USB Communication Security: Threats and Countermeasures

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Informatik
Computer Science

USB Communication Security: Threats and Countermeasures

Master Thesis

Der-Yeuan Yu
dyu@ethz.ch

Tuesday 23rd October, 2012

Advisers:
Prof. Dr. Srdjan Čapkun
Ramya Jayaram Masti
Claudio Marforio
Davide Zanetti

Department of Computer Science
ETH Zürich

To Grandma

Abstract

The Universal Serial Bus (USB) specification is a popular standard for communication between computers and their peripherals. Today, a variety of devices support the USB interface, ranging from mass storage to personal healthcare products. Although these devices may contain sensitive information, they are often security-agnostic (e.g., they send data to the host in plaintext). The commercial nature of these devices often implies that modifying them (or their accompanying software) is impractical.

In this thesis, we investigate the security of the USB communication stack. We discuss the effects of the compromise of hardware or software components in the stack and show that various attacks (e.g. impersonation of USB devices) are possible even if an adversary does not control the host OS. To prevent these attacks, we propose a solution consisting of two parts: a USB device access control module and a secure USB bridge. The USB device access control module uses security mechanisms existing in modern operating systems to prevent access from unauthorized or altered software. The secure USB bridge acts as a proxy between a USB device and the host, providing a secure channel between them. Our design is compatible with existing commodity products without further modification. We show the feasibility of realizing our solution through a prototype implementation of the secure USB bridge and evaluate its performance.

Acknowledgment

I would like to thank Professor Srdjan Capkun for providing a motivating environment for security research and directing me towards a future academic career. I thank my supervisors Ramya Jayaram Masti, Claudio Marforio, and Davide Zanetti for advising my thesis and providing helpful insights and feedback. I also thank my colleagues and friends Nikos Karapanos, Hubert Ritzdorf, Petar Tsankov, Michèle Feltz, Luka Malisa, Nan Zhong, and Ilias Rinis for their valuable input.

Contents

Chapter 1

Introduction

The Universal Serial Bus (USB) specification has been a popular standard for communication between computers and peripherals. Today, a large number of devices support the USB interface. The industry is now shipping more than 3 billion USB devices a year [1], ranging from keyboards, mice, to flash-based storage drives, smart phones, and healthcare devices. With the proliferation of USB, users are storing more data in USB devices, which may include sensitive data such as personal medical information. However, some USB products are designed without security considerations. For example, we have discovered a commercial blood pressure meter which sends out the measurements of patients using an unencrypted protocol [2]. Additionally, the devices and their accompanying software are often provided by manufacturers as-is and are therefore impractical to modify.

To interact with USB devices, users typically use software applications, provided either by their manufacturers or the operating system. Unfortunately, security-agnostic USB products expose their communication with computers to potential attacks. For example, in [3], implementation flaws were discovered in an automated external defibrillator (AED) with a USB programming interface and its accompanying software, allowing the AED firmware to be maliciously reprogrammed by an adversary. In addition, the lack of message-level security allows an adversary in the communication channel to intercept and insert data, posing a threat to both the USB device and the software application. Hardware adversaries such as USB sniffers and fuzzers [4] exist to passively eavesdrop or actively modify transmitted data. Software adversaries may also interfere with the communication by exploiting vulnerabilities of the underlying operating system.

Efforts in securing USB communication have mostly been focused on USB mass storage devices such as USB flash memory or hard disk drives by providing data encryption and user authentication [5, 6]. On the other hand, any other USB device which potentially exchanges data with software should also provide security guarantees. Additionally, securing software applications on the computer from USB-related attacks is also an important issue.

This research aims at investigating threats surrounding general USB communication, both to the device and its application. We explore possible attack vectors in the USB communication environment to identify adversary models and their capabilities. Our objective is to provide communication security for a generic USB device and its software, while also specifying and enforcing software policies on the computer to prevent malicious software from interfering with the communication.

Our solution consists of two parts:

- A **USB software access control module**, which uses the existing technologies for discretionary/mandatory access control and integrity verification in the OS to allow only authorized and unaltered software to access the USB device.
- A **secure USB bridge**, which acts as a proxy for a generic USB device to communicate with the host computer. The bridge creates a secure channel between the device and the host such that their communication provides message integrity and confidentiality.

This solution is transparent to both the USB device and its application, and is suitable for a generic, security-agnostic USB product. Most importantly, no modification is required on the USB device or its application.

Based on our design, we implement a prototype secure USB bridge and a Linux kernel module to serve as a proof-of-concept realization of secured USB communication. We measured the performance of our implementation with commodity USB devices. When encrypting communication (for message confidentiality) using RC4, our prototype bridge decreases the performance by 82.5% for file reading and 74.2% for file writing operations; when the bridge uses AES encryption, the performance decreases by 90.8% and 80.7% respectively. Based on a series of benchmarks, we evaluate the feasibility of our solution and discuss our findings.

This thesis begins by highlighting some background information on the USB specification and the Linux USB software stack in Chapter 2. In Chapter 3 we investigate related work to secure a communication between USB devices and host computers. We present our problem statement in Chapter 4. Chapter 5 analyses the attack surface in our system model based on various adversary capabilities. Based on the analysis, we propose our solution in Chapter 6. Chapter 7 covers the implementation and evaluation of the prototype bridge. We conclude our study in Chapter 8.

Chapter 2

Background

In this chapter, we first provide a highlight of the USB specification which is essential to the comprehension of this thesis. As a case study, we investigate how USB support is implemented in modern operating systems by examining the Linux USB host subsystem.

2.1 USB Specification

USB is a standard for communication between computers and electronic devices, introduced in 1996 and maintained by the USB Implementers Forum (USB-IF) [7]. The standard defines hardware connector specifications and communication protocols between a USB host computer and a USB peripheral device. All USB devices are connected to the host controller through USB hubs. The USB standard has progressed from version 1.0 to 1.1, 2.0, and the latest USB 3.0. For the scope of our study, we focus on USB 2.0, which is currently the most widely version supported by hardware manufacturers.

2.1.1 Device Classification

A USB device is categorized using a three-level classification defined by USB-IF, which consists of: a *base class*, a *subclass*, and a *protocol.* Base on this classification, the host computer OS often contains generic drivers which are developed for each device type such that they can communicate with devices regardless of their manufacturers. This classification information is stored in the device and sent to the host computer when it is plugged in. Table 2.1 shows some popular predefined base classes. Complete base class information as well as definitions of subclasses and protocols can be found in [7].

Base Class	Description
0x01	Audio
0x03	Human Interface Device
0x06	Image
0x07	Printer
0x08	Mass Storage
0x09	USB Hub
0x0B	Smart Card
0x0E	Video
0x0F	Personal Healthcare
0x10	Audio/Video Composite
0xE0	Wireless Controller
0xFF	Vendor Specific

Table 2.1: Common base classes defined by USB-IF

2.1.2 Device Hierarchy

The USB standard allows a single device to perform multiple *functions*; e.g., a keyboard integrated with a USB hub. Devices with multiple functions are known as *composite* devices. The USB specification utilizes a hierarchical tree-like data structure which defines each of these functions in detail. A USB device hierarchy consists of a top level *device* as the root, which contains numerous *configurations* from which the host chooses to activate one. Each configuration provides power demand information, as well as the number of *interfaces* this configuration offers when its active. Each interface represents a particular function, such as the keyboard or the USB hub in the previous example, and contains information on the *endpoints* used for this interface. Endpoints, as the name indicates, are the communication endpoint with which the host exchanges data, and are the terminal nodes in the hierarchy. The endpoints can be further classified into two types: `IN` endpoints, from which the host receives data, and `OUT` endpoints, to which the host transmits data. When a configuration is active, all of its specified interfaces and their corresponding endpoints are available for communication with the host. For example, a composite device functioning as a wireless dongle and a flash drive would expose the two functions as two independent interfaces in a single configuration, each defining their own sets of endpoints to communicate with the host computer. An example hierarchy tree is provided in Figure 2.1.

To assist the USB host computer in learning its functional hierarchy, the device reveals structural information through *descriptors* that the host queries. Descriptors are data structures with predefined data fields that contain information essential to the host. Based on the hierarchy of the USB device,

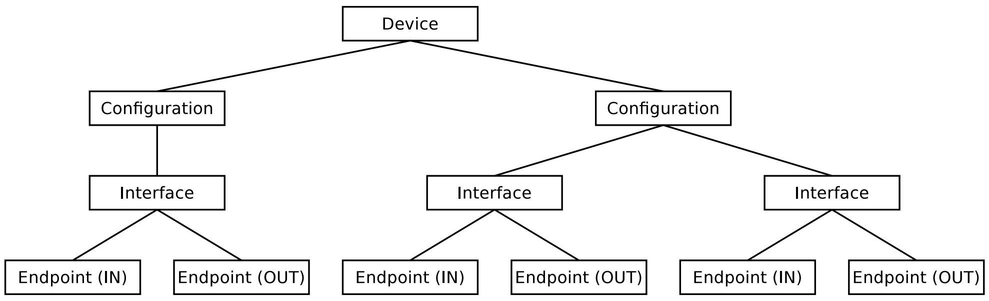


Figure 2.1: USB peripheral hierarchy. Each node is represented by a descriptor which will be sent to the USB host during the enumeration process.

there are five types of descriptors.

- **Device descriptors**, representing the device. The vendor and product IDs are specified here, as well as the classification information. The classification may also be empty, suggesting a composite device, in which case each function class is stored in its respective interface descriptors.
- **Configuration descriptors**, representing a configuration. The power requirements and the number of interfaces when this configuration is active.
- **Interface descriptors**, representing an interface. This descriptor contains the USB classification specific to this interface as well as the number of endpoints available. For composite devices, the classification information of each function is also specified here.
- **Endpoint descriptors**, representing an endpoint. The endpoint address, ranging from 1 to 15, and the direction of data transmission (`IN` or `OUT`) is specified here. Additionally, the endpoint descriptor also specifies the maximum number of bytes which the endpoint can transmit or receive at a time.
- **String descriptors**, storing an array of strings to which data fields in the above descriptors may refer. For example, the device descriptor contains a field specifying the device manufacturer name, which is an integer index to the array of strings defined here.

The USB host requests these descriptors from the device during the enumeration process to configure the host-side driver. By default, an *endpoint zero* is always available and is dedicated to host-device communication during enumeration.

2.1.3 Device Enumeration

USB communication is host-driven. Namely, data is only sent from the device upon request by the host. The USB host computer contains a *USB host controller*, which is commonly implemented as a hardware chip or directly integrated into the south bridge of the motherboard with a USB host firmware.

When a USB device is plugged in, the host initiates the driver enumeration process to identify the device and load the corresponding driver. In detail, the USB hub notifies the host that a new device is attached. The host then communicates with the device to identify the supported speeds, acquire its descriptors and finally assigning the device a unique *address* on the bus. The device class, vendor and product information contained in the descriptors act as the primary reference for the host computer to load the proper device driver. The driver communicates with the device through the USB stack of the OS, exposing the device resources in a logical level to the user, such as a webcam or storage space.

2.1.4 Communication Types

As previously mentioned, communication with the USB device involves sending and receiving data to and from endpoints. The host communicates with endpoint *zero* to obtain essential information such as USB descriptors during the enumeration step. After the endpoints of the interfaces are identified, the host sends *tokens* to them to initiate various types of data transfer.

The USB standard defines the following four data transfer types. Each endpoint takes upon one of these types.

- **Control transfers.** The host may use control transfers to acquire information, such as the USB descriptors, from the device during enumeration. Control transfers may also be used for any miscellaneous configuration requests defined by a USB vendor. Messages transferred during control transfers are specially formatted as *setup packets*. Each setup packet consists of either a standard command to the device (e.g., fetching a descriptor or assigning a device configuration) or a product-specific command defined by the vendor. During control transfers, an additional data field of variable length may also be associated with the setup packet. This field may be filled either by the host to send specific data to the device or by the device to contain its response (e.g., the device descriptor) to the command in the setup packet.

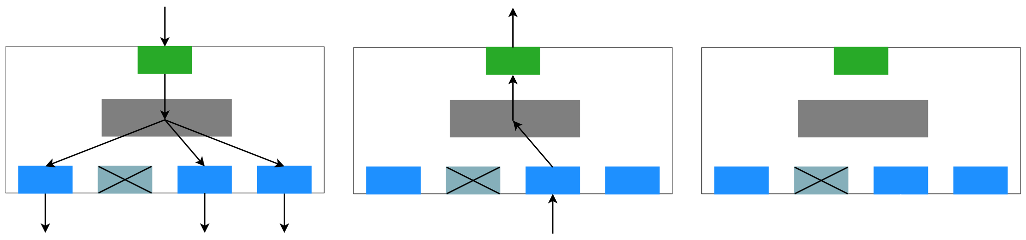
- **Bulk transfers.** The bulk transfer is the basic method of transmission and is typically used for high-volume data communication such as printing documents or storing files in mass storage. Bulk transfers guarantees data delivery but provides no dedicated bandwidth.

- **Interrupt transfers.** When a device requires the attention of the host, an interrupt transfer is used to send data. Since the communication is host-driven, the host should regularly poll to check if any interrupt data is available from the device. Interrupt transfers guarantee an upper-bound on the latency of transmission and provide error detection and retransmission mechanisms. A HID peripheral such as a mouse often uses interrupt transfers to achieve instantaneous response for the sake of smooth user experience.
- **Isochronous transfers.** Devices providing data with a limited time window of validity may use isochronous transfers to send it. Examples include streaming video from a webcam or streaming audio from a microphone. Isochronous transfers do not provide guaranteed delivery. However, for timely delivery of data, the host reserves a portion of the USB bandwidth for isochronous transfers.

2.1.5 Data Packet Format

The USB specification defines protocols for data packets exchanged between the host and the device. Each data packet contains information about the transfer and its corresponding data. The header information consists of the following fields.

- The `SYNC` field, a coded sequence used for data alignment and clock synchronization on the recipient hardware.
- The `Packet` field, containing the following information.
  - Packet identifier (`PID`), data type tokens (`IN`, `OUT`, `Start-of-Frame`, and `SETUP`) and various handshake information.
  - Function address (`ADDR`), containing the address of the function assigned by the host after enumeration.
  - Endpoint address (`ENDP`), containing the endpoint number of the function.
  - Frame number, a counter for the number of data frames transferred.
  - Data, the actual data to be transferred.
  - Cyclic redundancy check, allowing error checking of all fields except the PID field.



(a) Downstream broadcasting. The hub broadcasts downstream data packets to all down connected downstream ports.

(b) Upstream forwarding. The hub forwards data from one downstream port to the upstream port.

(c) Idle. All connectivity is disabled and no data packet is routed.

Figure 2.2: Hub connectivity illustration. The hub contains one upstream port (green) and multiple downstream ports (four in blue) as well as its routing logic (gray). In this scenario, three of the downstream ports are connected with USB devices.

2.1.6 Data Connectivity

Now we introduce the actual routing of data packets in the USB specification. Recall that all USB devices are connected to the host computer through USB hubs. The host controller exposes a *root hub* which serves as the first tier in the USB topology. A USB hub contains an upstream port, connected to the host side, and potentially multiple downstream ports, which are connected to devices. Routing logic built in the USB hub forwards data between the upstream and downstream ports in different ways depending on the direction of the data packets.

Figure 2.2 illustrates three states the hub may take upon during message transmission. When data packets are detected from the upstream port, the hub broadcasts them to all downstream ports. Devices connected to the downstream ports are responsible for observing the function address (`ADDR`) field and discarding any packets other than the ones which match their assigned address during enumeration. Before enumeration is complete, the default address for the device is 0, to allow enumeration over its default endpoint *zero*. When the hub detects data sent from a downstream port, it enables only its connection with the upstream port and forwards the data packet. As a result, no other downstream ports on the hub observes the data packet. An idle state is also defined for the USB hub in the absence of data packets from both the upstream and the downstream ports. In this case the USB hub disables all connectivity and awaits the detection of data packets from either end.

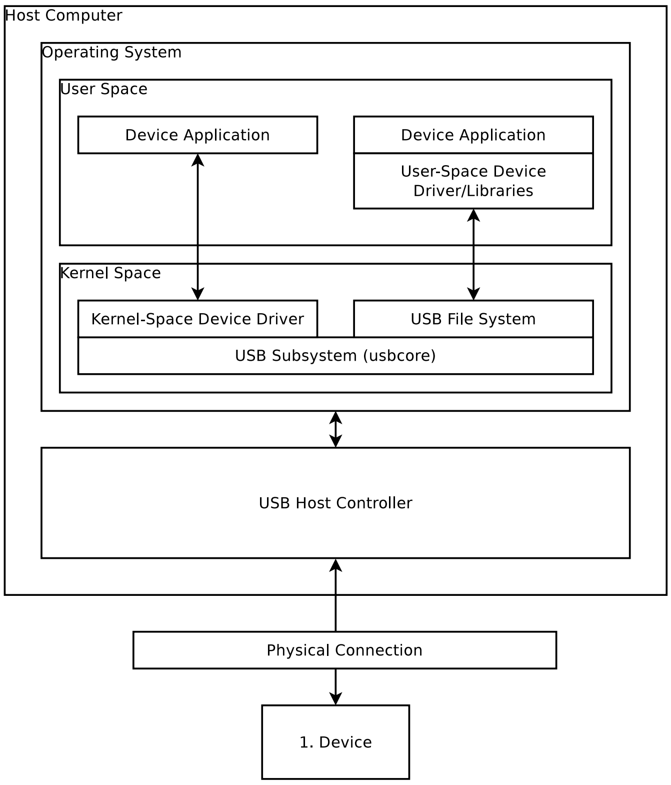


Figure 2.3: The Linux USB host software stack

2.2 Linux USB Host Subsystem

The Linux kernel supports USB host-side communication via two main components, `usbcore`, the central framework for USB communication, and a USB host controller driver, used to control a vendor-specific USB host controller in hardware to actually communicate with USB devices. `usbcore` provides a unified programming interface for software to access USB devices and interacts with host controller drivers, which is specific to various USB host controller manufacturers. The `usbcore` module exposes USB functionality through kernel-space functions for kernel-space software and a virtual file system (`vfs`) known as the USB File System (`usbfs`) for user-space applications. Figure 2.3 provides an overview of the USB subsystem and its interaction with USB software.

