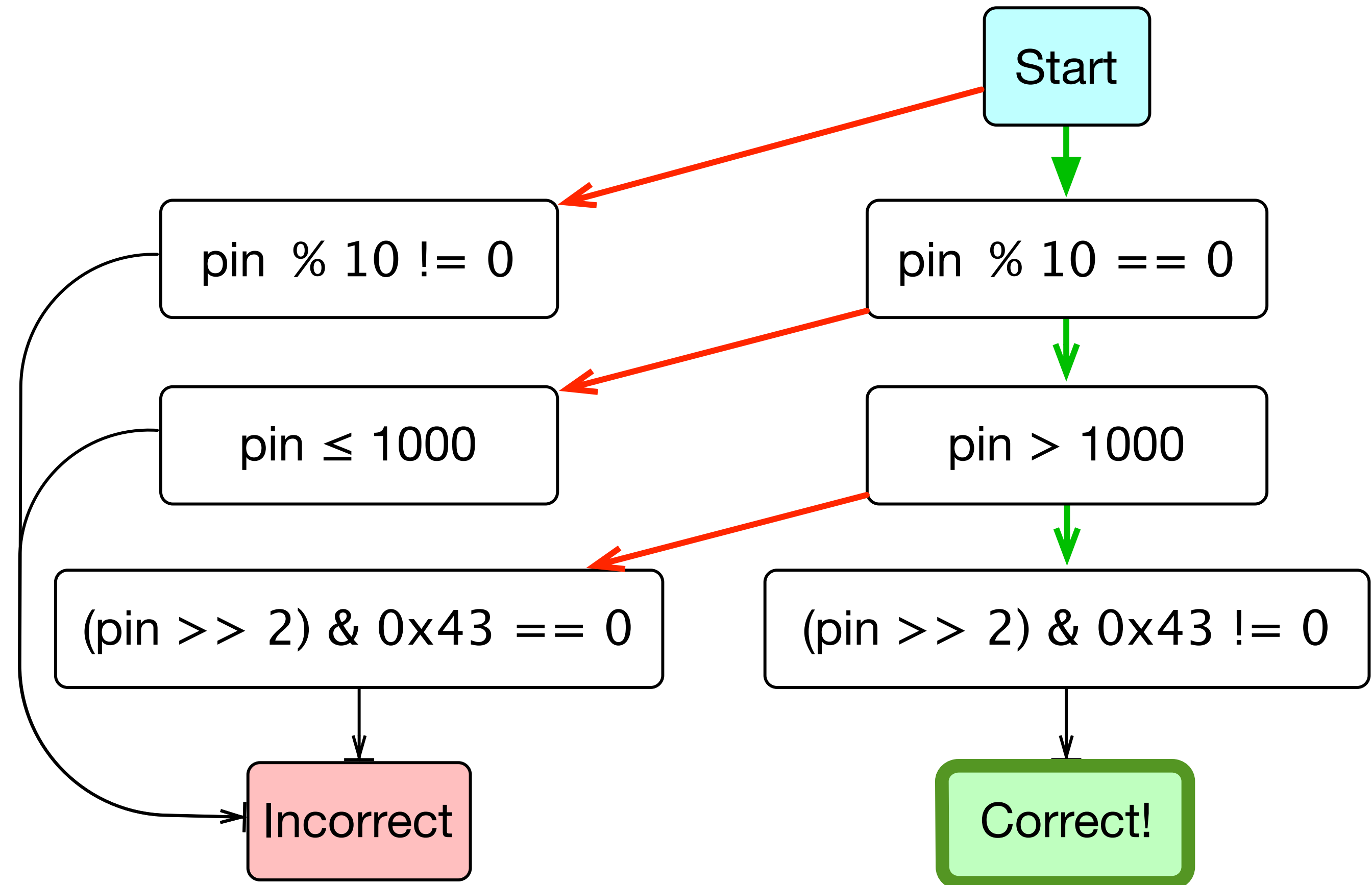
```
if(pincode % 10 == 0) {  
    if(pincode > 1000) {  
        if((pincode >> 2) & 0x43) {  
            printf("Correct!\n");  
        }  
    }  
}
```



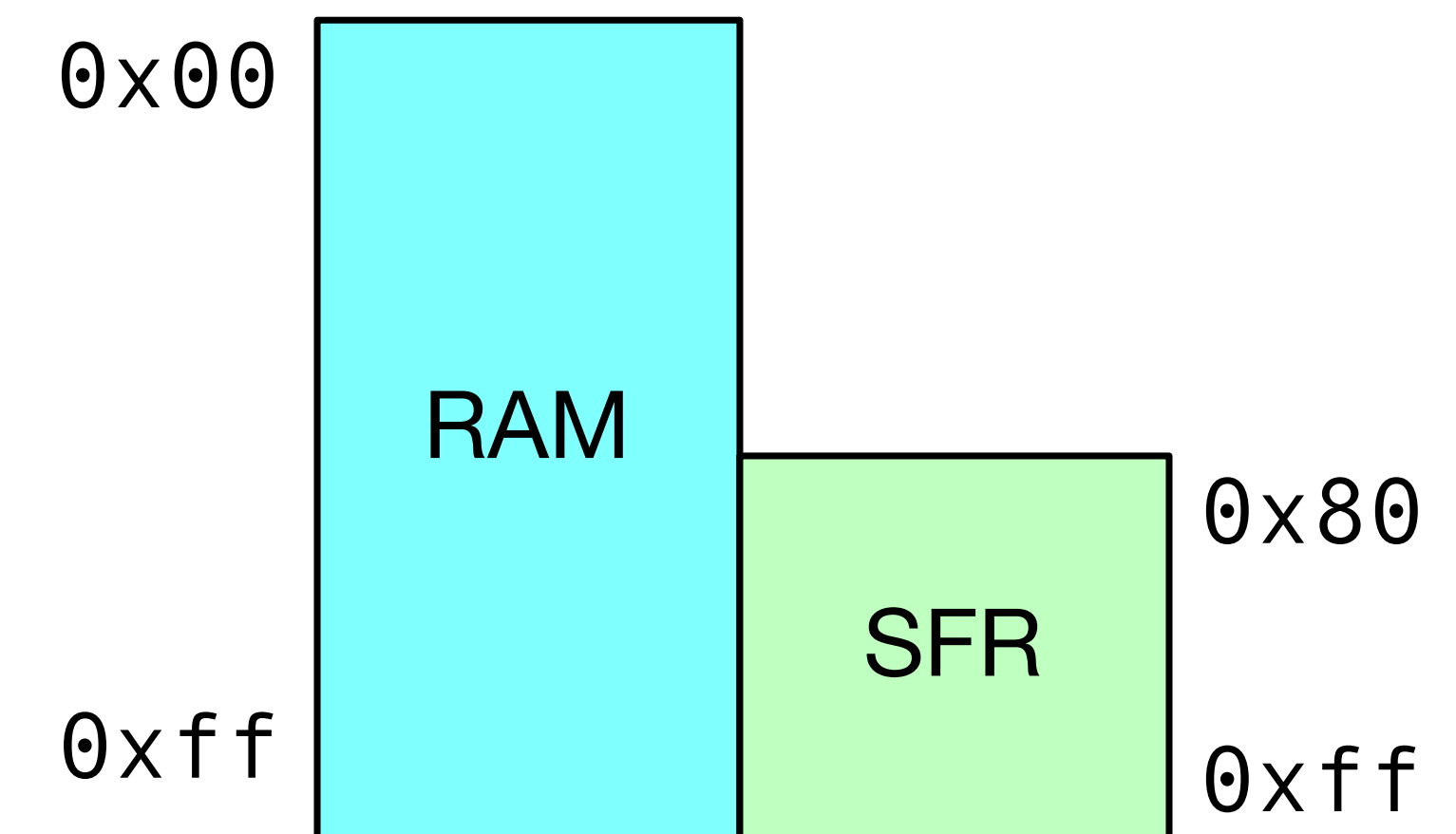
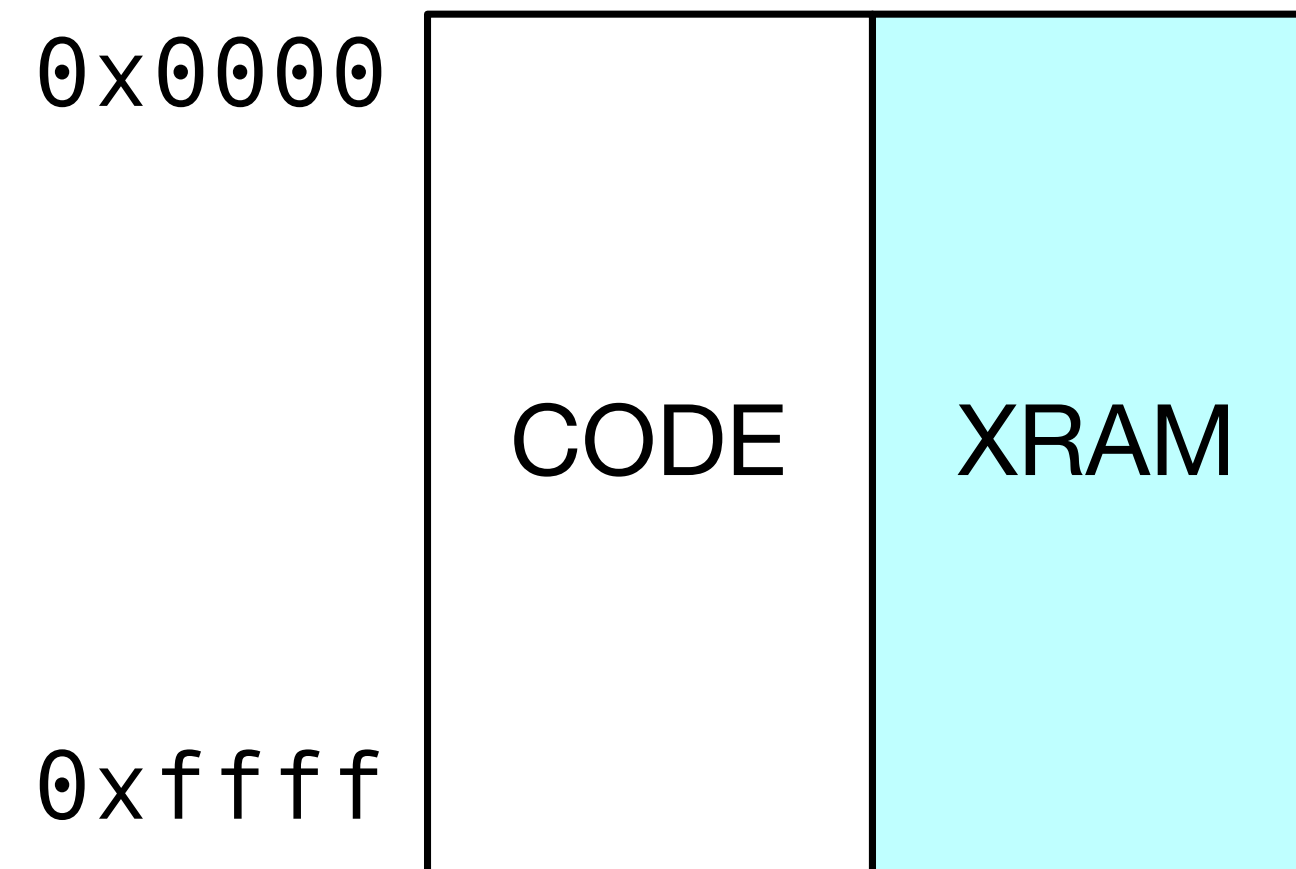
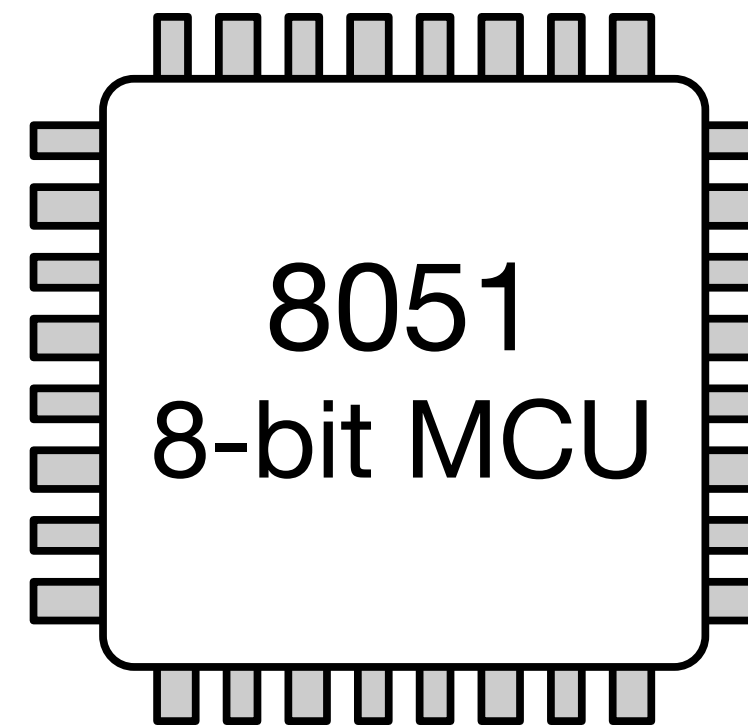
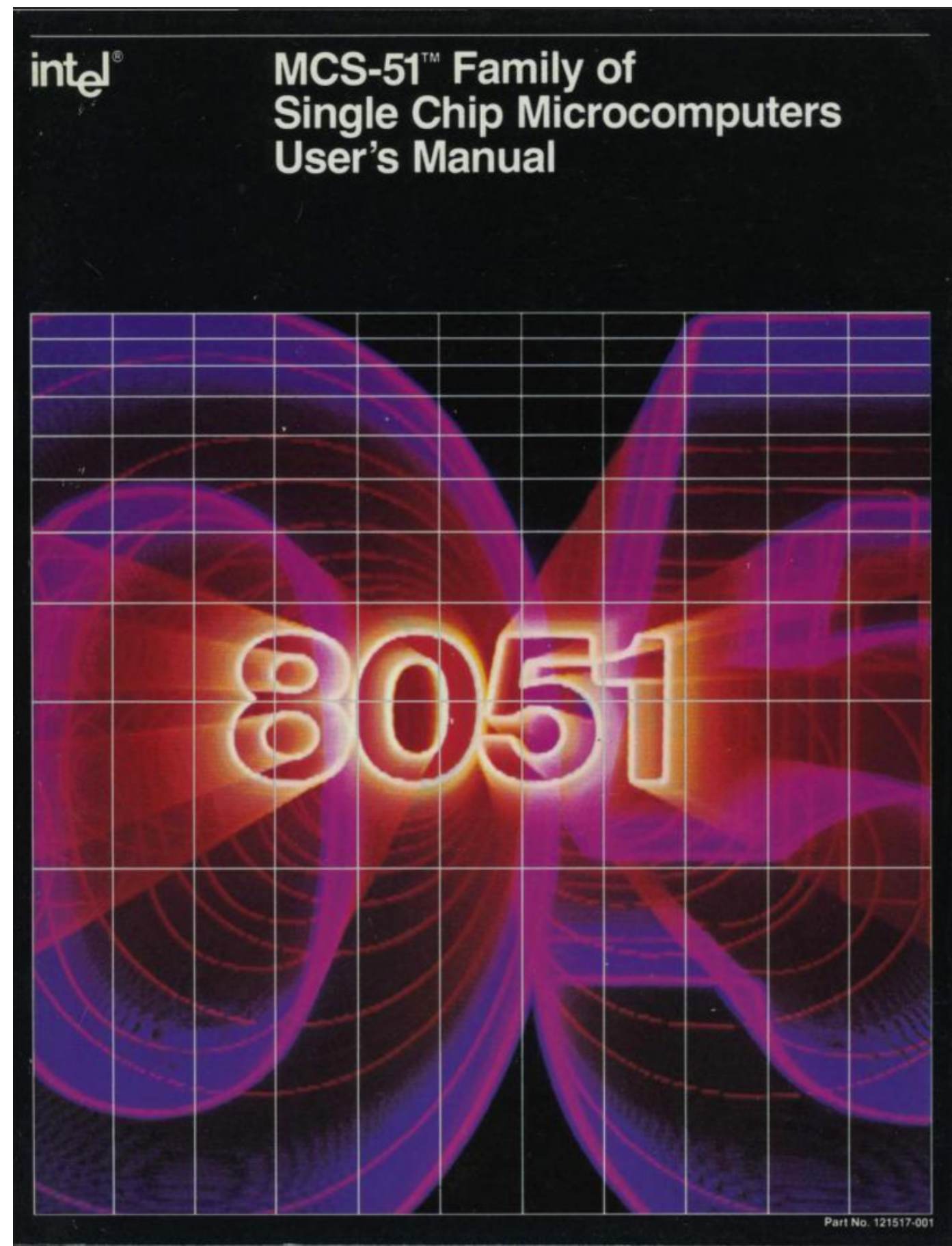
Symbolic Execution (Cont.)

```
unsigned short pincode = 0x????;
```

```
if(pincode % 10 == 0) {  
    if(pincode > 1000) {  
        if((pincode >> 2) & 0x43) {  
            printf("Correct!\n");  
        }  
    }  
}
```



