Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU WIEN UB Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Bad Things Happen through USB

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Security Engineering for Enterprise Environments

eingereicht von

Sebastian Neuner
Matrikelnummer 1227217

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl

Wien, 17. Oktober 2017

Sebastian Neuner	Edgar Weippl

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Bad Things Happen through USB

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Security Engineering for Enterprise Environments

by

Sebastian Neuner
Registration Number 1227217

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl

Vienna, 17th October, 2017

Sebastian Neuner Edgar Weippl

Erklärung zur Verfassung der Arbeit

Sebastian Neuner
Dr.-Kaserer-Weg 4
6170 Zirl

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 17. Oktober 2017

Sebastian Neuner

Abstract

The universal serial bus (USB) is the most widely-used connector for modern computer systems and serves to connect printers, cameras, storage media and many other different devices. Nowadays, almost no computer system is produced that does not have such a port. However, this port is not just used to connect benign peripheral devices, but also to infect the systems a USB device is attached to. This was impressively demonstrated by Stuxnet, a malware designed to sabotage uranium enrichment centrifuges in Iran's nuclear facilities via malware that initially infected the control systems through their USB ports.

Motivated by the ubiquitous usage of USB all over the world and the resulting severity of vulnerabilities triggered by USB, this thesis targets the specific problem of USB-related reflashing attacks. Those attacks make use of the convenience of modern operating systems offering plug and play functionality for USB devices in order to inject malicious keystrokes without the knowledge of the victim. A previously demonstrated instance of this attack species is BadUSB, which is able to reflash commodity USB devices, such as USB flash drives to perform keystroke injection attacks. Other BadUSB-like attacks, such as the Rubber Ducky also inject keystrokes to a victims operating system, but need dedicated hardware.

To gather insights into BadUSB and BadUSB-like attacks, in this thesis we present a large scale data collection and evaluation of USB data. We collect millions of benign USB packets over several months within a company-wide environment and from several volunteers included in a typing dynamics experiment. Based on an evaluation of this data against data we gather from BadUSB and BadUSB-like attacks, we propose several detection and prevention mechanisms on system- but also on a company-wide networking level.

Kurzfassung

Der Universal Serial Bus (USB) ist der weitest verbreitete Anschluss für moderne Computersysteme und dient der Verbindung von Druckern, Kameras, Massenspeichern und vielen anderen Geräten. Aktuell wird nahezu kein Computersystem ohne einen derartigen Anschluss hergestellt. Jedoch wird USB nicht nur verwendet um gutartige Geräte an Computersysteme anzuschließen, sondern auch um diese Systeme anzugreifen. Dies wurde eindrucksvoll von Stuxnet demonstriert- einer Schadesoftware, die mit dem Ziel entwickelt wurde, Irans Nuklearanlagen über die Steuersysteme für dessen Zentrifugen für Urananreicherung zu sabotieren. Diese waren, wie alle modernen Computersysteme, ebenfalls mit einem USB-Anschluss ausgestattet.

Angetrieben vonder weltweiten Verbreitung von USB und dem Risiko, welches bei durch USB ausgenützte Schwachstellen entsteht, befasst sich diese Abschlussarbeit mit USB-reflashing-Attacken. Diese Art von Attacken nützen die Plug and Play Funktionalität für USB Geräte moderner Betriebssysteme aus, um Tastaturanschläge ohne das Wissen des Benutzers in das System einzuschleusen. Die kürzlich vorgestellte Attacke BadUSB ist eine derartige USB-reflashing Attacke, die USB Geräte des täglichen Gebrauchs, wie etwa USB Massenspeicher reflashen kann um mit dem Gerät Tastaturanschläge einschleusen zu können. Ähnlich der BadUSB-Attacke ist die sogenannte Rubber Ducky Attacke, welche zwar auch Tastaturanschläge einschleust, jedoch eine eigens dafür gefertigte Hardware benötigt.

Um bessere Einblicke in BadUSB und ähnliche Attacken zu bekommen, führen wir in dieser Abschlussarbeit eine umfangreiche Datensammlung und Evaluierung dieser Daten durch. Wir sammelten über einige Monate Millionen von unmodifizierten USB Paketen innerhalb eines Unternehmens und von mehreren Freiwilligen, welche sich für ein Schreibexperiment meldeten. Basierend auf diesen Daten und dessen Auswertung in Verbindung mit Daten von BadUSB und ähnlichen Attacken, schlagen wir Erkennungs- und Abwehrmechanismen auf System- aber auch auf Netzwerkebene vor.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Problem 1
1.3 Aim of the Work 2
1.4 Methodology 3
1.5 Contributions 3

2 Background **5**
2.1 Linux Fundamentals 7
2.2 Windows Fundamentals 7

3 Related Work **9**
3.1 Attacks on USB 9
3.2 Defenses against USB Attacks 12
3.3 Keystroke Behaviour 13

4 Design **17**
4.1 System Overview 17
4.2 Design – Problem 1 17
4.3 Design – Problem 2 19
4.4 Design – Problem 3 20

5 Implementation **21**
5.1 General Design 21
5.2 Implementation: Prototype 1 21
5.3 Implementation: Prototype 2 22
5.4 Implementation: Problem 3 23
5.5 Data Storage 23

6 Evaluation **27**
6.1 Data Set: Problem 1 27
6.2 Data Set: Problem 2 31
6.3 Data Set: Problem 3 32
6.4 Data Set: Attacks 36

7 Discussion **41**
7.1 Research Question 1 . . . 41
7.2 Research Question 2 . . . 42
7.3 Research Question 3 . . . 42
7.4 Research Question 4 . . . 44
7.5 Limitations and Future Work . . . 45

8 Conclusions **47**

A Appendix **49**

Bibliography **65**

CHAPTER 1

Introduction

1.1 Motivation

In a horrible scenario an attacker finds a way to exploit every computer in the world with just a single attack vector – even atomic power plants. This attack vector is called universal serial bus (USB). USB is the most widely-used connector for modern computer systems to connect printers, cameras, storage media and many other different devices. Nowadays, there is almost no computer system produced that does not have such a port. Even uranium enrichment centrifuges are affected, as Stuxnet, a malware designed to sabotage Irans nuclear facilities, impressively demonstrated. This highly-sophisticated malware infected a computer running Windows via a USB device and then spread to SCADA and PLC systems within the same facility. Once those systems were infected, Stuxnet was able to manipulate the centrifuges to cause severe damage[32] [41] [29].

1.2 Problem

Motivated by the high availability of USB all over the world and the resulting severity of vulnerabilities triggered by USB, this thesis targets the specific problem of USB-related reflashing attacks. An instance of this attack, BadUSB, has previously been presented by Nohl et al. [46]. Those attacks make use of the convenience of modern operating systems offering plug and play functionality for USB devices. This plug and play functionality automatically selects a driver based on the kind of USB device inserted by the user. In general, a driver is a piece of software which is used to establish a connection between any hardware and the operating system and acts as an interface between those. This automatic, convenient selection causes the problem, that the user has to rely on the operating system to properly select a driver (without having the possibility to intervene). Additionally, the normal computer user has no possibility to control whether the device behaves as intended. To illustrate, consider the following examples: When inserting a

USB mass storage device, the user expects the mass storage driver be loaded so that the user can interact with the USB storage. When inserting a webcam with built-in microphone the user expects a camera driver and a microphone driver to be loaded. However, inserting a USB device that is reflashed into a BadUSB device [46], causes multiple drivers to be loaded, carrying out malicious tasks, such as the rapid injection of keystrokes. Those keystrokes can modify the users' system without being noticed by the user, for instance, to implant a backdoor. Due to the fact that every user dealing with USB devices can be targeted and there is no effective mitigation so far, this and other attack vectors described in Chapter 3 are a high-impact attack surface.

1.3 Aim of the Work

The overall goal of this thesis is to answer the following research questions:

1. How prevalent are connected USB devices over a certain time frame and how to measure the prevalence at scale (i.e., in an enterprise network)?

2. What is the entropy of a typical user typing a specific pre-defined paragraph that is not known in advance, with particular regards to the timing information?

3. Is there a statistical significant pattern of connected devices that can be used for a per-user whitelisting approach of benign USB devices?

4. Is there a significant difference between the collected data and emulated keystrokes as it would be done by BadUSB or a BadUSB-like device?

To answer those questions, a significant amount of USB data, such as USB packets and USB metadata are needed. To tackle research question 1, a Python application is implemented that collects messages sent by the kernel module usbmon [76] from kernel to userland. Having this application in place, the first experiment is its installation on a typical users' machine and observing the described metrics over a period of three months.

Research question 2 is answered by collecting data via an implemented kernel module for the Linux operating system. This kernel module consists of several parts e.g. an interface to the userland level of the operating system and gathers data on the lowest level of the Linux operating system, namely the kernel [33]. In contrast to usbmon, this kernel module only sends timing information to the userland to preserve the privacy of the user typing. To get a representative amount of data, this kernel module is tested in a user study including 33 people to show its applicability. Those 33 users are typing a pre-defined pattern, which is then compared to the data of the long-term gathering. Those subjects are selected out of a large number of project partners at SBA Research[1] and have to fulfill the requirement of being an office employee working with computers on a daily basis.

[1] `https://www.sba-research.org/`

Both prototypes allow the collection of statistical data. In case of the first prototype, important data to be collected, includes among others the number of interfaces on each USB device, product and vendor ID. However, both prototypes collect the most important data: the arrival time delta of keystrokes on the operating system.

An additional way to get typical users' data is the extraction of the Windows Registry which stores historical data about every USB device plugged-in from the time of installation of the operating system up until the time of lookup [71] [35]. Therefore, the Windows USB device history of all employees of SBA Research[2] was collected, by extracting the required information from the registries of all clients. Combined with the results of research question 1, this answers the question if a whitelisting approach is possible and therefore also answers research question 3.

Ultimately, to answer research question 4, the collected data of question 1 to 3 is evaluated and compared against extracted data of the successful exploitation of a Linux computer by BadUSB.

1.4 Methodology

An extensive state-of-the-art assessment has to be carried out on related research is the foundation for the basic points of this thesis and is vital to develop novel approaches. Furthermore, a survey on the available technologies is important for the succeeding step of this thesis.

A proof-of-concept prototype for research question 1, the userland receiver is implemented. Additionally, the prototype for research question 2, namely the kernel module is implemented. This implementation includes the gathering of USB data as well as the interfaces to the userland (e.g. to write files to the filesystem). Then, the evaluation starts by gathering USB data on a typical users' machine, proceeding by letting users type a pre-defined pattern and gathering the data from clients at SBA Research. The conclusion of the evaluation phase is the comparison between the captured user data and the data from attacks like BadUSB. This comparison mainly focuses on comparing the interarrival time of keystrokes between a user and the attack by BadUSB but also the amount of interfaces bundled with product and vendor ID compared to the BadUSB attacking device. Possible future work is the creation of an algorithm that detects anomalies between the typical user data, which can be based on the data collected in this thesis and the emulated keystrokes by an attack like BadUSB.

1.5 Contributions

This work provides novel insights into the usage of USB devices in regards to reflashing attacks such as BadUSB. To this end, this work contains a large amount of USB-related data as well as the tools utilized to gather and evaluate this data. In more detail, the contributions are as follows:

[2] About 100 employees

- The data collected for research question 1, 2 and 3 and their interpretation.
- The prototype implementations used to collect and evaluate the data.
- An analysis of the gathered data to answer the question if it is suitable to defend against reflashing attacks.

Please note that none of the gathered data contains any sensitive information. All published data has been modified to ensure strong respect to privacy information.

The thesis is structured as follows: Chapter 2 provides the necessary background, where Chapter 3 describes related work for this thesis. Chapter 4 describes the design of the data collection phase as well as of the prototypes whose implementations are outlined in Chapter 5. The results of the collection phase the prototype executions are stated in Chapter 6. Chapter 7 discusses those results, the thesis' limitations and describes possible future work. Finally, Chapter 8 concludes the thesis.

CHAPTER 2

Background

The Universal Serial Bus (USB) is the world's widest used computer peripheral connector and was initially developed in 1990. USB operates in various speeds, comes in different sizes and uses varying pin assignments. USB is based on a tiered-star topology, which specifies one dedicated master controller. Besides the controller, a hub manages the USB devices. In case of the master controller, this hub is called root hub. Every USB hub uses seven bits to address connected USB devices, which leads to a limitation of 127 attachable (administrable) USB devices per hub.

A fully configured USB hub is waiting for events in idle state. Those events include plugging-in of a new USB device, which is actually an interrupt signalling a port change on the hub. This port change is signaled into the userland to the daemon `udev` [28]. Contained in the information sent by the kernel is, across others the vendor ID and the product ID of the attached USB device. The daemon now uses modprobe to load the corresponding driver to the USB device [69]. The selection of which driver is loaded for which USB device is chosen by the vendor ID and product ID of the USB device. This mechanism is called plug-and-play and is the root of the so-called `reflashing attacks`.

Taken from the USB 2.0 standard [15] Figure 2.1 shows the message flow between the USB device and the host as well as involved components. As described above, the USB hub waits for new devices to be plugged-in. Upon connection channels for communication are created: so-called endpoints. Craig Peacock [52] describes USB endpoints as sources and sinks of data. Those sources and sinks of data are logically grouped together to interfaces and announced to the host via `interface descriptors`. For each announced interface descriptor, the host selects a driver as mentioned above and binds the appropriate driver to the interface.

To clarify this with an example: A sophisticated USB mouse offers the capabilities of a mouse (human interface device (HID)), as well as of a display to show the mouses' sensitivity level. This means, this device has two interfaces, where each of it gets a

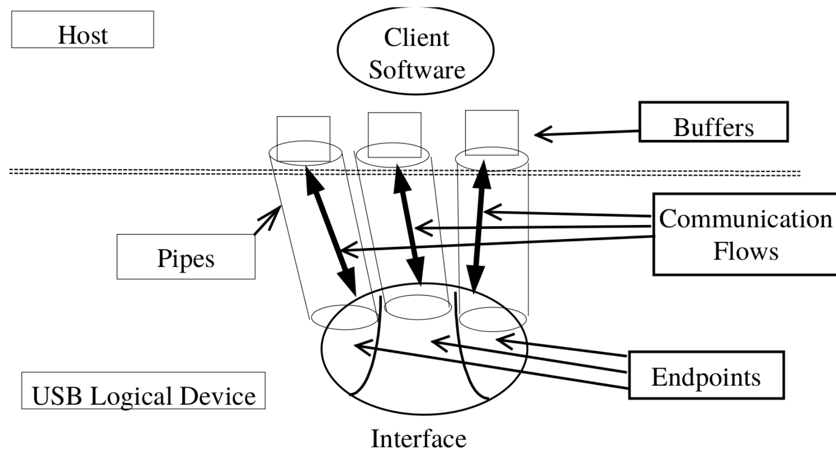


Figure 2.1: Low level message flow in USB [52].

different driver bound to it.