Kernel-space device drivers for common USB devices such as keyboards, mice, and mass storage, communicate with USB devices using kernel-space functions provided by `usbcore` [8]. *USB Request Blocks* (URBs) are basic elements to initiate data transfer with USB devices, consisting of essential information including

- USB device information
- Transfer type and various settings
- Pointers to memory space either containing the data to transmit or the buffer prepared to store received data
- A completion handler function, invoked after the USB transfer is completed

Kernel modules may also make use of Linux memory `I/O` primitives such as Direct Memory Access (DMA) or Scatter/Gather data structures to improve the efficiency of transmission [9]. DMA relieves the burden on the CPU for data copying by delegating communication between the main memory and the USB host controller to an external DMA controller. The Scatter/Gather data structure allows transferred data to be stored in non-contiguous memory space.

With various helper functions to initiate a URB data structure, kernel-space USB drivers pass the URB data structure to the kernel function (named `usb_submit_urb`) to schedule the transfer. After the transfer completes successfully or fails due to errors, the completion function is called to post-process the transmission and execute clean-up routines. To user-space, kernel USB drivers ultimately expose USB device functionality through device files in the file system, which may be accessed by user-space programs. For example, Linux webcam drivers abstract the low-level USB communication specifics and provide the device files `/dev/video*` for user-space webcam software to access using `ioctl` calls [10].

In the absence of a kernel-space driver for a particular USB device, user-space software may still interact, using `ioctl`, with USB devices by accessing the `usbfs`, which directly exposes USB devices as device files. The `usbcore` module processes requests sent through `usbfs` and creates URB requests similar to those issued by kernel-space drivers. In this case, the user-space software is responsible for implementing the communication protocols to communicate with the device. Software libraries such as `libusb` [11] provide an abstraction layer to these device files and allow easier access for user-space software.

Chapter 3

Related Work

3.1 USB Security

The widespread use of external computer peripherals has raised security awareness regarding their impact on overall system security [12]. Particularly, the popularity of USB in the consumer electronics market has led to security research in USB devices. In [13], the authors introduce multiple ways of compromising a host computer such as abusing auto-run features in the OS to automatically execute malware or exploiting host USB drivers by unexpected protocol messages. In [14], the author explores possibilities of programmable HID devices sending mouse or keyboard signals to the host computer to issue administrative commands or execute malicious programs. Hidden channels were identified in [15], which allow a USB device and a host computer program to collude and leak host-side information. For example, malicious software may send personal information found in the host computer to a malicious USB keyboard using special sequences of change notifications for Caps Lock, Num Lock and Scroll Lock states. To prevent against attacks from malicious devices, it has been suggested that in a security-sensitive environment, USB devices be audited and whitelisted when plugged into the computer, allowing only recognised devices to interact with host drivers [16].

Another aspect in USB security lies in identifying potential attacks over the communication channel between a legitimate USB device and a host [3]. Generally, USB devices capable of exchanging sensitive data should implement security features to ensure message integrity or confidentiality. However, studies have shown that these secure protocols, although existent, have not been able to provide high security guarantees due to design and implementation flaws. In [17], the authors investigated communication between personal health record devices and their corresponding device applications, and identified vulnerabilities in password storage and during authentication processes. Furthermore, these devices often contain backdoor functions for emergency purposes, allowing partial data access without password authentication but posing as a threat to personal information privacy. In [18], some commercially available USB flash drives with data encryption features were analysed, revealing weaknesses in communication protocols and retrieving access passwords. As a solution, various USB communication protocols have been proposed to provide secure user authentication [19]. Secure communications specifically for USB mass storage devices have also been explored and evaluated for their performance impact [5, 6]. Additionally, in [20], a patent was filed disclosing a mechanism which makes use of trusted execution environments in the host computer to process sensitive USB traffic.

Previous efforts, as mentioned above, explore security issues for specific USB devices. However, there has been limited research in analysing security of a communication environment consisting of generic USB devices and their software applications, which is addressed by this thesis.

3.2 Software Security

There has been extensive research in providing a secure environment to protect legitimate applications and enforcing policies during software execution. Here we classify these techniques into three categories: *launch-time*, *run-time*, and *post-execution* software security.

3.2.1 Launch-time Security

Before execution, the system (the entity that loads the binary) must decide whether the binary is allowed to run. Efforts can be classified into *blacklisting* and *whitelisting*, as listed below.

- Blacklisting
  - Antivirus software currently available in the market provides detection of malware such as rootkits, worms, and quarantines suspect malware to prevent user execution. The software functions by scanning the computer and searching for known malware signatures collected in a database. In some cases, certain heuristics are also used to identify possible zero-day attacks [21, 22].
- Whitelisting
  - Verified kernels like Coyotos [23] and seL4 [24] are efforts to construct small and trustworthy kernels through development with formal methods, providing a reliable basis for software execution. However, their stress on formally-proven reliability and does not scale well for use in modern OSes.

  - Trusted computing techniques aim at guaranteeing code integrity. Fail-stop mechanisms can be designed to prevent software execution whenever its integrity measurement is inconsistent with that of a known good state [25, 26, 27, 28].
  - In the particular case with device drivers, authenticating driver software based on their signatures allows only legitimate vendor drivers to be loaded into the system. This mechanism is currently employed in modern Microsoft Windows operating systems [29]. Moreover, all modern operating systems make efforts to include drivers of most devices in the kernel or a trusted repository for installation [30, 31]. Regardless of these safeguards, users intervention are often required to allow loading of unauthenticated drivers. As a result, the threat of malware infection still remains due to user negligence.

3.2.2 Run-time Security

Run-time security measures address two issues: protecting the system environment and ensuring correct software execution [32]. The common solution to achieving either goal is isolation. In the former case, a sandbox is often created to envelop an application from unwanted access to the host environment. In the latter case, guaranteeing platform integrity can normally be ensured by applying the measures described previously to ensure a safe execution environment, thereby creating a Trusted Computing Base (TCB). Furthermore, there have been efforts to reduce the code size of the TCB such that the system can be booted with some compromised components yet still guarantee correctness of program execution.

- Sandboxing
  - Virtualization and hypervisors such as VirtualBox [33] and Xen [34] allows users to create individual virtual environments in which untrusted applications may run in isolation from other system software.
  - Program shepherding [35] is a method for monitoring control flow transfers during program execution, and specifically 1) restricts executions of code, 2) restricts control transfers, and 3) enforces software policies.
  - Information flow control [36, 37] is similar to program shepherding but focuses on containing the transfer of important data. Trusted software modules oversee the exchange of sensitive data with untrusted software to ensure confidentiality and integrity.

  - Mandatory access control, which is implemented by software like AppArmor [38] and SELinux [39], restricts privileges of individual software based on custom defined rules and constraints.

  - SUD [40] is an effort to secure the Linux kernel by migrating kernel-space drivers to user-space such that malicious drivers do not compromise the kernel. In addition, further constraints are imposed to restrict device hardware from accessing memory by using existing components like the IOMMU.

- Trusted Computing Base Minimization

  - Flicker [41] allows applications to run in trusted virtualized environments provided by modern Intel and AMD processors, effectively removing the need to trust a potentially compromised operating system. This method requires modification of the application and constant use of the physical Trusted Platform Module (TPM), which is often slow [42].

  - TrustVisor [43] is an improvement of Flicker which is a hypervisor that utilizes a software dynamic root of trust on guest virtualizations. This removes the computational overhead of cryptographic functions in the TPM.

  - CARMA [44] exploits the Cache-as-RAM mechanism on Modern CPUs, which is originally used to load the memory controller during bootstrap, and uses it as an isolated execution environment for generic software. As a result, the correctness of the application relies on the integrity of the CPU. However, this solution depends on specific hardware setup.

  - TRESOR [45] is a patch to the Linux kernel which implements the AES encryption algorithm and key management solely on the processor, thereby also excluding the memory from the TCB. TRESOR makes use of the CPU support for AES-NI instructions for AES encryption but is not applicable to generic software applications.

- Specialized security hardware is also another direction in research.

  1. AEGIS [46] is a processor architecture which contains physical random functions and off-chip memory protection mechanisms. However, the requirement for special hardware for these solutions does not fit our user environment, which consists of general personal computers.

3.2.3 Post-execution Security

After software has completed execution, an intuitive solution to guaranteeing correctness is often done by comparing the system state with a known good state. With the help of the TPM, we may perform *remote attestation* [47], which proves to an external entity the integrity of the system.

Another more extreme solution which shifts the reliance on a single computer to multiple computers is *secure multi-party computation* [48], from which a correct computation result can be generated through cryptographic protocols and consensus despite the existence of compromised computers. This solution aims at computation correctness through redundancy, and are therefore infeasible for general personal computing.

Chapter 4

Problem Statement

After introducing background information, this chapter provides an overview discussion of our problem. We begin by introducing the system model of USB communication in a general setting and identify the goals of the adversary. Based on the system model and adversary goals, we outline the objectives of our thesis.

4.1 System Model

Generally, a USB communication between a device and a host computer consists of the following components.

- A USB device provided by a vendor, later referred to as the *device*.
- A computer, later referred to as the *host*, which runs an OS supporting both the device and its application. In our study, we focus on the Linux OS.
- A software application which communicates with the device, later referred to as the *device application*. This includes user applications and device drivers included in the host or provided by the vendor of the device.

The interaction between the host computer is depicted in Figure 4.1.

In this environment, a typical use case includes the following steps.

1. As an optional step, the user first installs the device driver.
2. The user plugs the device into the computer.
3. The user starts the device application.
4. The user interacts with the device through the device application.

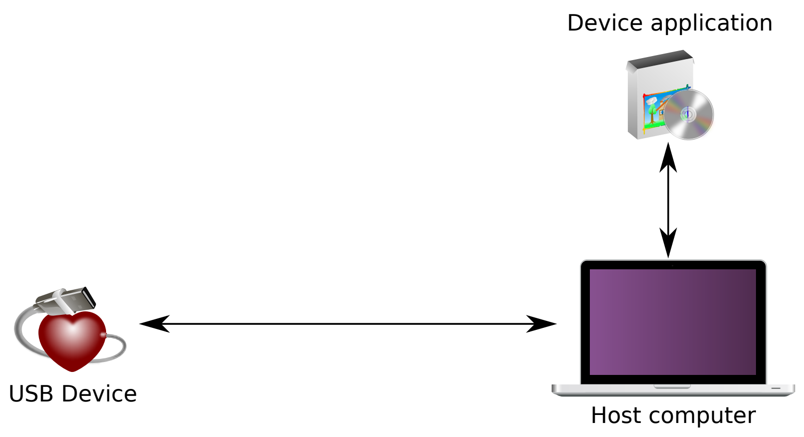


Figure 4.1: The system model for general USB communication, consisting of the USB device, the device application, and the host computer.

The Linux kernel USB subsystem, as introduced in Section 2.2, provides the abstraction for software applications to interact with the USB device. The device application may interact with the device through kernel-space device drivers or the `usbfs` virtual file system.

4.2 Goals of the Adversary

From the security perspective, we differentiate between two types of malicious goals of the adversary.

- **Steal information.** The adversary may wish to passively obtain sensitive data from the communication between the device and the device application. The sensitive data could be, for example, medical records of a patient stored on a USB personal healthcare device. Information stealing can be achieved by eavesdropping the USB data transferred during communication. In addition, an unsecured communication protocol also allows the adversary the possibility of communicating directly with the device or its application.
- **Interfere with communication.** In contrast to the passive attack of stealing information, the adversary may intend to perform an active attack by interfering with the messages exchanged between the endpoints. For example, in the case of a personal healthcare device, bogus personal records may be sent to the device application. Erroneous configurations may also be sent to the device to affects its proper functions. The adversary may achieve this by compromising certain components of the system, or communicate directly with the endpoints if the communication protocol is insecure.

4.3 Thesis Objectives

In this thesis, we investigate potential security issues with USB communication. We first survey previous work, as summarized in Chapter 3, on the security of USB communication and discuss their differences with this study. We then identify the attack surface under the system model with a Linux-based host computer, and analyse all possible attacks the adversary may perform to achieve her goals as well as the feasibility of various attacks. Based on the feasibility analysis, we define a realistic model of adversary capabilities, and provide a solution to ensure communication security for generic security-agnostic USB device and their applications under this threat model.

Chapter 5

Security Analysis of Linux USB Communication

In this chapter, we provide definitions of attacks by the adversary, and discuss how they can be realized in the USB communication stack such that the adversary may achieve her goals defined in Section 4.2.

5.1 Definition of Attacks

To achieve the goals as previously defined, the adversary utilizes the following five attack primitives over the communication.

- Eavesdropping
- Modification
- Insertion
- Deletion
- Impersonation

The first four attacks assumes that two legitimate endpoints[1] have established connection and started communicating. We now continue to the defining these attacks.

We break down communication to single-direction message transmissions between legitimate endpoints A and B, namely, the device and its application, or the application and the device respectively; the adversary is represented by E. The attack primitives are illustrated in Figure 5.1.

[1]We use the term *endpoint* here to refer to the entities on either end of a communication channel. This is not to be confused with the endpoint defined in the USB specification.

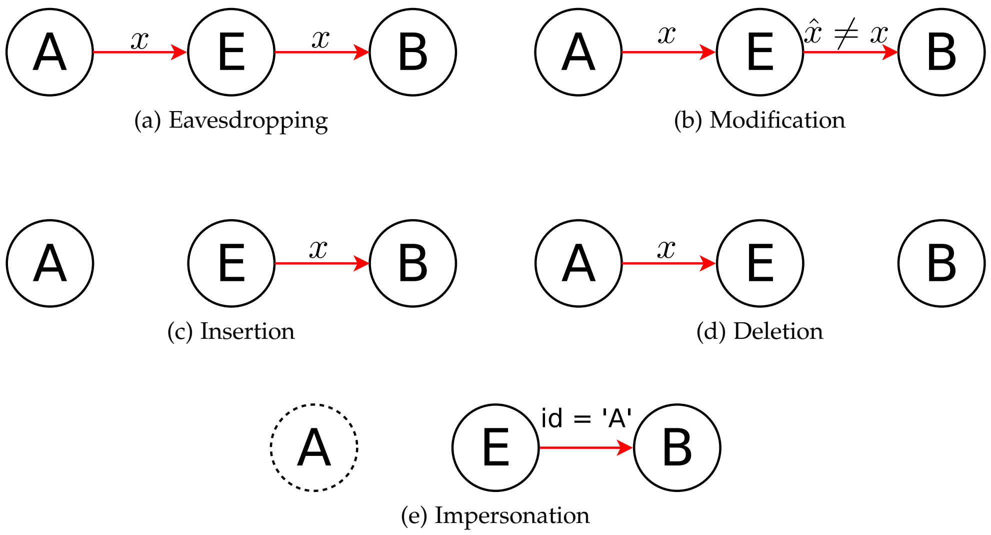


Figure 5.1: The attack primitives for message eavesdropping, modification, insertion, and deletion. An variant of the insertion attack is the impersonation attack, in which the adversary sends false identity information to one endpoint. Dotted lines imply that the entity may or may not be active in the communication session.

Eavesdropping

Eavesdropping is the act of obtaining messages exchanged between two endpoints while they communicate with each other.

- Endpoint A sends $x \in \Sigma$
- Endpoint B receives x
- Adversary E receives x

Modification

Modification is defined to be modifying messages sent between endpoints such that a message received at the destination endpoint is different from the one sent from the source endpoint.

- Endpoint A sends $x \in \Sigma$
- Endpoint B receives $\hat{x}$, where $\hat{x} \neq x$.

Insertion

Insertion is defined as a synthesis of a message to send to one endpoint, misleading it to interpret the synthesised message as originated from the source endpoint currently in communication.

- Endpoint A does not send any message.
- Endpoint B receives x.

Deletion

Deletion is defined to be the elimination of a message during transit such that a message sent from a source endpoint does not reach its destination endpoint.

- Endpoint A sends x.
- Endpoint B does not receive any message.

Impersonation

Impersonation is defined as the adversary establishing a communication channel independent of a legitimate endpoint with the other endpoint while claiming itself as be legitimate. Note that with impersonation attacks, only the targeted endpoint needs to exist.

5.2 Feasibility of Attacks

After identifying the different types of attacks, we investigate how they can be accomplished by the adversary.

From the Linux host USB subsystem introduced in Section 2.2, the following components could be controlled by the adversary.

- A malicious device
- Compromised physical connection between the device and host
- Compromised user account in the host OS
- Compromised host OS kernel
- Compromised host hardware

These different levels of system compromise grants the adversary various capabilities to mount the attacks defined in Section 5.1. We discuss the feasibility of these attacks and their requirements. Table 5.1 overviews the capabilities of the adversary in these different levels. We further base the following discussions on how attacks can be achieved. The summary of these dicussions are shown in Tables 5.2, 5.3, and 5.4.

Controlled Entity		Essential Components	Achievable Attacks
Malicious device		A USB device which is connected to the host	Impersonation of device, eavesdropping data from host, and inserting data to host
Physical connection		USB cables, hubs	Any
Host	User-level privileges	User-space resources such as files, environment variables, or runtime memory space	Potentially any attack, depending on how the resources used by the device application can be exploited
	OS	Kernel-space resources such as modules and the USB subsystem	Any
	Hardware	CPU, memory components, USB host controllers, etc.	Any

Table 5.1: Summary of the capabilities of the adversary under various levels of compromise

A malicious device

Consider the case where the adversary is a malicious USB device (except for a USB hub) plugged into the host computer. Given an insecure communication protocol between the legitimate device and its application, she can replicate insecure communication protocols and therefore achieve impersonation attacks by pretending to be a legitimate device while communicating with the device application.

As an instance, the malicious device is plugged into the computer along with the legitimate device. The device application starts and observes two compatible devices available for use. The attack would be successful if the malicious device is selected, either with or without user intervention.