pincode = [20930, 19190, ...]



- **Harvard Architecture**
- **44 instructions**
- **256 encodings**
- **128 bytes of RAM**
- **32 registers**
- **64KB of code**

Why 8051?



BADUSB - ON ACCESSORIES THAT TURN EVIL

USB has become so commonplace that we rarely worry about its security implications. USB sticks undergo the occasional virus scan, but we consider USB to be otherwise perfectly safe – until now.

Why 8051?

- The original BadUSB work hijacked a *Phison 2251-03* firmware



BADUSB - ON ACCESSORIES THAT TURN EVIL

USB has become so commonplace that we rarely worry about its security implications. USB sticks undergo the occasional virus scan, but we consider USB to be otherwise perfectly safe – until now.

Why 8051?

- The original BadUSB work hijacked a *Phison 2251-03* firmware
- *Many* Phison USB controllers use 8051



BADUSB - ON ACCESSORIES THAT TURN EVIL

USB has become so commonplace that we rarely worry about its security implications. USB sticks undergo the occasional virus scan, but we consider USB to be otherwise perfectly safe – until now.

Why 8051?

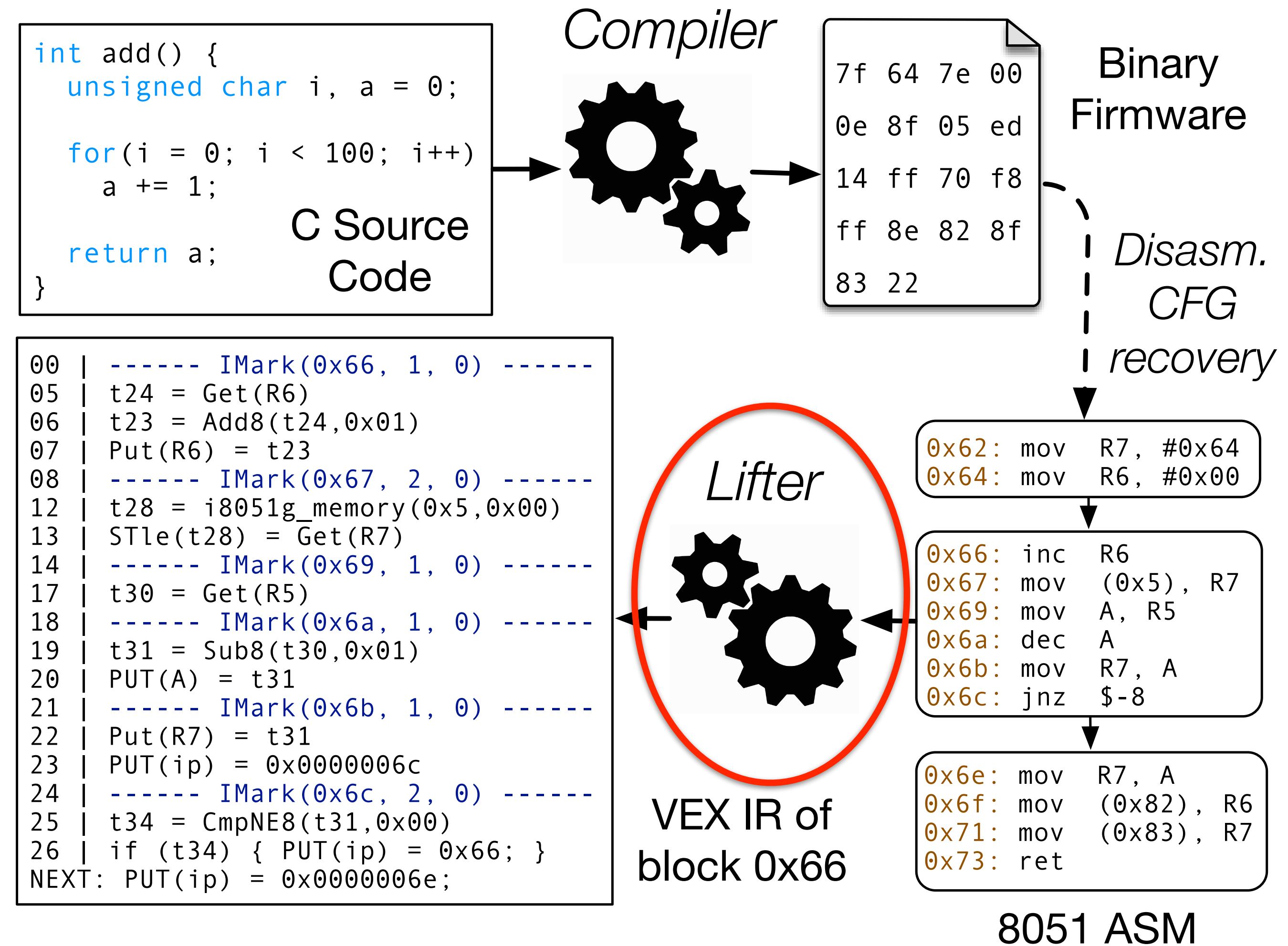
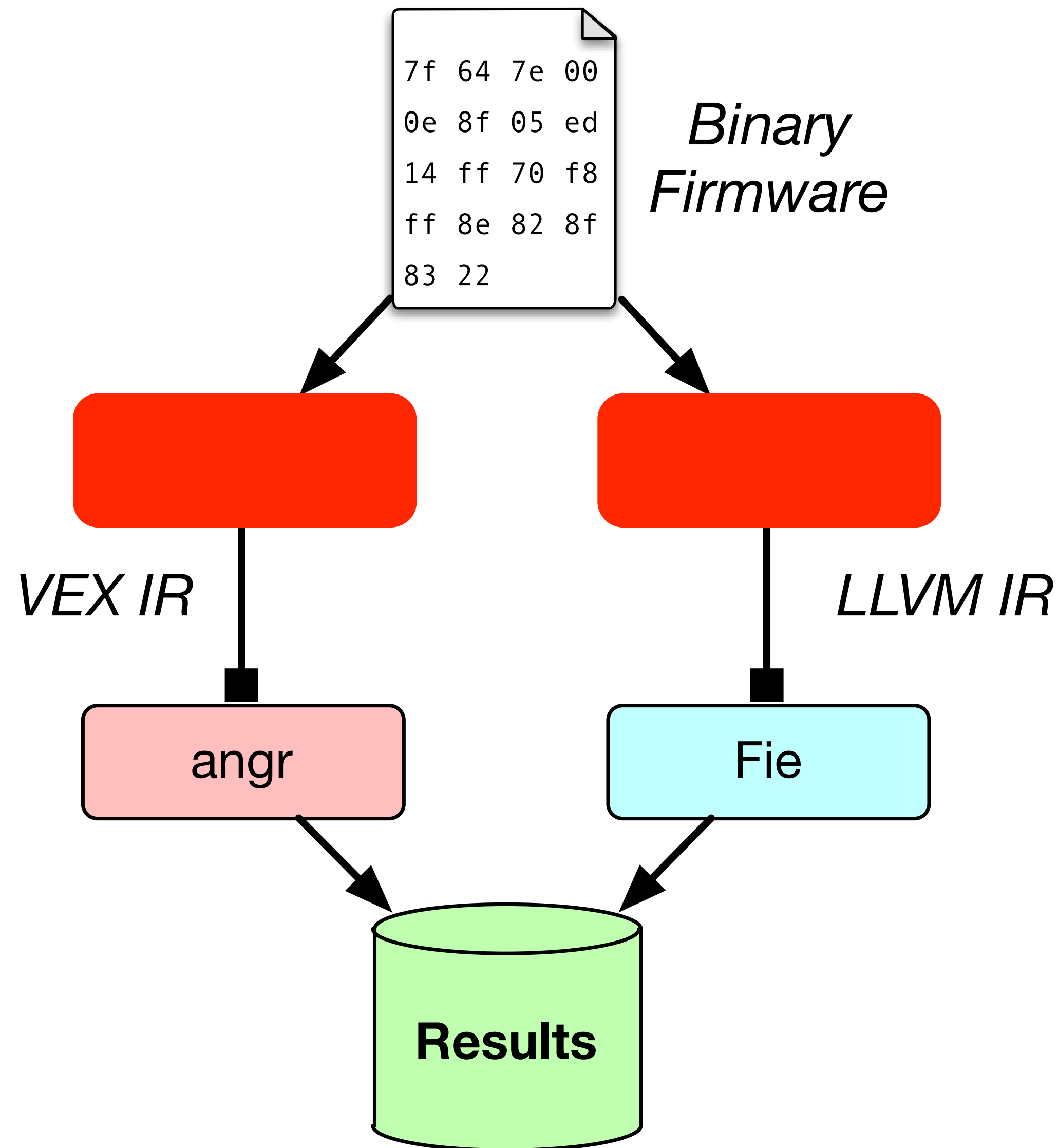
- The original BadUSB work hijacked a *Phison 2251-03* firmware
- *Many* Phison USB controllers use 8051
- **No symbolic execution support for angr or Fie**
 1. Create Intermediate Representation lifter (VEX IR & LLVM IR)
 2. Create architecture definition (Registers, memory map, I/O & interrupts)



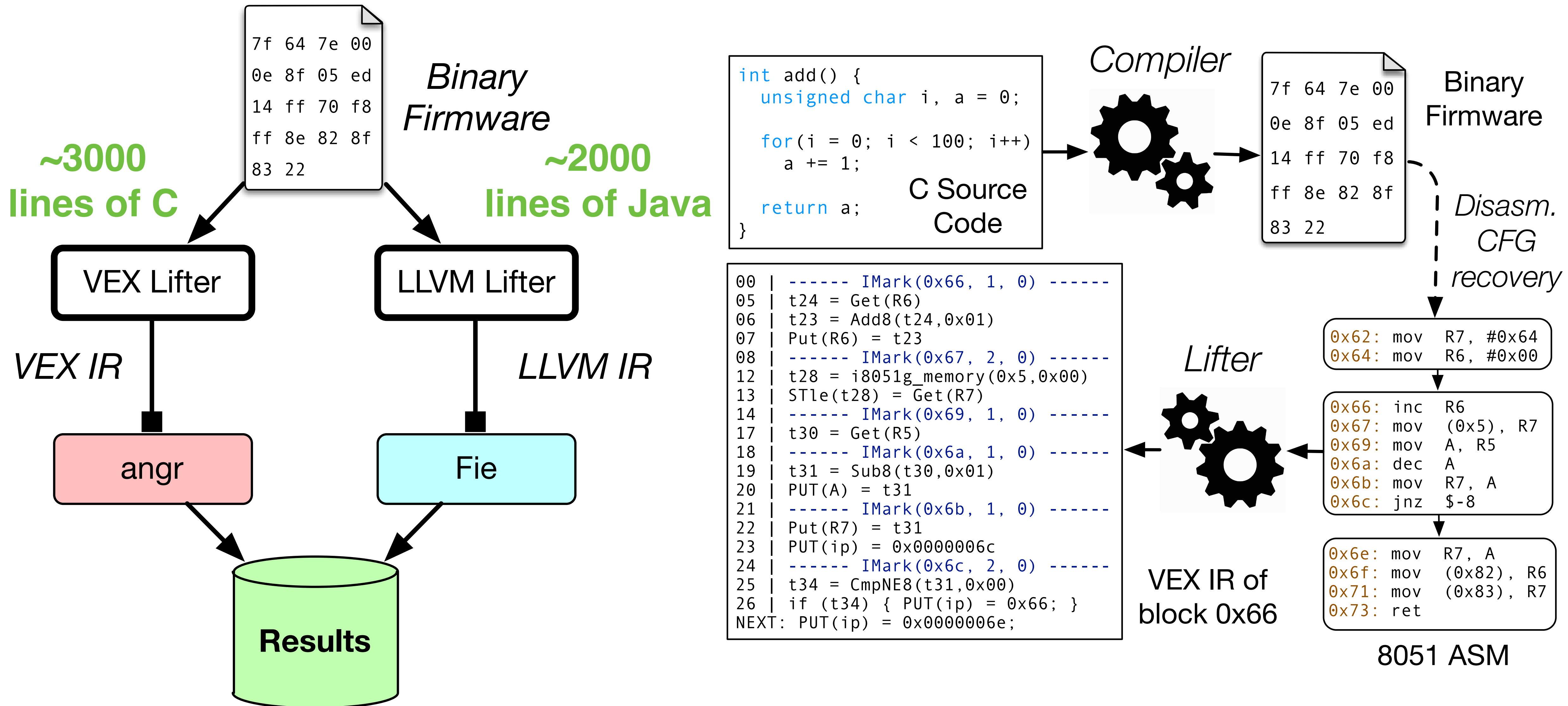
BADUSB - ON ACCESSORIES THAT TURN EVIL

USB has become so commonplace that we rarely worry about its security implications. USB sticks undergo the occasional virus scan, but we consider USB to be otherwise perfectly safe – until now.