Several problems arise with the automatic binding of USB drivers and no checks in place. One of the most severe form of those problems are reflashing attacks. The host sets the device up as described above: After getting VID and PID from the device, the first appropriate driver is bound to the device to enable the operating system to interact with it. As soon as the legit (expected) function is announced by the USB device via an interface descriptor, a second function is announced the same way by the USB device. Figure 2.2 illustrates this with an example of a USB mass storage device: In the shown

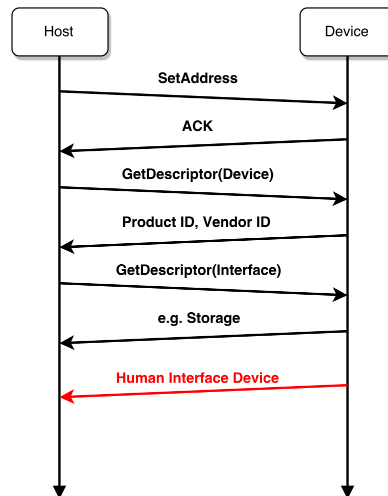


Figure 2.2: USB packet flow during a reflashing attack.

case, the user expects the loading of one single driver to the one expected functionality:

the mass storage functionality of the USB device. However, shortly after the storage functionality is announced to the host by the interface descriptor, another interface descriptor is sent to the host. This second descriptor forces the plug-and-play-ready host to bind also a HID driver to the USB device. As soon as the second driver is bound to the device, the further attack is the injection of keystrokes, while the unaware user interacts with the mass storage device.

2.1 Linux Fundamentals

To understand later chapters of this thesis, describing implementation details on Linux, this section is vital as well. Linux on modern processors follows a simple privilege model, where the kernel has the most permissions and the user the least [13]. The high privileged kernel can be extended with modules, so-called kernel modules. A messaging system between a Linux userland application and a kernel module is the netlink interface [43]. This interface is socket based and therefore bi-directional. An alternative for using netlink sockets would be writing to the filesystem, such as the `Procfs` or the `Debugfs` [26] from the kernel module. This would give a userland application access to the kernel module data. However, using this approach a file has to be monitored for changes – using the netlink approach, the data is directly sent to the listener application in userland.
The usbmon kernel module as the de-facto standard to transmit USB messages from the kernel to the userland uses a netlink derivative [76]. This module sends every USB event seen by the kernel to userland. To be more specific, it writes those events to the RAM-based debug filesystem located at
`/sys/kernel/debug/usb/usbmon/[0-9][u|s|t]` [47]. The number of the usbmon socket describes the USB bus to capture messages from, where the character describes different formats of the outputted messages. However, `0` is a special case, since it captures USB events from all USB buses.
The standard application to capture traffic, which is also used in this thesis is the successor of Ethereal, namely Wireshark[1] [56]. This application captures traffic that is transmitted over networks, such as wired and wireless networks. Since version 1.2.0, Wireshark makes use of the usbmon interface to capture traffic on USB ports. A tool that is extracted from the Wireshark code, is the command line tool `dumpcap`. This tool captures packets on a specified interface and writes it to a Wireshark compatible file, without the need of a GUI.

2.2 Windows Fundamentals

Microsoft Windows maintains a huge collection of metadata within its so-called registry. Regarding Carvey [11] the registry is also an excellent source for forensic data. The top layers are called Hives, which are filled with keys, values and data. Including logging data such as wireless networks accessed and saved in the past, the registry saves an

[1]The command line version of Wireshark is TShark

important log for this thesis, namely the USB device history [39]. One use of this USB device metadata by Windows is the maintenance of a device – driver correlation based on the product ID and vendor ID of the device [23]. Without this permanent mapping, the same driver would have to be installed every time the same USB device is inserted. To extract the information about all USB devices off the registry, several hives and their keys, values and data have to be correlated, as described by McQuaid [37]:

1. The vendor information, brand and serial number are stored in the `HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Enum/USBSTOR` hive of the registry.

2. Using the information extracted off the `USBSTOR` hive, the serial number can be used to extract the matching drive letter at the time the USB device was inserted out of the `HKEY_LOCAL_MACHINE/SYSTEM/MountedDevices` hive.

3. Correlating the information from 1 and 2, the extracted information of the `HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Enum/USB` hive is the product ID, vendor ID as well as the first and last connection times of each USB device.

For every USB device there exist a `USB class` device entry in the registry as outlined by Ibrahim [21]. In contrast to Linux, which also lists interface classes on demand, Windows saves every interface of every USB device in a separate value looking the same as the USB device itself. The following example illustrates this with an example: A USB keyboard is plugged in. The keyboard needs the HID driver on the only interface of the device to work properly. The USB device itself will show the device class `0x00`, indicating that the functionality is specified on interface level as specified in the USB standard. This interface class code is saved as its own `USB class` value in the registry, in case of the keyboard as `0x03`. By matching the vendor ID and product ID of the USB classes `0x00` and `0x03` to each other, the result is one physical device with two entries in the Windows registry.

Finally, the important data extracted from the Windows registry is among others: The vendor ID and product ID, first time of the USB drive to be plugged in as well as the last time and furthermore the device (interface) classes. Chapter 6 and Chapter 7 provide deeper insights, why these values are important and vital for the evaluation.

CHAPTER 3

Related Work

In recent years several severe attacks on USB were described in academic literature as well as appeared in media. This chapter will give an overview on related work, divided into already published attacks on the one hand and defense techniques on the other hand.

3.1 Attacks on USB

The most media-hyped attack through USB is the so-called BadUSB [46] vulnerability, which was discovered by a German research team in 2014. BadUSB makes use of the reflashing possibility of certain USB firmware chips [70] and the flexibility of drivers bound to USB devices. So far different attack vectors using the BadUSB vulnerability are known, including the following:

BadUSB [46]: Reflashing the chip to enable an USB storage device to act not only as SCSI device (mass storage) but furthermore as HID device (e.g. a keyboard). By acting as a keyboard, victims operating system binds an additional HID driver to the USB device and interprets data coming from the USB device as keystrokes. Considering the worst case, the attackers payload on the USB device implants a backdoor on the victims machine and is in total control from that point on.

badAndroid [44] considers the threat model, “a user connects an alleged benign Android device to a victims computer via USB, to charge its battery”. In the typical scenario no notification informs the user about modifications in the routing table, like changing the default gateway to the Android devices' IP. From that moment on, every traffic is routed via the mobile phone, enabling the attacker to inspect the whole bi-directional traffic as well as change DNS entries and therefore redirecting traffic to attacker-controlled servers.

badBIOS [45]: In case the driver for the malicious USB device is bound to it at boot time, the modified device emulates keystrokes while the operating system executes

essential, initial functions (e.g. invoking the boot loader). By emulating keystrokes at boot time, a maliciously crafted BIOS hidden on the USB device could be installed on the computer. This BIOS overrides the original BIOS and is from then on the default BIOS to boot from- enabling an attacker to execute commands before the actual user operating system is loaded.

The security company Sophos [14] discussed recently the applicability of badBIOS on modern Smart TVs. Evidently indicating that every device providing a USB port is possibly endangered by reflashing attacks, badBIOS is used in this example to connect air-gapped devices. This connection is achieved by forcing the Smart TV to produce high-frequent audio signals, carrying information to other devices.

Other USB attacks besides the media-hyped BadUSB attack include the work by Mulliner et al. [42]. The paper describes an attack, where an USB drive emulates a mass storage device. Certain Smart TVs (e.g. selected Samsung Smart TVs) carry out two read attempts before installing a firmware update, namely `check` and `install` phases. Therefore the research team modified the USB drives firmware to deliver different firmware update files on each read attempt by the TV. For the first read attempt of the TV reads a benign firmware update file, right before the second read will be presented a malicious firmware update files. This so-called time-of-check-to-time-of-use vulnerability (TOCTTOU) is a subset of the better known race-condition vulnerabilities.

Maskiewicz et al. [36] launched their proposed attack by manipulating the firmware update application of the Logitech gaming mouse G600. Since the G600 is a gaming mouse with several keys on top (besides the usual mouse buttons: left, right and scroll) it registers itself as composite device to the host, namely two HID devices: mouse and keyboard. By reverse engineering the update software, the research team was able to inject an attacker-controlled firmware into the mouse and launching further attacks over this vector. In their evaluation they opened `calc.exe` by injecting keystrokes over the mouse, to show the potential exploitability of the system. Besides showing that this kind of composite devices are a potential attack vector, they proposed signing the firmware by the vendor. Additionally to the signed firmware, a corresponding signature verification has to be in place on the mouse to verify the integrity.

A similar approach was shown by Chen in 2009 [12], by reverse engineering Apples firmware update mechanism for the Apple Aluminium Keyboard. Although a CRC32 is in place, the update check for the firmware in the update tool is circumventable, which leads to the update of an attacker controlled firmware to the keyboard. After installing the malicious firmware on the device, it injects keystrokes into the target system to compromise it and even act as a keylogger. Chen further describes the possible of denial-of-service attacks: If the firmware update process is interrupted, the device is most likely bricked. As stated by the author, this update process can also be launched (and interrupted) by visiting a malicious website with a vulnerable browser.

In 2011 at the annual hacker conference Blackhat [5], Wang et al. [68] demonstrated three different attack vectors using smartphones. Those three kinds are:

i. phone-to-computer-attacks,

ii. computer-to-phone-attack, and

iii. phone-to-phone-attacks

For for the first attack vector (i) the smartphone loads the HID driver upon connection and acts as a keyboard to inject keystrokes. By injecting a certain series of keystrokes the mass storage of the phone gets loaded into the system to install a new autorun.inf [10] file on the victims machine as well as a malicious file, which is automatically opened on insertion of new USB devices. Within the second vector (ii) the computer roots the mobile phone via fastboot (included in the Android SDK [3]). As soon the device is rooted, a pre-installed application on the smartphone clicks "yes" upon warning that the device is going to be unlocked. After carrying out these two steps, the computer is able to install malware on the target phone. This attack (ii) takes four minutes to complete. Finally, (iii) is very similar to (ii), except the Android phones host mode has to be enabled. After that, the two phone are connected with a specially crafted cable that allows exploitation of the target device.

Adrian Crenshaw presented an overview over several USB related attacks at Shmoocon [7] in 2011. The first attack discussed in the paper is the dated attack, in which the mass storage part of the USB drive contains malware that should be executed by a victim. The second described attack vector is using the now abandoned technology of Sandisk: U3 [61]. Here, the attacker creates a malicious autorun payload to be executed as soon as the U3 USB device is plugged into a computer. Other discussed attack vectors in the paper are the usages of USB as hardware keyloggers and the usage of programmable USB devices as keyboards, that act similar to the BadUSB vulnerability mentioned above (emulation of keystrokes).

Although there is a great set of attack vectors using USB as a medium described in literature, there is still research that aims to find more flaws. One of those publications is written by Schumilo et al. [58]. This paper describes the high performance USB fuzz testing [63] framework vUSBf which aims to find flaws in USB drivers. The results in form of found vulnerabilities range from null-pointer-dereferences and segmentation faults to kernel panics. As stated by the authors, several of the found bugs could not only be used for denial-of-service attacks but also for injection of malicious code.

To launch several of the above mentioned attacks but especially reflashing attacks like BadUSB, an attacker needs very specific chipsets on commodity hardware to be stealth. However, there is specific hardware designed to instrument and audit USB drivers, ports and related software. One of those hardware devices is the teensy USB development board [62]. This board is reprogrammable for various attacks in regards to emulation of keystrokes and mouse movements on a victims machine.

Making use of the teensy board, Samy Kamkar developed the attack tool USB-driveby [24]. In its current version, this tools targets Apple iOS to manipulate routing entries in the system to redirect a victim to spoofed websites and install a backdoor in case the user detects and/or modifies the routes. By also emulating mouse movement, the time-frame between plugging the device into the computer and successful exploitation is about 30 seconds as shown on the authors website. Enhancing visual stealthiness to the

victim, the attack, especially the typing could theoretically be a lot faster which would drastically affect the time of exploitation as mentioned above.

Developed by hak5 [1], the Rubber Ducky [19] is designed to deliver an attacker defined payload to a victim. On plug-in of the device, it acts as a human interface device, meaning: deliver keystrokes to the target computer. Since it simply needs the operating systems HID driver to emulate keystrokes, it works on the vast majority of operating systems available (e.g. Linux, Windows, MAC OS). Backed by a strong community, the Rubber Ducky has a simple to use scripting language [4], namely Ducky Script as well as ready-to-use payloads for different occasions available [8] [2].

A device invented to enhance a users security is the USB armory [50]. It can be used e.g. as external hardware security module. However, due to its ability to be reprogrammed, it can be reflashed with malicious payload and then used as attack device to emulate keystrokes and mouse movements.

3.2 Defenses against USB Attacks

Several security firms offer protection against reflashing vulnerabilities like BadUSB. One of those companies is the German AV vendor Gdata with its USB Keyboard Guard [6]. Once a new HID device is connected to the "protected" computer, the tool pops up a window informing the user that a new HID device is about to be loaded into the operating system. Depending on the users decision, the devices' product ID and vendor ID is either whitelisted or blacklisted, respectively. In regards to a well-known German security and electronics magazine [57], this system is essentially flawed: If an attacker manages to find the combination of product ID and vendor ID of a whitelisted device, the attackers' device will be allowed and loaded into the system, enabling it to carry out whatever malicious activity it is intended to do.

Another commercial solution to tackle reflashing attacks is the IronKey [22] produced by Imation. This USB drive was originally intended to offer encryption to ensure confidentiality of users' data. When media-hyped attacks like BadUSB appeared, IronKey started to ensure blocking unauthorized firmware updates. This should be ensured by signing the firmware and hardware based security keys, making the drive unusable if the firmware integrity check fails.

Since severe attacks like Stuxnet target industrial control systems (ICS) Yang et al. [72] describe in their paper the trust management scheme (TMSUI) for USB storage devices in the area of ICS. This kind of protection does not target technical aspects of security but organisational. Therefore TMSUI aims to protect ICS by allowing USB storage devices only on certain, protected terminals and furthermore only for a certain time-frame. The paper additionally evaluates this system regarding forensic aspects as well as revocation of trusted devices.

A recent approach to protect computers against reflashing attacks like BadUSB was published in 2015 by Tian et al. [65], namely GoodUSB. The structure of GoodUSB is manifold: The main part is a userland module, namely gud (GoodUSB daemon). This userland module furthermore consists of several modules itself: A policy engine, used

for USB device policies. These policies map a USB device to certain, allowed drivers, e.g. a USB storage device is only allowed to load the drivers for mass storage activities. A drawback here, is the general allowance for vendor specific drivers, that are always allowed (for the users' convenience). A central part of gud is the graphical user interface, which allows the user to interact with GoodUSB, e.g. to allow or deny a newly connected device. If the user allows a device, this is stored in the device database, that is also a part of gud. After allowing the device, the corresponding driver is bound to it and it is connected to the operating system for normal use.
Connected to the userland program via netlink interface is the corresponding kernel module. The kernel module is able to prevent the automatic binding of drivers to USB devices on a very low system level. For the special cases of USB headsets, the kernel uses a limited HID driver, that is implemented by the authors of the paper to allow three certain keystrokes: volume up, volume down and mute.
In case the USB device connected, is marked as malicious by the user, its control is transfered to a USB honeypot, namely HoneyUSB. HoneyUSB offers the possibility to monitor and profile the USB devices' activity for further (forensic) analysis. This USB honeypot is a Linux host, virtualized by QEMU-KVM.

In contrast to the approaches mentioned in the related work above, this thesis' approach is a non-purely-technical approach. By considering the research questions in Chapter 1, that means besides gathering the data from several different workstations and users, also their behaviour is an important factor: How is a user typing? How many devices does the user plug-in over a certain period of time? Considering the company wide Windows data: How are USB devices distributed in a mid-sized company? Is it possible to whitelist certain devices, based on this data?