Recall in Section 2.1.6, the USB hubs broadcasts all data sent from the host to all downstream devices, placing the responsibility of ignoring unrelated data on USB devices. As a result, the malicious device may eavesdrop on USB data sent from the host by recording all traffic received, regardless of the values in the function address field. When dealing with upstream data sent from devices to the host, USB hubs forwards data packets without verification from downstream ports to the host whenever available. This grants the adversary the potential to mount insertion attacks on the host by sending out data packets to the host with a forged function address and endpoint address.

Other attacks, namely, eavesdropping data sent from other devices, inserting data to devices, modifying or deleting data sent from either the host or the device would not be accomplished due to the lack of connectivity defined by the specification of USB hubs.

Compromised connection between the device and host

The physical connection, including the USB cables and USB hubs, can be compromised by the adversary through techniques from basic wire-tapping to controlling a USB hub. In this event, all attacks can be achieved if the communication protocol is insecure. The adversary processes all traffic between the device and its application and trivially achieves eavesdropping, modification, insertion, and deletion of USB traffic. Additionally, she may also actively send bogus messages to either end and achieve impersonation attacks.

Compromised user account in the host OS

The device application is executed by a user, and uses a set of *resources* in the host OS to which the user is granted access. These resources may include the following.

- Files such as pipes, dynamically-linked libraries, software configurations, special files mapping to system states and components
- Environment variables
- System calls
- Memory

Compromising these resources may be achieved if the adversary has the access rights to them under the discretionary access control model in Linux, or if the adversary succeeds in an application level exploit to escalate her privileges. Once a resource accessed by the device application is compromised by the adversary, she can potentially attack the device as well as the application. The level of threats depends on the resources used by the application. Here we provide a few examples.

- A dynamically-linked library file which provides functions for handshaking, reading from, or writing to the device may be replaced with a malicious version. The malicious library may either passively store all messages transmitted, or actively modify, insert, and delete messages. In addition, impersonation attacks on the application can also be achieved by creating a bogus instance of a USB device.

- Environment variables, should they be used to contain configurations of the device application, can also be altered to disrupt its execution. A poor application design might read the USB device file path stored from an environment variable. By changing the this value, the adversary can redirect its communication to a malicious USB device and achieve an impersonation attack.
- The run-time memory of an application is often a major target of software attacks. One example would be exploiting the software vulnerabilities, such as buffer overflow, to execute arbitrary code. In the case where the software is bug-free, the adversary may also attach debuggers like `gdb` to the runtime execution and modify the stack and heap memory. Consider the example of the device application opening the USB device node file using the `open` system call, where the arguments are stored in the stack memory. By modifying the file path in `gdb`, the adversary may have the application open another device without triggering a warning, thereby performing a successful impersonation attack. Generally, when the adversary attaches `gdb` to a running application, its stack and heap memory space are open for modification, and any variables used by the application can be altered to achieve any attack possible.

We listed some example methods which may be employed by an adversary with the necessary access rights to compromise the components. Other resources such as signals or network sockets, if compromised, may also pose a threat to the correct execution of the device application.

Note that in addition to attacking the device application, the adversary may even simply execute a malicious application of its own and communicate with the device, thereby achieving an impersonation attack while the legitimate device application is either blocked from accessing the device or not even executing.

Compromised host OS kernel

The host computer OS kernel is the heart of the software that provides the device application an execution environment to communicate with the device. The major component involved in the kernel is the USB subsystem. If the kernel is compromised, any attack is possible. Here we lists some attacks that combined will cover all those we have defined.

- The kernel module `usbmon` provides an interface where all USB messages are stored for debugging purposes. A malicious takeover of the kernel could imply sending these privileged data to a remote storage.
- The `usbcore`, which handles all USB communication in Linux, can be compromised by the adversary to achieve any attack.

- A compromised kernel may start any user level program, including a malicious version of the device application, to communicate with a legitimate device, thus achieving an impersonation attack on the device endpoint.
- System calls provide an interface for user-space applications to interact with the OS kernel. Should their functions be altered, the adversary may also accomplish certain attacks. Impersonation attacks can be performed by modifying the `open` system call, which is used to open USB device nodes. Eavesdropping, modification, insertion, and deletion attacks can be achieved by modifying the `ioctl`, `read`, `write` system calls.

Generally, if the OS of a host computer is compromised by the adversary, any attack we have defined is possible.

Compromised host hardware

All attacks are achievable when important USB hardware is compromised. If the adversary controls a USB host controller, then she may easily eavesdrop, modify, insert, or delete messages sent to USB devices. Even forging a fake device to the OS as well as sending arbitrary messages to the device is possible, thus achieving impersonation attacks.

Adversary Presence	Attack	Adversary Implementation
Malicious Device	Eavesdrop	Record all broadcast data sent from host
	Modify	N/A
	Insert	Send data with forged function addresses of other devices to the host
	Delete	N/A
	Impersonate	*Impersonate device:* implement protocol to communicate with host PC
Physical Connection	Eavesdrop	Implement malicious hub to monitor and alter data
	Modify	
	Insert	
	Delete	
	Impersonate	*Impersonate device:* implement protocol to communicate with application *Impersonate application*: implement protocol to communicate with device
Host Hardware	Eavesdrop	*Malicious CPU:* Execute arbitrary code to achieve all attacks *Malicious memory components:* Monitor or alter memory contents to corrupt data or executable code to achieve all attacks *Malicious USB host controller:* Monitor or alter data to be sent to either endpoint, generate bogus data to impersonate device or application
	Modify	
	Insert	
	Delete	
	Impersonate	

Table 5.2: Implementing attacks external to the host computer

Adversary Presence	Application Privilege	Driver Execution Space	Attack	Adversary Implementation
Regular user	Administrator	Kernel-space	Eavesdrop	N/A
			Modify	N/A
			Insert	N/A
			Delete	N/A
			Impersonate	N/A
	Regular user	Kernel-space	Eavesdrop Modify Insert Delete	1. If device application is running, kill it 2. Modify application resources (libraries, environment variables, configurations) to covertly eavesdrop, modify, insert or delete data 3. If device application was killed, restart it
			Impersonate	*Impersonate application:* release application claim on USB interface, execute malicious application
	Regular user	User-space	Eavesdrop Modify Insert Delete	1. If device application is running, kill it 2. Modify user-space driver or application resources (libraries, environment variables, configurations) to covertly eavesdrop, modify, insert or delete data 3. If device application was killed, restart it
			Impersonate	*Impersonate application:* release application or driver claim on USB interface, execute malicious application or driver

Table 5.3: Implementing attacks as a regular user of the host OS. Here we have to distinguish the execution privileges of the device application and the execution space of the device driver.

Attack	Adversary Implementation
Eavesdrop Modify Insert Delete	Generic way: 1. If device application is running, kill it 2. Replace driver with malicious version, install malicious USB subsystem, or modify application resources (libraries, environment variables, configurations) to covertly eavesdrop, modify, insert or delete data 3. If the device application was killed, restart it To eavesdrop, use `usbmon`
Impersonate	*Impersonate application:* release any claim on USB interface, load malicious module or application *Impersonate device:* forge device file, load malicious driver

Table 5.4: Implementing attacks as the administrator (root) of the host OS. Note that since the adversary in this scenario is given root privileges, we combine the discussion of application execution privileges and execution space of device drivers.

5.3 Refined Adversary Model

As discussed in the previous text, security issues of USB communication stems from the fact that various compromised components in the system model allow the adversary to perform attacks in different ways. We now establish an adversary model for our problem to define the scope of the research.

The adversary as a malicious USB device or part of the physical connection to the host computer poses a viable threat to the computer. Considering the limited previous work on generic USB communication security, we are interested in finding a solution to this problem.

For adversaries on the host computer with access to the OS kernel (this includes adversaries in user space with root privileges), solutions already exist to provide isolated execution environments for software such that applications can be protected from attacks from the OS. For solutions employing TCB minimization, the isolated execution environment would have to include any resource from the OS that is accessed by the device application, which significantly increases the code size of the isolated environment and therefore contradicts the original intent of minimizing trusted code size. Additionally, all USB software would require modification of its source code to make use of modern isolated execution technologies, which is impractical to implement on commodity USB products. On the other hand, for isolated execution based on hypervisors, the entire OS of the host computer has to be re-installed, which is also infeasible to realize. To provide a solution for generic USB devices, device applications, and the use case in the general system model, we therefore assume that the OS of the host computer is trusted and is not compromised by the adversary. This allows us to focus our research on the security of USB communication.

We focus on a refined adversary model to design our solution. The adversary we consider in this thesis is composed of the following.

- Malicious hardware. The adversary may control malicious USB devices, USB hubs, cables, and even the USB host controller on the host computer.
- Malicious software with regular user privileges with access rights to certain secure-agnostic USB devices and device applications.

Chapter 6

Design of a Secure USB Communication Stack

We introduce the solution to our problem by first specifying the requirements based on our refined adversary model, followed by the design which fulfils them. Finally, we analyse the security of our architecture and discusses how it prevents various attacks under our adversary model.

6.1 Requirements

We first identify the requirements for our design of a secure communication environment against the adversary model mentioned in 5.3.

Requirement 1 *The solution should ensure authorized access to the device from the host computer.*

This requirement is essential to prevent the software adversary from compromising the communication between the device and application.

Requirement 2 *The solution should provide a secure communication channel which provides message confidentiality and integrity to messages that are exchanged between the device and application.*

The fulfilment of this requirement prevents the hardware adversary from compromising the communication between the device and the device application to achieve eavesdropping, modification, insertion, deletion, and impersonation attacks.

Requirement 3 *The device and the device application are not changed.*

Considering the infeasibility of modifying the hardware of the device and software of the device application for generic USB products, the design should provide the security guarantees described in the first two requirements independently from the USB product implementation.

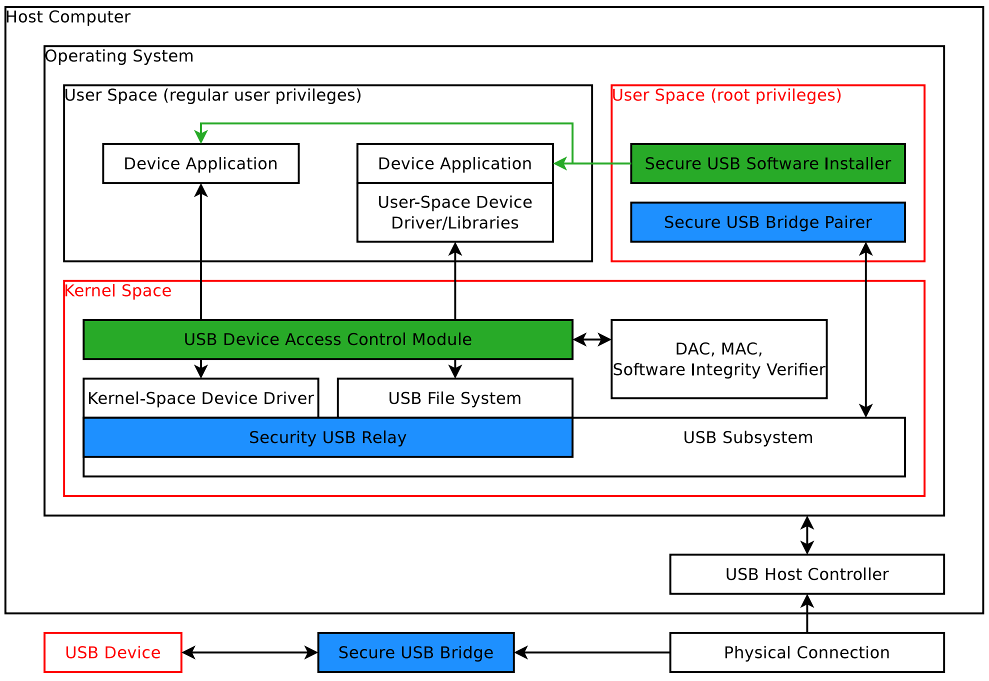

Host Computer
Operating System
User Space (regular user privileges)
User Space (root privileges)
Device Application
Device Application
User-Space Device Driver/Libraries
Secure USB Software Installer
Secure USB Bridge Pairer
Kernel Space
USB Device Access Control Module
DAC, MAC, Software Integrity Verifier
Kernel-Space Device Driver
USB File System
Security USB Relay
USB Subsystem
USB Host Controller
USB Device
Secure USB Bridge
Physical Connection

Figure 6.1: Solution design. Green marks the USB device access control module (and its associated software) which uses existing access control mechanisms in the OS to allow only authorized device applications to access the device. Blue components belong to the secure USB bridge solution which ensures communication channel security. Red boxes represent trusted regions in the environment.

6.2 Design

To fulfil the requirements, our design introduces various components into the original system model, as shown in Figure 6.1. The design consists of the following two parts.

- **USB Device Access Control Module.** The USB device access control module uses existing DAC, MAC, and software integrity checks in the host OS to control access to USB devices from software applications.
- **Secure USB Bridge.** The secure USB bridge, which establishes a shared secret with the host computer and communicates with the host OS to provide message confidentiality and integrity over the communication channel. The host OS uses a *secure USB bridge pairer* software to establish the shared secret with the bridge, and communicates with the bridge securely using a *secure USB relay* module.

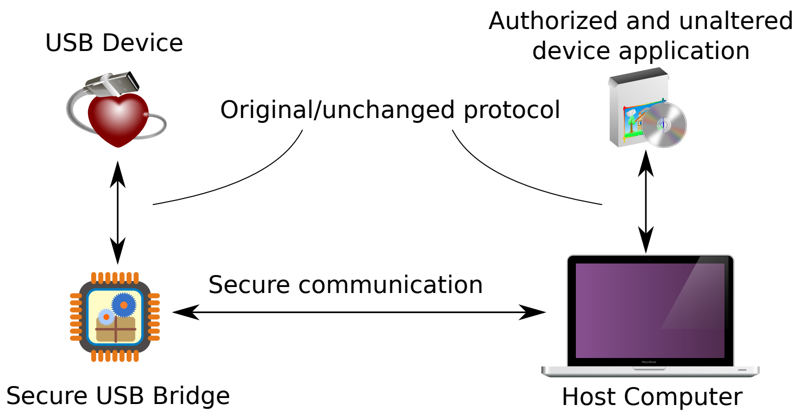

USB Device
Authorized and unaltered device application
Original/unchanged protocol
Secure communication
Secure USB Bridge
Host Computer

Figure 6.2: Illustration of the secured communication channel between the bridge and host computer.

To make use of the solution, the user now interacts with the device through the application in the following steps.

1. Optionally, the user installs the device driver and the device application using the secure USB software installer, which measures the various features of the device application such as the hash of the executable file.
2. The user attaches the bridge to the host computer.
3. The secure USB bridge pairer is executed to establish a shared secret between the bridge and relay through a key exchange protocol. After successful pairing, the bridge now waits for the device to be connected.
4. The user attaches the device to the bridge.
5. The bridge enumerates the device to obtain basic device information and exposes a USB instance containing the device information to the connected host computer.
6. The user starts the device application, which is granted access by the USB device access control module based on a series of access control checks and integrity verifications.
7. The user uses the device application to communicate with the device, during which the bridge and relay encapsulates communication between the USB subsystem and the bridge in a secure channel.

The resulting secured communication environment is illustrated in Figure 6.2.

We now describe how the USB device access control module (*control module*) and the secure USB bridge (*bridge*) are designed.

6.2.1 USB Device Access Control Module

To avert attacks from malicious software, we propose a solution to enforce access control policies on the host computer OS based on whitelisting. The solution stores information about programs which are authorized to access USB devices through an initial setup step, and verifies the program state during launch-time. This prevents unauthorized or altered software from accessing the USB device.

The existing discretionary access control (DAC) in the OS is insufficient for preventing malicious software from accessing USB devices, since an application access to system resources is granted/denied based solely on the access rights associated with the user. As a result, mandatory access control (MAC) has to be introduced to restrict access of applications to only a subset of all resources based on predefined policies. In Linux, implementations of MAC include as AppArmor [38], SELinux [39], TOMOYO Linux [49], and SMACK [50], which are all part of the Linux kernel. Linux MAC implementations make use of the Linux Security Modules [51] framework to restrict resource access from applications based on access policy rules, either predefined during installation or added by administrators.

In addition to MAC, the USB device access control module also verifies the integrity of applications before they access resources in the OS. Various approaches for software whitelisting and fail-stop mechanisms have been widely investigated and designed [25, 26, 27, 28]. These solutions allow the execution of an application only if the integrity of its executable is verified.

To provide a tighter access control, the control module combines the well-established use of DAC, MAC and integrity fail-stop mechanisms. A *Secure USB Software Installer* registers an application to a USB device which it is allowed to access and measures the application files and system environment for future integrity verification. As an example, measurement of files and system environment may include hashes of files or certificates. The registration and application information is stored in an access control database which is only accessible by the kernel. Whenever an application accesses a USB device, the control module fetches the its registered information in the database and allows access only if all of the following conditions are met.

- The user account is granted access to the USB device.
- The device application is granted access to the USB device.
- The integrity of the device application is verified.

Under this mechanism, it is therefore important to identify outstanding features of device software such that a comprehensive measurement of its legitimate state can be obtained for integrity verification. This is orthogonal to this thesis and is therefore considered as future work.

6.2.2 Secure USB Bridge

To achieve communication security, we encapsulate the original communication protocols between the device and application in a secure channel. Specifically, we establish a secure channel in their communication to provide confidentiality and integrity. In addition, the secure communication channel does not modify the device and application, according to Requirement 3. Therefore, we introduce the secure USB bridge solution which actually consists of the following entities.

- The Secure USB Bridge, referred to as the *bridge*, which is connected between the USB device and the host computer
- A software module called the Secure USB Relay, later referred to as the *relay*, on the host computer

Since many USB devices, such as mass storage or digital imaging devices, communicate with the host computer with high data rates in the order of a few megabytes per second, we adopt symmetric encryption to secure data communication. This requires a shared secret known only between the bridge and relay, and we therefore introduce an initialization phase for secret establishment.

A software module called the Secure USB Bridge Pairer, later referred to as the *pairer*, is used to establish a shared secret between the relay and the bridge using key exchange protocols, such as the Diffie-Hellman key exchange [52]. To prevent adversaries from disrupting the protocol (e.g., man-in-the-middle attacks), messages exchanged between the bridge and relay must be verified for their authenticity. This can be achieved, for example, using digital signatures based on public key cryptography or password-authenticated key exchange protocols. The sequence diagram for the pairing process is shown in Figure 6.3.