Supporting 8051



Supporting 8051



Lifting Challenges

- Supporting condition bit-codes was cumbersome and required many IR statements to be emitted

Lifting Challenges

- Supporting condition bit-codes was cumbersome and required many IR statements to be emitted
- Overlapping RAM, XRAM, and CODE regions (address 0x0)

- Supporting condition bit-codes was cumbersome and required many IR statements to be emitted
- Overlapping RAM, XRAM, and CODE regions (address 0x0)

**Typical microcontroller patterns are
difficult to support with LLVM and VEX**

- Use USB domain knowledge to find key parts of firmware images
- Find code references (XREFs) to these addresses and feed them as *targets* for the symbolic execution stage

Pattern Name	Byte Pattern	Data Address	Cross-Reference
DEVICE_DESC	12 01 00 ?? 00	0x302b	0xb89
CONFIG_DESC	09 02 ?? ?? ?? 01 00	0x303d	0xbd5
HID_REPORT	05 01 09 06 A1	0x3084	0xbf1

Table: Found patterns in the Phison firmware.

- Use USB domain knowledge to find key parts of firmware images
- Find code references (XREFs) to these addresses and feed them as *targets* for the symbolic execution stage

[Length] [Type] [...]

Pattern Name	Byte Pattern	Data Address	Cross-Reference
DEVICE_DESC	12 01 00 ?? 00	0x302b	0xb89
CONFIG_DESC	09 02 ?? ?? ?? 01 00	0x303d	0xbd5
HID_REPORT	05 01 09 06 A1	0x3084	0xbf1

Table: Found patterns in the Phison firmware.

Query Engine & Semantic Analysis

- Employ static and symbolic analysis to answer questions about the firmware

Query Engine & Semantic Analysis

- Employ static and symbolic analysis to answer questions about the firmware
- Write in Python (for angr) or C++ (for Fie)

- Employ static and symbolic analysis to answer questions about the firmware
- Write in Python (for angr) or C++ (for Fie)
- **Example query:**
 1. Recover CFG, find USB signatures statically
 2. Symbolically execute towards *targets*
 3. For each found target, print the path condition

```
$ ./firmusb -i firmware1.bin -q query-type -o fw1.log
```


Query I: The Claimed Identity

Query I: The Claimed Identity

- I. Determine the USB type through a combination of static and symbolic analysis

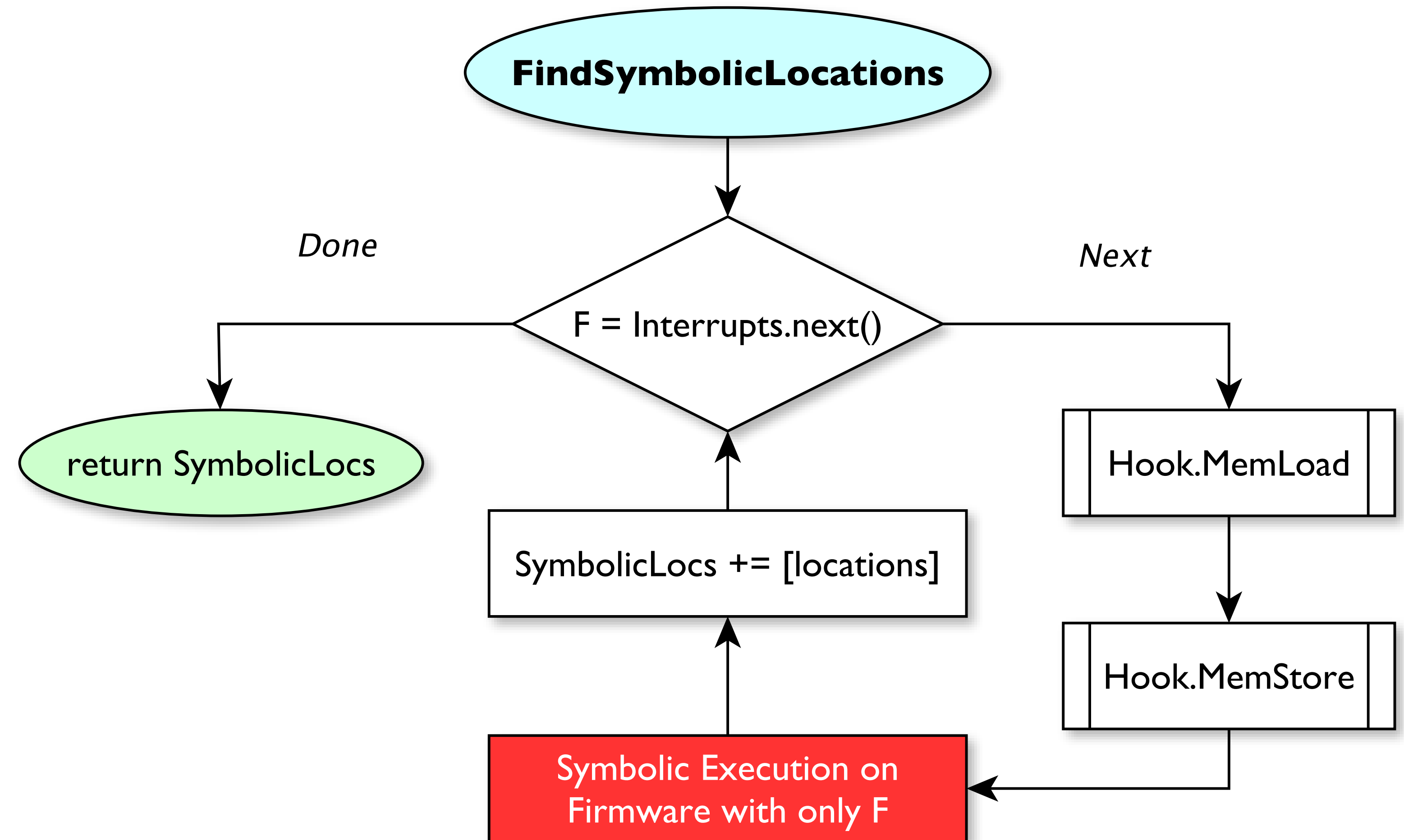
Query I: The Claimed Identity

1. Determine the USB type through a combination of static and symbolic analysis
2. Find USB descriptor signatures and determine referencing code addresses

Query I: The Claimed Identity

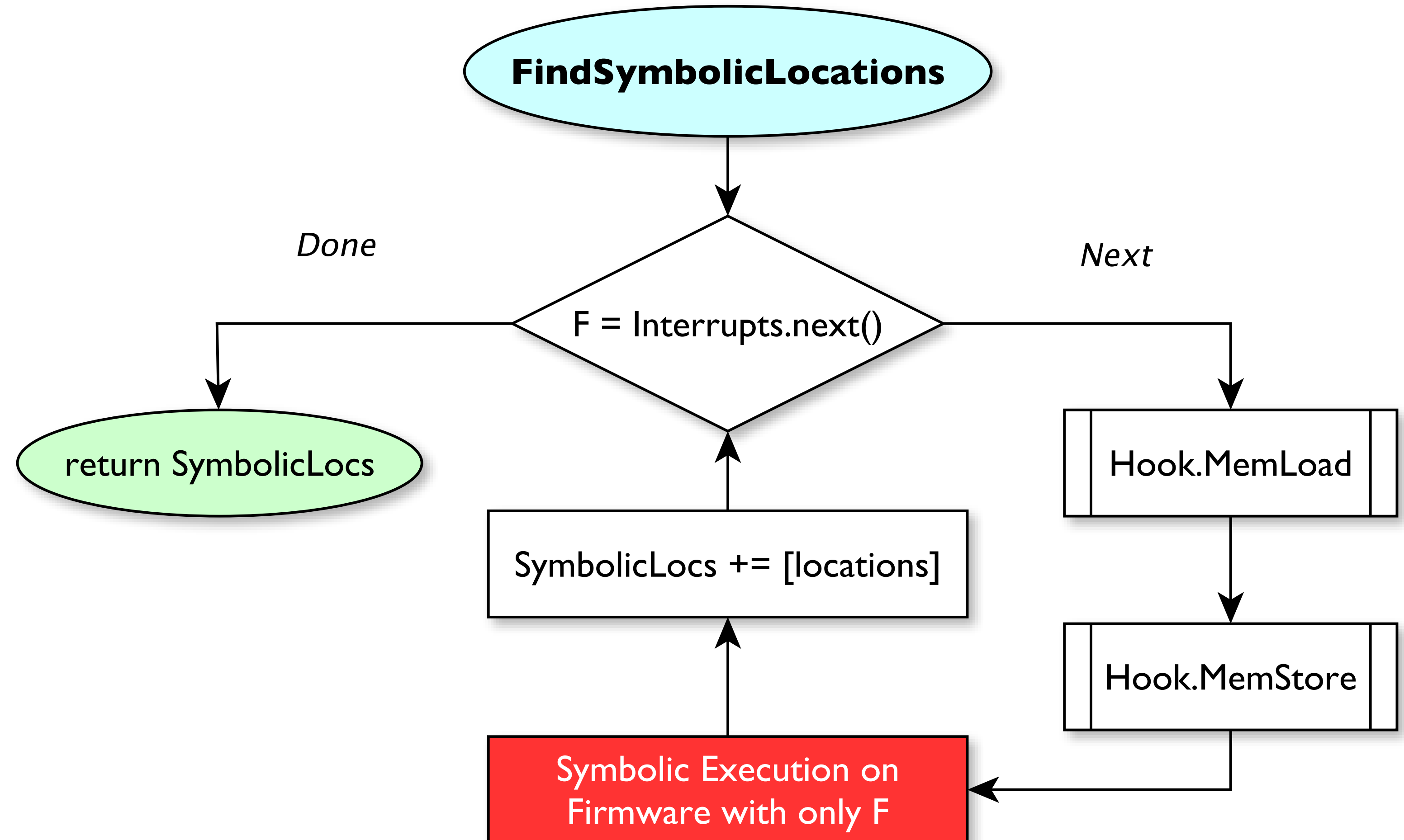
1. Determine the USB type through a combination of static and symbolic analysis
2. Find USB descriptor signatures and determine referencing code addresses
3. Symbolically execute to these 'targets' and determine path conditions