3.3 Keystroke Behaviour

If an application could automatically determine the difference between a real user and a device emulating keystrokes, attacks described above would face a hard barrier before successfully exploiting a target computer. A faster approach with stronger possibilities for enforcement, would be a kernel module defending against such attacks. Instead of notifying a userland application, the kernel could decide directly and due to its proximity to the hardware (USB devices) also faster. However, strong considerations against modifications of the kernel are security aspects: Is the modification of the kernel vulnerable in any kind, the highest level of privileges on Linux would be compromised.

So far research has not specifically dealt with keystroke dynamics in regards to reflashing attacks like BadUSB. However, machine learning effort is a possible solution to such problems.

As shown by Lane et al. [31] profiles based on behavioral sequences can be used for anomaly detection. The research team furthermore stated that this approach can be extended to keystroke rates and therefore anomaly detection in a users' typing behaviour.

Several other approaches were proposed in literature that show how machine learning is used to classify typing behaviour and keystroke dynamics. Raghu et al. [54] describe in

their paper how neural networks [18] can be used to identify and authenticate a user by using his keystroke dynamics in a web application. Compared to k-nearest-neighbor [27], the proposed approach by Raghu et al. has an average error rate of one percent, when identifying a user and comparing to an imposter.
In contrast, Yu et al. [74] use several different machine learning approaches to differentiate between a user and an imposter for identification: supported vector machines (SVM) [20] for novelty detection, genetic algorithms [16] and a genetic algorithm SVM wrapper for feature subset selection. However the evaluation is based solely on the genetic algorithm SVM wrapper. Regarding the results of the proposed approach: The average false rejection rate (FRR) is 15.78%, where the average FRR with an improved feature selection is 3.54%.
Revett et al. [55] propose in their work a behavioural biometric based on keystroke dynamics. By using a probabilistic neural network a classification accuracy of 3.9% is reached.

A common problem is the classification and feature selection for machine learning techniques. Therefore very specific metrics have to be defined by the creators of such systems.
Two of those metrics are the keystroke interval and the keystroke time (and pressure) as stated by Young et al. [73]. In their patent they describe how to identify individuals (users) by their typing behaviour, based on keystroke payload (typed characters) and a timing encoder coupled to the keyboard to extract inter-keystroke-timings. Based on these features a template for every user is created, which is later used to ensure the identity of those users.
By relying on keystroke latency pattern analysis Song et al. [60] show how to recognize users. In contrast to user identification as described above the approach by Song et al. creates one reference sample that is further used for authenticating every single user. The user input is continuously sampled for further learning of the template and identification of the users. However, this paper suggests the usage of neural networks to overcome certain limitations of their probability generator.
In 2005 Araujo et al. [9] published a paper to differentiate between users for authentication. This approach includes the following parameters: ASCII code of the typed key, two keystroke latencies as well as the duration the key is pressed down. The system is able to differentiate between an user and an imposter with a false rejection rate of 1.45% and a false acceptance rate of 1.89%.
Karnan et al. [25] summarize metrics stated above. The metrics in their paper include well-known approaches used in keystroke dynamics from 1990 until 2010. This summary includes a detailed list of features as well as feature extraction methods, e.g. for keystroke pressures the extraction method Fast Fourier Transformation as proposed by Loy et al. [34] is applied. Additionally, the paper evaluates the performance of the proposed techniques.
Concluding this chapter, Peacock et al. [51] compare several techniques for identification of users using keystroke biometrics. This comparison is done in regards to the different

proposed approaches' application, classifier accuracy, usability and the confidence in their evaluation results. The paper concludes with security and privacy issues raised by keystroke dynamics, e.g. an attacker, that successfully guesses the timing to avoid being detected.
To put this into perspective of reflashing attacks: Although a machine is learned to specific user typing behaviour, an attacker might be able to imitate that user within the attack, e.g. delaying the inter-keystroke-time until it matches the users' timing.

CHAPTER 4

Design

4.1 System Overview

The system is split into three parts, where each part addresses a research question described in Chapter 1 . This section describes the design of those three problems and the design of the two prototypes used therefore. The implementation details are described in Chapter 5 and the evaluation of the data collected in Chapter 6.

The similarities between the two prototypes (Problem 1 and Problem 2) are the general flow of the data: In both cases the user produces input via a USB device, which is routed through the kernel to an userlands' analysis application. This application normalizes the data for further evaluation. The normalized data is then stored into a database for later access or public release respectively. Those steps and system parts are described in each section below.

The source code of this thesis is partially available in the Appendix of this thesis. The RAW-data used for evaluation will not be available, due to the high content of sensitive data (e.g. passwords). However, the full dataset used for the evaluation (in an anonymized format) is available online[1].

4.2 Design – Problem 1

The prototype for the first problem is shown in Figure 4.1. The specific parts of the prototype are described in the following sections.

4.2.1 Kernel Level

The user produces input via arbitrary USB devices, e.g. keyboards, mice, storage devices, etc. This input is translated by a corresponding driver to understandable signals for

[1] Available at: `https://www.sba-research.org/badThingUSB`

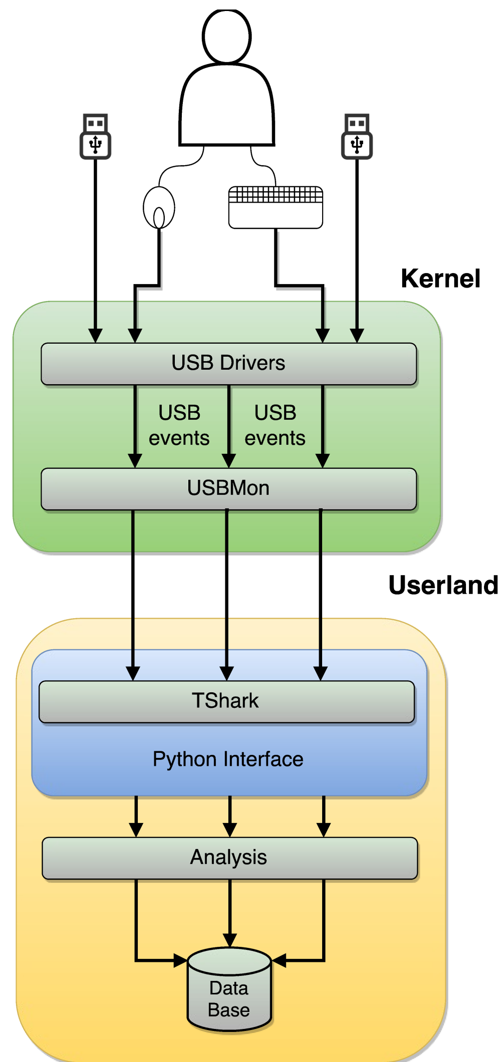


Figure 4.1: Design of the first prototype.

the kernel, that corresponds to the green area in Figure 4.1. Within prototype one, the kernel module usbmon consolidates USB events and acts as a middleware between the kernel and the userland. As shown in Figure 4.1 the USB events are sent to the userland by usbmon.

4.2.2 Userland

In the yellow area, namely the userland, TShark as part of the Wireshark bundle listens to USB events sent by usbmon. Wireshark displays those low-level messages in a convenient format to the user. The process of listening to specific USB interfaces or the whole USB bus respectively and saving USB events to a `pcap`-format file is automated by a Python

script. This Python wrapper program is illustrated by the blue area in Figure 4.1. The analysis program reads the produced `pcap`-files for further evaluation, before the data is aggregated into the database.

4.3 Design – Problem 2

Recall that the objective is the collection of USB data which is sent from the devices to the operating system. Therefore the second prototype is similar to the first in terms of the general flow of information: device(s) $\rightarrow$ kernel $\rightarrow$ userland. However, certain specifics are different as shown in Figure 4.2 and as described in the following sections.

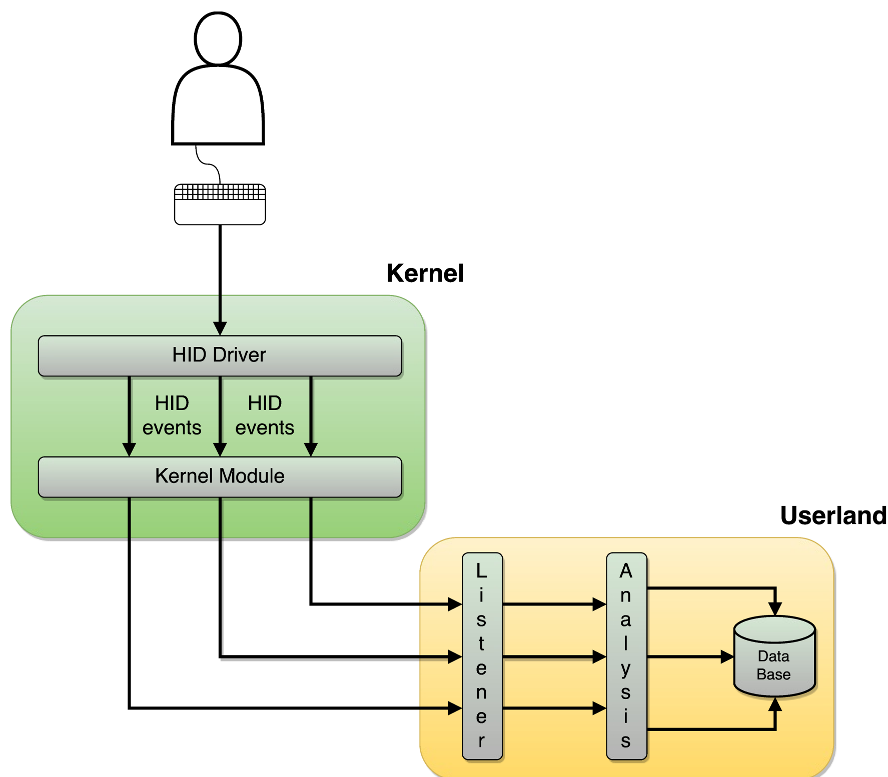


Figure 4.2: Design of the second prototype.

4.3.1 Kernel Level

The green area in Figure 4.2 marks the kernel land of the operating system. Similar to Prototype 1 input is created by the user, however the input devices are known in advance. This knowledge is derived from the evaluations' design and is further described in Chapter 6. Therefore the operating systems' built-in HID driver takes care of incoming HID hardware signals. Those HID events are captured by a kernel module, privacy enhanced by removing anything but the timestamps and then routed to the userland.

4.3.2 Userland

In userland a listener application listens to those privacy enhanced events coming from the kernel. Those messages are sent by the kernel module and specially crafted to only contain the information needed for further evaluation, namely the timing. The listener application forwards the messages to the analysis routine. This routine acts very similar to the routine in Prototype 1, however it stores different data formats in the database. The exact database format as well as other implementation related details are discussed in Chapter 5.

4.4 Design – Problem 3

Problem 3 affects the data acquisition on the Windows clients as needed for answering research question 3. To solve Problem 3, the internal structure of the network was used to accumulate the data needed. Since all of the clients are connected to one centralized server, the data can be pulled from one location. Chapter 5 offers a detailed description for this process and the used tools.

CHAPTER 5

Implementation

This chapter describes the specific implementation of the aforementioned design of this thesis. It contains application descriptions and high-level descriptions alike. For detailed descriptions of most functions (where appropriate), each corresponding listing can be consulted.

5.1 General Design

The prototypes for Problem 1 and Problem 2 are implemented for derivatives of the Linux operating system, stated in each corresponding section. This is mainly caused by the open source availability of every part of Linux. This not only includes the userland but also the kernel. Additionally, working on an open source platform offers the possibility to extend the kernel with modules as needed for Prototype 2.

For both of the implemented prototypes, but also for solving Problem 3 the backend database is a SQLite database [49]. This type of database has several advantages. In comparison to systems like MySQL and PostgreSQL [64] especially the following: It is file-based and therefore easy to share. This means, the normalized and anonymized evaluation data for each problem can be distributed within a single `.sqlite` file that is offered for download. Additionally, the main programming language used in this thesis (Python) offers excellent support for SQLite databases.

5.2 Implementation: Prototype 1

As stated in Chapter 4, Prototype 1 consists of several parts. On kernel side, the basis for Prototype 1 is usbmon. It is pre-compiled in the standard Linux kernel (any derivative). This is also true for the Linux derivate Xubuntu in version 15.10, which is delivered with kernel version 4.2 by default and was used for developing Prototype 1.
The second part of Prototype 1 works in Userland. The userland part is manyfold: An

initial setup script (i), a preparation script for capturing (ii), a wrapper application for the usbmon messages gathered by TShark (iii), as well as the analysis application that is also the connection to the data storage (iv). (i) as well as (ii) are developed as bash compatible scripts and therefore only usable on Linux systems including a shell or Windows systems with emulation software such as Cygwin[1] respectively. (i) prepares the host and the current user for capturing USB packets sent by the kernel. This is done by adding a capturing group, assigning the current user to it and adjusting the permissions of dumpcap (`/usr/bin/dumpcap`) to `754`. This permission system is the octal representation of the actual permissions and used in Linux operating systems, where `7` means read, write and execute rights for the owning user, `5` read and execute rights for the owning group and `4` read rights for everyone else on the system (users as well as groups). The preparation script (ii) loads the kernel module usbmon, if not loaded by default. Additionally, it adds privileges for users in the Wireshark group to listen to traffic crossing the usbmon devices [2] (`/dev/usbmon[0-9]`).
The wrapper application (iii) running in userland is developed in Python for the 2.7 interpreter. It takes the usbmon interface number as parameter, on which the application should listen. The captured USB messages are saved into a pre-defined location as a Wireshark capture file (`.pcap`).
Finally, the last part of Prototype 1 is the analysis application (iv) as shown in Listing A.1. Executing the main function of (iv) extracts the information needed from the given capture files and saves them into the stated sqlite database. The directory containing the capture files has to be supplied as parameter one when calling the application. The structure of the data storage is described in more details in Section 5.5.

5.3 Implementation: Prototype 2

Chapter 4 lines the design of Problem 2 out. Similar to Problem 1, Problem 2 is split into two parts: kernel part and userland. In kernel land, the kernel module reacts on every keystroke send by a user. Each keystroke is checked for its type: Either `KEY_DOWN` if a key on the keyboard is pressed or `KEY_UP` if the pressed key is released.
If the type is `KEY_DOWN` the exact time of pressing as a UNIX timestamp in nanosecond resolution is sent to the userland application. It receives the timestamps and stores them into a sqlite database.
Both, the kernel module as well as the userland application are communicating over Linux netlink sockets. The kernel module is opening the socket on initial loading of the kernel module into the kernel. Then, the socket sends multicast messages to a certain multicast group, namely `31`. Netlink allows up to 32 (0-31) netlink multicast groups.

[1] https://www.cygwin.com/ (Accessed: 15.10.2017)

[2] The default permissions of files (everything is a file in Linux based operating systems) in the device tree of Linux (`/dev/`) are limited to the root user and the corresponding group. This limitation also affects Wireshark/TShark which accesses those files while capturing USB messages from the kernel. Executing most applications as the root user is not advisable from a security perspective. Therefore access to the usbmon capture files are granted for users in the Wireshark group

Every application listening to the messages of this group receives those messages. In our case, the listening application is the userland application.

5.4 Implementation: Problem 3

Problem 3 mainly describes how to solve research question 3. The vital part of research question 3 is the problem of gathering the required data from the clients of SBA Research. To realise this gathering, the System Center Configuration Manager [38] of Microsoft delivers a piece of software to the clients for execution. This delivered software is USBDeview [59] developed by NirSoft. To be more specific, after installing the software on each client the precise execution command looks as follows: `USBDeview.exe /scomma /AddExportHeaderLine 1 export.csv`. This command exports the data needed for the further evaluation into a comma separated file.