After a shared secret is established between the bridge and the relay, the actual USB communication between the device and application is enabled. The same key is used for secure communication of any devices that may be attached to the bridge after the key exchange. When the device is sending data, the bridge encrypts and sends it to the relay. Upon reception, the relay on the host computer decrypts the ciphertext, verifies its integrity and forwards the resulting plaintext to the device application. Similarly, when the device application sends data to the device, the relay encrypts the data and sends it to the bridge. The bridge decrypts the ciphertext, verifies its integrity, and forwards it to the USB device in its original plaintext format.

The bridge should be directly connected to the device to alleviate concerns of malicious intermediate hardware such as USB sniffers. A sequence diagram illustrating the communication between the device and application after the adding the bridge and the relay is shown in Figure 6.4.

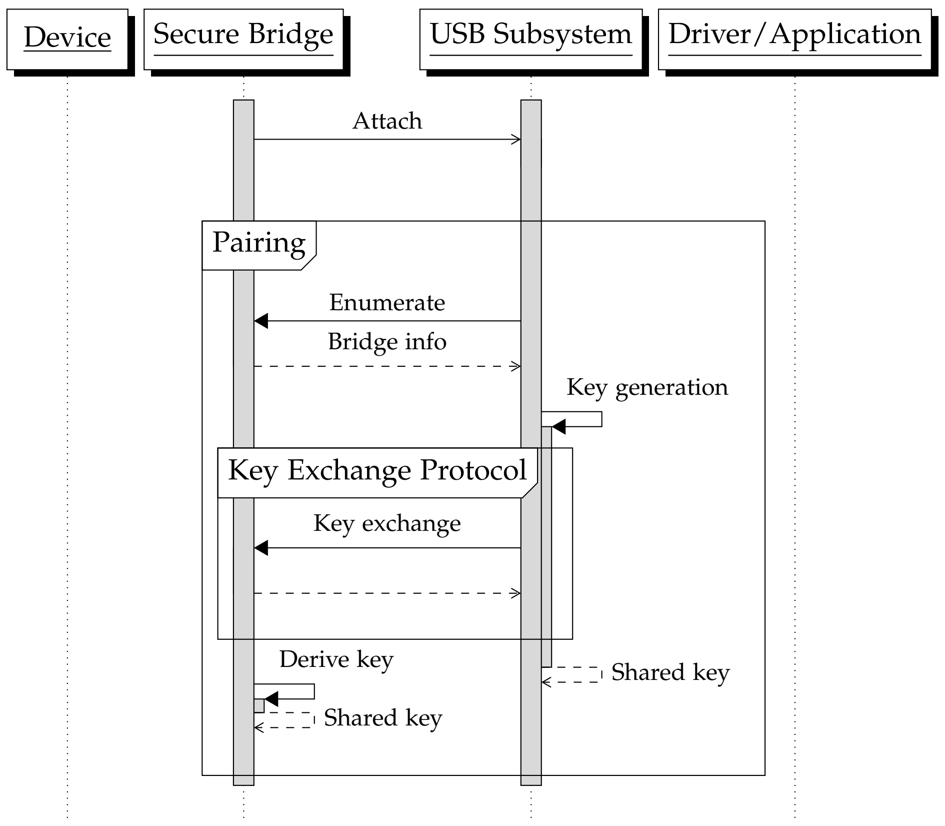


Figure 6.3: Pairing process between the enhanced host OS and the secure bridge. After pairing, a shared secret is generated for efficient secure communication.

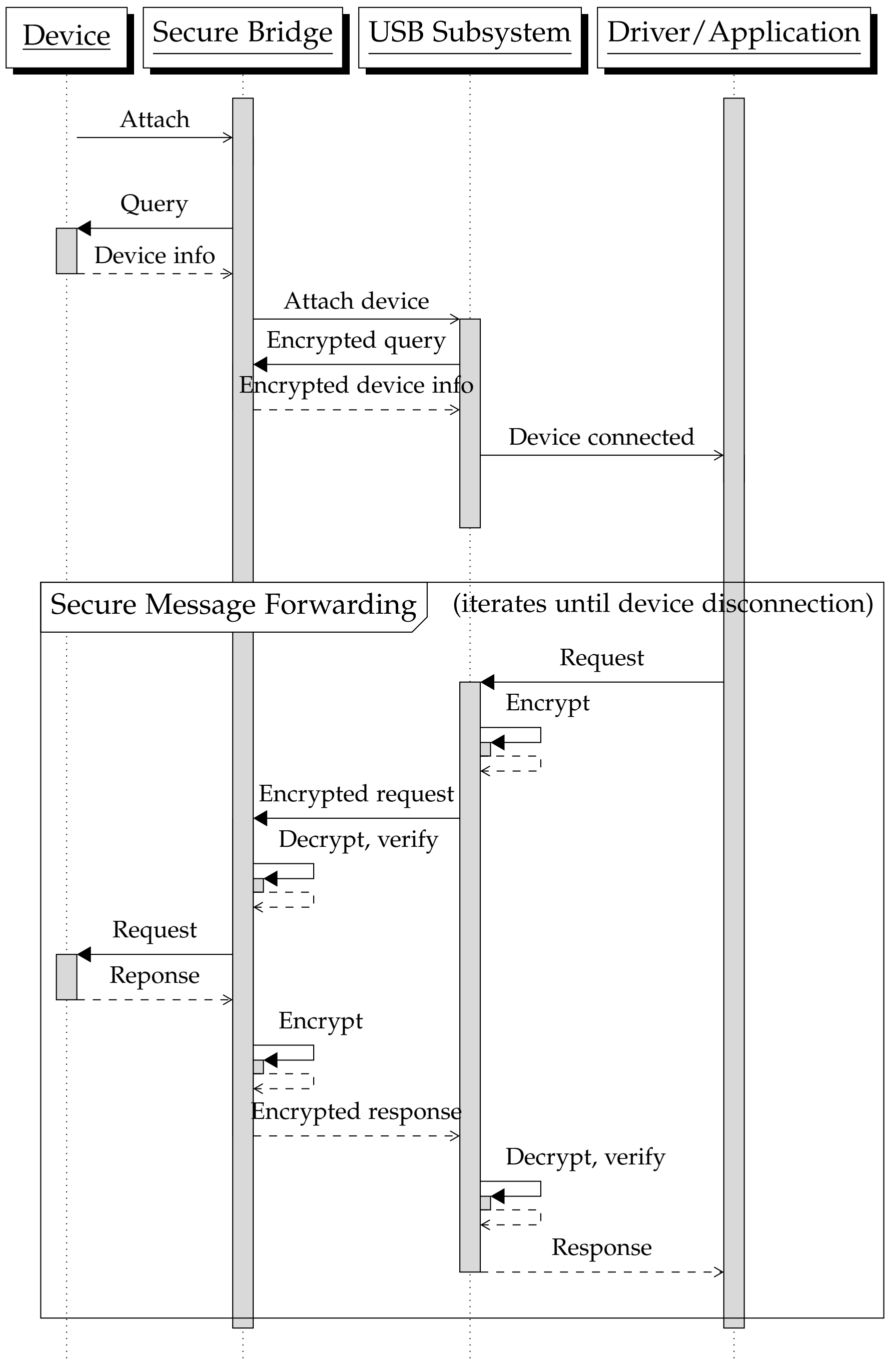


Figure 6.4: Secure communication between the device and an authorized driver/application. Here the security of messages are implied by the shared secret established between the secure bridge and the host OS during the pairing process.

6.3 Security Analysis of Our Design

Our USB device access control module aims at preventing the software adversary, discussed in 5.3, from achieving any of the the attacks defined in 5.1. The secure USB software installer measures the device application during installation. The control module makes use of existing solutions for DAC, MAC, and software integrity verification to prevent unauthorized access of the device from an unregistered or altered device application. These security features fulfil Requirement 1.

We now discuss the secure USB bridge design to aim at fulfilling Requirement 2. The secure key exchange protocol ensures that no entity may derive the shared key without the knowledge of the secret on either the bridge or the host. Additionally, the shared key is secure against man-in-the-middle attacks since the host and bridge verify the authenticity of the messages which they receive during the pairing process. As a result, the encrypted communication channel protects against passive adversaries from obtaining meaningful information by eavesdropping. The integrity checks of decrypted messages performed by the bridge and relay further prevents insertion and modification attacks from an active adversary. Impersonation attacks being an instance of insertion attacks, are also prevented.

However, our solution does not resolve insertion or impersonation attacks from the device side of the bridge, which may require techniques such as device identification. Minimally, we need to be able to write an identifier to the device, which is not supported by many USB devices and therefore remains a limitation of our solution.

For deletion attacks, the adversary in the secure communication channel may still be able to deny the transmission of encrypted messages. For example, a malicious hub may simply refuse to route the encrypted messages to the bridge or the host. Our solution therefore does not address denial-of-service attacks by the adversary, though such attacks would result in the disruption of communication which is often noticeable by the user.

Chapter 7

Implementation and Evaluation

Given that there has already been significant research on software security, as discussed in Chapter 3, we focus on realizing the design of the secure USB bridge. Realizing such a channel in a generic way is a challenge because there are several types of USB devices each with their own speed and performance requirements. Moreover, even devices of the same class differ with one another in the way they define interfaces and endpoints.

We implemented a prototype implementation of the secure USB bridge using the BeagleBone [53] development board with an ARM Cortex-A8 [54] microprocessor produced by Texas Instruments [55]. The board consists of a USB controller supporting Direct Memory Access (DMA) and exposes a USB host-side port to connect to USB devices as well as a USB device-side port which connects to a USB host such as a computer. By connecting the host-side port to the USB device and the device-side port to the computer running the device application, the development board can act as a proxy for the USB device which processes the communicated data. We implemented a bare-metal software module using the StarterWare library [56] (i.e., it does not run on an OS) on the development board to forward data between the device and the host computer.

For the host computer, we use a Lenovo ThinkPad T420 laptop running Ubuntu 11.10, with specifications listed in Table 7.1. We implement a kernel module on the host OS to act as the relay which communicates with the bridge.

For cryptographic operations during implementation, we used PolarSSL [57] for its compatibility with bare-metal platforms.

The following sections discuss how key exchange and message encryption are implemented.

Lenovo ThinkPad T420	
CPU	Intel Core i7-2620M @ 2.7GHz
Memory	4GB
OS	Ubuntu Linux 11.10
Kernel version	3.0.0.26-generic-pae

Table 7.1: The host computer specifications

7.1 Key Exchange

For the key exchange phase, we adopt the Diffie-Hellman key exchange protocol with RSA signatures (DHE-RSA [58]) for the messages transferred between two parties. The bridge and the relay each generates a 1024-bit RSA keypair. The bridge is given the public key of the relay; similarly, the relay is given with the public key of the bridge. This allows the bridge and the relay to verify the authenticity of the messages exchanged during the Diffie-Hellman key exchange process.

The bridge, upon attachment to the host computer, first exposes itself as a USB device with one configuration containing a bridge handshake interface. The bridge handshake interface consists of one endpoint for `IN` type bulk transfers and one endpoint for `OUT` type bulk transfers. Since USB devices are passive, the bridge waits for the host computer to initiate the pairing process.

On the host side, we implement a user-space program which initiates the pairing process. This program is executed with root privileges to prevent access from software adversaries with as defined in 5.3. The program proceeds as follows.

1. Generating Diffie-Hellman parameters.
    - The modulus prime number p
    - The generator prime number g
    - A randomly chosen secret integer a
    - The key contribution $A = g^a \mod p$
2. Obtain hash h_A of $p|g|A$ using SHA1. [1]
3. Sign h_A with its private key to obtain s_A.
4. Send $p|g|A|s_A$ to the USB bridge handshake interface OUT endpoint.

[1]The | symbol is the concatenation operation.

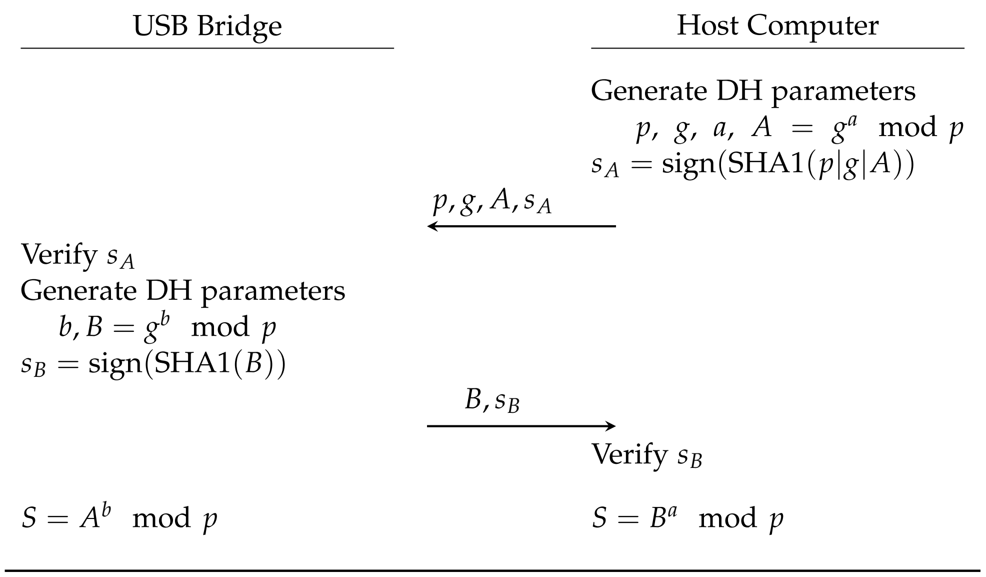


Figure 7.1: An example sequence of successful pairing under our implemented DHE-RSA protocol for key exchange between the bridge and the host computer.

5. Obtain the bridge key contribution B' and the signature of its hash s'_B from its response.
6. If the authenticity of the Diffie-Hellman contribution passes verification, derive the shared secret $S_a = (B')^a \mod p$. Otherwise report failure and terminate execution.

The bridge passively waits for the data to be sent into its `OUT` endpoint. It proceeds as follows.

1. Wait until data is received through the OUT endpoint.
2. Unpack the message to obtain p', g', A', and s'_A.
3. Obtain hash h'_A of $p'|g'|A'$ using SHA1.
4. If the signature s'_A is verified with the public key of the host, proceed. Otherwise, the message is discarded and the bridge returns to Step 1.
5. Generate its Diffie-Hellman parameters.
    - A randomly chosen secret integer b.
    - The key contribution $B = (g')^b \mod p'$.
    - Obtain hash h_B of B using SHA1.

- Sign h_B with its private key to obtain s_B.
- Place $B|s_B$ in the buffer of the USB bridge handshake interface IN endpoint for the host to read.
- Derive the shared secret $S_b = (A')^b \mod p'$.

If a key exchange session is successful, then

$$\begin{aligned} p' &= p \\ g' &= g \\ A' &= A = g^a \\ B' &= B = g^b \end{aligned}$$

which implies an identical shared secret key is established.

$$\begin{aligned} S_b &= (A')^b \mod p' \\ &= g^{ab'} \mod p \\ &= (B')^a \mod p \\ &= S_a \end{aligned}$$

As an example, a successful execution of the key exchange protocol during the bridge-relay pairing process is illustrated in Figure 7.1. Failures in data interpretation or signature verification would reset the protocol state of both the bridge and relay.

7.2 Message Encryption

Based on Figure 6.4, after establishment of a shared secret, the bridge waits for a USB device to connect to its host-side port. When a device is connected, the bridge enumerates the device and obtains a copy of its descriptors. The bridge then activates its device instance to the connected host computer, providing (upon request) the copy of descriptors appended with an additional bridge interface to each configuration. The appended bridge interface consists of a set of bulk `IN/OUT` endpoints, similar to that during bridge-relay key-exchange, and is used for encrypted message transmission and reception. As a result, after the host computer enumerates the new device instance exposed by the bridge, the actual USB device information is passed to the host. This enables successful detection of the original device and execution of its driver or application. To avoid inadvertent communication between host USB drivers and the bridge interface, we specify the USB class of this interface as Vendor Specific (`0xFF` in Table 2.1).

As discussed in Section 2.2, software which communicates with the USB device ultimately invokes the `usb_submit_urb` kernel function in the `usbcore` module to send or receive messages in URB data structures. In our solution, the requests for USB data transfer are detected by hooking calls to `usb_submit_urb` using `jprobes` [59]. The `jprobes` hook function manipulates the data buffer, endpoint information, transmission type in the URB structure to redirect traffic to communicate with the bridge interface. This reduces the risk of the adversary distinguishing the traffic data based on its endpoint addresses and transfer types.

Our implementation currently redirects control transfers after enumeration, and any `IN` or `OUT` requests sent to device endpoints. Since the request to any endpoint of the USB device is routed to the bridge interface, the redirection logic appends the destination information and transfer type of the original URB as header information to the data sent to the bridge. The following describes a simple message forwarding protocol we implement on the `jprobes` relay and the bridge for each type of transfer. The format of data sent to the bridge for each type of transfer are shown in Figure 7.2.

When encrypting the message, we apply either RC4 or AES (in CTR mode) encryption to the plaintext data, which consist of essential header information prepended to the original USB message and a SHA1-128 hash of the entire message. The decryption process consists of decrypting the ciphertext and verifying its hash. Messages with invalid hashes after decryption are ignored.

- **Control** transfers. Recall in Section 2.1.4 that a control transfer consists of a setup packet and an associated buffer field containing data to be sent to or received from the device. When the relay detects the control transfer, it proceeds as follows.

  1. Create a new data buffer, containing the concatenation of the setup packet and the data buffer, prepended with header information.

  2. Encrypt the data buffer on the host and send it to the bridge `OUT` endpoint.

  3. Alter the current URB to request data from the bridge `IN` endpoint.

  4. Decrypt received data on the host and place it into the original URB data buffer

  5. Invoke the URB completion handler originally specified by the driver.

The bridge, processes these control transfers as follows.