Symbolic Set Algorithm



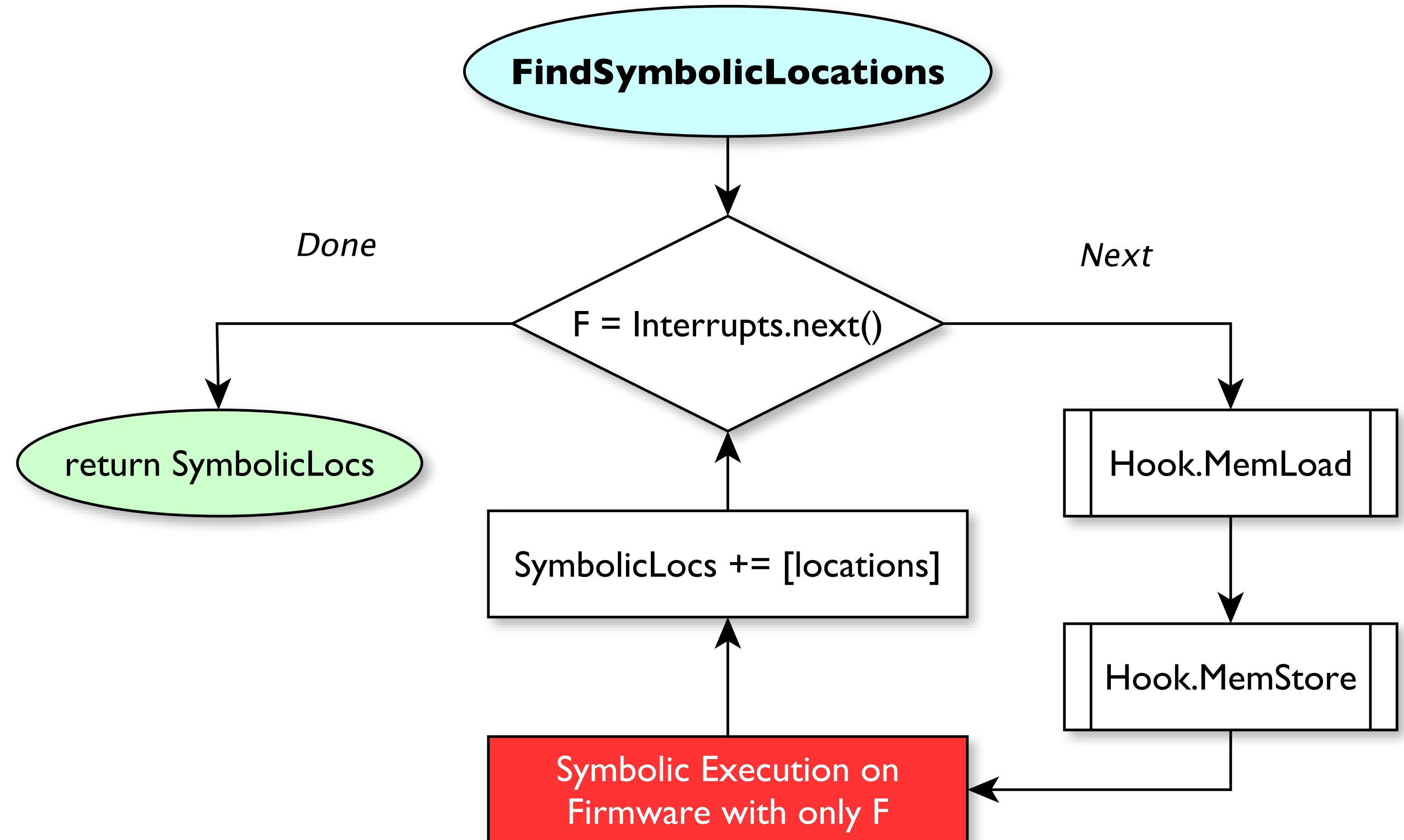
Symbolic Set Algorithm

- Determine a more minimal set of symbolic variables



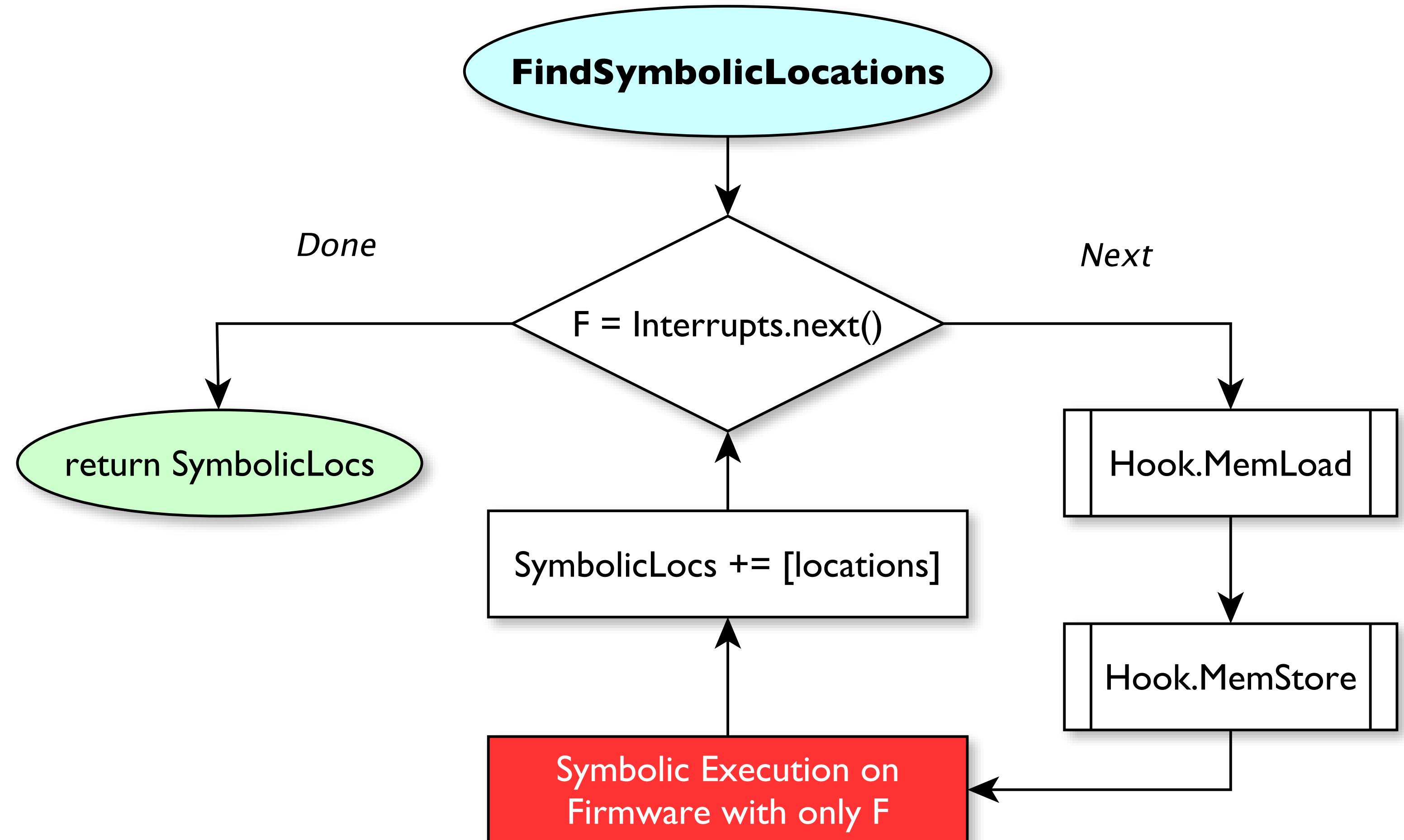
Symbolic Set Algorithm

- Determine a more minimal set of symbolic variables
- Relies on knowledge of 8051 interrupts



Symbolic Set Algorithm

- Determine a more minimal set of symbolic variables
- Relies on knowledge of 8051 interrupts
- Greatly speeds up symbolic execution vs. fully-symbolic (less state explosion)



USB Domain Constraining

- Speeds up symbolic execution by lowering state explosion
- Focus **only** on the code which interacts with USB
- **Example:** apply constraints to the USB I/O SETUP to *assume* certain values

```
Symbol(SETUP[1]) == 6      // bRequest  – Descriptor
Symbol(SETUP[2]) == 34     // wValueH    – HID Report
Symbol(SETUP[3]) == 0      // wIndexL    – Keyboard Index
```

USB Domain Constraining

- Speeds up symbolic execution by lowering state explosion
- Focus **only** on the code which interacts with USB
- **Example:** apply constraints to the USB I/O SETUP to *assume* certain values

```
Symbol(SETUP[1]) == 6      // bRequest – Descriptor  
Symbol(SETUP[2]) == 34     // wValueH   – HID Report  
Symbol(SETUP[3]) == 0      // wIndexL   – Keyboard Index
```


USB Domain Constraining

- Speeds up symbolic execution by lowering state explosion
- Focus **only** on the code which interacts with USB
- **Example:** apply constraints to the USB I/O SETUP to *assume* certain values

Symbol1(**SETUP**[1]) == 6

Symbol1(**SETUP**[2]) == 34

Symbol1(**SETUP**[3]) == 0

// **bRequest** – Descriptor

// **wValueH** – HID Report

// **wIndexL** – Keyboard Index

Evaluation Targets

- Target 1: BadUSB Phison
 - Original firmware extracted from Phison device and modified to inject keystrokes
 - Displays both mass storage and keyboard
 - Size: 13KB



Evaluation Targets

- Target 1: BadUSB Phison

- Original firmware extracted from Phison device and modified to inject keystrokes
- Displays both mass storage and keyboard
- Size: 13KB



- Target 2: EzHID Firmware

- Generic HID firmware platform
- When triggered, injects keystrokes from hard coded buffer
- Size: 3.4 KB



Evaluation Results — Query I



Time to Target (seconds)	Fie (Config.)	Fie (HID)	angr (Config.)	angr (HID)
Phison (Full)	384.40	43.49 s	—	—
Phison (Q1+Domain)	7.68	5.64 s	70.28	70.09
EzHID (Full)	—	—	10.76	24.04
EzHID (Q1+Domain)	9.45	9.87	5.18	11.13

- **Finding USB Specific Code**
 - Discover targets in the firmware and symbolically execute towards them
 - Speedup achieved when using symbolic set algorithms domain knowledge

Time to Target (seconds)	Fie (Config.)	Fie (HID)	angr (Config.)	angr (HID)
Phison (Full)	384.40	43.49 s	—	—
Phison (Q1+Domain)	7.68	5.64 s	70.28	70.09
EzHID (Full)	—	—	10.76	24.04
EzHID (Q1+Domain)	9.45	9.87	5.18	11.13

- **Finding USB Specific Code**
 - Discover targets in the firmware and symbolically execute towards them
 - Speedup achieved when using symbolic set algorithms domain knowledge

Time to Target (seconds)	Fie (Config.)	Fie (HID)	angr (Config.)	angr (HID)
Phison (Full)	384.40	43.49 s	—	—
Phison (Q1+Domain)	7.68	5.64 s	70.28	70.09
EzHID (Full)	—	—	10.76	24.04
EzHID (Q1+Domain)	9.45	9.87	5.18	11.13

50x

- **Finding USB Specific Code**
 - Discover targets in the firmware and symbolically execute towards them
 - Speedup achieved when using symbolic set algorithms domain knowledge

Time to Target (seconds)	Fie (Config.)	Fie (HID)	angr (Config.)	angr (HID)
Phison (Full)	384.40	43.49 s	—	—
Phison (Q1+Domain)	7.68	5.64 s	70.28	70.09
EzHID (Full)	—	—	10.76	24.04
EzHID (Q1+Domain)	9.45	9.87	5.18	11.13

7x

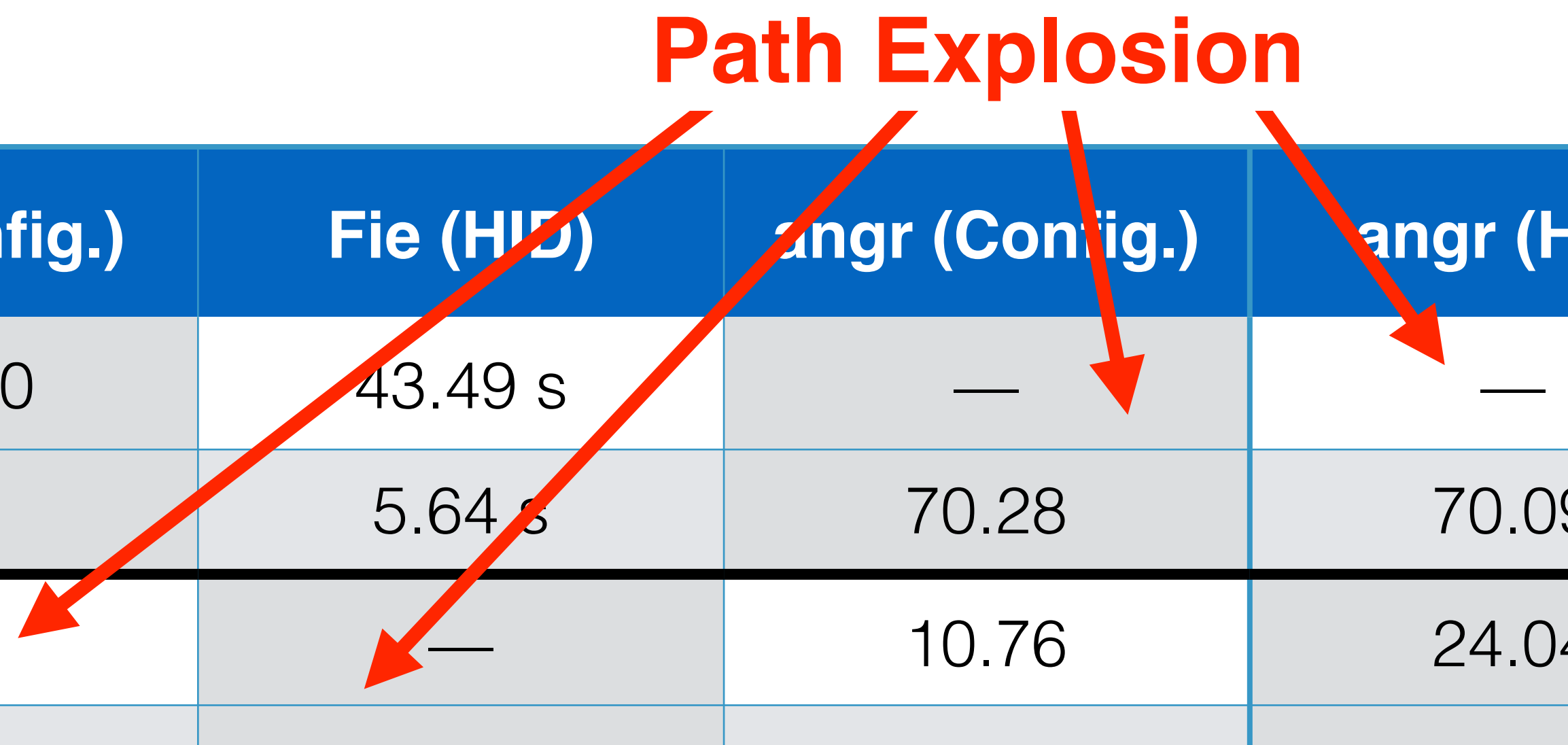
- **Finding USB Specific Code**
 - Discover targets in the firmware and symbolically execute towards them
 - Speedup achieved when using symbolic set algorithms domain knowledge

Time to Target (seconds)	Fie (Config.)	Fie (HID)	angr (Config.)	angr (HID)
Phison (Full)	384.40	43.49 s	—	—
Phison (Q1+Domain)	7.68	5.64 s	70.28	70.09
EzHID (Full)	—	—	10.76	24.04
EzHID (Q1+Domain)	9.45	9.87	5.18	11.13

2x

- **Finding USB Specific Code**

- Discover targets in the firmware and symbolically execute towards them
- Speedup achieved when using symbolic set algorithms domain knowledge

A red text label 'Path Explosion' is positioned above the table. Four red arrows originate from this label and point to specific cells in the table: one to the 'Fie (HID)' cell for 'Phison (Full)', one to the 'angr (Config.)' cell for 'Phison (Full)', one to the 'Fie (HID)' cell for 'EzHID (Full)', and one to the 'angr (HID)' cell for 'EzHID (Full)'.

Time to Target (seconds)	Fie (Config.)	Fie (HID)	angr (Config.)	angr (HID)
Phison (Full)	384.40	43.49 s	—	—
Phison (Q1+Domain)	7.68	5.64 s	70.28	70.09
EzHID (Full)	—	—	10.76	24.04
EzHID (Q1+Domain)	9.45	9.87	5.18	11.13

Query 2: Consistent Behavior

Query 2: Consistent Behavior

- How are USB endpoints used in the firmware image?