5.5 Data Storage

As described, the evaluation data for each problem is stored into a SQLite database file. Figure 5.1 illustrates the database structure of Problem 1. Problem 1 refers to the

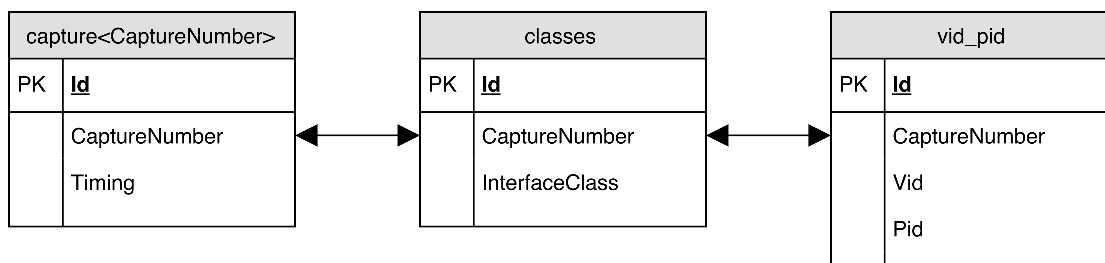


Figure 5.1: Structure of database for Problem 1.

long-term capture of USB messages on a single client. For every capture file (`.pcap` file) there is a corresponding table in the database, indicated by the table name `capture`, followed by the capture-file-number. These tables store the timing for each file. Those timing values are the timing differences (deltas) of each consecutive keystroke `KEY_DOWN` event sent by the keyboard. The time is stored in seconds with a precision of 10^{-6}, except for Problem 2, where the timing values are stored as UNIX timestamps (with nanosecond granularity).
The table `classes` contains the interface classes extracted from all capture-files. The table `vid_pid` contains the vendor ID's and product ID's collected across all captures as the table name suggests. Additionally, each table contains the capture number of the capture files. This offers the possibility to cross-correlate with the other tables of this database.

The database for Problem 2 consists of a table for each person participated in the user-study, namely `person`, followed by an incrementing number for each person. Each table contains the timing for each `KEY_DOWN` event in nanosecond granularity, as sent by the second prototype from the kernel module to the userland application. Figure 5.2 illustrates the scheme.

person<PersonNumber>	
PK	**Id**
	Timing

Figure 5.2: Structure of a sample table for Problem 2.

Problem 3 refers to the extracted data from 60 Windows registries across the company SBA Research. Figure 5.3 lines the scheme out. For each client's registry, a table is

client<ClientNumber>	
PK	**Id**
	Vid Pid Class CreatedTime LastPlugTime Serial

Figure 5.3: Structure of a sample table for Problem 3.

stored into the database with the name `client` and a number for each client. This number is incrementing and caused by process of capturing and is mainly for preserving the privacy of each participating client. Each of those tables consist of the vendor- and product ID's found on each client for each USB device. Additionally, information about the device classes and times of first and most recent plug-in are stored.
As shown in Chapter 6, the serial number of each device is needed to uniquely identify a device across all clients. All released data sets are modified upon release to ensure the privacy of every user and client. Therefore the serial numbers of each device in each table is substituted by the corresponding SHA256 hash value. This measure ensures the uniqueness of the values in case the evaluation wants to be reproduced after release of this thesis.

Additionally, the usage of SHA256 with an appended salt[3] hinders the calculation of the original serial number [53]. This is necessary to ensure the computational security of the calculated hash values. In case the serial numbers are hashed without salts, most of them could be reversed with little effort [75], since the serial numbers collected consist only of numbers. This factor would highly reduce the search space for a brute force attack. To the best of the authors knowledge, no attack on non-reduced SHA256 is known to date [30]. The code of this anonymizing application is shown in Listing A.2.

Finally, database 4 contains the timing information of two attack scenarios:

- **Rubber Ducky attack:** Table `Rubber Ducky` contains timing information on a successful attack by a Rubber Ducky.
- **BadUSB attack:** Table `badusb` contains timing information on a successful attack by a USB device, reflashed to a BadUSB device.

Chapter 6 provide deeper insights into the evaluation carried out as well as the exact values stored in each database. Additionally, the code will be partially available in the Appendix of this thesis. The databases containing data for evaluation will be fully available online[4].

[3]Note: The database with salts will not be published.

[4]Available at: `https://www.sba-research.org/badThingUSB`

CHAPTER 6

Evaluation

The following chapter consists of several parts, mainly oriented on the research questions. As described in the previous chapter, the data is extracted by the implemented prototypes and USBDeview respectively and written into a SQLite database. The evaluation itself is then carried out on the database, which is shared with the scientific community. This achieves a high reproducibility of the applied metrics.
The aim of the evaluation is applying metrics to show the differences between maliciously reflashed USB devices carrying out an attack and “normal” user behaviour.

6.1 Data Set: Problem 1

The data set of problem 1 relates to the data which is captured over three months on a single device. Since the data was acquired in an office environment, those three months match 60 days (5 working days a week). For each of those 60 days, one corresponding capture file was created. For reproducibility reasons, the specifications of this device are as follows:

- Processor: Intel i5-5200U
- Memory: 16GB
- Drive: 1TB Solid State

The database `P1_LongTerm.sqlite` contains this data as described in the previous section.

6.1.1 Problem 1: Evaluation

Every line of the capture tables within the Problem 1 database, contains the time between two KEY_DOWN events on the used keyboard. In total, 466,000 keys were pressed in

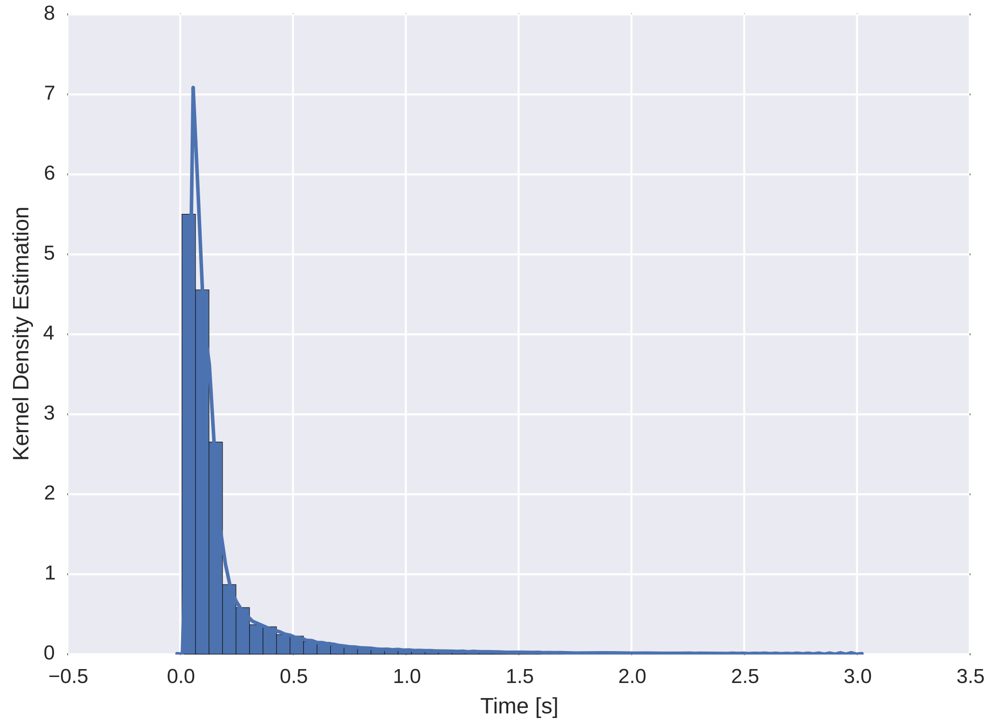


Figure 6.1: Distribution of the time delta values of Problem 1.

those 60 days. The distribution of those values is illustrated in Figure 6.1. The values are oriented mainly on the left near the Y-axis, identifying this distribution as either an exponential or a F-distribution. The largest amount of the time values are between `0.1` and `0.5` seconds. By summing up those delta values and dividing them by the number of values, the average time between two KEY_DOWN keystrokes adds up to `0.21` seconds, whereas the statistical median is `0.1` seconds. Looking at the distribution shown above, the calculated values seem to be sound.
However, the analysis application excludes delta timing values which are larger than 3 seconds from the data set. The reason behind this step is twofold: The user typing is not using the keyboard but something else (e.g. the mouse) or is away from keyboard. Therefore those values would poison the distribution and the comparative character to reflashing attacks.

Figure 6.2 shows the delta values of Problem 1 in an empirical cumulative distribution function. The trend clearly shows the largest increase between `0.0` and `0.5`. This fits the average value calculated at the beginning of this section: `0.21` seconds. Regarding this cumulative distribution, about 98% of all delta values are lower than one second.

Another question to be answered is the usage of different USB devices over three months. As shown in Figure 6.3, ten different (unique by VID and PID) USB devices are plugged into the computer over a period of three months. Across those devices, ten

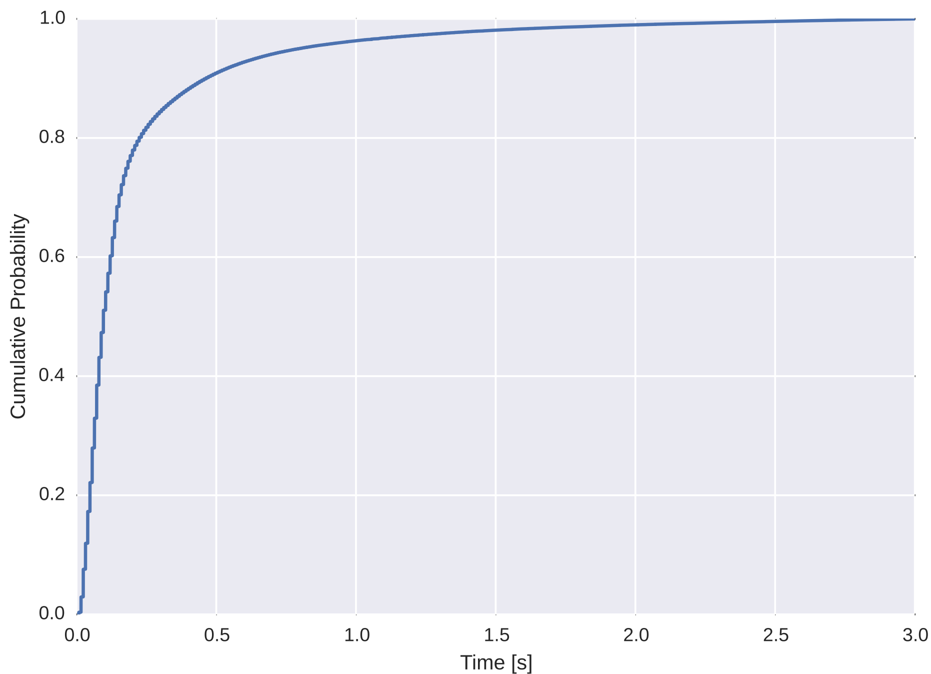


Figure 6.2: Empirical cummulative distribution of the timing values of Problem 1.

devices are unique by their VID:PID combination, and nine devices are additionally unique by their vendor ID. This means, over across three months, the user working with the computer for Problem 1 plugs-in nine different vendors.
On the other hand, the two devices on the right are plugged in every single day of those three months. Those two devices from Lenovo and IBM are the mouse and the keyboard used respectively for Problem 1 as the main input devices.

To deceive a user and distract from the existence of a maliciously reflashed USB device, those devices usually provide more than one interface, as described in Chapter 2. Therefore an analysis of the interface classes was carried out. The following listing describes which classes were present over three months and how often those classes were plugged-in:

- Human Interface Device (HID): Plugged-in 207 times.
- Mass storage device: Plugged-in 15 times.
- Smart card device: Plugged-in three times.

The vast majority of devices are either mice or keyboards for interaction. Of those occurrences, 211 plug-in events are triggered by a device with only one interface and 7 are triggered by a device with two interfaces. In more detail, the following listing provides the interfaces and interface combinations and their frequency of occurrence:

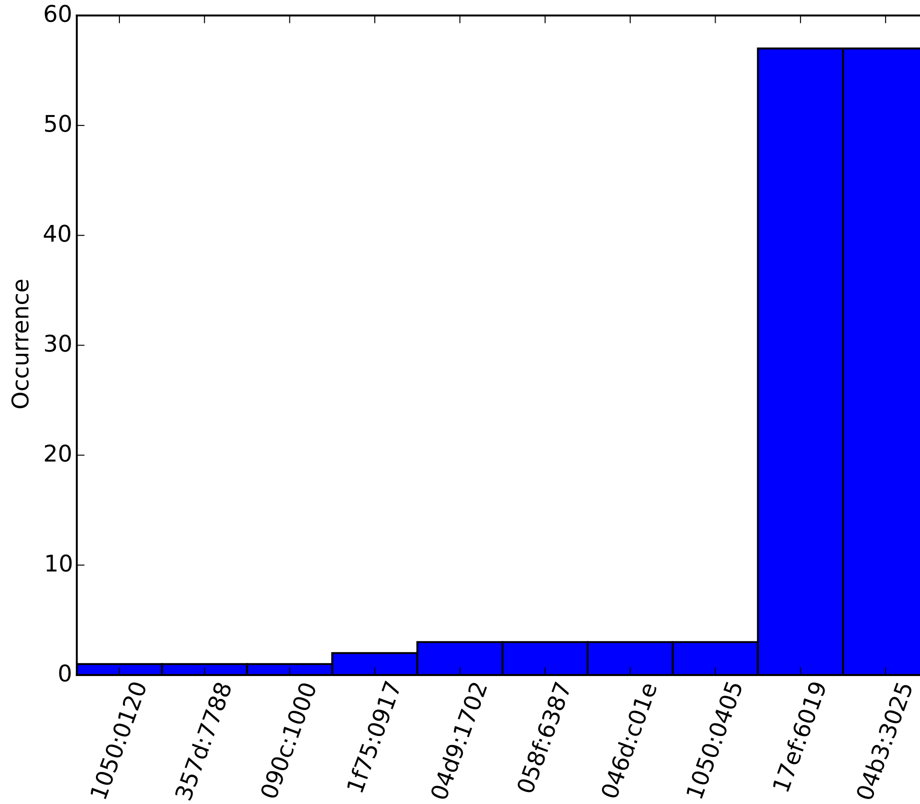


Figure 6.3: Vendor ID and product ID distribution of Problem 1.

- Single interface HID: Plugged-in 198 times.
- Single interface mass storage: Plugged-in 13 times.
- Double interface `HID:HID`: Plugged-in three times.
- Double interface `HID:Smart Card`: Plugged-in three times.
- Double interface `Mass Storage:Mass Storage`: Plugged-in once.

As derived from the vendor and product ID within the Linux lookup table [17], the combination `HID:HID` is caused by a keyboard from Holtek Semiconductor[1]. This keyboard uses a second HID interface **without** any apparent reason (e.g. no second HID device like an integrated mouse). The interface combination `HID:Smart Card` is part of a device manufactured by Yubico[2], namely their 2FA USB device. This device acts as the second factor for secure authentication. Finally, the combination `Mass Storage:Mass`

[1] `http://www.holtek.com/english/` (Accessed: 15.10.2017)
[2] `https://www.yubico.com/` (Accessed: 15.10.2017)

`Storage` is caused by a docking station for SATA devices, providing ports for several storage devices.