1. Decrypt the relay request and issue an actual control transfer request to the device.

2. Encrypt the data returned by the device and place them in the buffer of the bridge `OUT` endpoint.

- **IN** token transfers. In the case with `IN` token transfers, the relay should also indicate to the bridge which endpoint of the device is addressed by the URB. The relay takes the steps similar to handling control transfers, as listed below.

  1. Create a new data buffer which consists of the header information, appended by the size of the data buffer the URB is requesting (in bytes).

  2. Encrypt the data buffer on the host and sent it to the bridge `OUT` endpoint.

  3. Alter the current URB to request data from the bridge `IN` endpoint.

  4. Decrypt received data on the host and place it into the original URB data buffer.

  5. Invoke the URB completion handler originally specified by the driver.

  The bridge handles `IN` requests accordingly.

  1. Decrypt the relay request to obtain the device endpoint and issue an IN request on the specified `IN` endpoint of the device.

  2. Encrypt the data returned by the device and place them in the buffer of the bridge `OUT` endpoint.

- **OUT** token transfers. Sending data to devices involves a simpler step for the relay since no data is returned from the device. The relay takes the following steps.

  1. Create a new data buffer which consists of the header information, appended by the size of the data buffer, and then the data to be sent to the device.

  2. Encrypt the new data buffer and send it to the bridge `OUT` endpoint.

'C'	ep	setup packet	data (optional)

(a) Control transfers consisting of a `SETUP` request

'I'	ep	t	length of requested data

(b) `IN` request

'O'	ep	length of data	data

(c) `OUT` request

Figure 7.2: Packaged data sent by the relay to the bridge for each type of URBs intercepted. *ep* represents the designated endpoint address. In the case with `IN` requests, an additional parameter *t*, representing the type of transfer (bulk, isochronous, interrupt) is also specified as additional information for the bridge.

3. After transmission is confirmed from the host controller driver, invoke the URB completion handler originally specified by the driver.

The bridge proceeds as follows.

1. Decrypt the relay request and send the data to the endpoint specified by the header information.

With each URB request generated/modified, the relay assigns/substitutes custom completion handler functions which perform clean-up and trigger the original driver completion function. When handling `IN` request completions, the relay places the decrypted original data from the device into the original buffer allocated by the driver.

To prevent disrupting USB devices which are not connected to the host via the bridge, the relay module only intercepts URB requests which are sent to a predefined list of devices which are known to be connected through the bridge.

7.3 Preliminary Evaluation

Given the implementation of the prototype bridge and relay, we provide a preliminary evaluation of its performance to observe the communication overhead it introduces. We ran benchmarks using two devices: a USB bulk device which contains a source and a sink for data transmission, and a USB flash drive.

Metrics

During communication we measure the time it takes for the software application to transfer data. In addition, time measurements of data transmissions are converted to throughput whenever applicable, which is calculated as:

$$\text{throughput} = \frac{\text{data size}}{\text{transfer time}}$$

We also measure the time the relay spends in processing `usb_submit_urb` requests, which represents the time spent on cryptographic operations, receiving the response, and necessary data processing. To get an insight into the communication overhead caused by the bridge, we measure time on the bridge when it processes a request from the relay. This involves host-side `I/O`, device-side `I/O` and cryptographic operations.

Measurement Setup

We measured the performance of our implementation using the above mentioned metrics in the following four scenarios.

- Original connection without bridge or relay
- Connection through the bridge and relay with no encryption
- Connection through the bridge and relay with SHA1-RC4
- Connection through the bridge and relay with SHA1-AES

In each scenario, we tested our implementation using the bulk device (refer to Section 7.3.1) and flash drive (refer to Section 7.3.2). Additionally, we enabled data and instruction cache on the BeagleBone, as well as its DMA functionalities when communicating with the device. However, the BeagleBone DMA features between itself and the host computer is not enabled due to unstable DMA implementation in StarterWare [60, 61, 62].

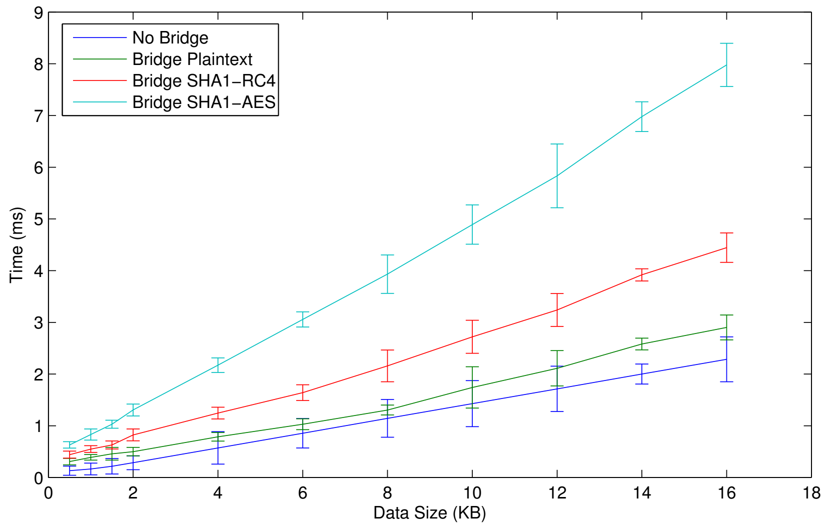
7.3.1 Bulk Device

We implemented a USB bulk device consisting of an interface with two endpoints: one `OUT` endpoint as a sink which receives any data sent from the host, and one `IN` endpoint as a source which continuously sends bytes of zeros to the host upon request. The device is implemented as a user-space USB gadget under the Linux gadget framework [63] and runs on a Freescale i.MX53 Quick Start Board [64].

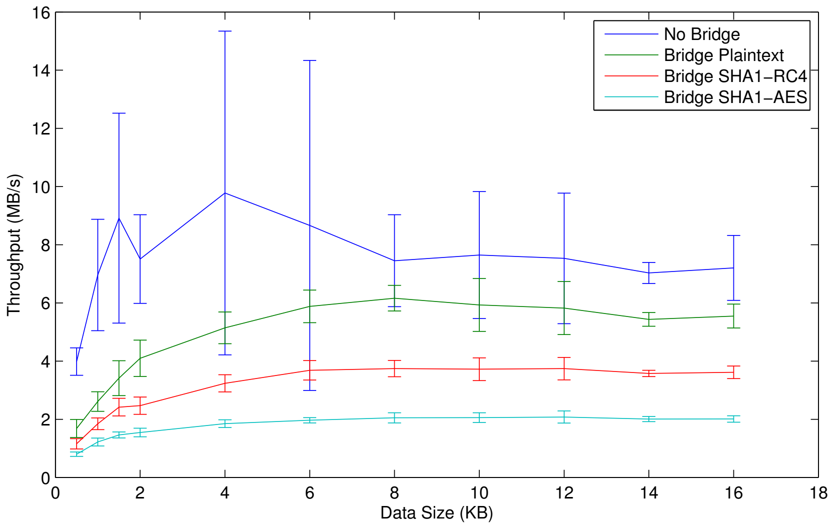
From the host side, we wrote a simple program which reads or writes data to or from the bulk device using `libusb` [11] and measures the transfer time. For our experiments, we have conducted 1000 test iterations for each individual setting.

Figures 7.3 and 7.4 shows the performance of read and write operations for various data sizes. In both cases we can observe that throughput saturates for data transfer sizes larger than 8KB. For example, when the bridge is just forwarding plaintext and has device-side DMA and cache enabled, the saturated throughput for the reading and writing is 3.8MB/s and 2.5MB/s respectively. In addition, the throughput decreases with the use of encryption algorithms (RC4 and AES), which is consistent with literature [65]. More specifically, throughput is highest for direct connection, followed by plaintext message forwarding through the bridge and forwarding with using SHA1-RC4 and SHA1-AES encryption. For example, in the case with SHA1-RC4 and SHA1-AES encryption, the throughput for read operations is about half and one fourth of that with direct connection, respectively. In the case with write operations, the drop in performance appears more drastic, due to the asymmetric performance of the bridge when communicating with the host computer. Specifically, the bridge does not use DMA to transfer data with the host, thereby relying on the CPU and its software code to handle data transmissions; however, the code used to transfer and receive data are implemented differently and therefore produces different performance. Figures 7.5 through 7.7 shows the time measured on the bridge for reading and writing 16KB of data from and to the bulk device, from which we observe the bridge taking significantly more time in receiving data from the host computer.

During the implementation of the solution, we noticed significant improvement of the bridge implementation on the BeagleBone after enabling data/instruction cache for the microprocessor and DMA for the USB controller when communicating with the device. Comparisons of various cache and DMA configurations are shown in Figures 7.8 and 7.9 for plaintext bridging, Figures 7.10 and 7.11 for SHA1-RC4 encrypted bridging, and finally Figures 7.12 and 7.13 for SHA1-AES encrypted bridging. In all benchmark results for write operations, the improvement after enabling BeagleBone USB DMA with the device can not be clearly observed due to the bottleneck introduced by the bridge when receiving data from the host computer without bridge-to-host DMA.

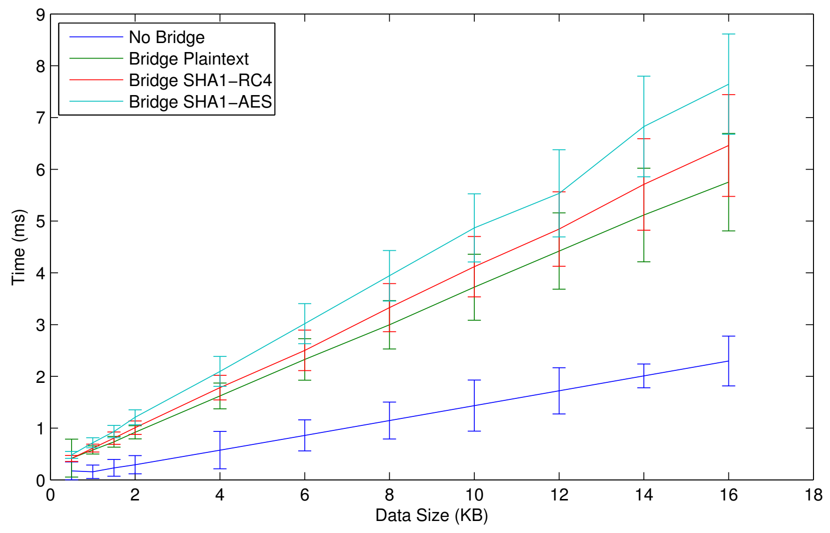


(a) Time

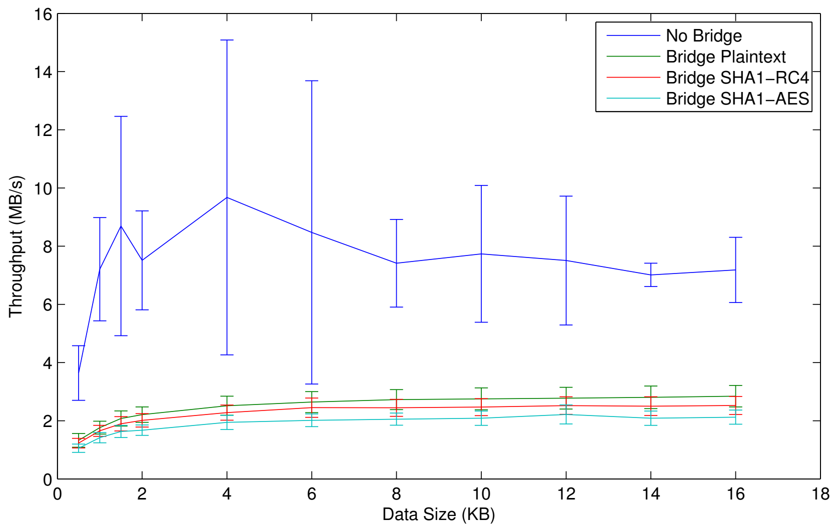


(b) Throughput

Figure 7.3: Comparison of perceived read performance for various bridge setups on the bulk device. Here we can observe the impact of our solution implementation, in which a RC4 and AES encryption schemes reduce the performance more drastically.

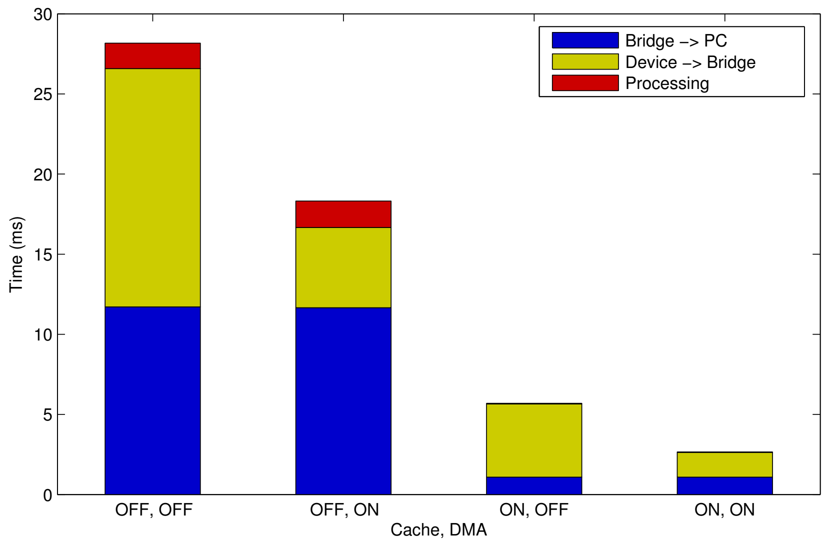


(a) Time

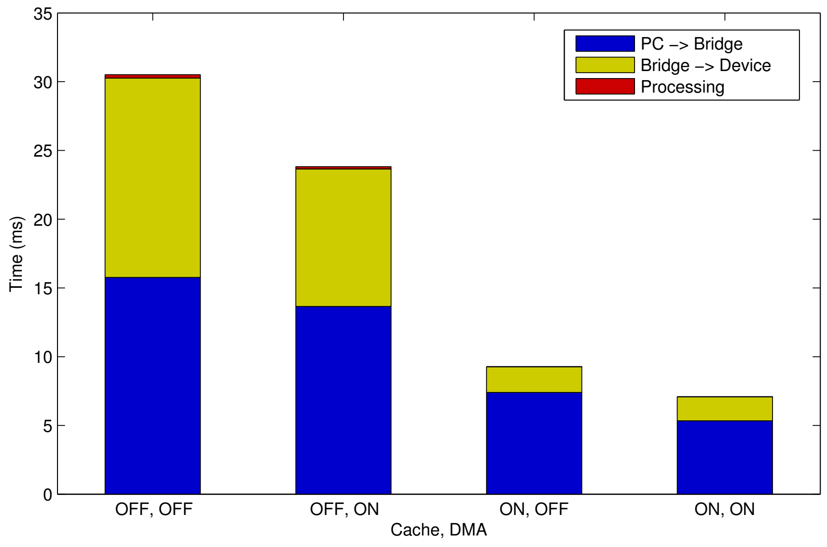


(b) Throughput

Figure 7.4: Comparison of perceived write performance for various bridge setups on the bulk device. Similar to Figure 7.3, the performance is affected by the encryption scheme chosen for secure communication. We also notice the overhead of receiving data from the host inherent in the StarterWare implementation, causing an overall decrease in performance.

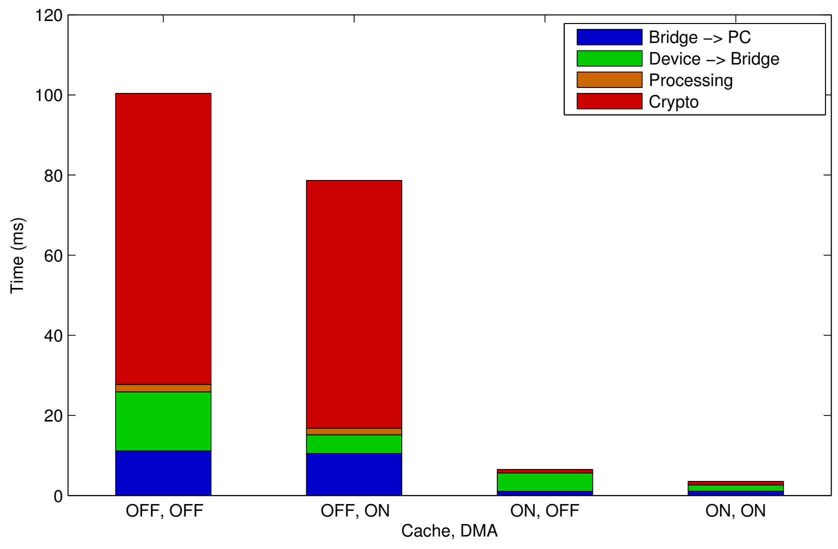


(a) Read 16KB

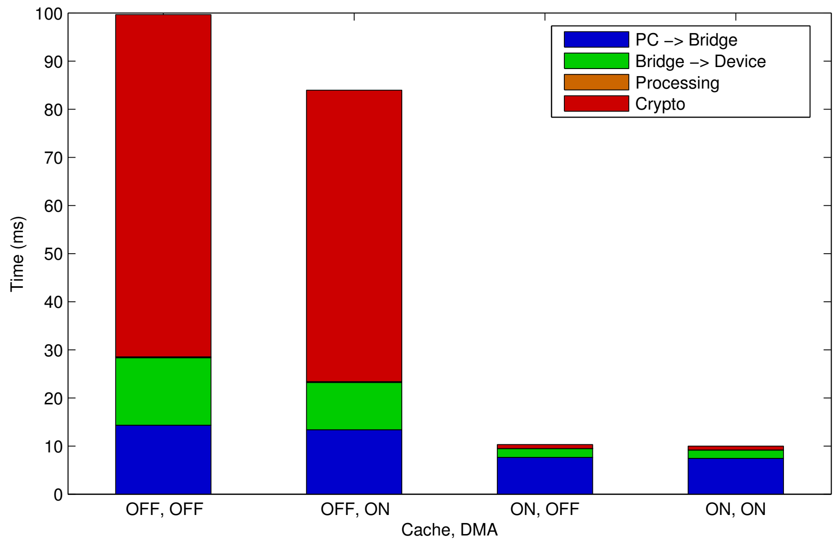


(b) Write 16KB

Figure 7.5: Comparison of bridge overhead for various settings of cache/DMA features on the bulk device. The performance is benchmarked with reading or writing 16KB of data without encryption. Enabling cache significantly reduces cryptographic operations while enabling DMA reduces communication overhead between the bridge and the device.