Query 2: Consistent Behavior

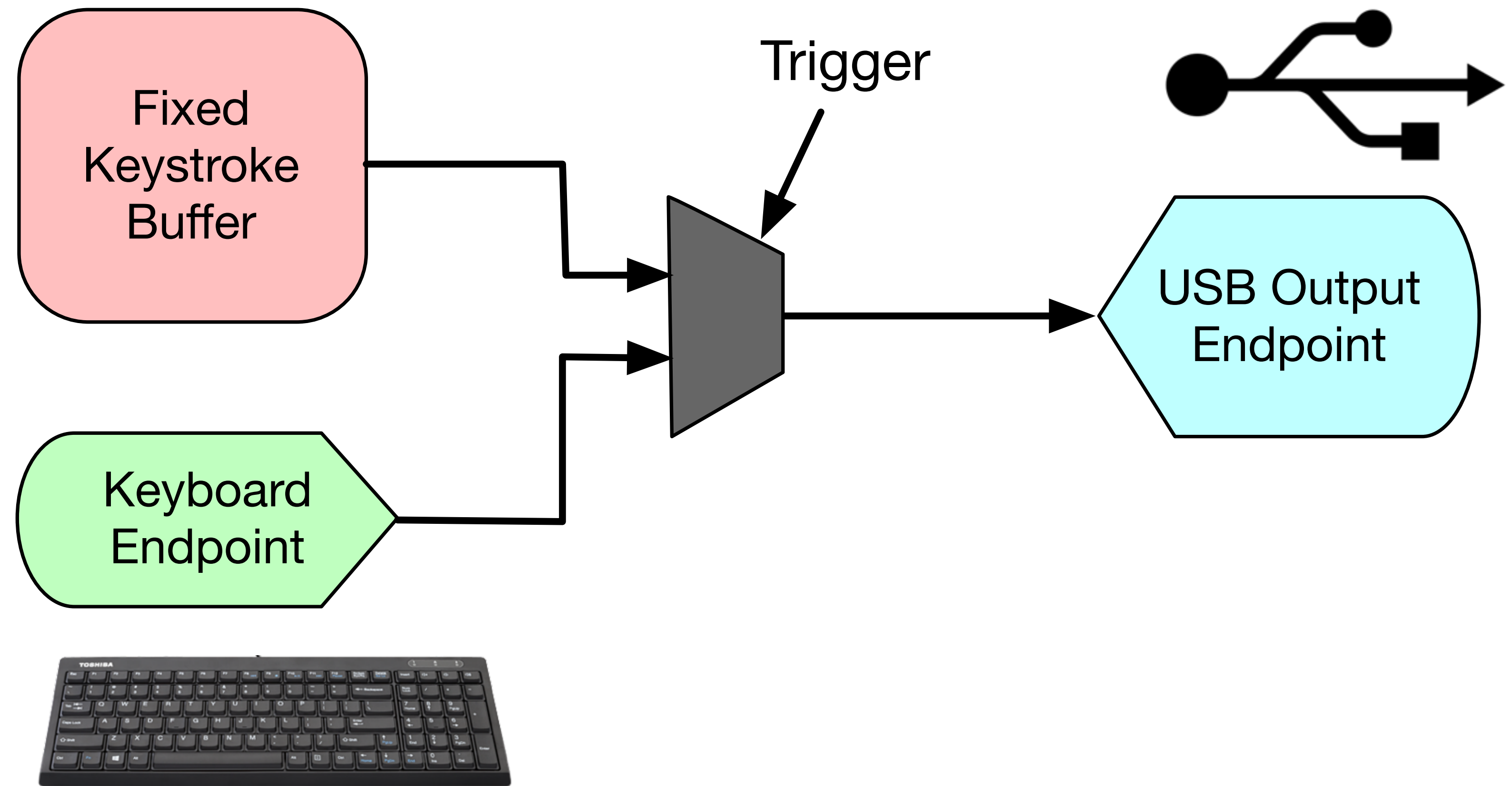
- How are USB endpoints used in the firmware image?
- Does this usage change throughout its execution?

Query 2: Consistent Behavior

- How are USB endpoints used in the firmware image?
- Does this usage change throughout its execution?
- **Example:**
 - Keyboard device reads keyboard data from an I/O port then passes it to the USB output
 - Suddenly it decides to inject hardcoded keystrokes

Inconsistent Flows

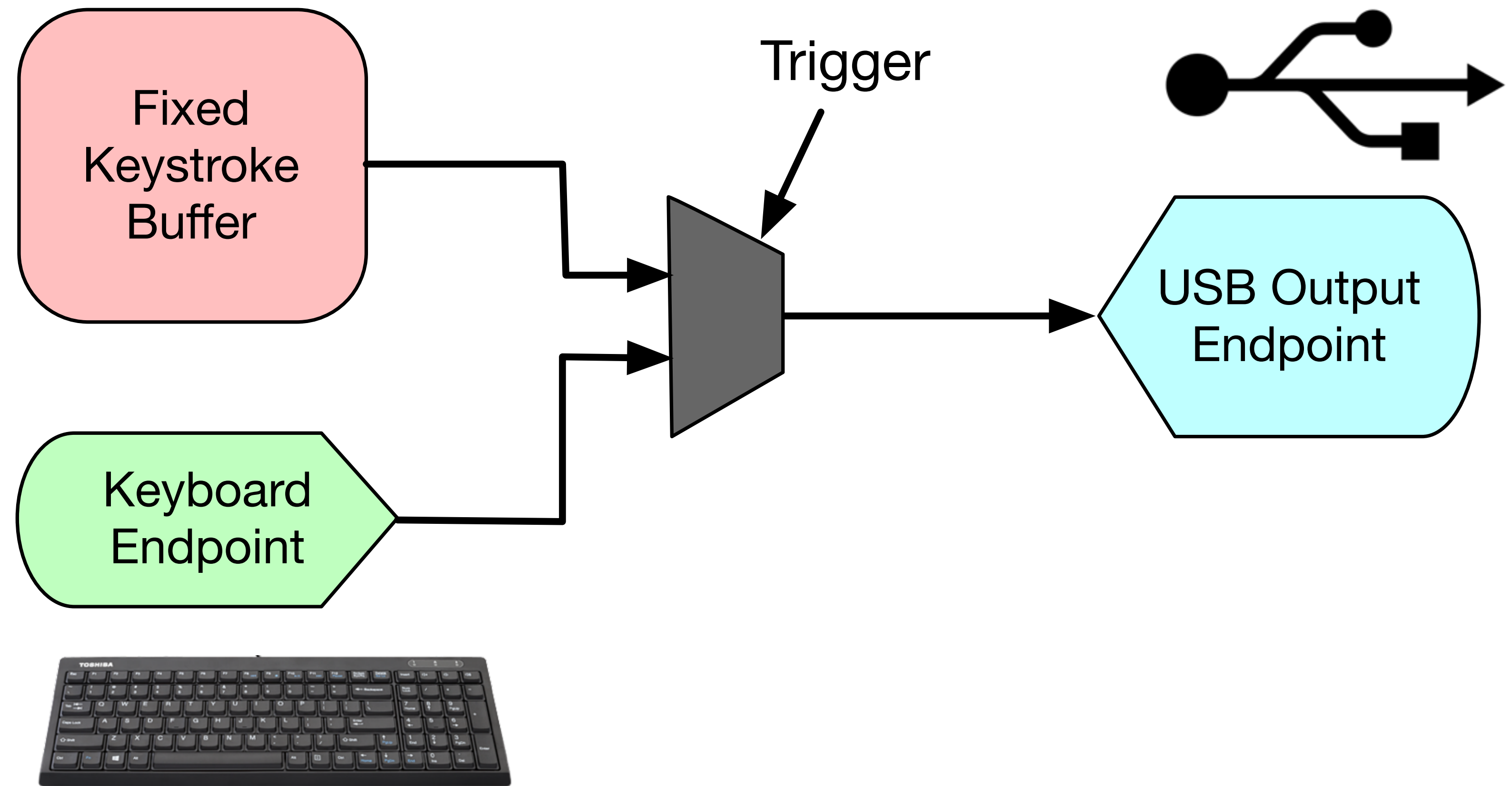
```
char inject[] = { 'c', 'm', 'd', '.', 'e', 'x', 'e', ...};
```



Inconsistent Flows

- Certain USB Endpoints should NOT receive constant data

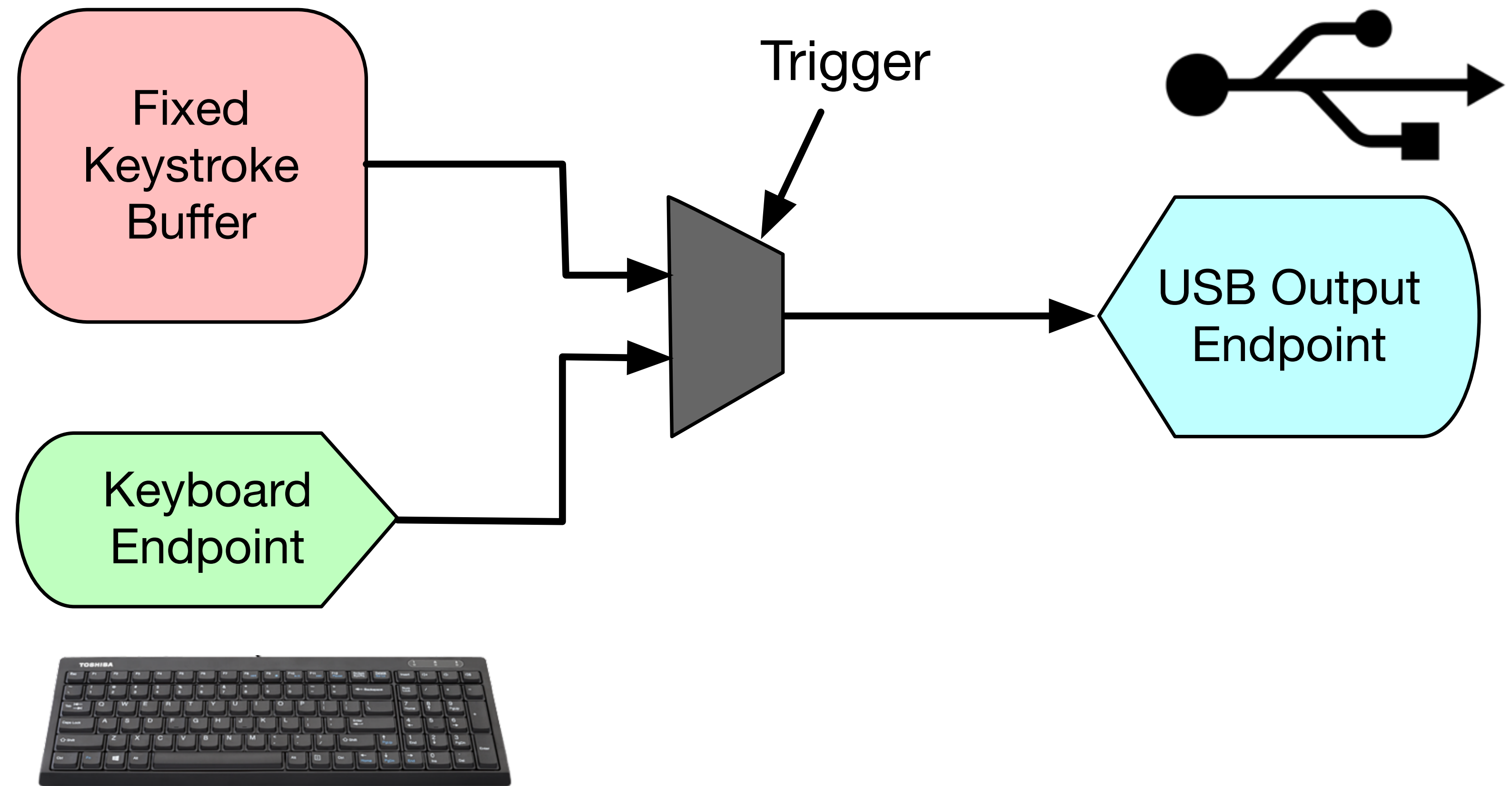
```
char inject[] = { 'c', 'm', 'd', '.', 'e', 'x', 'e', ...};
```



Inconsistent Flows

- Certain USB Endpoints should NOT receive constant data
- Record all memory stores during symbolic execution

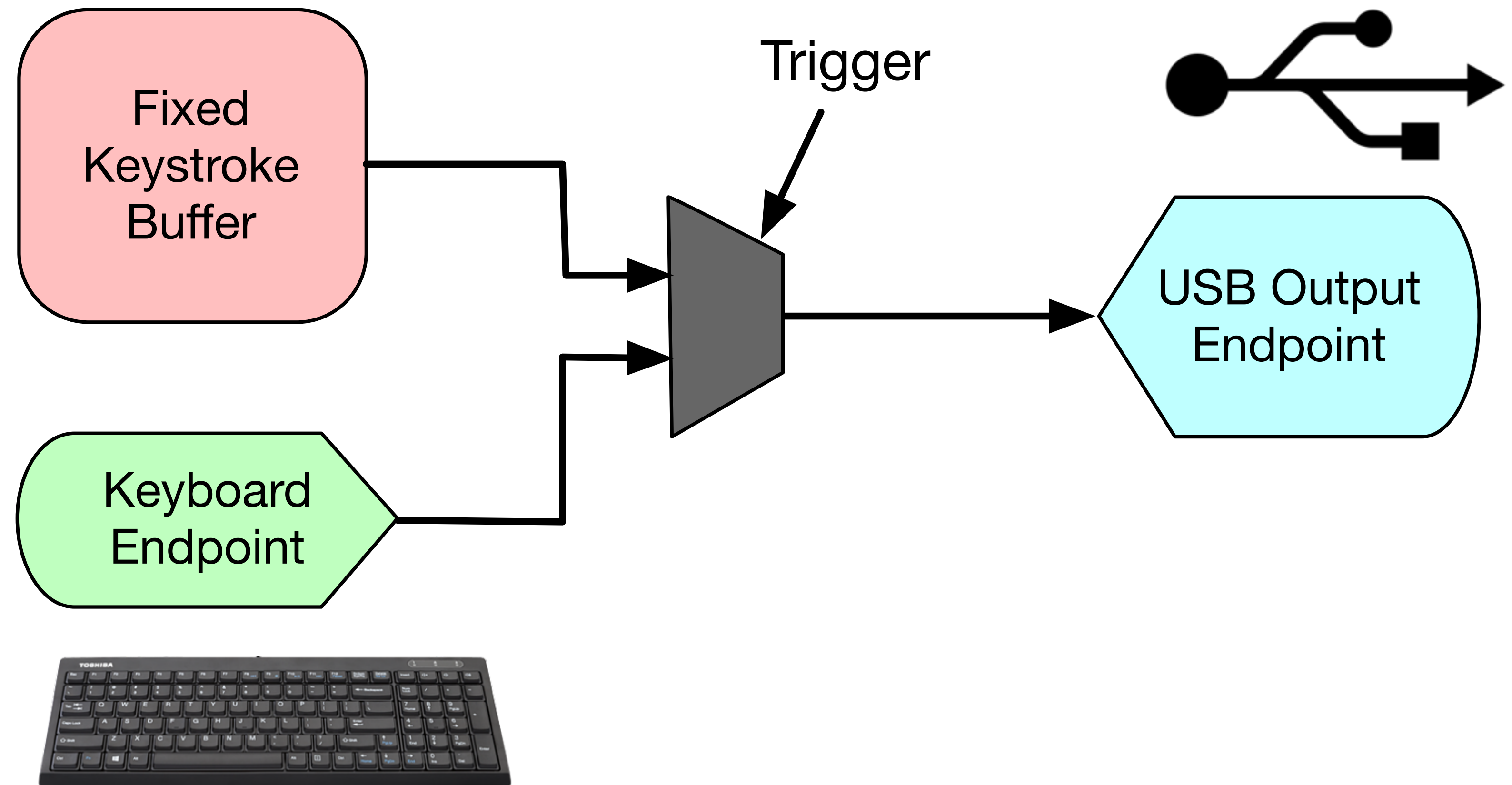
```
char inject[] = { 'c', 'm', 'd', '.', 'e', 'x', 'e', ...};
```



Inconsistent Flows

- Certain USB Endpoints should NOT receive constant data
- Record all memory stores during symbolic execution
- Track symbolic vs. concrete and writer instruction addresses

```
char inject[] = { 'c', 'm', 'd', '.', 'e', 'x', 'e', ... };
```



Evaluation Results — Q2

- Discover all inconsistent memory addresses
- Track when and where writes take place
- Execute for 30 minutes to accumulate I/O port flows

Write Address	Writers	Symbolic Name	Concrete Values
0x7e80 - 0x7e87	0x991, 0xa7e	scancode[0-7]	0x0, 0xe2, 0x3b, 0x1b, 0x17, 0x08, 0x15, ...