6.2 Data Set: Problem 2

Specified users have to type a pre-defined pattern. This pattern is a part of the online available text generator bacon ipsum[3]. To be more precise, two sentences of a one-paragraph generated bacon ipsum text.
Since those sentences have to be typed by 33 different users across SBA Research, the underlying hardware has to be portable. Therefore, problem 2 is implemented on a Raspberry Pi 2 model B, running an Ubuntu server 15.10 optimized for ARM processors. This setup is illustrated by Figure 6.4 (a) and a close up shot of the Raspberry Pi 2 in (b).

(a) (b)

Figure 6.4: (a) Setup of Problem 2, (b) close up shot of the setup.

6.2.1 Problem 2: Evaluation

In case of this study, 33 employees of SBA Research were asked to type a text containing 71 characters. Figure 6.5 shows the empirical cumulative distribution of all participants – one curve per participant. Every shown curve appears to be skewed. This is most likely caused by the study design, in which the participants had no visual feedback on the typed text. However, we entered into the compromise that favors portability of our design over visual feedback. This decision is based on the selection of the participants: office workers, typing on keyboards on a daily basis. This extensive usage of keyboards, should not require visual feedback of the typed text. Considering the top most and the bottom most curve, there appear differences. A 98% of the timing values of the top most curve are below `0.5` seconds, whereas just 58% of the timing values the bottom most

[3]`http://baconipsum.com/` (Accessed: 15.10.2017

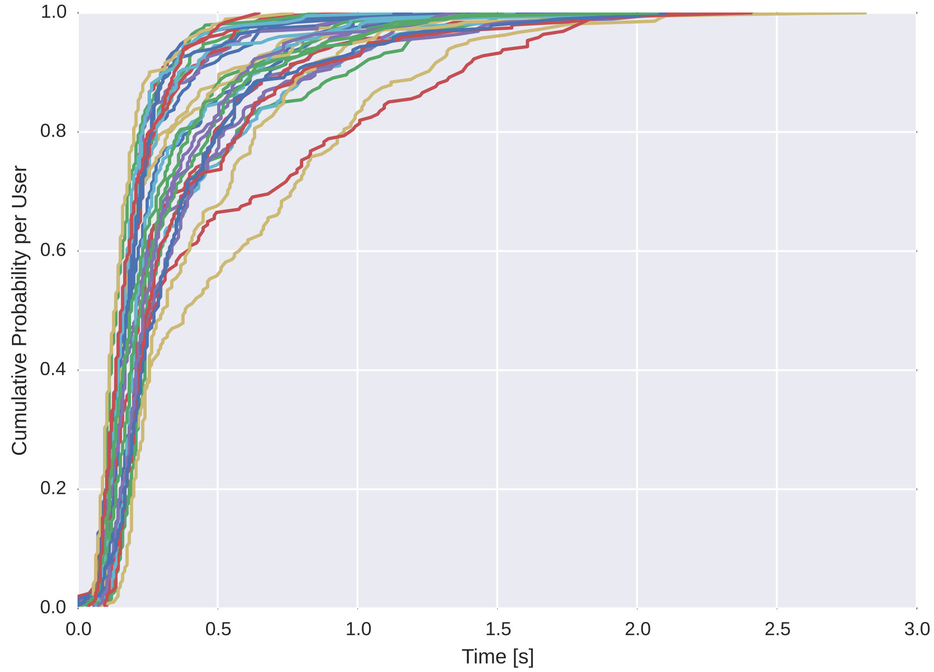


Figure 6.5: Distribution of 33 participants within Problem 2.

curve are below `0.5` seconds. The statistical median between keystrokes is `0.2` seconds, while the average value is `0.3` seconds. Processing the underlying database for Problem 2, 98% of all time values are below one second.

6.3 Data Set: Problem 3

The data set to evaluate Problem 3 consists of Windows registry data. This data is captured from 60 clients from 60 employees within SBA Research. However, SBA Research employs more than 60 employees. The data was gathered, as soon as a client, which was not seen before registered in the internal network- either locally or remote via VPN. The whole data set is stored withing the SQLite database `P3_SBA.sqlite` and a table for each client therein. Important to mention: Devices not containing a serial number are excluded from the evaluation of problem three.

6.3.1 Problem 3: Evaluation

The first step of the evaluation for Problem 3 is the distribution of the vendor ID and product ID across the 60 clients. Figure 6.6 shows how many times a unique `VID:PID` combination occurs across all clients. Three certain combinations occur 20 times, followed by 15 and 13 times of other combinations. The three combinations with the highest number of appearance (namely 20) and their corresponding vendor are as follows:

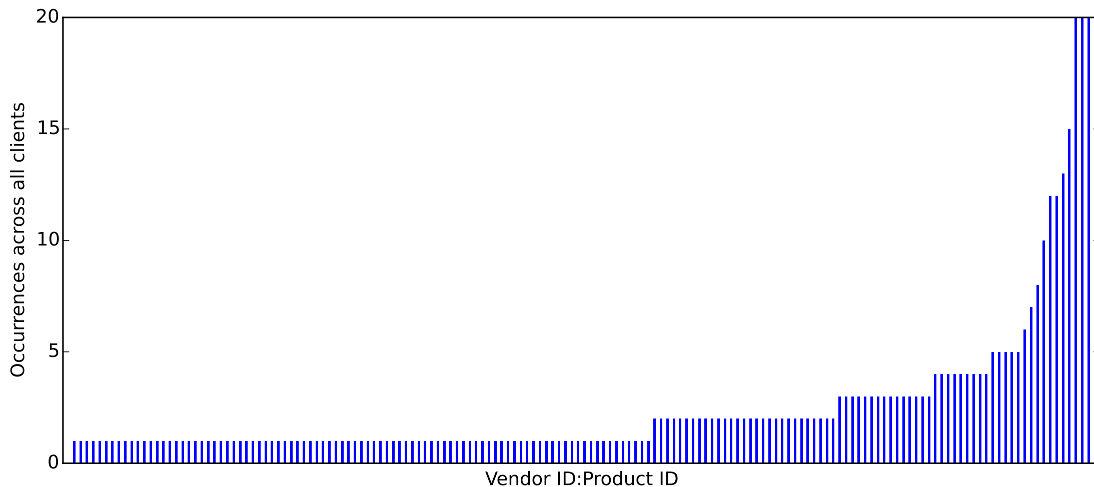


Figure 6.6: Distribution of the vendor ID and product ID of Problem 3.

- `8564:1000`: Transcend, which provides mass storage products.
- `138a:003f`: Validity Sensors, which produces fingerprint sensors. The high frequency of this vendor and product combination could be caused by a bulk order of computing devices with the same fingerprint sensor.
- `0e0f:0001`: VMWare, which provides virtualization solutions.

However, this distribution does not show if the VID:PID combination is unique across the clients. In fact, on client 6 uniquely different Transcend Jetflash mass storage devices were found. The remaining 14 devices with this VID:PID combination are distributed across other 8 clients.

Across all devices, certain interface classes appear more often than others. As described by the USB Implementers Forum [67], 00 indicates that the device class is specified on interface level. Therefore devices with just 00 as device class and no interface class are considered as errors in the Windows registry and thus omitted. In more detail, 124 devices offer one interface, introduced by the device class 00 and one device provides four different interfaces (00 excluded). This device with the VID:PID combination `04e8:681c` is a Samsung mobile phone (Galaxy). The offered interface classes are '00', '02', '06', '08' as well as 'ff'. The class 02 refers to the Communications and CDC Control class, which is commonly found on modern smartphones for tethering, class 02 refers to the Still Imaging class as described in the corresponding standard [66]. The interface class 08 refers to a mass storage device and ff to vendor specific functionalities a vendor can implement. Those vendor specific capabilities often require certain non-standardized drivers. All those values are common interfaces of modern smartphones to provide their expected functionalities.
Other classes found in the data set of Problem 3 include classes like 03, 08, e0 and others.

Notably are two values: The class for the HID devices appears only three times in the data set. This could have several reasons: For one, if the clients are laptop computers, no mouse or keyboard is needed to interact. Another reason could be wrong devices. Since devices without serial numbers are excluded from the evaluation (but still stored in the published data set), a large amount of HID devices could lurk in there. The other notable value is 08 for mass storage devices and appears 84 times over all clients.

As shown, there is a high number of duplicate devices across the 60 clients. The question answered by Figure 6.7 is how many different devices does each client provide. As for every evaluation step, devices without a serial number are omitted. This, on

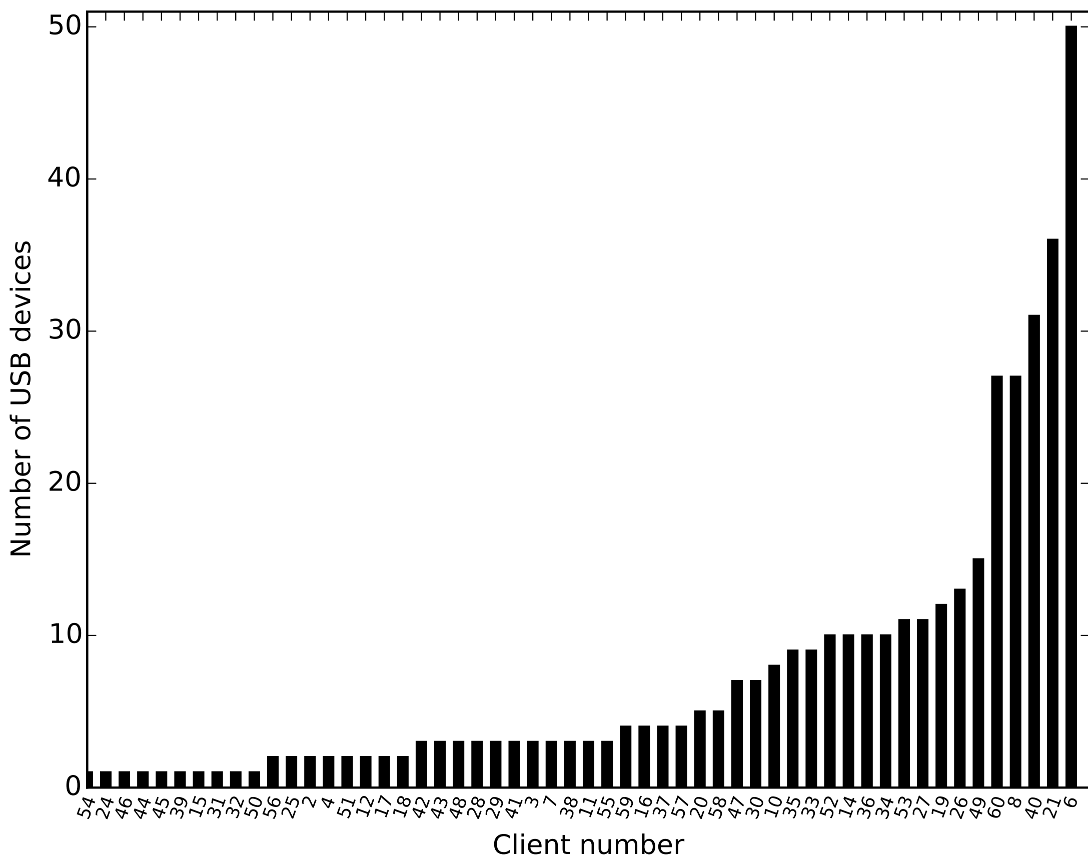


Figure 6.7: Number of USB devices of each client.

the one hand excludes certain clients from Figure 6.7, e.g. client 1, 5 and 9. However, this still leads to a high number of unique USB devices on each client. Considering the most rightwards bar, this client (client 6) contains 50 different USB devices. The next two highest values are 36 and 31 on client 21 and client 40 respectively. However, the trend indicates also several clients with only one or two USB devices. This could have multiple reasons: For one, this could be a newly setup computer, without any USB

history saved in the Windows registry. Another possible reason is a very sensitized user, taking meticulous care of every device inserted into the computer.

As shown there are a number of unique devices on each client. The question answered in the following paragraph, is how many of those devices are found across all 60 clients. Figure 6.8 illustrates this answer. This Figure shows the uniqueness of the devices in

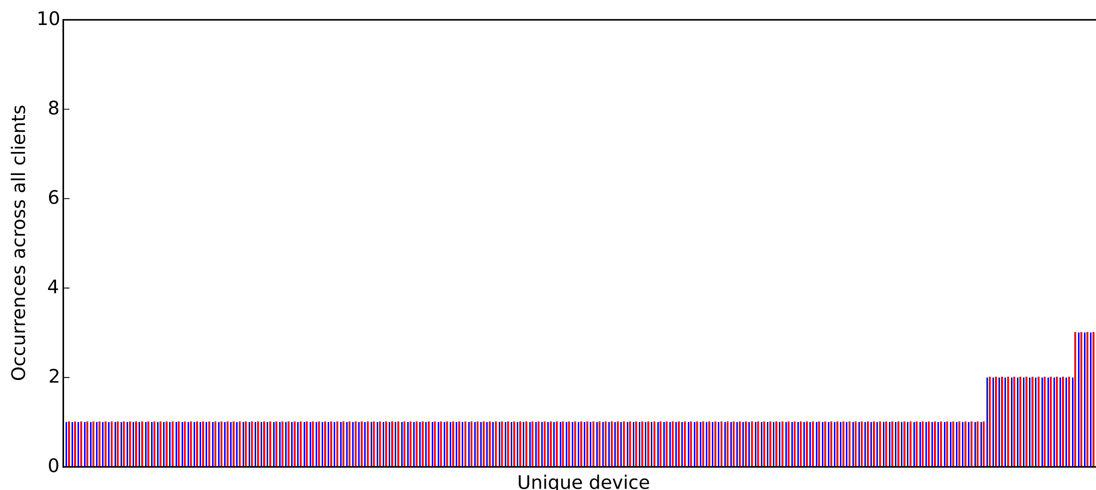


Figure 6.8: Appearance of unique devices across all clients.

more detail, where each bar corresponds to a unique device across the data set. This is due to the fact, that Figure 6.6 only takes the vendor ID and product ID into account (although omitting empty serial numbers). However, Figure 6.8 also takes the serial number into account to pinpoint a total of 341 unique devices. As the most rightwards bar shows, one device, namely the tuple `05e3:0727:`
`765ec81815afd0d0b909d4949f033da3e272af3efc359caa08246117287d01d2`
appears on ten different clients. Since the serial number should be unique worldwide (for this device), false-positives can be excluded. This furthermore means, that bringing a certain device into this company then different clients could be delivered with the same malware.

This part of the thesis' evaluation identified several important factors of USB devices in a mid-sized company (SBA Research). One important factor is the detection of unique devices across the whole data set. Building upon this knowledge, Figure 6.9 shows a timeline of devices that appear multiple times uniquely over the data set. Additionally, this Figure shows the date and time as UNIX timestamps on X-axis. Correlating the information shown, it is possible to track certain devices in time and space. Several devices occur more than once. One example is device 6 which occurs on ten different clients. As expected, this device 6 matches the device found in Figure 6.8, namely the device with the serial `765ec81815afd0d0b909d4949f033da3e272af3efc359caa08246117287d01d2`.