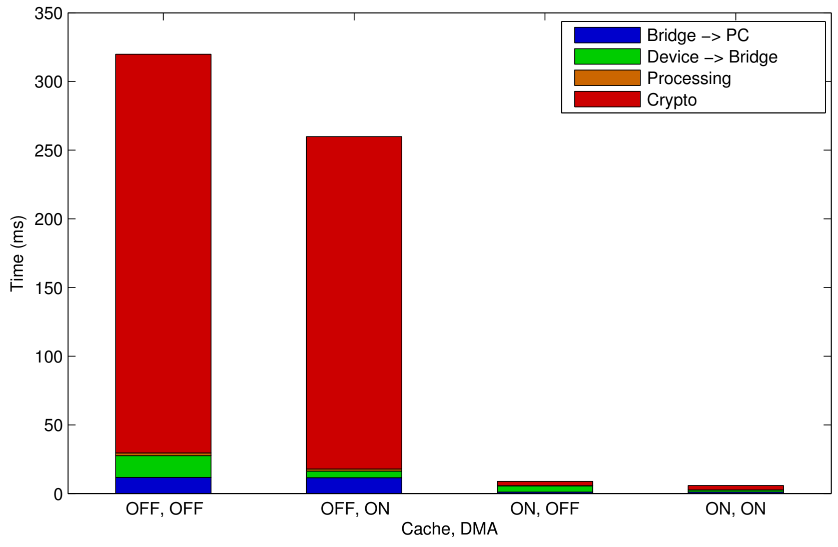


(a) Read 16KB

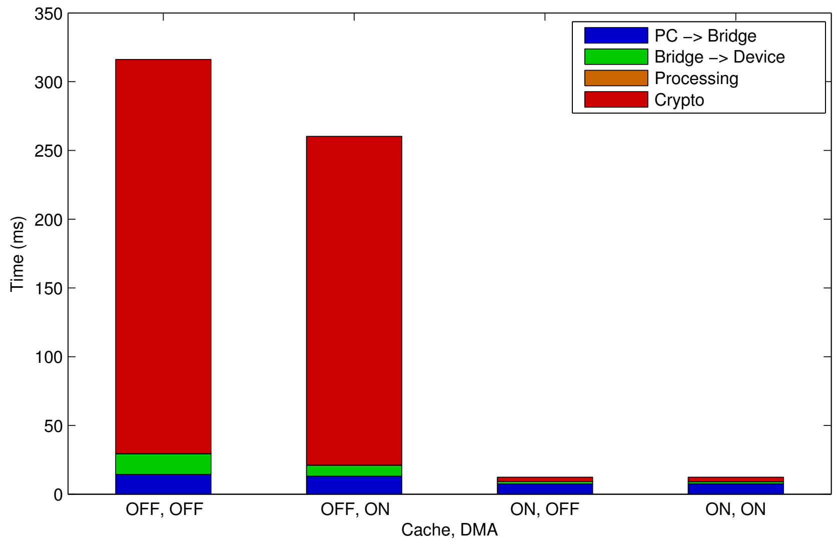


(b) Write 16KB

Figure 7.6: Comparison of bridge overhead for various settings of cache/DMA features on the bulk device. The performance is benchmarked with reading or writing 16KB of data with SHA1-RC4 encryption. Enabling cache significantly reduces cryptographic operations while enabling DMA reduces communication overhead between the bridge and the device.

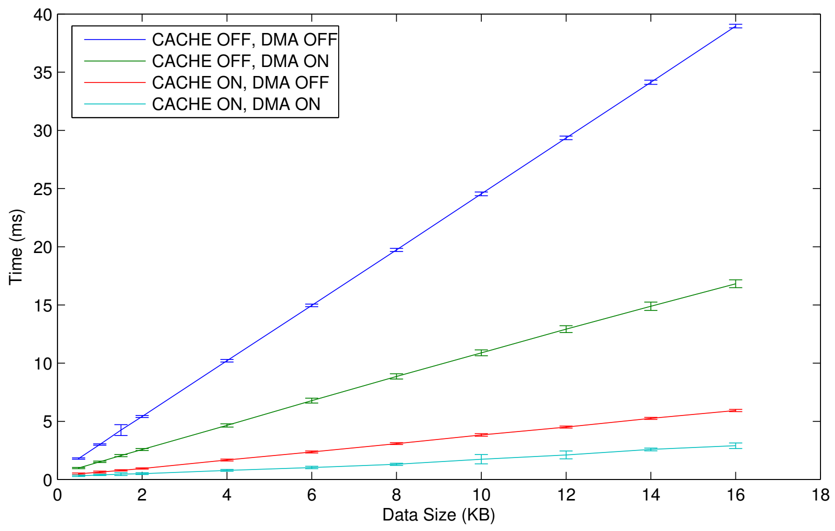


(a) Read 16KB

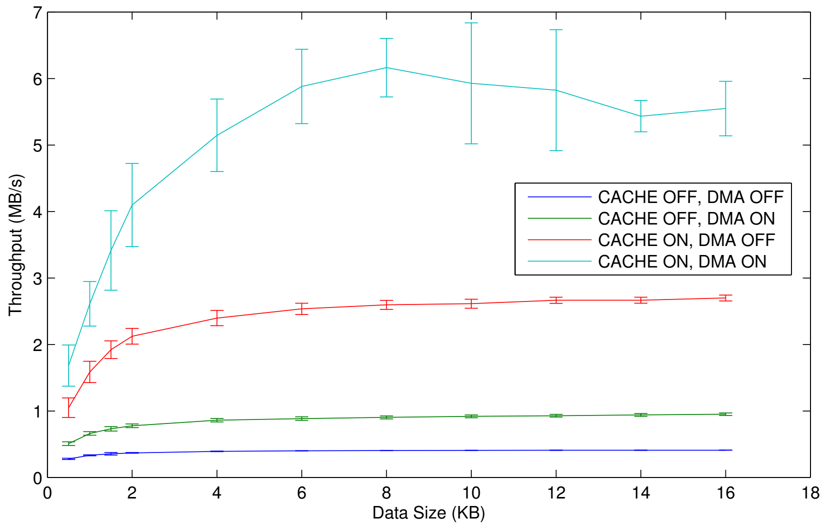


(b) Write 16KB

Figure 7.7: Comparison of bridge overhead for various settings of cache/DMA features on the bulk device. The performance is benchmarked with reading or writing 16KB of data with SHA1-AES encryption. Enabling cache significantly reduces cryptographic operations while enabling DMA reduces communication overhead between the bridge and the device.

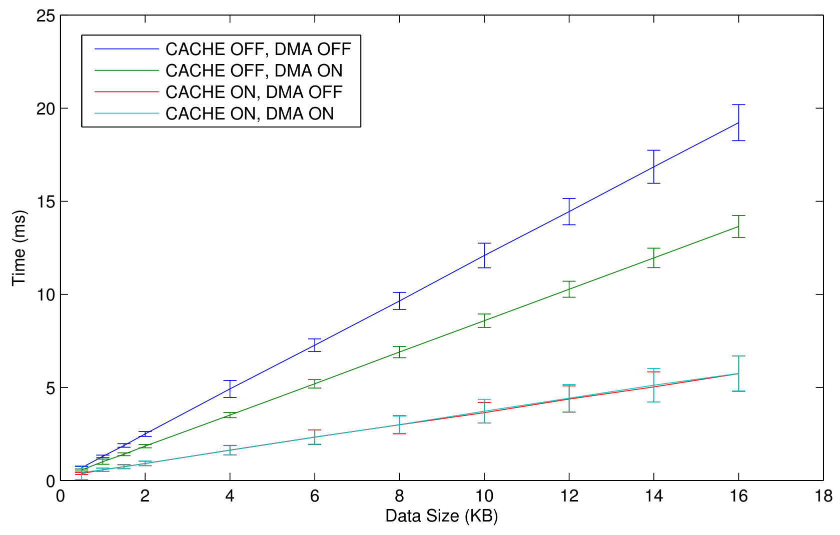


(a) Time

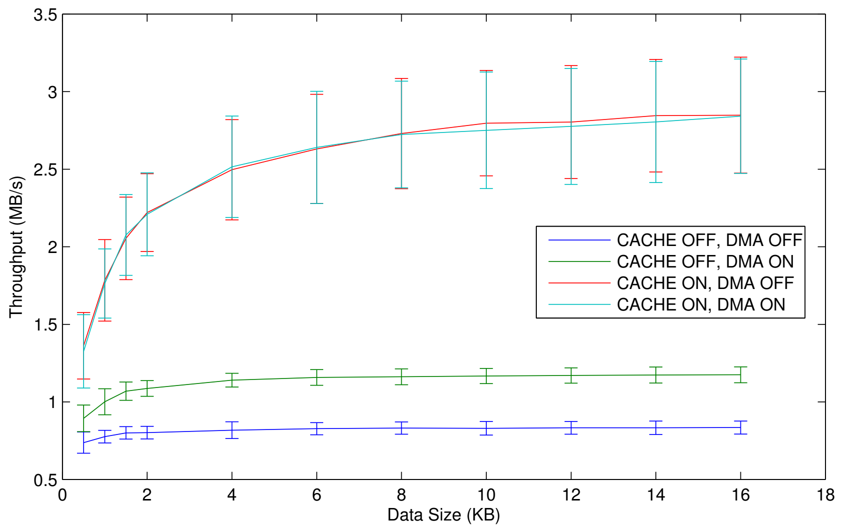


(b) Throughput

Figure 7.8: Comparison of perceived read performance for various cache/DMA features on the bulk device when the bridge is forwarding data without encryption. We can observe that the main factor which impacts performance is caching.

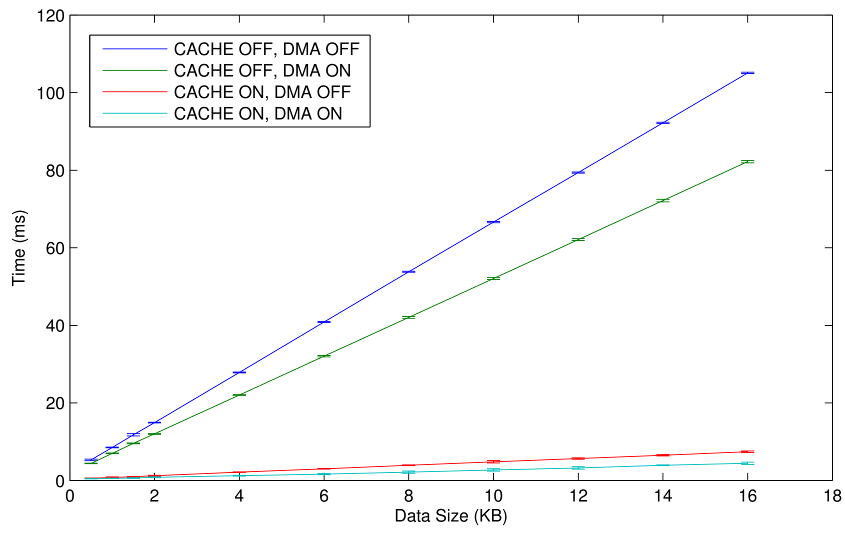


(a) Time

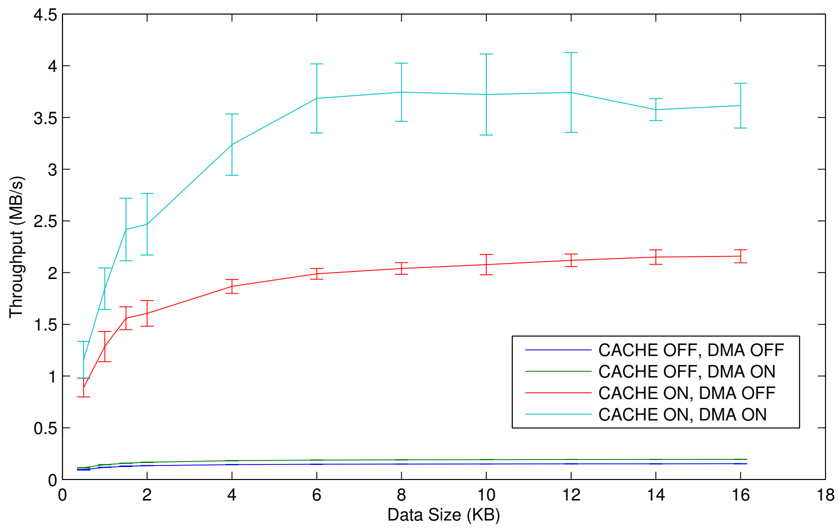


(b) Throughput

Figure 7.9: Comparison of perceived write performance for various cache/DMA features on the bulk device when the bridge is forwarding data without encryption. We can observe that the main factor which impacts performance is caching.

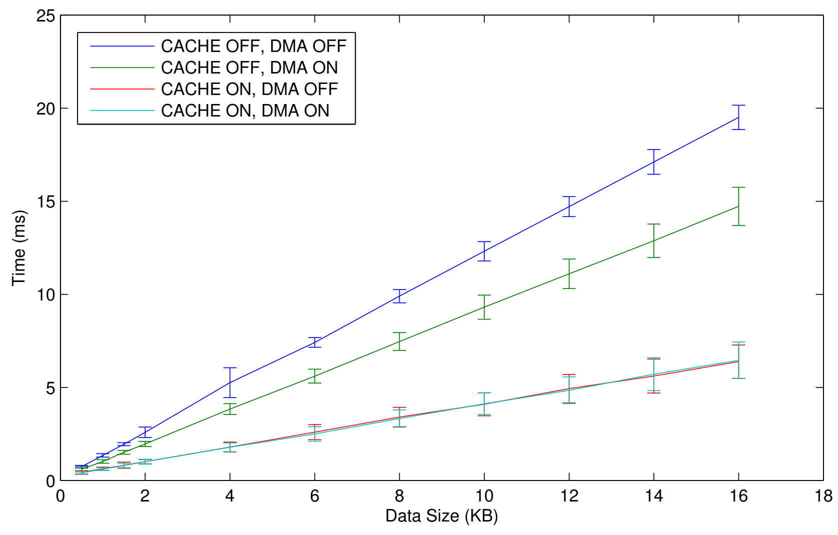


(a) Time

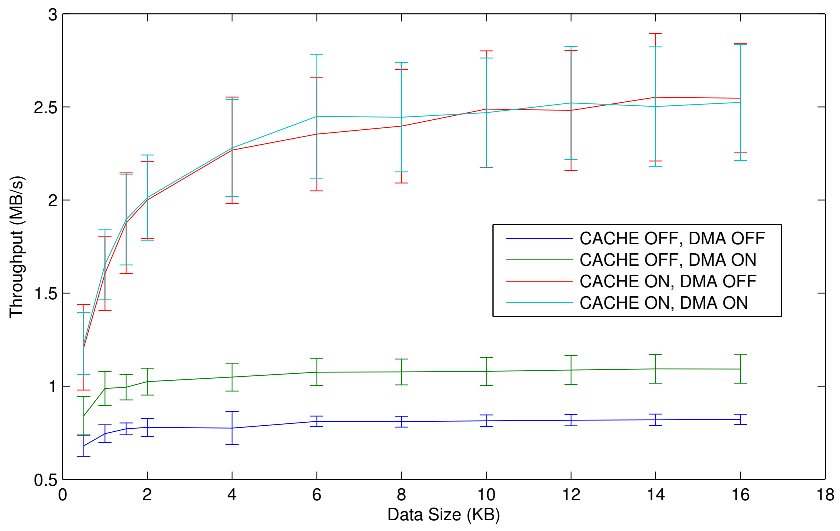


(b) Throughput

Figure 7.10: Comparison of perceived read performance for various cache/DMA features on the bulk device when the bridge is forwarding data with SHA1-RC4 encryption. We can observe that the main factor which impacts performance is caching.

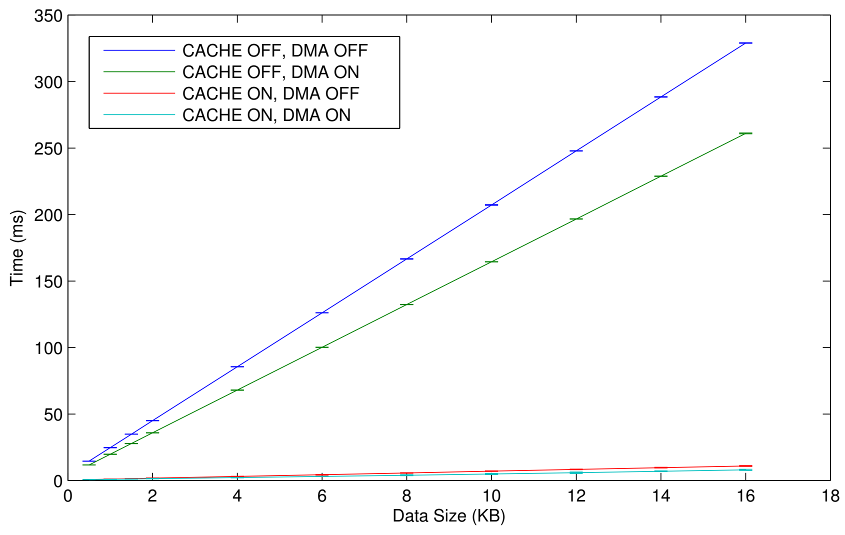


(a) Time

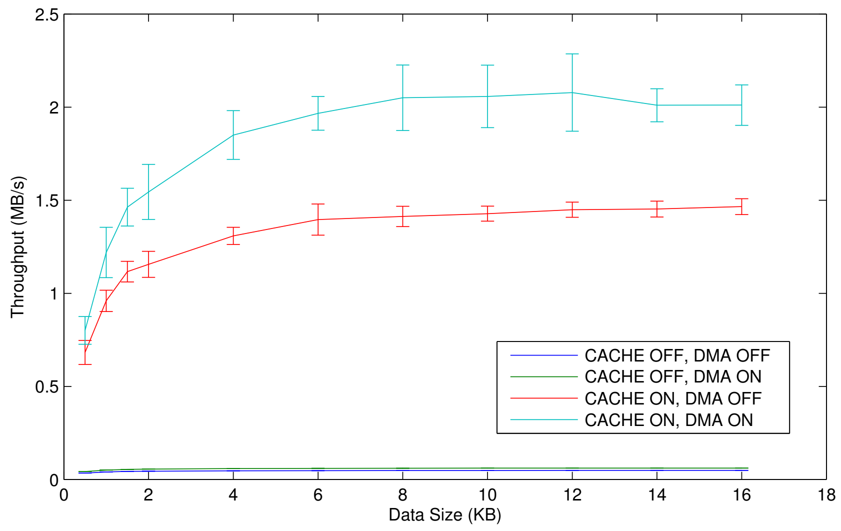


(b) Throughput

Figure 7.11: Comparison of perceived write performance for various cache/DMA features on the bulk device when the bridge is forwarding data with SHA1-RC4 encryption. We can observe that the main factor which impacts performance is caching.