Table: EzHID Query 2 Results

- **Neither were easy to bring 8-bit architecture support to**
 - Both required lifters & architecture definitions
 - angr had no interrupt support and less path heuristics
 - Environment support (I/O) difficult

- **Neither were easy to bring 8-bit architecture support to**
 - Both required lifters & architecture definitions
 - angr had no interrupt support and less path heuristics
 - Environment support (I/O) difficult
- **Neither IR was ideal, but VEX IR is the better choice for binary-only analysis**
 - VEX IR assumes *bottom-up* approach, no types, and no CFG
 - LLVM IR comes from a *top-down* perspective

- **Automatic device extraction is difficult and controller specific**
 - How do we scale FirmUSB to more firmware?
- **No trusted path to USB devices or any device attestation**
 - How can we trust automatically extracted firmware?
- **More work required to handle adversarial firmware (obfuscation)**
 - Adversarial firmwares may cause path explosion or prevent static analysis

- We develop an embedded **firmware analysis framework**
- **Analyze 8051 USB firmware** to determine intent
- Apply **domain-informed symbolic execution** to target specific code paths and improve performance
- **Side-by-side analysis of existing symbolic execution engines** and the ease of supporting a new architecture in each



Thanks!

Questions & Comments

grant.hernandez@ufl.edu

Fighting back against BadUSB

- **Prompting** — *GoodUSB* (ACSAC'15), Allow or Deny for USB devices

Fighting back against BadUSB

- **Prompting** — *GoodUSB* (ACSAC'15), Allow or Deny for USB devices
 - ✗ Requires users to make security sensitive decisions

Fighting back against BadUSB

- **Prompting** — *GoodUSB* (ACCSAC'15), Allow or Deny for USB devices
 - ✗ Requires users to make security sensitive decisions
- **Sandboxing** — *Cinch* (USENIX'16), Sandboxing the USB stack

Fighting back against BadUSB

- **Prompting** — *GoodUSB* (ACSSAC'15), Allow or Deny for USB devices
 - ✗ Requires users to make security sensitive decisions
- **Sandboxing** — *Cinch* (USENIX'16), Sandboxing the USB stack
 - ✗ Requires an active virtual machine

Fighting back against BadUSB

- **Prompting** — *GoodUSB* (ACSAC'15), Allow or Deny for USB devices
 - ✗ Requires users to make security sensitive decisions
- **Sandboxing** — *Cinch* (USENIX'16), Sandboxing the USB stack
 - ✗ Requires an active virtual machine
- **Firewalling** — *USBFILTER* (USENIX'16), iptables for USB

Fighting back against BadUSB

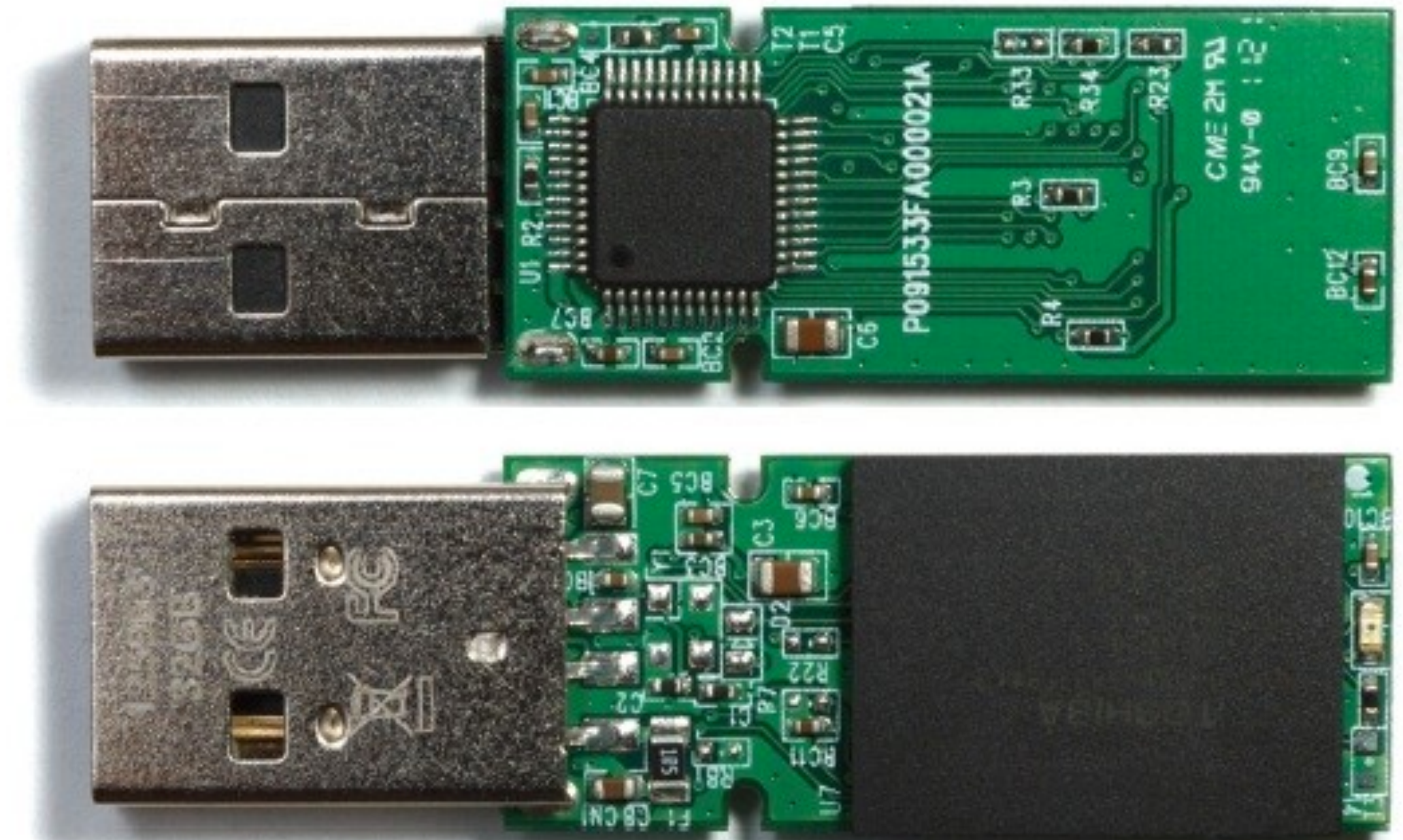
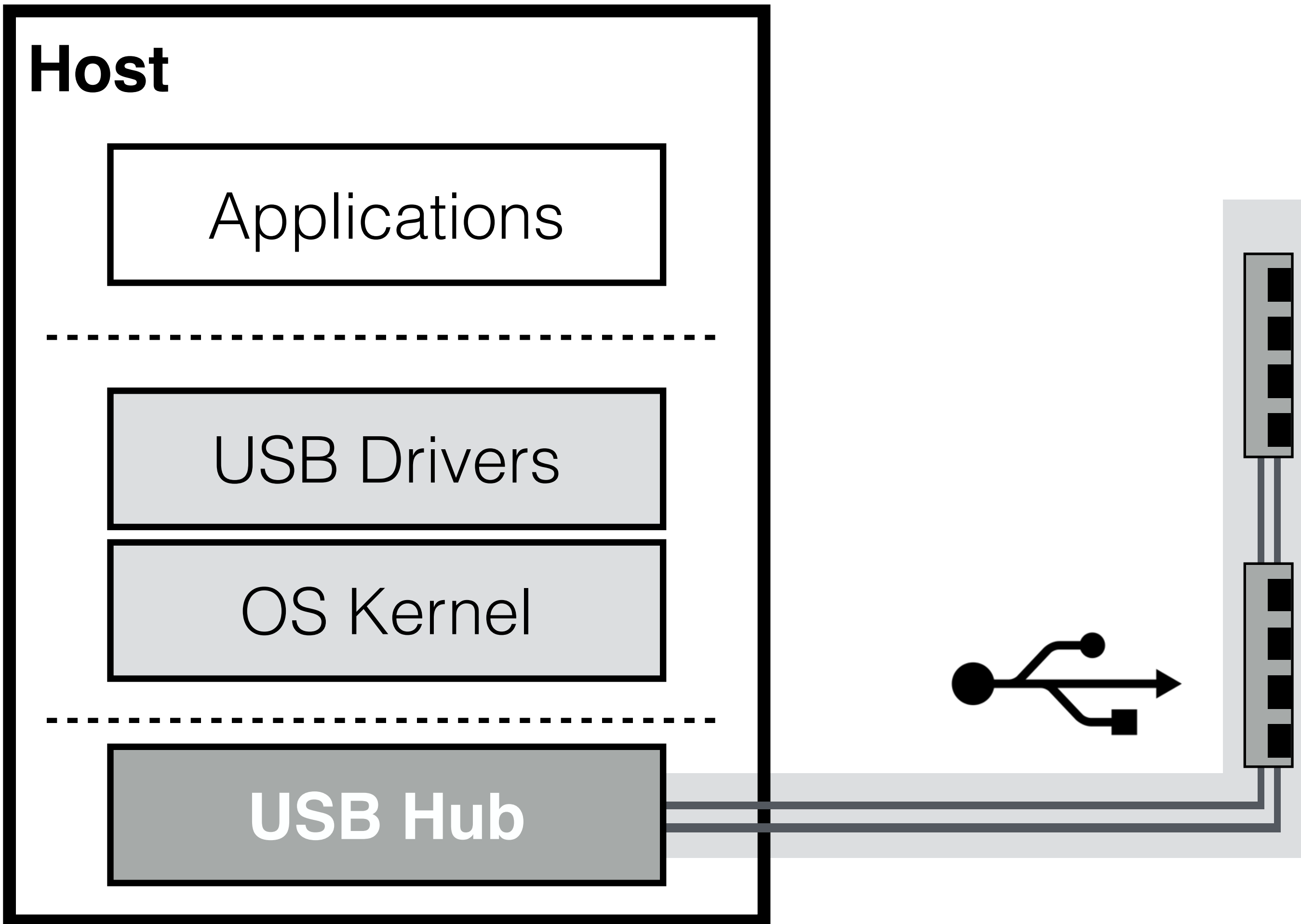
- **Prompting** — *GoodUSB* (ACSAC'15), Allow or Deny for USB devices
 - ✗ Requires users to make security sensitive decisions
- **Sandboxing** — *Cinch* (USENIX'16), Sandboxing the USB stack
 - ✗ Requires an active virtual machine
- **Firewalling** — *USBFILTER* (USENIX'16), iptables for USB
 - ✗ Requires complex policies and trusted hardware

Fighting back against BadUSB

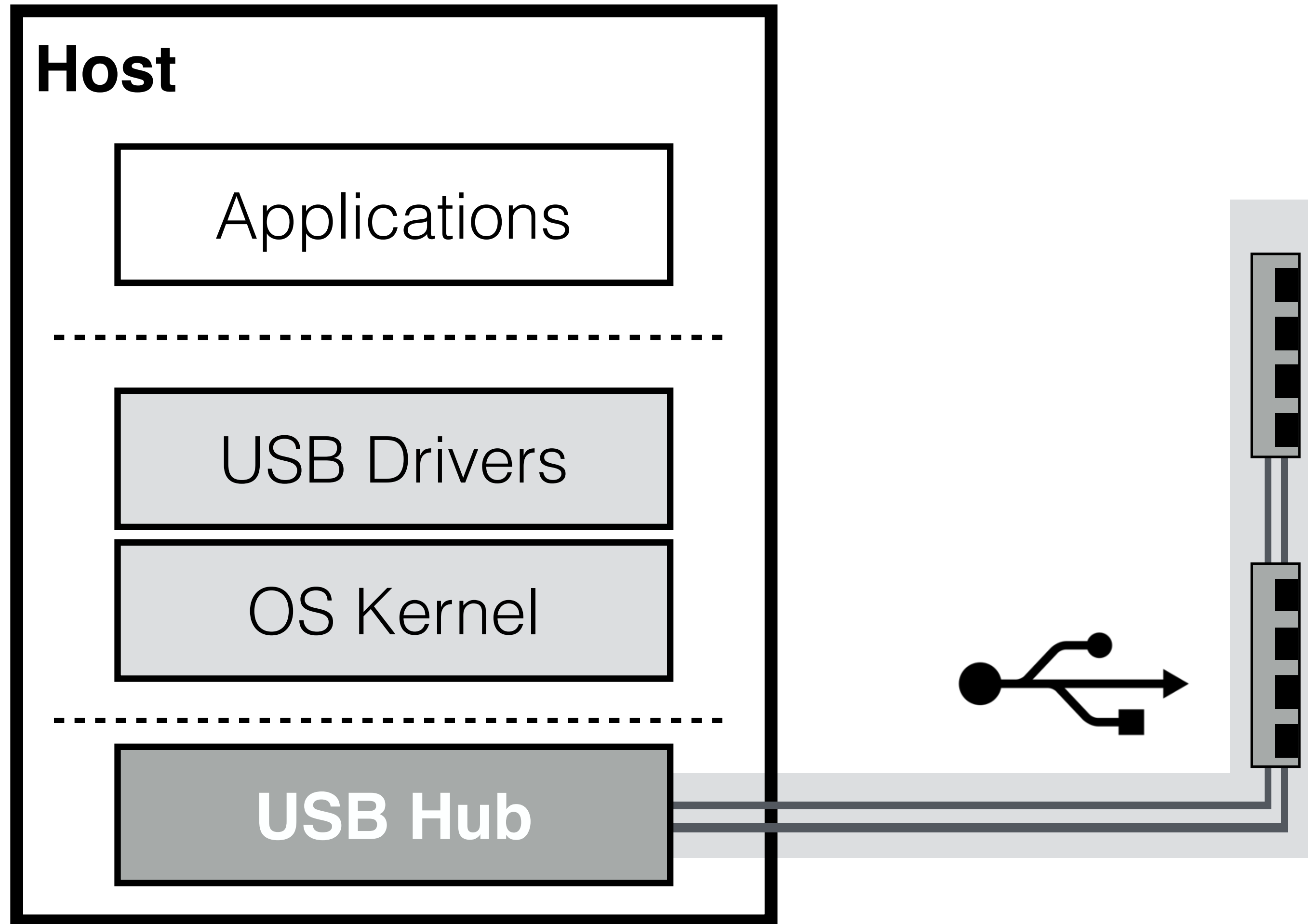
- **Prompting** — *GoodUSB* (ACCSAC'15), Allow or Deny for USB devices
 - ✗ Requires users to make security sensitive decisions
- **Sandboxing** — *Cinch* (USENIX'16), Sandboxing the USB stack
 - ✗ Requires an active virtual machine
- **Firewalling** — *USBFILTER* (USENIX'16), iptables for USB
 - ✗ Requires complex policies and trusted hardware