Since only a few time-based outliers affect visibility within Figure 6.9, Figure 6.10 provides insights with better readability. In more detail, Figure 6.10 contains the time

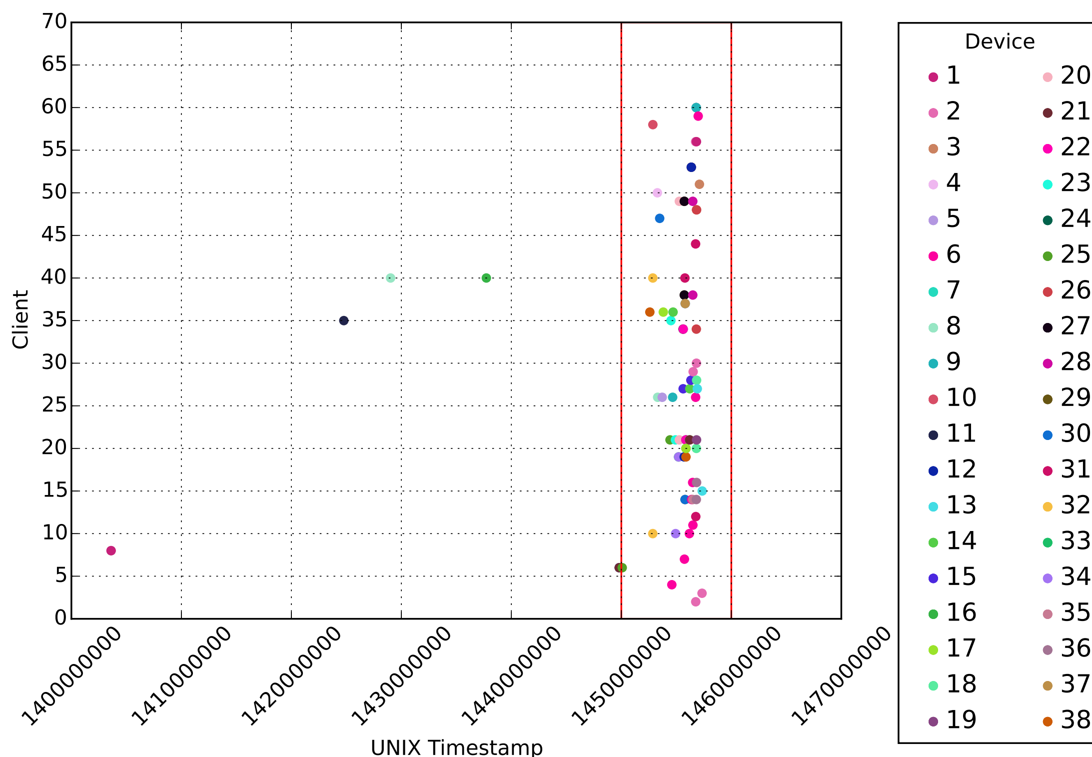


Figure 6.9: Tracking USB devices in time and space across data set 3.

frame marked red in Figure 6.9 (omitted time span: 1403608865 to 1450080458). As (now) clearly shown, device 6 appears multiple times. Table 6.1 illustrates the track of device 6 across the ten clients it was plugged into, sorted by the timestamp. This timestamp is the first connection time of device 6 on the corresponding client. The start of device 6 was on December 11th 2015 on client 6. After roughly three months, on 3rd of March 2016 the device ended up on client 59. This means, that a possible infection of ten clients in the network would take about 80 days.
The anonymized data that led to these figures and the table above, is shown in Listing A.3 as well as Listing A.4 respectively. The remaining data can be downloaded[4].

6.4 Data Set: Attacks

As later sections will show, there is a significant difference between the normal user behaviour and attacks, which are carried out by maliciously reflashed USB devices. This section covers the evaluation of the attacks. To capture and analyze the necessary data, the same applications as for Problem 1 were used, however the data is stored in the database `P4_attacks.sqlite`. Therefore also the attacks target is the same computer

[4]Available at: `https://www.sba-research.org/badThingUSB`

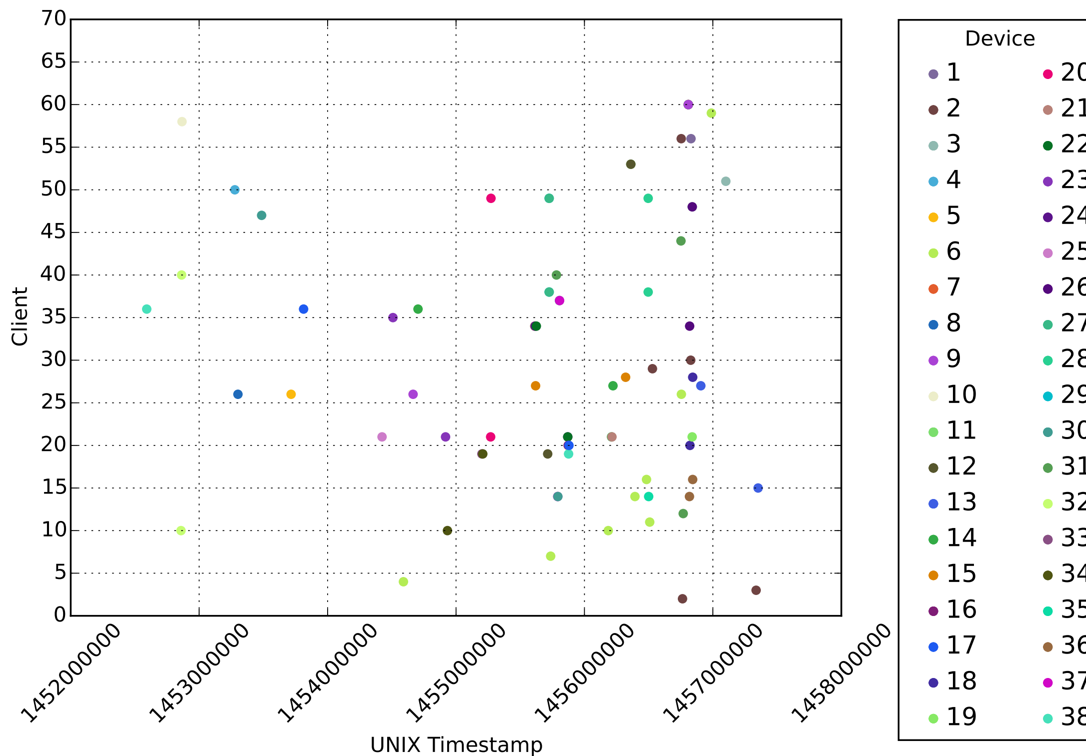


Figure 6.10: Taking a closer look at the timeline.

as used for Problem 1. The attacking USB devices are the following: One USB mass storage device, reflashed to a BadUSB device as described by Wilson [70]. The second device is a HAK5 Rubber Ducky, which emulates pre-defined keystrokes. To be more precise, the two attacking USB devices are the following:

BadUSB: The important step to carry out a BadUSB attack is the need of a USB device, controlled by a Phison chip in version 2251-03. This kind of chip is reflashable to carry out malicious activities. However, upon acquiring a USB device, only the manufacturer knows, which chip is built-in. The USB device used for the evaluation is a USB thumb drive from Kingston, manufactured by Patriot[5] (Patriot Tab 32gb).

Rubber Ducky: The latest Rubber Ducky of HAK5 as of December 2015.

6.4.1 Problem 4: Evaluation

To launch an attack using a maliciously reflashed BadUSB device, the Ducky Script language is used. This scripting language is the same as for the Rubber Ducky attack.

[5]Note: 50 USB devices were acquired from 12 different websites. Only one found to be having the right Phison chip built-in

Client	UNIX Timestamp	Timestamp
06	1449812828	Fri, 11 Dec 2015 05:47:08 GMT
04	1454590565	Thu, 04 Feb 2016 12:56:05 GMT
07	1455736927	Wed, 17 Feb 2016 19:22:07 GMT
10	1456185139	Mon, 22 Feb 2016 23:52:19 GMT
14	1456393160	Thu, 25 Feb 2016 09:39:20 GMT
16	1456483149	Fri, 26 Feb 2016 10:39:09 GMT
11	1456508222	Fri, 26 Feb 2016 17:37:02 GMT
26	1456754597	Mon, 29 Feb 2016 14:03:17 GMT
60	1456808626	Tue, 01 Mar 2016 05:03:46 GMT
59	1456987996	Thu, 03 Mar 2016 06:53:16 GMT

Table 6.1: Timeline of device 6 across the collected clients.

The USB device used for evaluation and testing was partially destroyed while flashing it with a modified firmware, making it impossible to change the USB devices' firmware again. The payload of the reflashed USB device (BadUSB device) is the following: Open an editor by sending the superkey and `r`, emulating the keystrokes for `editor` and finally emulate the keystrokes for `Hello world!!!`. This payload is non-malicious, but works as a proof-of-concept for malicious payloads. The Ducky Script, emulating these keystrokes is outlined in Listing A.5.
Examining the delta time values stored in the `badusb` table of the database `P4_attacks.sqlite`, the average value of `0.38` seconds can be calculated. Calculating the average values within the evaluation ensures a basis for further comparison. Looking at the timing values (deltas between KEY_DOWN keyboard events), the ECDF is shown in Figure 6.11. This figure shows, that over 80% of the delta values are below `0.25` seconds. The almost vertical line consists of values which differ in a range from `0.0000001` and `0.00001` seconds. However, some delayed keystrokes sent by the malicious device are above 2 seconds.
As of the vendor ID and product ID of the BadUSB device, the values 13fe as well as 5201 are in place. The vendor ID corresponds to Kingston Technologies, where the product ID is unknown to the Linux lookup table.
The whole process of reflashing a suitable USB device to a BadUSB device is described here [70], including ready to use code.

To launch an attack with the Rubber Ducky, the Ducky Script language is used. For this evaluation, the attack is non-malicious, namely opening a Linux editor and typing one paragraph of the Bacon Ipsum text, which was also used in Problem 2. Listing A.6 shows the Ducky Script used and the text emulated by the Rubber Ducky. The results of this emulation is stored within database `P4_attacks.sqlite` and table `rubberducky`. The stored timing values are the delta values between two KEY_DOWN events sent by the Rubber Ducky. Taking a closer look on the timing of the Rubber Ducky attack, the average time between those KEY_DOWN events is `0.01` seconds. This is illustrated in

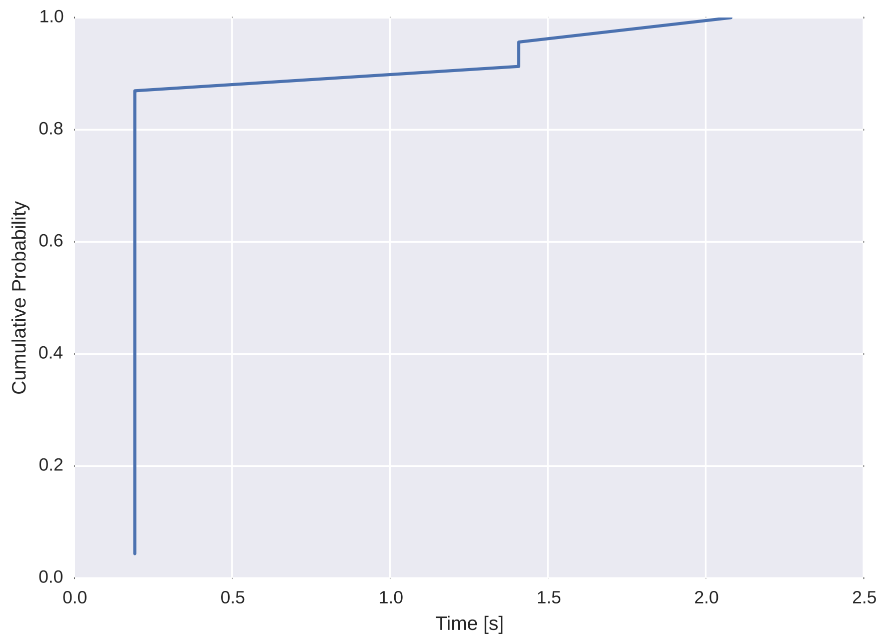


Figure 6.11: ECDF plot of the delta timing values of the BadUSB attack.

Figure 6.12 and shows the ECDF plot of the delta values. The figure shows an almost

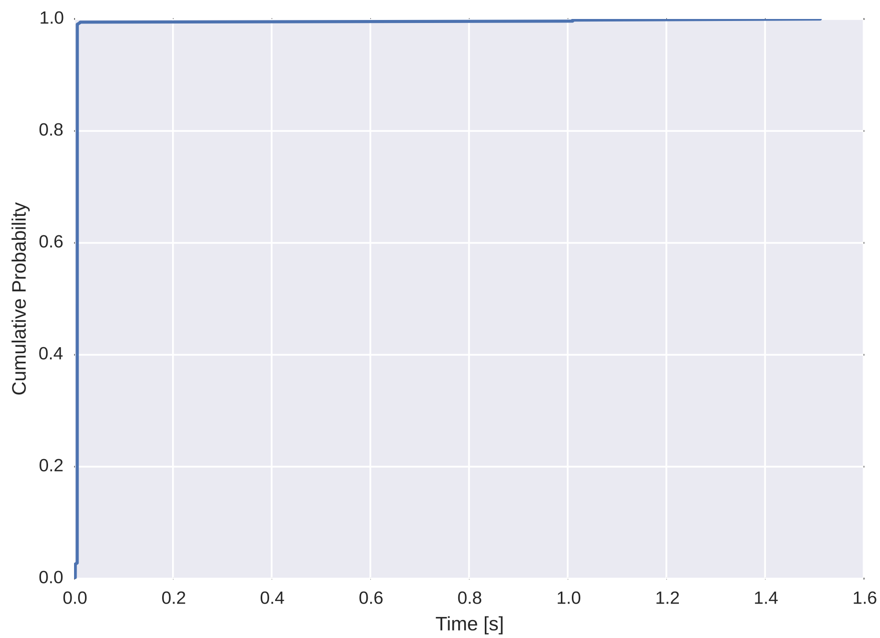


Figure 6.12: ECDF plot of the delta timing values of the Rubber Ducky attack.

non-existing curve. This is caused by the fact, that the keystrokes are sent very fast by the emulating device. This furthermore causes most values being slightly above `0.01`. As of the ECDF, over 99% values are below `0.02` seconds delta between keystrokes.
Regarding the vendor ID and product ID of the Rubber Ducky: The values 03eb and 2401 mark the USB device after purchasing. Those values correspond to the vendor Atmel Inc. and an unknown product ID.

CHAPTER 7

Discussion

As stated in the Introduction of this thesis, the important parts are answering the following research questions:

1. How prevalent are connected USB devices over a certain time frame and how to measure the prevalence at scale (i.e., in an enterprise network)?
2. What is the entropy of a typical user typing a specific pre-defined paragraph that is not known in advance, with particular regards to the timing information?
3. Is there a statistical significant pattern of connected devices that can be used for a per-user whitelisting approach of benign USB devices?
4. Is there a significant difference between the collected data and emulated keystrokes as it would be done by BadUSB or a BadUSB-like device?

Before going into details of this Section, it is important to mention that the data and prototypes are thesis specific and for the sake of answering the research questions of this thesis we opted for the chosen specifics. This means, the user that carried out the long-term study used a Linux based operating system, the enterprise network for data collection is Windows based and the prototypes are implemented in Python and C. However, the underlying method is completely independent from the thesis specifics and could be transferred to other use-cases (collecting other USB data in different environments) as well.