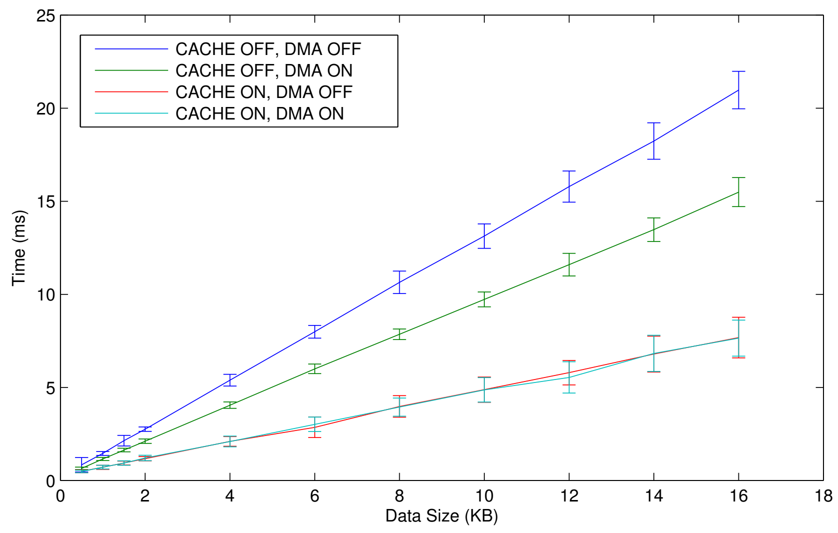


(a) Time

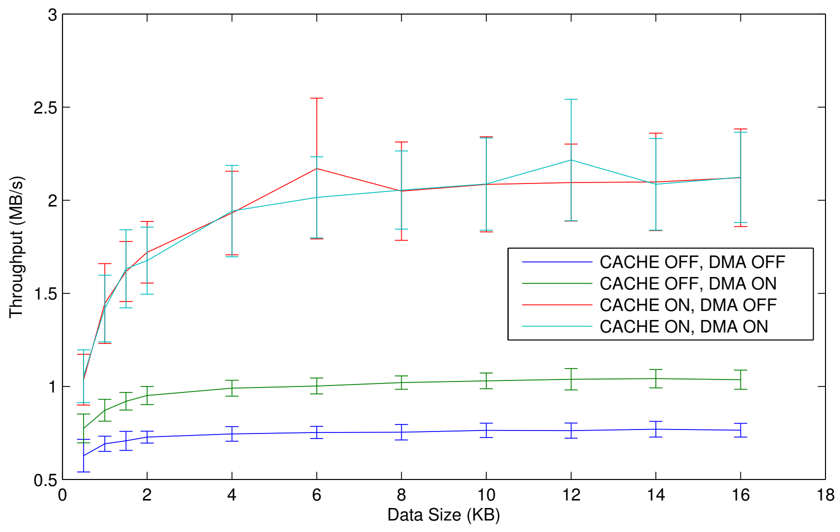


(b) Throughput

Figure 7.12: Comparison of perceived read performance for various cache/DMA features on the bulk device when the bridge is forwarding data with SHA1-AES encryption. We can observe that the main factor which impacts performance is caching.



(a) Time



(b) Throughput

Figure 7.13: Comparison of perceived write performance for various cache/DMA features on the bulk device when the bridge is forwarding data with SHA1-AES encryption. We can observe that the main factor which impacts performance is caching.

7.3.2 Flash Drive

While the bulk device gives us information about the raw data rate, most USB devices use additional protocols to transfer data. For example, USB mass storage devices adopt SCSI [66] commands to transfer files. In order to understand the performance of our solution in the context of these commodity products. In our evaluation, we have chosen to evaluate our solution using a USB flash drive for its widespread usage.

We benchmark our bridge and relay solution by copying files between a USB flash drive using `dd`. The following results are obtained with 100 test iterations for each individual setting. To overcome performance variations when transferring small files, we copied large files between the host and the flash drive to obtain stable measurements.

We benchmarked reading a 5MB file from the flash drive to the host computer. The average performance is shown in Figure 7.14. On average, our solution reduces the throughput from 24.66MB/s down to 4.32MB/s and 2.27MB/s for secured communication using SHA1-RC4 and SHA1-AES respectively.

For file writing operations, we performed tests with writing a 15MB file, and the results are shown in Figure 7.15. As observed during the benchmark with the bulk device, the write performance is affected more by the overhead of the bridge receiving data from the host than by various encryption algorithms. The average throughput drops from 6.64MB/s using direct connection down to 1.71MB/s and 1.28MB/s for SHA1-RC4 and SHA1-AES respectively.

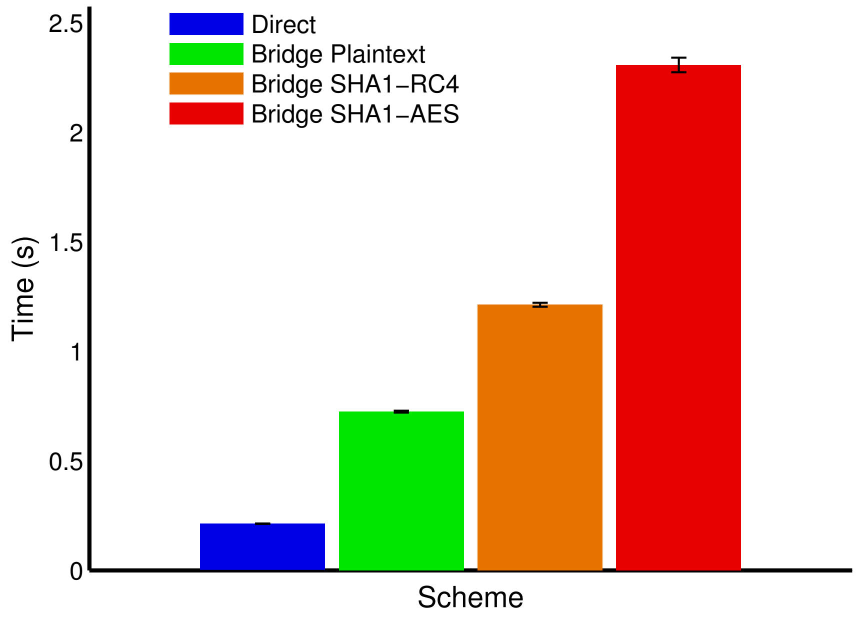
The asymmetric performance in data transmission between the bridge and the host is caused by our implementation of the bridge, and can be confirmed by profiling the host computer kernel using `sysprof` [67], obtained from reading and writing 1GB of data while the bridge is forwarding plaintext. Here we focus on the `usb_stor_control_thread` kernel thread used by the USB storage driver to transfer data, as shown in Figure 7.16. The kernel function spends more time on the function `usb_sg_wait`, which loops until the host controller driver reports completion of transfer, when writing to the device [68].

We also measure the time spent on the bridge for both reading and writing files, as shown in Figure 7.17. Additionally, a best performance potentially offered by the BeagleBone hardware is also estimated based on various benchmarks of various software efficiently making use of the underlying hardware features such as DMA and cryptographic acceleration [69, 70] as shown in Table 7.2. Figure 7.18 shows the time spent on kernel module acting as the relay. Here we clearly observe that most time is spent on data transfer. The increased time with SHA1-RC4 and SHA1-AES encryption al-

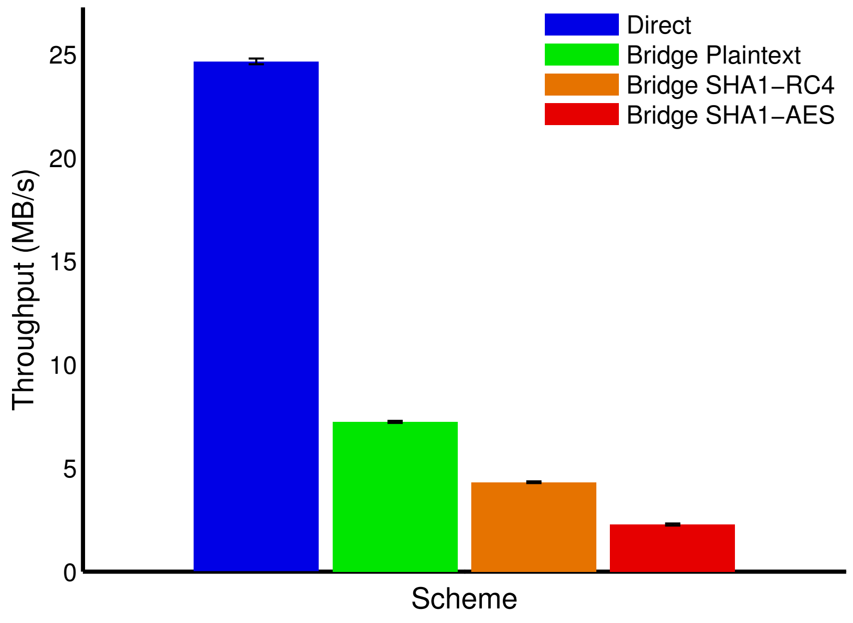
Reading USB device	22.35 MB/s
Writing USB device	19.56 MB/s
Read from USB host	16.77 MB/s
Write from USB host	15.15 MB/s
AES	25.35 MB/s
SHA1	54.54 MB/s

Table 7.2: Highest achievable speeds provided by the BeagleBone when optimized software libraries make use of hardware features such as DMA and cryptographic accelerations [69, 70].

gorithms is attributed to the time spent waiting for bridge operation. Based on these observations, the communication efficiency between the bridge and host could be improved with proper use of DMA on the BeagleBone baremetal software.

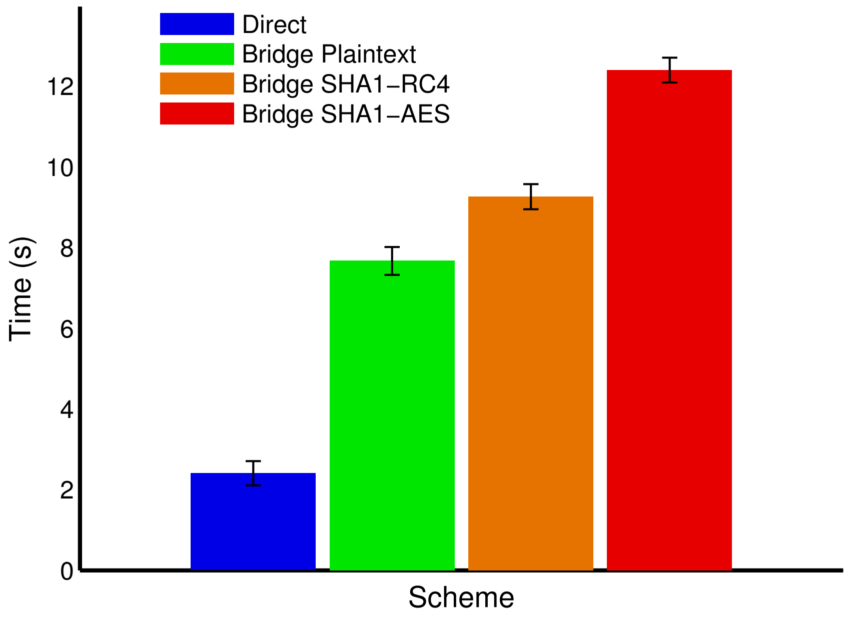


(a) Time

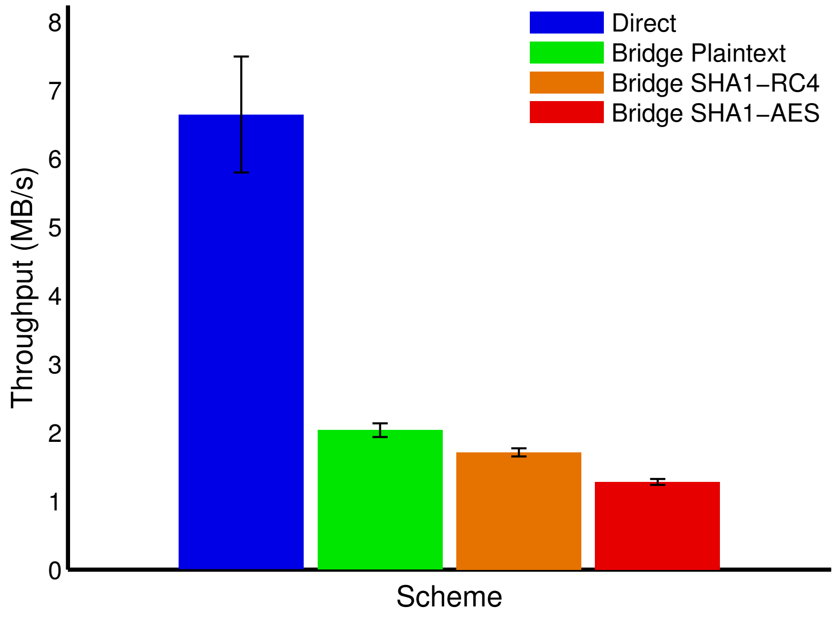


(b) Throughput

Figure 7.14: Comparison of perceived read performance for 5MB with various bridge setups on the flash drive.



(a) Time



(b) Throughput

Figure 7.15: Comparison of perceived write performance for 15MB with various bridge setups on the flash drive.

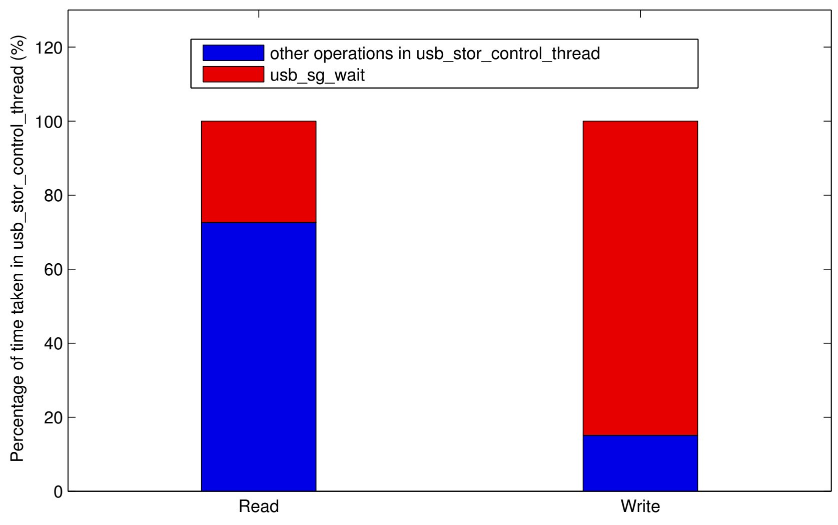
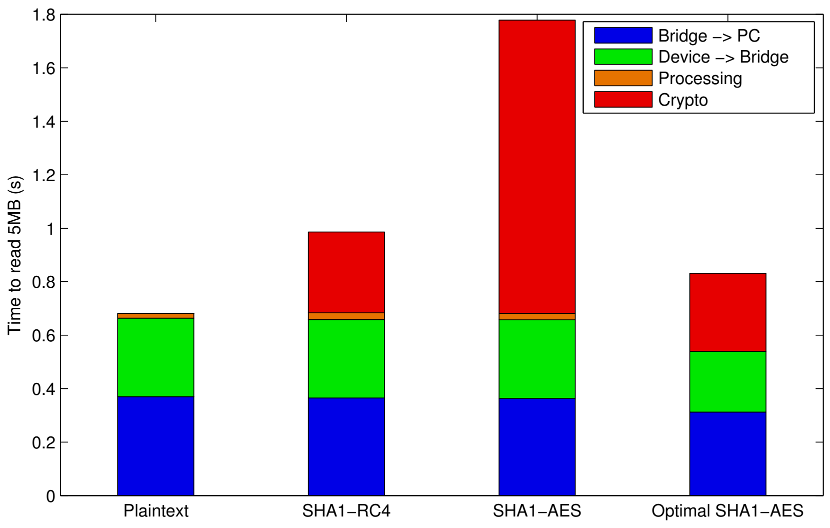
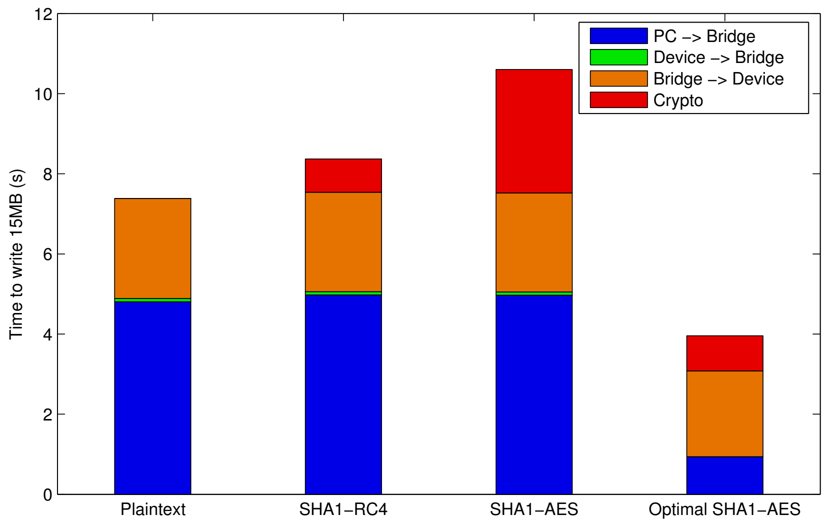


Figure 7.16: Kernel profiling results for the kernel function `usb_stor_control_thread` when the host computer reads from and writes to the USB flash drive through the bridge which forwards data without encryption. Waiting for the bridge to respond takes up more time for the host when it is writing data.