These solutions all rely on *runtime behavior*

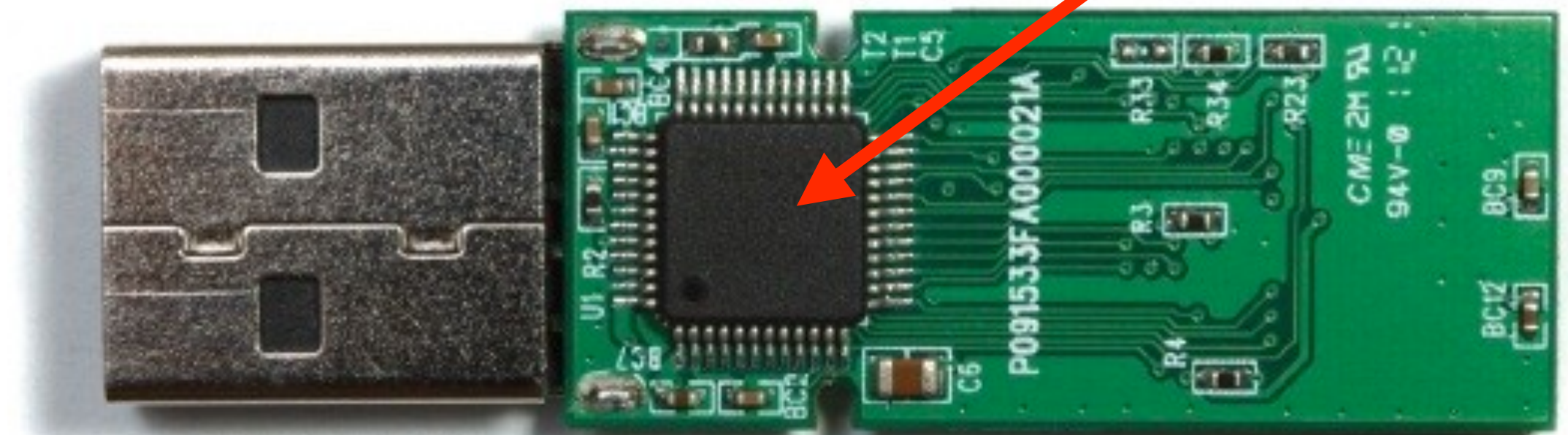
FirmUSB & Related Work



FirmUSB & Related Work

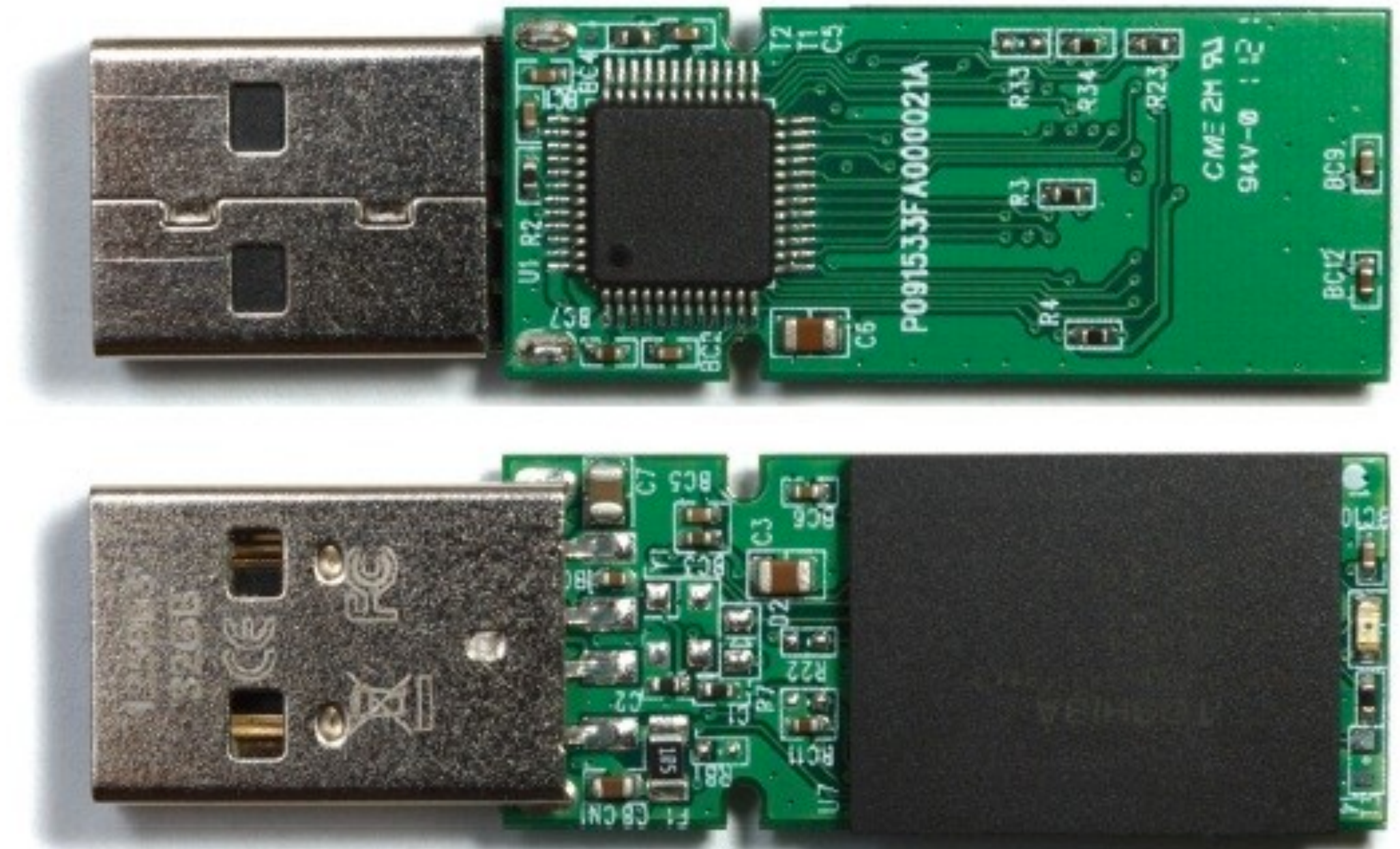
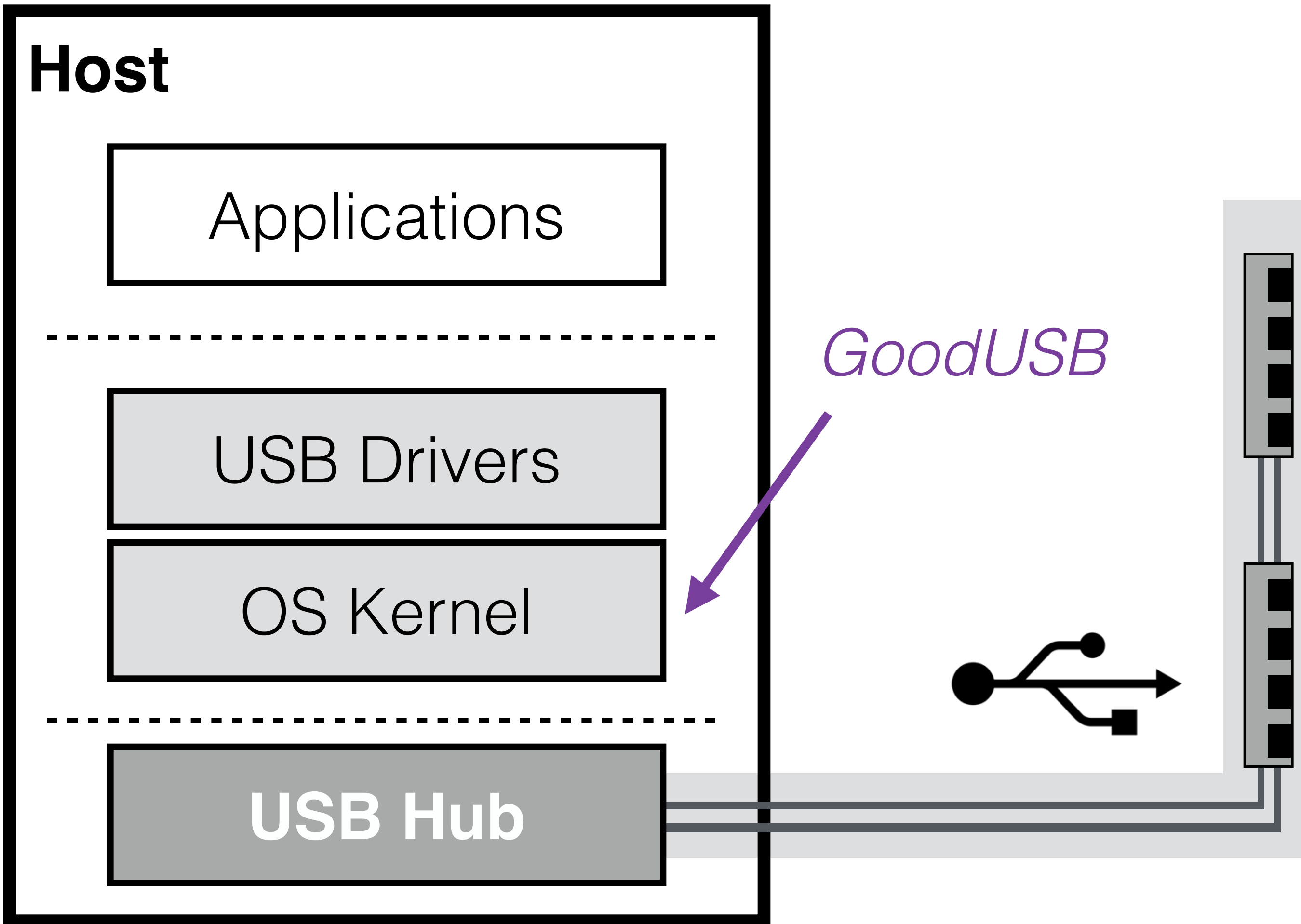


USB Controller

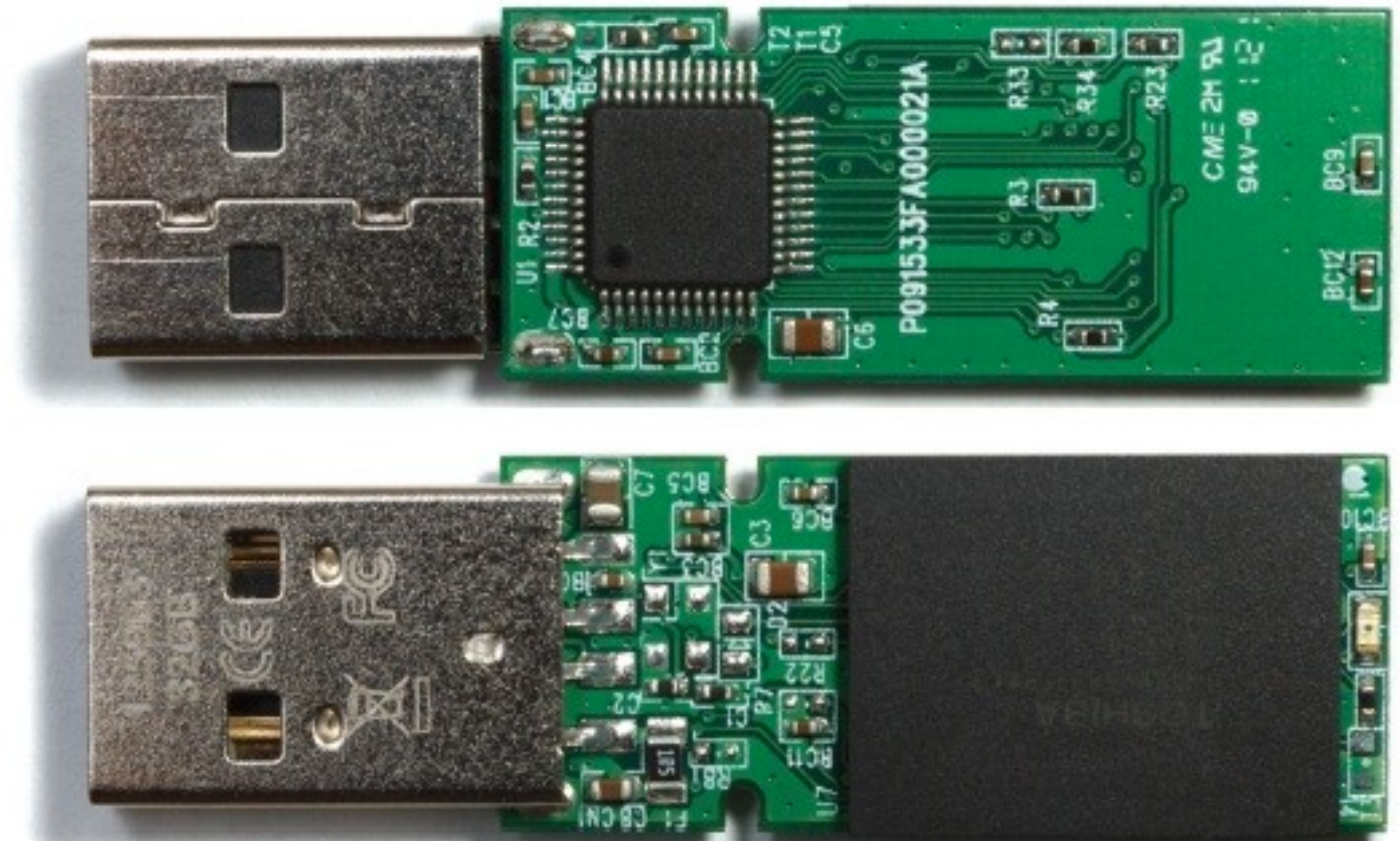
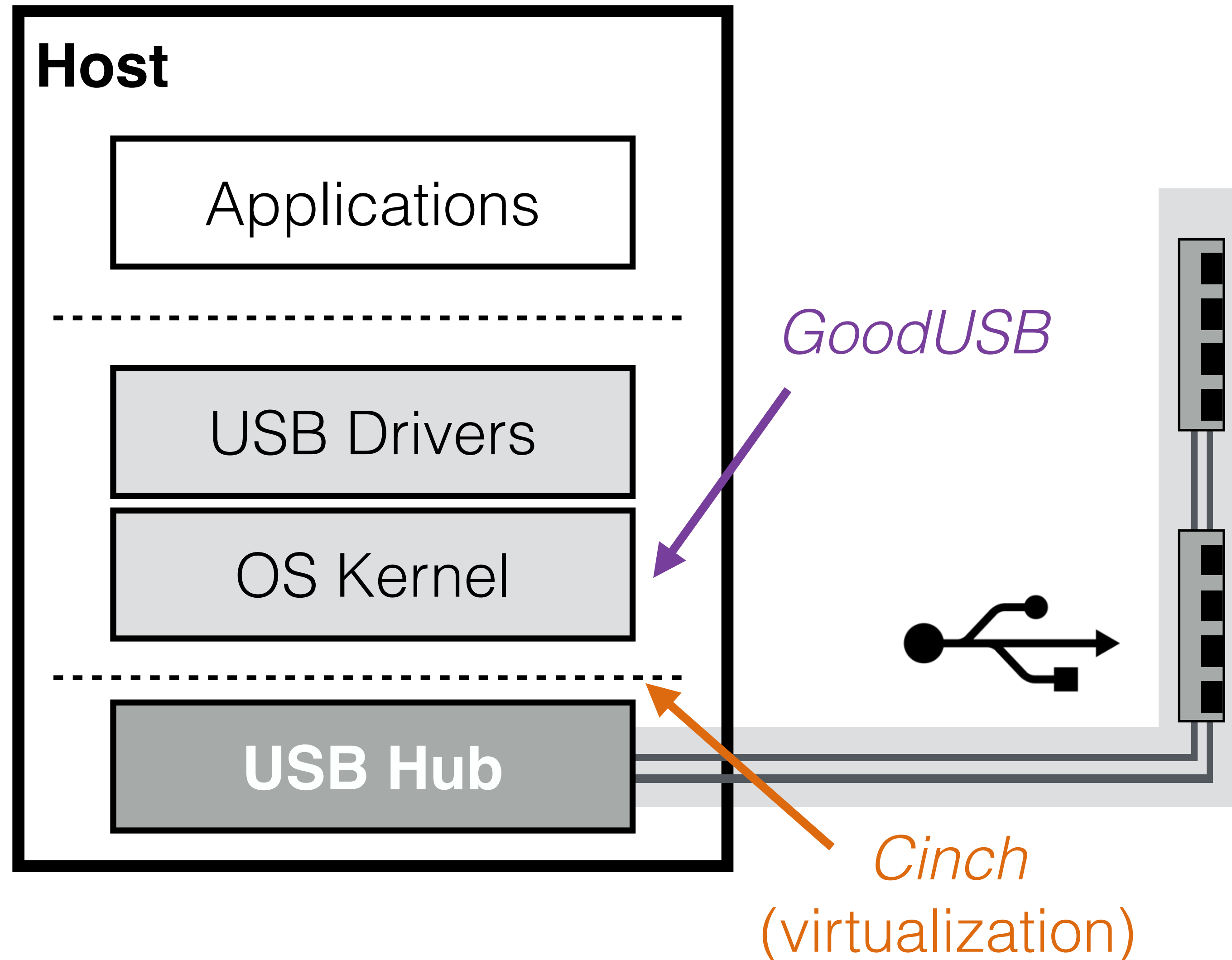