7.1 Research Question 1

This question affects both, Problem 1 and Problem 3. For question one, the usage of USB devices over three months was measured. As of question three, 60 clients were inspected. The inspected the time frame of the USB devices connected on those clients, range from

June 24th, 2014 to March 7th, 2016. This means, the timespan for Problem 3 is almost 2 years.
As shown in the evaluation, over three months on a single computer, only ten (uniquely) different USB devices are plugged in. As for the two years timespan, several different results are shown in the evaluation: Some clients seem to have only a very few different USB devices plugged-in. Those clients, e.g. client 54 has the first plug-in time of its single USB device on March 1st 2016 (time of extraction on the clients: beginning of March 2016). This means, the timeframe is too short to come into consideration as a candidate, since one unique USB device in this short timespan is a realistic value. On the other hand, client 40 has 31 different USB devices plugged-in, over a period of eleven months (April 2015 until March 2016). The top client with the most different USB devices (client 6) is a special case, since the timestamp of every unique device is very close (between December 11th and December 15th 2015). Since the origins of the data (clients) are anonymous, there is no possibility to check on this special data by hindsight.
Table A.1 in the Appendix shows the unique devices per client and the time of first plug-in of any USB device and also the last time of plug-in of any other or the same device respectively. Comparing this data to the data gathered in Problem 1, no conclusion seem to be significant. In Problem 1, ten unique devices over three months are plugged-in, where in Problem 3, several different behaviours are shown: Taking client 6 as an example, where 50 devices are plugged-in over a short period of four days, whereas e.g. the user of client 8 has 27 different devices plugged-in over a period of almost two years (June 2014 to February 2016). Other extreme values are provided by client 11, where three unique devices are plugged-in over a period of roughly two years (June 2014 to February 2016).

7.2 Research Question 2

As shown in Chapter 6, the distribution of timing values is very similar between the 33 participants of the study, excluding a very low number of outliers. To take also the timing values of Problem 1 into account: The statistical median of the timing value between two KEY_DOWN events in Problem 1 is `0.10` seconds, whereas the median for Problem 2 is `0.20` seconds. Looking at Figure 7.1, the median value of `0.10` seconds resides between the lower and upper quartile of most boxplots. This, on the other hand means, that the long term study carried out in Problem 1 produces statistically similar timing values to the timing values of Problem 2.
Two factors limit the impact of the gathered data: The short text as well as the low number of samples. However, the number of participants was limited by the number of employees of SBA Research volunteering to participate in the study. Additionally, the length of the text to type is a trade-off with regards to the user convenience.

7.3 Research Question 3

Due to a high number of different USB devices, a general whitelisting approach is hard to specify. However, there are a few remarkable factors that could lead to basic whitelisting

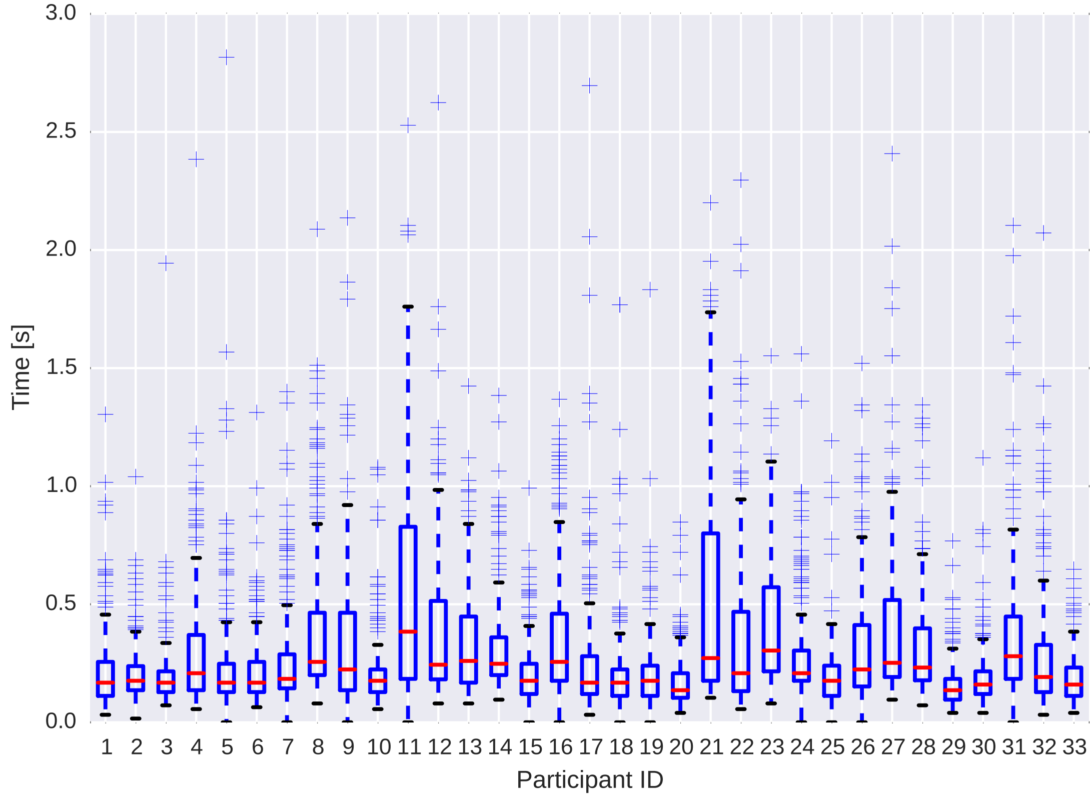


Figure 7.1: Boxplots for typing timings in Problem 2.

approaches. Further insights and outlooks how approaches in that directions could look like, are discussed in depth in Section 7.5 since they are beyond this thesis. Those observed, basic metrics are:

- **Global number of interfaces:** On a global scale, USB devices with more than one interface could be denied in a whitelisting approach. As observed in the evaluation, across all 60 clients, 124 devices offer one interface and one device offers 4. Based on this observation, devices with more than one interface could be denied on a global scale. Users that need to plug-in devices with more than one interface need special clearance. On a usability point of view regarding this small amount of affected users: This is an acceptable downside.
  Possible whitelist:
  Globally denying devices with more than one interface per device (excluding 00 as initial identifier).
  At an example:
  A user plugs a mass storage device in, which was not seen in this network before. The device tries to register two interfaces: At first it tries to register the interface 08 for a mass storage device, but shortly afterwards also 03 for a keyboard. The

device will instantly be blocked.

- **Local time analysis:** As illustrated in Figure 6.7, some clients only have a few unique USB devices (e.g. one), where other clients have a huge amount of different USB devices (e.g. 50). This whitelisting approach is very similar to Research Question 1 (prevalence of USB devices over time). However, it is important to mention, that this specific whitelisting approach only works, if the target client is monitored over a long period of time and is constantly used by one specific user. Considering multiple users using the same client, the behaviour could not be specifically determined for each client. Taking the two examples from above, client 8 and client 11 could be candidates for a whitelisting approach: Calculating the average of USB devices plugged into the client over two years, client 8 experiences one new USB device every two weeks. The average value for client 11 looks very different with roughly one new USB device every year. The high suitability of those clients is their distinction.
  Possible whitelist:
  The client to protect is monitored over a certain timespan. The number of devices connected mark the baseline of the whitelist to enforce. However, the longer the client is monitored, the higher the accuracy of the enforced whitelist will be.
  At an example:
  A client is monitored over 1 year. The user working on the monitored client connects 12 unique USB devices to the client. The average of devices connected is calculated: One unique USB device per month. If the user connects an unusual high number of new, unique devices to the client in month 13, the whitelist is enforced. This enforcement blocks the connection of every new device until unblocked.

- **Local vendor whitelisting:** Another approach is whitelisting USB devices by their vendor. If a non-whitelisted vendor is detected, the new device is blocked. Counting the unique vendor ID's per client, a high number seems to be unique based on the total number of unique USB devices.
  Possible whitelist:
  Restrict the number of allowed vendors from the beginning or monitor the client over time to learn the users' preferences in USB vendors. As for other whitelists, the longer the client is monitored, the better.
  At an example:
  Taking client 60 as an example: As shown in the evaluation, this client has 27 unique USB devices, where 21 do not share the same vendor. Building upon this knowledge, those vendors could be whitelisted and each new vendor denied. This would effectively block rogue devices, but also a high number of false positives.

7.4 Research Question 4

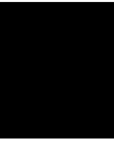
Due to the similarity of keystroke pattern between Problem 1 and Problem 2, both are treated as the same dataset to answer the current research question. Therefore the

reference dataset to the attack is the data gathered in Problem 2. The evaluation has shown, that there exist a significant difference in the timing values of a carried out Rubber Ducky attack and Problem 2. This is proved by the average timing of `0.02` seconds during the attack and an average timing value of `0.30` seconds.
In the case of the BadUSB attack, it seems to be not possible to differentiate between a legit user typing and the maliciously reflashed USB device attacking. The average timing values of `0.37` seconds for the attack and `0.30` seconds for a user typing are too close to each other. Even more, when considering errors in the timekeeping. However, due to the fact that the USB drive used for evaluation was partially destroyed while reflashing, a successful flash probably produce the same timing values as a Rubber Ducky attack. This is even more realistic, when considering that the BadUSB flashing technique uses the Ducky Script to perform the attack.

7.5 Limitations and Future Work

To disguise the presence of a maliciously reflashed USB device, an attacker might use delays between keystrokes that are as near as possible to the timing values collected in this thesis. An attacker could refine the average and median values until the timing values of Problem 1 and Problem 2 are reached. There exist efforts to reach a maximum level of indistinguishability between users and therefore a perfect platform to support a BadUSB-like attack as shown by Monaco et. al [40].
The hak5 forums [48] provides efforts to evade a VID/PID based whitelisting approach. It reflashes a Rubber Ducky with random vendor ID and product ID, making it impossible to deny certain devices. However, a whitelisting approach, by using long-term data as described at the beginning of this section may still be possible.
Another limitation of the approach of this thesis is clearly the strong emphasis on the Linux based operating system used throughout the thesis. This is due to the availability of every source code needed and also the possibility to highly interact with the operating system. Taking Windows as an example, tasks like extending the kernel with loadable modules are not possible to be performed. Furthermore, the USB stacks' implementation is closed source as well as the built-in USB drivers are (as all the other parts of Windows are). Only drivers published by a small amount hardware vendors are open source.
Several limitations are directly connected to the user study: For one, the number of participants is rather low, as well as the text to type is very short. Additionally, the participants in the study are selected within one company (SBA Research), which limits the study to one social graph.

The effectiveness of the proposed whitelists is seen as future work. This effectiveness evaluation, however, needs a lot more data we had no access to. A possible evaluation in this direction could take the USB data of a campus wide network of thousands of participants. This approach would either show the effectiveness or the uselessness.
Further future work on this topic and not covered in this thesis, could be an algorithm that differentiates between a legit user and an attack. The basis of this algorithm could be the timing data provided by this thesis.

CHAPTER 8 

Conclusions

BadUSB and BadUSB-like attacks, like the Rubber Ducky attack, are a severe threat. Therefore we presented in this thesis a large scale data collection and evaluation of USB data: Both kinds of data, BadUSB/BadUSB-like attack data, as well as benign USB data. This benign USB data correlates to over 8 million USB packets. The data was collected by two implemented prototypes over three months and from more than 60 different users in a company-wide environment. Taking a closer look on malicious and benign data, it turned out to be a non-trivial task to defend against BadUSB attacks, but an easy task to detect Rubber Ducky attacks. Therefore we proposed several detection and prevention mechanisms based on this data. On the one hand we described possible whitelisting approaches, which can be applied to a company-wide IT environment. Two of those discussed whitelists are the blocking of devices with multiple interfaces and the blocking of devices per user, after analyzing its device usage behaviour. On the other hand we described a detection mechanism of BadUSB-like attacks, based on the differentiation of keystroke timing. This differentiation is based on typing data collected in a user study from 33 different people typing a pre-defined text. The same text was emulated by a Rubber Ducky to simulate a BadUSB-like attack on a victims computer. Adapted from the correlation of those two sources, there exist a statistical significant difference between a legit user typing on a keyboard and a malicously reflashed USB device (Rubber Ducky) emulating keystrokes. Once infected, the collected company-wide enters the picture: This data can assist an investigator in tracking the origin of an infection and in preventing further infections by the same or similar devices respectively.