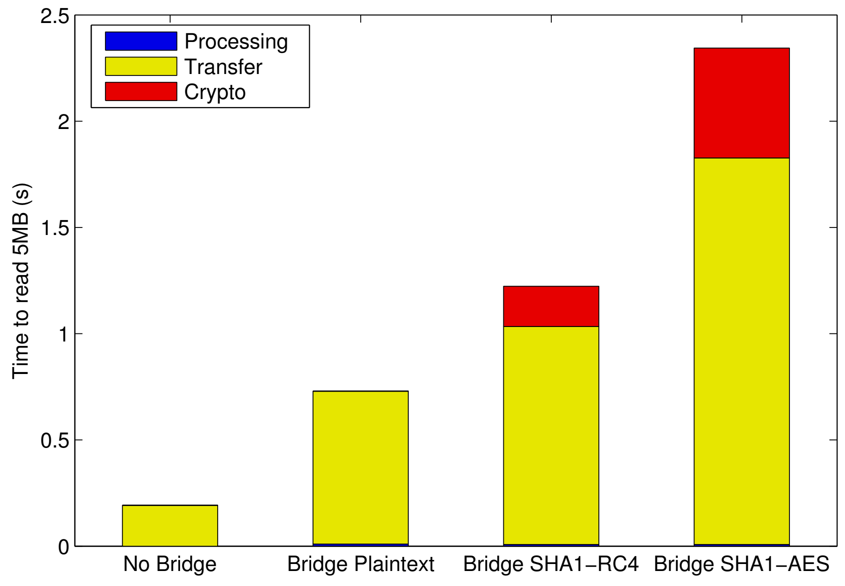


(a) Read 5MB

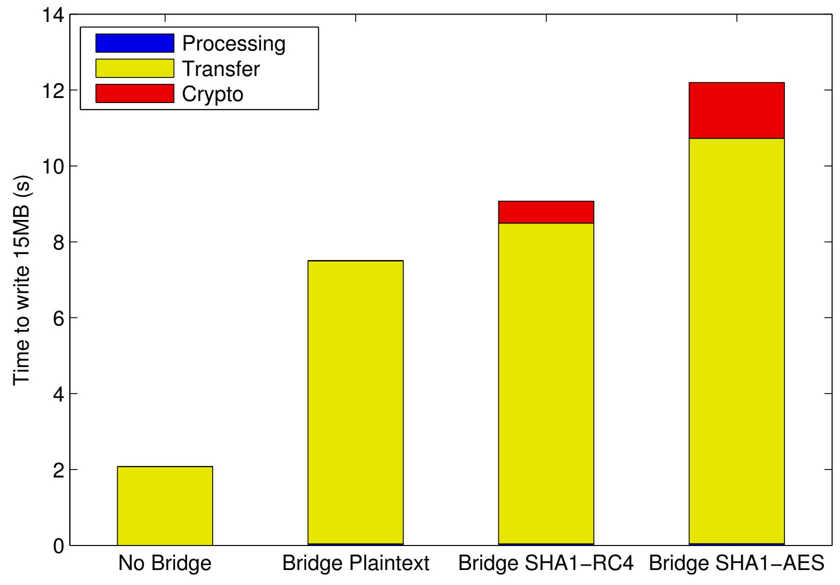


(b) Write 15MB

Figure 7.17: Comparison of bridge overhead for accessing flash storage in various encryption schemes. We also add an estimated best performance which may be obtained on the BeagleBone in the SHA1-AES encryption scheme.



(a) Read 5MB



(b) Write 15MB

Figure 7.18: Comparison of the kernel module overhead for accessing flash storage in various encryption schemes. We also provide the original overhead from issuing USB requests via `usb_submit_urb` without the bridge solution.

7.3.3 Summary

The preliminary benchmark result shows a significant performance impact by the secure USB bridge due to the following reasons.

- When the bridge forwards data transfers securely, as designed in Chapter 6, it must first receive the data, process it, and sent it to either the device or the host computer. The overhead of receiving data and sending it without parallelism implies an 50% penalty in terms of performance.
- Our implementation on the BeagleBone does not make full use of its hardware capabilities such as cryptographic acceleration and DMA.

Our prototype implementation is therefore not suitable for securely encapsulating high speed USB communication. However, we notice that prototype is still useful for device which require low bandwidth, such as USB keyboards, mouse, and even simple healthcare devices which sends or receives data in the size of a few bytes. With specialized hardware design such as field programmable gate arrays (FPGA), dedicated cryptographic processors and faster USB communication hardware, the secure USB bridge may be implemented to reduce performance penalty.

Chapter 8

Conclusion and Outlook

In this thesis, we investigated security threats surrounding USB devices during their interaction with a host computer running a Linux OS. We focused on a scenario in which users interact with these devices through software applications either provided by the device manufacturers or built into the OS. We identified various attack vectors and their feasibility based on the the compromise of various software or hardware components of the USB stack. These attacks allow the adversary to eavesdrop, insert, modify, and delete data transferred over the USB communication channel; it also allows the adversary to impersonate the device or its application. Since device applications often rely on many resources provided by the OS, we focus on devising a solution that does not require significant modifications of the OS, such as hypervising. We therefore restrict our problem by defining an adversary model in which the adversary may compromise a user account in software or control malicious hardware in the USB communication channel.

Based on the above chosen adversary model, we provided a solution consisting of two parts: the USB device access control module and secure USB bridge. The USB device access control module employs existing DAC, MAC and software integrity checking mechanisms to ensure that only authorized and unaltered device applications may access the USB device. The secure USB bridge acts as a proxy for the USB device and secures the communication channel by encrypting and verifying the integrity of the original communication with a software module in the host OS. With the exception of deletion attacks, our secure USB bridge design prevents eavesdropping, insertion, and modification attacks between the bridge and the host computer. On the other hand, the solution does not protect against malicious devices connected to the host computer through the bridge.

Since software-based solutions are well-known and implemented in most OSes, we focus on the implementation of the secure USB bridge. We implemented a prototype bridge to communicate with a kernel module on a Linux host computer. We have performed a preliminary performance analysis of our prototype by testing it with a USB bulk device and a flash drive. We observed that our secure channel using SHA1-RC4 and SHA1-AES encryption significantly reduces data throughput, and we have identified the cause of the performance impact to be the bottleneck on the prototype bridge implementation.

To conclude, our solution has provided a general approach to improving communication security on commodity USB products without requiring the modification of either the device or the device software.

Further research may be pursued in the following directions.

- **Device identification.** By measuring certain characteristics, such as power usage profiles, the bridge may be able to individually identify USB devices, which is helpful in preventing attacks from malicious devices connected directly to the bridge.
- **Key exchange protocols.** There remains many other pairing mechanisms which could be explored to identify a suitable key exchange protocol in the use case of USB devices.
- **Performance improvement.** Our current prototype induces significant performance overhead, which could be reduced by specialized hardware. Additionally, the bridge could be enhanced to prefetch data if it is suitable for the device type, thereby improving overall throughput. For example, if a webcam is detected, the bridge may fetch its data while waiting for requests from the host computer.
- **Extensive USB device support.** To improve the compatibility of our solution, more USB device types with different communication protocols may be tested.
- **USB On-the-go.** USB On-the-go (OTG) [71] is a standard which allows a USB device to communicate with another USB device instead of a host computer. It would therefore be interesting to investigate this use case and adapt our security solution accordingly.
- **Secure USB standards.** Instead of providing a bump-in-the-wire solution like the secure USB bridge, a more robust secure communication environment could be achieved by incorporating security features in future USB specifications.

Bibliography

[1] Mashable: Did You Know 3 Billion USB Products Are Shipped Every Year? http://mashable.com/2012/10/08/usb-history/.

[2] Omron 790IT Protocol Documentation. http://docs.nonpolynomial.com/libomron/devel/asciidoc/omron_protocol_notes.html.

[3] Steven Hanna, Rolf Rolles, Andrés Molina-Markham, Pongsin Poosankam, Kevin Fu, and Dawn Song. Take Two Software Updates and See Me in the Morning: the Case for Software Security Evaluations of Medical Devices. In *Proceedings of the 2nd USENIX conference on Health security and privacy*, HealthSec'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[4] Facedancer USB: Exploiting the Magic School Bus. http://recon.cx/2012/schedule/events/237.en.html.

[5] B. Dolgunov. Enabling Optimal Security for Removable Storage Devices. In *Security in Storage Workshop, 2007. SISW '07. Fourth International IEEE*, pages 15 –21, sept. 2007.

[6] Fuw-Yi Yang, Tzung-Da Wu, and Su-Hui Chiu. A Secure Control Protocol for USB Mass Storage Devices. *Consumer Electronics, IEEE Transactions on*, 56(4):2239 –2343, november 2010.

[7] USB.org. http://www.usb.org.

[8] Linux Kernel Documentation for USB Request Blocks. http://www.kernel.org/doc/Documentation/usb/URB.txt.

[9] Linux Kernel Documentation for USB DMA and Scatter/Gather IO. http://www.kernel.org/doc/Documentation/usb/dma.txt.

[10] Linux ioctl manpage. http://www.unix.com/man-page/linux/2/ioctl/.

[11] libusb, version 1.0.0. http://www.libusb.org.

[12] I. Arce. Bad Peripherals. *Security Privacy, IEEE*, 3(1):70 –73, jan.-feb. 2005.

[13] Darrin Barrall and David Dewey. "Plug and Root," the USB Key to the Kingdom. http://www.blackhat.com/presentations/bh-usa-05/BH_US_05-Barrall-Dewey.pdf.

[14] Adrian Crenshaw. Plug and Prey: Malicious USB Devices. http://www.irongeek.com/i.php?page=security/plug-and-prey-malicious-usb-devices.

[15] John Clark, Sylvain Leblanc, and Scott Knight. Compromise through USB-based Hardware Trojan Horse Device. *Future Generation Computer Systems*, 27(5):555 – 563, 2011.

[16] A. Tetmeyer and H. Saiedian. Security Threats and Mitigating Risk for USB Devices. *Technology and Society Magazine, IEEE*, 29(4):44 –49, winter 2010.

[17] Adam Wright and Dean F. Sittig. Technical brief: Encryption characteristics of two usb-based personal health record devices. *JAMIA*, pages 397–399, 2007.

[18] Hanjae Jeong, Younsung Choi, Woongryel Jeon, Fei Yang, Yunho Lee, Seungjoo Kim, and Dongho Won. Vulnerability Analysis of Secure USB Flash Drives. In *Memory Technology, Design and Testing, 2007. MTDT 2007. IEEE International Workshop on*, pages 61 –64, dec. 2007.

[19] Sun-Ho Lee, Kang-Bin Yim, and Im-Yeong Lee. A Secure Solution for USB Flash Drives Using FAT File System Structure. In *Network-Based Information Systems (NBiS), 2010 13th International Conference on*, pages 487 –492, sept. 2010.

[20] Idan AVRAHAM, John C. DUNN, Constantyn KOEMAN, Mark WILLIAMS, and David R. WOOTEN. Accessing a USB Host Controller Security Extension Using a HCD Proxy. Patent, 02 2011. US 7886353.

[21] S.J. Stolfo. Worm and Attack Early Warning: Piercing Stealthy Reconnaissance. *Security Privacy, IEEE*, 2(3):73 –75, may-june 2004.

[22] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: An Emulator for Fingerprinting Zero-day Attacks for Advertised Honeypots with Automatic Signature Generation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 15–27, New York, NY, USA, 2006. ACM.

[23] Jonathan Shapiro, Ph. D, Michael Scott Doerrie, Eric Northup, and Mark Miller. Towards a Verified, General-Purpose Operating System Kernel. In *In Klein [10*, pages 1–19, 2004.

[24] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, HOTOS'07, pages 20:1–20:6, Berkeley, CA, USA, 2007. USENIX Association.

[25] W.A. Arbaugh, D.J. Farber, and J.M. Smith. A Secure and Reliable Bootstrap Architecture. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 65 –71, may 1997.

[26] Bill Horne, Lesley Matheson, Casey Sheehan, and Robert E. Tarjan. Dynamic Self-Checking Techniques for Improved Tamper Resistance. In *in ACM Workshop on Security and Privacy in Digital Rights Management*, pages 141–159. Springer-Verlag, 2001.

[27] David Aucsmith. Tamper Resistant Software: An Implementation. In Ross Anderson, editor, *Information Hiding*, volume 1174 of *Lecture Notes in Computer Science*, pages 317–333. Springer Berlin / Heidelberg, 1996.

[28] Hoi Chang and Mikhail J. Atallah. Protecting Software Code by Guards. In *Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, DRM '01, pages 160–175, London, UK, UK, 2002. Springer-Verlag.

[29] Microsoft Developer Network. http://msdn.microsoft.com.

[30] Ubuntu. http://www.ubuntu.com/.

[31] Security Basics - Fedora Project. http://fedoraproject.org/wiki/SecurityBasics#Security_Measures_in_Fedora_Systems.

[32] C. Cowan. Software Security for Open-source Systems. *Security Privacy, IEEE*, 1(1):38 – 45, jan.-feb. 2003.

[33] VirtualBox. https://www.virtualbox.org/.

[34] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[35] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[36] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM.

[37] Andrew C. Myers and Barbara Liskov. Protecting Privacy Using the Decentralized Label Model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, October 2000.

[38] AppArmor. `http://wiki.apparmor.net/`.

[39] Security-enhanced Linux. `http://www.nsa.gov/research/selinux/index.shtml`.

[40] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.

[41] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, April 2008.

[42] OSLO: Improving the Security of Trusted Computing. `http://static.usenix.org/event/sec07/tech/full_papers/kauer/kauer_html/`.

[43] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

[44] Amit Vasudevan, Jonathan McCune, James Newsome, Adrian Perrig, and Leendert van Doorn. CARMA: A Hardware Tamper-Resistant Isolated Execution Environment on Commodity x86 Platforms, 2012.

[45] Tilo Müller, Felix C. Freiling, and Andreas Dewald. TRESOR Runs Encryption Securely Outside RAM. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 17–17, Berkeley, CA, USA, 2011. USENIX Association.

[46] G. Edward Suh, Charles W. O'Donnell, and Srinivas Devadas. AEGIS: A Single-chip Secure Processor. *Information Security Technical Report*, 10(2):63 – 73, 2005.

[47] Liang Gu, Xuhua Ding, Robert Huijie Deng, Bing Xie, and Hong Mei. Remote Attestation on Program Execution. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, STC '08, pages 11–20, New York, NY, USA, 2008. ACM.

[48] Andrew C. Yao. Protocols for Secure Computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[49] TOMOYO Linux. http://tomoyo.sourceforge.jp/.

[50] The Smack Project. http://schaufler-ca.com/.

[51] Chris Wright, Crispin Cowan, and James Morris. Linux Security Modules: General Security Support for the Linux Kernel. In *In Proceedings of the 11th USENIX Security Symposium*, pages 17–31, 2002.

[52] W. Diffie and M. Hellman. New Directions in Cryptography. *Information Theory, IEEE Transactions on*, 22(6):644 – 654, nov 1976.

[53] BeagleBoard.org - BeagleBone Rev A6. http://beagleboard.org/bone.

[54] ARM Cortex-A8 Processor Product Page. http://www.arm.com/products/processors/cortex-a/cortex-a8.php.

[55] Texas Instruments AM3359 ARM Cortex-A8 Microprocessor. http://www.ti.com/product/am3359.

[56] StarterWare for ARM based TI microprocessors, version 2.00.00.07. http://www.ti.com/tool/starterware-sitara.

[57] Light-weight, modular cryptographic and SSL/TLS library in C - PolarSSL, version 1.1.4. http://polarssl.org/.

[58] The Transport Layer Security (TLS) Protocol Version 1.2. http://tools.ietf.org/html/rfc5246.

[59] Linux Kernel Documentation for Kernel Probes. http://www.kernel.org/doc/Documentation/kprobes.txt.

[60] StarterWare Forums - BeagleBone High-speed bulk USB transfer. http://e2e.ti.com/support/embedded/starterware/f/790/t/184201.aspx.

[61] StarterWare Forums - AM335x USB Host CppiDmaModeSet() function for setting Per EndPoint RNDIS mode does not match TRM. http://e2e.ti.com/support/embedded/starterware/f/790/t/207941.aspx.

[62] StarterWare Forums - USB DMA RNDIS question. http://e2e.ti.com/support/embedded/starterware/f/790/t/211686.aspx.

[63] Linux USB Gadget API Framework. http://www.linux-usb.org/gadget/.

[64] Freescale i.MX53 Quick Start Board Product Summary Page. http://freescale.com/iMXQuickStart.

[65] A. Nadeem and M.Y. Javed. A Performance Comparison of Data Encryption Algorithms. In *Information and Communication Technologies, 2005. ICICT 2005. First International Conference on*, pages 84 – 89, aug. 2005.

[66] Peter M. Ridge and David Deming. *The Book of SCSI*. No Starch Press, San Francisco, CA, USA, 1995.

[67] Sysprof, System-wide Performance Profiler for Linux. http://sysprof.com/.

[68] The Linux Kernel Archives. http://kernel.org/.

[69] AM335x-PSP 04.06.00.06 Features and Performance Guide. http://processors.wiki.ti.com/index.php/AM335x-PSP_04.06.00.06_Features_and_Performance_Guide.

[70] AM335x Crypto Performance. http://processors.wiki.ti.com/index.php/AM335x_Crypto_Performance.

[71] USB.org - USB On-the-go. http://www.usb.org/developers/onthego.

Internet references were accessed on Tuesday 23rd October, 2012.