Flash Chip

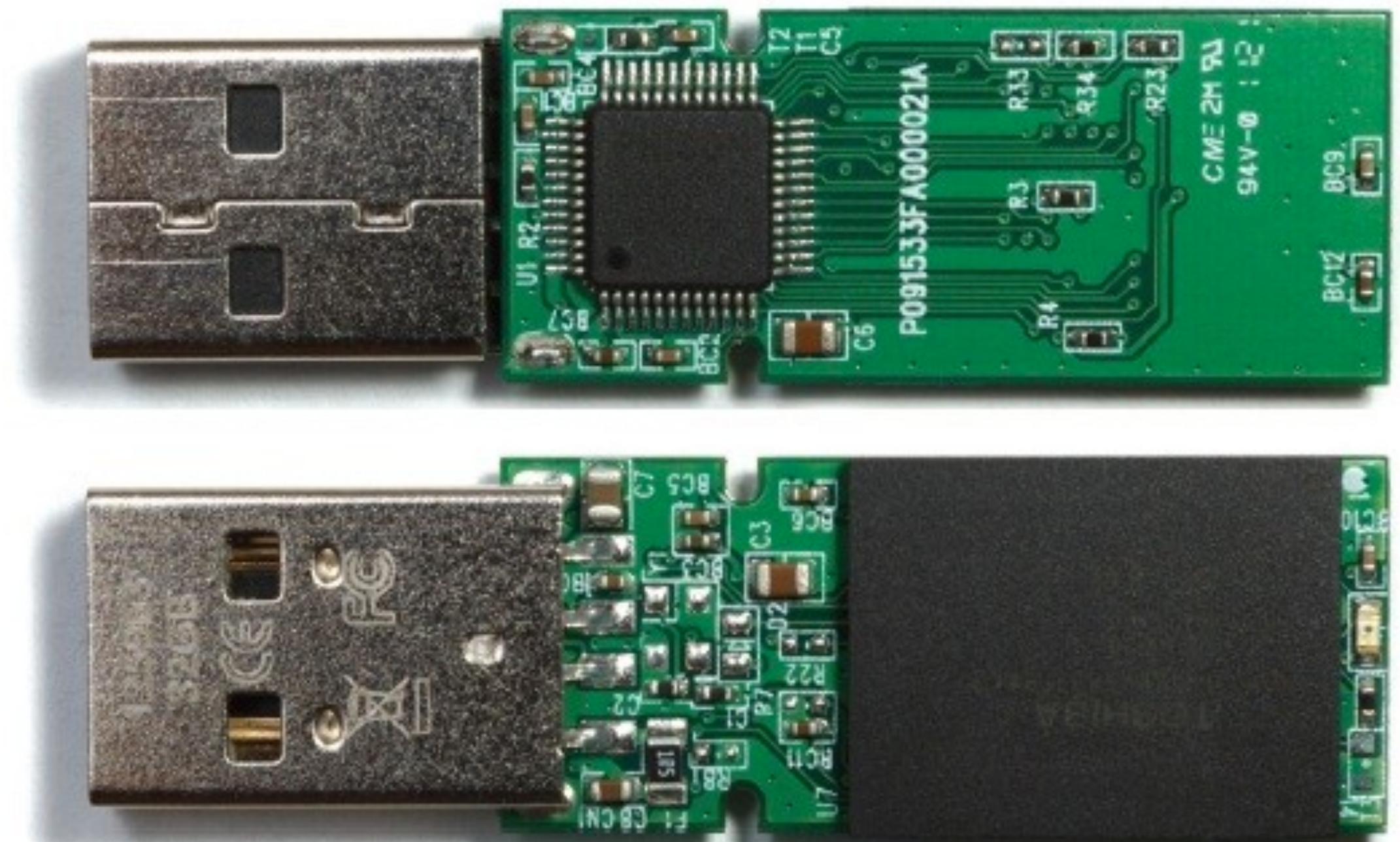
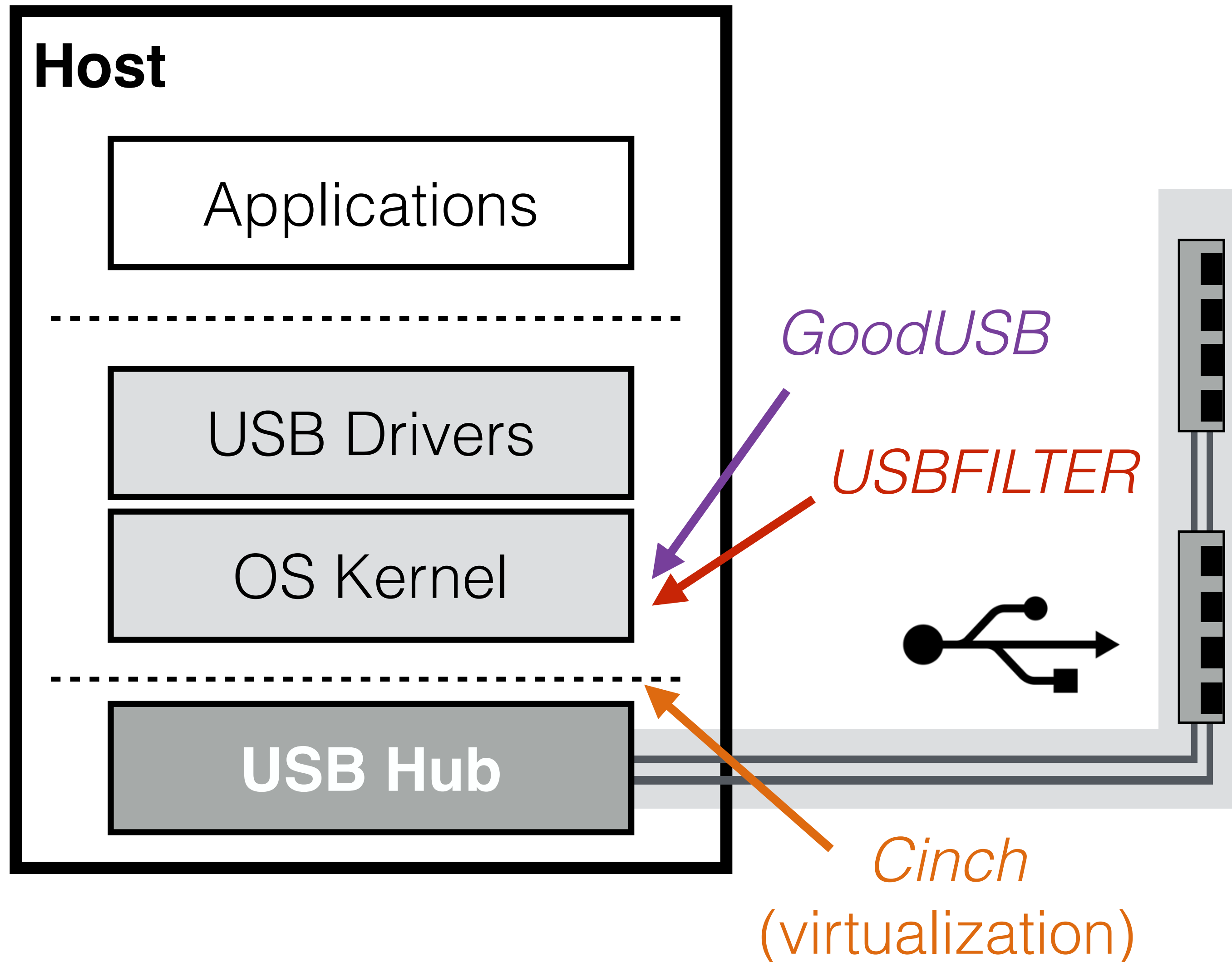
FirmUSB & Related Work



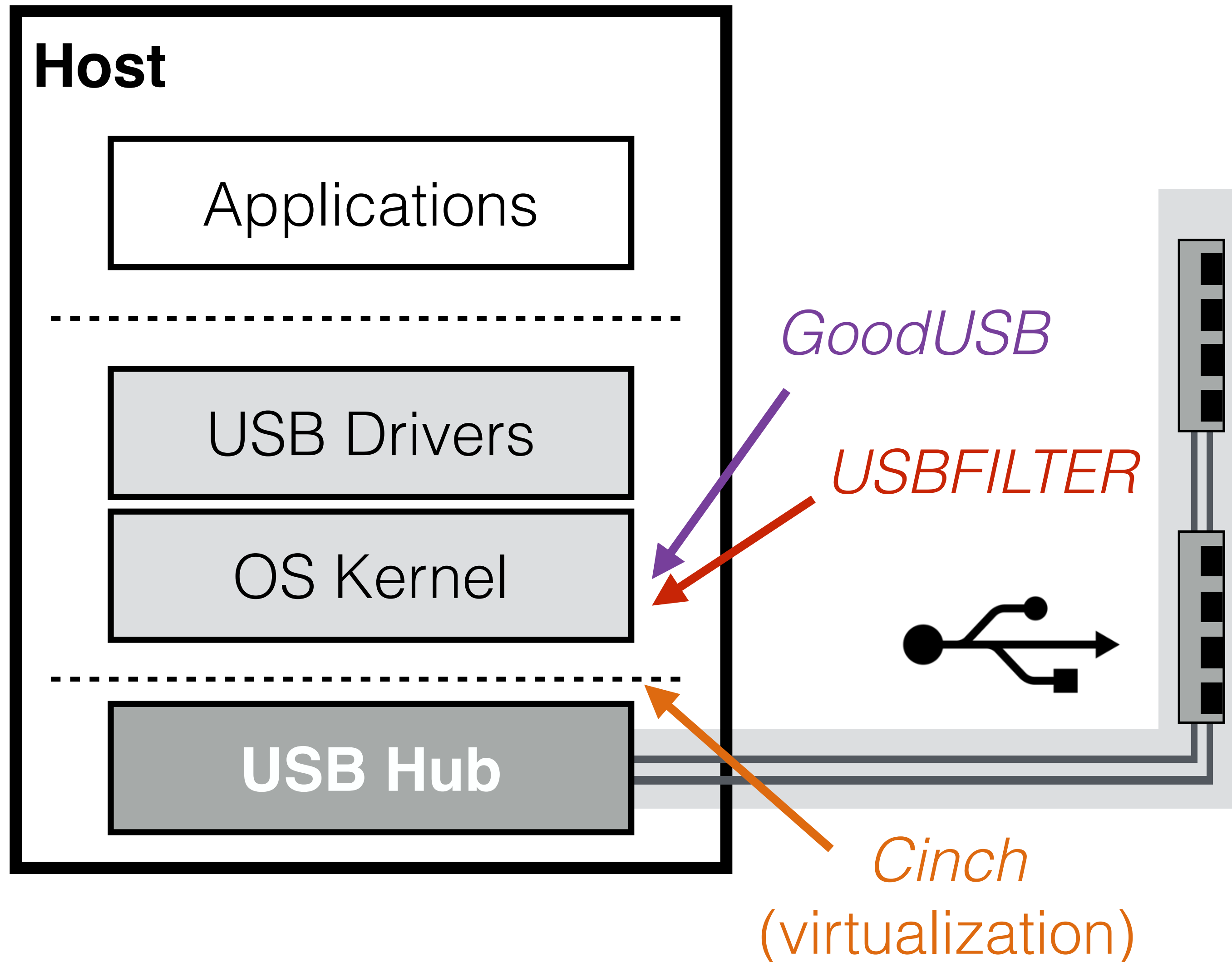
FirmUSB & Related Work



FirmUSB & Related Work



FirmUSB & Related Work



FirmUSB