APPENDIX A

Appendix

Listing A.1: P1 analysis application.

```
1  #!/usr/bin/env python
2
3  import os
4  import tempfile
5  import subprocess
6  from collections import defaultdict, Counter
7  import sys
8  from os import listdir
9  from os.path import isfile, join
10 import sqlite3
11 import re
12
13 def getKeystrokeTiming(pcap, deviceAddress, tableName, c, conn):
14   captureNumber = re.findall('\d+', tableName)[0]
15
16   # It's time to get the keystrokes for the selected device
17   # Not 00:00[...] because they are key_release
18   keystrokes = subprocess.Popen("tshark -r " + pcap + " -Y \"usb.
       device_address==" + str( deviceAddress ) + "&&usb.transfer_type
       ==URB_INTERRUPT&&usb.capdata!=00:00:00:00:00:00:00:00\"", stdout
       =subprocess.PIPE, shell=True).stdout.read()
19
20   keystrokes = keystrokes.split("\n")
21
22   if len(keystrokes) < 2:
23     return 0
24
25   times = []
26
```

```
27	for ks in keystrokes:
28		temp = ks.split(" ")
29
30		if temp[0] == '':
31			temp = temp[1:]
32		if len( temp ) > 1:
33			temp = temp[1:]
34
35			for t in temp:
36				if t == '':
37					continue
38				else:
39					times.append( float(t) )
40					break
41
42	delta_list = []
43	counter = 0
44	i=0
45	while i<len(times)-1:
46		delta = times[i+1] - times[i]
47
48		# If the delta is greater than 3, the button is either held or the user is afk
49		if delta > 3:
50			i+=1
51			continue
52		else:
53			c.execute("""INSERT INTO """ + tableName + """(CaptureNumber, Timing) VALUES ( ?, ? );""", ( captureNumber, delta ) )
54			i+=1
55
56	conn.commit()
57
58
59 def getAverageTime(pcap, tableName, c, conn):
60	c.execute('CREATE TABLE IF NOT EXISTS ' + tableName + '( Id INTEGER PRIMARY KEY, CaptureNumber INTEGER, Timing REAL );')
61
62	deviceClass = subprocess.Popen("tshark -r " + pcap + " -V -Y usb.bDeviceClass", stdout=subprocess.PIPE, shell=True).stdout.read()
63	deviceClass = deviceClass.split("\n\n")
64
65	directClasses = []
66
67	# If the device class is 0, then class is specified on interface level
68	# If not 0x00: The device has only one usage, that is specified directly
69	for d in deviceClass:
```

```
70      for dc in d.split("\n"):
71        if "bDeviceClass" in dc:
72          ddc = dc.split(" (")[1].split(")")[0]
73          if ddc != "0x00":
74            print "The device class is specified directly: " + str(ddc)
75
76    iClasses = subprocess.Popen("tshark -r " + pcap + " -V -Y usb.
        bInterfaceProtocol", stdout=subprocess.PIPE, shell=True).stdout.
        read()
77    iClasses = iClasses.split("\n\n")
78
79    device_addr = ""
80    hidAddresses = defaultdict(list)
81
82    for frame in iClasses:
83      device_addr = ""
84
85      for line in frame.split("\n"):
86        if "Device:" in line:
87          device_addr = line.split("Device: ")[1]
88
89        # Exclude 0x00 from the protocols (is "None" regarding the
            specs)
90        if "bInterfaceProtocol" in line and "(" in line:
91          hid = line.split(" (")[1].split(")")[0]
92
93          # Check for keyboard
94          if hid == "0x01":
95            if device_addr != "" and device_addr not in hidAddresses[
                hid]:
96              hidAddresses[hid].append( device_addr )
97
98    # Get timing for every keyboard
99    for addr in hidAddresses["0x01"]:
100     getKeystrokeTiming(pcap, str(addr), tableName, c )
101
102
103 # Calculate the pid and vid distribution per PCAP (and the
      combinations vid:pid)
104 # Key is vendor ID, content of every key are the product ids
105 def getPidVID(pcap, captureNumber, c, conn):
106   c.execute('CREATE TABLE IF NOT EXISTS vid_pid( Id INTEGER PRIMARY
        KEY, CaptureNumber INTEGER, Vid CHARACTER(4), Pid CHARACTER(4) )
        ;')
107
108   ids = subprocess.Popen("tshark -r " + pcap + " -V -Y \"usb.idVendor
        ||usb.idProduct\"", stdout=subprocess.PIPE, shell=True).stdout.
        read()
109   ids = ids.split("\n\n")
```

```
110
111   pidVid = defaultdict(list)
112
113   # VID and PID have to be done frame by frame to keep the relation
        of PID and VID
114   for i in ids:
115     vid = []
116     pid = []
117
118     i = i.split("\n")
119
120     for ii in i:
121       if "idVendor" in ii:
122         vid.append( ii.split("(")[1].split(")")[0] )
123       if "idProduct" in ii:
124         pid.append( ii.split("(")[1].split(")")[0] )
125       else:
126         continue
127
128     for v in vid:
129       for p in pid:
130         if p not in pidVid[v]:
131           pidVid[v].append( p )
132
133   for vi, pi in pidVid.iteritems():
134     for pp in pi:
135       c.execute("""INSERT INTO vid_pid(CaptureNumber, Vid, Pid)
          VALUES ( ?, ?, ? );""", ( captureNumber, vi[2:], pp[2:] ) )
136       conn.commit()
137
138
139 # Get the device class distribution
140 # Function returns class per device --> gives the possibility to see
      weird
141 # combinations like {mass storage : HID}
142 def getDeviceClasses(pcap, captureNumber, c, conn):
143   c.execute('CREATE TABLE IF NOT EXISTS classes( Id INTEGER PRIMARY
        KEY, CaptureNumber INTEGER, InterfaceClass TEXT );')
144
145   directClasses = subprocess.Popen("tshark -r " + pcap + " -V -Y \"
        usb.bDeviceProtocol!=0\"", stdout=subprocess.PIPE, shell=True).
        stdout.read()
146   if directClasses != '':
147     print "Classes directly specified"
148     print directClasses
149
150   interfaceClasses = subprocess.Popen("tshark -r " + pcap + " -V -Y
        \"usb.bInterfaceProtocol\"", stdout=subprocess.PIPE, shell=True)
        .stdout.read()
```

```
151    interfaceClasses = interfaceClasses.split("\n\n")
152
153    deviceCounter = 1
154    deviceDict = defaultdict(list)
155
156    # Get interface classes and protocols
157    # E.g. {"HID": ["Mouse", "Keyboard"]}
158    for ic in interfaceClasses:
159      inClass = []
160      inProto = []
161      device = "device" + str(deviceCounter)
162      deviceCounter+=1
163
164      ic = ic.split("\n")
165
166      for i in ic:
167        if "bInterfaceClass" in i:
168          inClass.append( i.split("bInterfaceClass: ")[1].split(" (")
               [0] )
169        if "bInterfaceProtocol" in i:
170          # The protocol is currently unused
171          inProto.append( i.split("bInterfaceProtocol: ")[1].split(" ("
               )[0] )
172        else:
173          continue
174
175      for inc in inClass:
176        deviceDict[device].append( inc )
177
178    for dev, classes in deviceDict.iteritems():
179      temp = ""
180      for cl in classes:
181        if temp == "":
182          temp = str(cl)
183        else:
184          temp = temp + ":" + str(cl)
185
186      c.execute("""INSERT INTO classes(CaptureNumber, InterfaceClass)
           VALUES ( ?, ? );""", ( captureNumber, temp ) )
187      conn.commit()
188
189
190  if __name__ == '__main__':
191    sqlite_file = 'P1_LongTerm.sqlite'
192    conn = sqlite3.connect(sqlite_file)
193    c = conn.cursor()
194
195    pcap_dir = sys.argv[1]
```

```
196    onlyfiles = [ f for f in listdir( pcap_dir ) if isfile(join(
          pcap_dir ,f)) ]
197
198    counter = 1
199
200    for pcap in onlyfiles:
201      pcap = pcap_dir + pcap
202      getAverageTime(pcap, "capture" + str(counter), c, conn)
203      getDeviceClasses(pcap, str(counter), c, conn)
204      getPidVID(pcap, str(counter), c, conn)
205      counter += 1
206
207    conn.commit()
208    conn.close()
```

Listing A.2: P3 anonymizer application.

```
1 #!/usr/bin/env python
2
3 import sqlite3
4 import hashlib
5
6 if __name__ == '__main__':
7   sqlite_file = 'P3_SBA.sqlite'
8   conn = sqlite3.connect(sqlite_file)
9   c = conn.cursor()
10
11   i=1
12
13   while i<=60:
14     for row in c.execute("""SELECT Id, Serial FROM client""" + str(i)
          + """;""").fetchall():
15         serial = ""
16         anon_serial = ""
17
18         if row[1] != "":
19           serial = str(row[1])
20           hash_object = hashlib.sha256( serial )
21           anon_serial = str(hash_object.hexdigest())
22           c.execute("""UPDATE client""" + str(i) + """ SET Serial=?
                WHERE Id=?;""", (anon_serial, row[0]))
23           conn.commit()
24         else:
25           continue
26
27     i+=1
28     conn.commit()
29
30   conn.commit()
```

```
31    conn.close()
```

Listing A.3: Anonymized data for the timeline.

```
1  1403608865 8 4
2  1403608865 8 35
3  1403608865 8 1
4  1424767695 35 11
5  1429020302 40 8
6  1437726707 40 16
7  1449812828 6 19
8  1449812828 6 6
9  1449812828 6 22
10 1449812828 6 7
11 1449812828 6 33
12 1449812828 6 34
13 1449812828 6 21
14 1450080458 6 25
15 1452591901 36 38
16 1452858816 10 32
17 1452863207 40 32
18 1452866601 58 10
19 1453277992 50 4
20 1453302023 26 8
21 1453486408 47 30
22 1453716000 26 5
23 1453813139 36 17
24 1454424091 21 25
25 1454508863 35 23
26 1454590565 4 6
27 1454665756 26 9
28 1454704061 36 14
29 1454918494 21 23
30 1454933197 10 34
31 1455201428 19 33
32 1455208344 19 34
33 1455269589 21 20
34 1455272179 49 20
35 1455612508 34 21
36 1455616509 34 24
37 1455619575 27 15
38 1455625163 34 22
39 1455713717 19 12
40 1455725160 49 29
41 1455725160 38 29
42 1455725281 49 27
43 1455725281 38 27
44 1455736927 7 6
45 1455781974 40 31
```

```
46 1455791801 14 37
47 1455792961 14 30
48 1455805362 37 30
49 1455806344 37 37
50 1455869948 21 24
51 1455869949 21 22
52 1455873068 20 16
53 1455876363 19 38
54 1455879274 20 17
55 1456185139 10 6
56 1456210864 21 14
57 1456214163 21 21
58 1456222103 27 14
59 1456320450 28 15
60 1456360953 53 10
61 1456360953 53 11
62 1456360953 53 12
63 1456393160 14 6
64 1456483149 16 6
65 1456496079 49 28
66 1456496079 38 28
67 1456499945 14 35
68 1456508222 11 6
69 1456529336 29 2
70 1456751462 44 31
71 1456753400 56 2
72 1456754597 26 6
73 1456763430 2 2
74 1456768947 12 31
75 1456808626 60 5
76 1456808626 60 2
77 1456808626 60 6
78 1456808626 60 7
79 1456808626 60 8
80 1456808626 60 9
81 1456817517 14 36
82 1456818969 34 26
83 1456821504 20 18
84 1456827386 30 2
85 1456829548 56 1
86 1456838675 21 19
87 1456839358 48 3
88 1456839358 48 26
89 1456842606 16 36
90 1456842845 28 18
91 1456906056 27 13
92 1456987996 59 6
93 1457100721 51 3
94 1457336511 3 2
```

```
95 1457351966 15 13
```

Listing A.4: Closer look at the anonymized data of the timeline.

```
1 1452591901 36 38
2 1452858816 10 32
3 1452863207 40 32
4 1452866601 58 10
5 1453277992 50 4
6 1453302023 26 8
7 1453486408 47 30
8 1453716000 26 5
9 1453813139 36 17
10 1454424091 21 25
11 1454508863 35 23
12 1454590565 4 6
13 1454665756 26 9
14 1454704061 36 14
15 1454918494 21 23
16 1454933197 10 34
17 1455201428 19 33
18 1455208344 19 34
19 1455269589 21 20
20 1455272179 49 20
21 1455612508 34 21
22 1455616509 34 24
23 1455619575 27 15
24 1455625163 34 22
25 1455713717 19 12
26 1455725160 49 29
27 1455725160 38 29
28 1455725281 49 27
29 1455725281 38 27
30 1455736927 7 6
31 1455781974 40 31
32 1455791801 14 37
33 1455792961 14 30
34 1455805362 37 30
35 1455806344 37 37
36 1455869948 21 24
37 1455869949 21 22
38 1455873068 20 16
39 1455876363 19 38
40 1455879274 20 17
41 1456185139 10 6
42 1456210864 21 14
43 1456214163 21 21
44 1456222103 27 14
45 1456320450 28 15
```

```
46 1456360953 53 10
47 1456360953 53 11
48 1456360953 53 12
49 1456393160 14 6
50 1456483149 16 6
51 1456496079 49 28
52 1456496079 38 28
53 1456499945 14 35
54 1456508222 11 6
55 1456529336 29 2
56 1456751462 44 31
57 1456753400 56 2
58 1456754597 26 6
59 1456763430 2 2
60 1456768947 12 31
61 1456808626 60 5
62 1456808626 60 2
63 1456808626 60 6
64 1456808626 60 7
65 1456808626 60 8
66 1456808626 60 9
67 1456817517 14 36
68 1456818969 34 26
69 1456821504 20 18
70 1456827386 30 2
71 1456829548 56 1
72 1456838675 21 19
73 1456839358 48 3
74 1456839358 48 26
75 1456842606 16 36
76 1456842845 28 18
77 1456906056 27 13
78 1456987996 59 6
79 1457100721 51 3
80 1457336511 3 2
81 1457351966 15 13
```

Listing A.5: BadUSB Ducky Script.

```
1 DELAY 3000
2 GUI r
3 DELAY 500
4 STRING notepad
5 DELAY 500
6 ENTER
7 DELAY 750
8 STRING Hello World!!!
9 ENTER
```

Listing A.6: Rubber Ducky Ducky Script.

```
1 DELAY 500
2 ALT F2
3 DELAY 500
4 STRING mousepad
5 ENTER
6 DELAY 500
7 STRING Bacon ipsum dolor amet turducken pork chop bresaola fatback
    jowl, ribeye meatloaf cupim pork belly short ribs short loin. Pig
    pork loin biltong boudin meatball pastrami picanha landjaeger
    swine. Turducken leberkas pork chop landjaeger porchetta ribeye
    short ribs sirloin filet mignon meatloaf ground round brisket
    spare ribs. Pork chop doner shoulder bacon. Andouille sirloin cow,
     flank turducken rump landjaeger picanha. T-bone ball tip swine,
    beef chuck kevin ribeye pastrami frankfurter biltong sirloin
    alcatra.
8 ENTER
```

Client	Number of devices	Date of first device / Date of last device
Client 02	2	1452675125/1456763430
Client 03	3	1451994189/1457336511
Client 04	2	1454510958/1454590565
Client 06	50	1449812828/1450080458
Client 07	3	1454679141/1455736927
Client 08	27	1403608865/1455871066
Client 10	8	1452601837/1456323236
Client 11	3	1403537537/1456508225
Client 12	2	1456768947/1456859691
Client 14	10	1455791801/1456831927
Client 15	1	1457351966/1457351966
Client 16	4	1456483149/1456848510
Client 17	2	1457339436/1457339476
Client 18	2	1455365787/1456925699
Client 19	12	1450283300/1456489464
Client 20	5	1455873068/1456821504
Client 21	36	1453813432/1456838675
Client 24	1	1456735560/1456735560
Client 25	2	1456998821/1457438593
Client 26	13	1443814351/1456754597
Client 27	11	1454596435/1456906306
Client 28	3	1456320450/1456842845
Client 29	3	1455818890/1456529336
Client 30	7	1453716489/1456827386
Client 31	1	1457101292/1457101292
Client 32	1	1455784972/1455784972
Client 33	9	1453797234/1456417210
Client 34	10	1433755484/1456818990
Client 35	9	1424767695/1457077909
Client 36	10	1386014297/1455617840
Client 37	4	1455805362/1457352766
Client 38	3	1455725160/1456496079
Client 39	1	1456823447/1456823447
Client 40	31	1429020302/1456570764
Client 41	3	1450371320/1456838351
Client 42	3	1449042704/1456815862
Client 43	3	1454160868/1456993754
Client 44	1	1456751462/1456751462
Client 45	1	1456395710/1456395710
Client 46	1	1457344453/1457344453
Client 47	7	1453486408/1456837266

Client 48	3	1456839358/1456839363
Client 49	15	1455200971/1456914222
Client 50	1	1453277992/1453277992
Client 51	2	1457100721/1457100728
Client 52	10	1452688706/1456907690
Client 53	11	1456360953/1456360953
Client 54	1	1456829604/1456829604
Client 55	3	1440426694/1456992438
Client 56	2	1456753400/1456829548
Client 57	4	1456329197/1456329197
Client 58	5	1452363529/1456259700
Client 59	4	1454583228/1456987996
Client 60	27	1456808626/1456808626

Table A.1: Number of devices per client with first and last plug time

List of Figures

2.1 Low level message flow in USB [52]. 6
2.2 USB packet flow during a reflashing attack. 6

4.1 Design of the first prototype. 18
4.2 Design of the second prototype. 19

5.1 Structure of database for Problem 1. 23
5.2 Structure of a sample table for Problem 2. 24
5.3 Structure of a sample table for Problem 3. 24

6.1 Distribution of the time delta values of Problem 1. 28
6.2 Empirical cummulative distribution of the timing values of Problem 1. 29
6.3 Vendor ID and product ID distribution of Problem 1. 30
6.4 (a) Setup of Problem 2, (b) close up shot of the setup. 31
6.5 Distribution of 33 participants within Problem 2. 32
6.6 Distribution of the vendor ID and product ID of Problem 3. 33
6.7 Number of USB devices of each client. 34
6.8 Appearance of unique devices across all clients. 35
6.9 Tracking USB devices in time and space across data set 3. 36
6.10 Taking a closer look at the timeline. 37
6.11 ECDF plot of the delta timing values of the BadUSB attack. 39
6.12 ECDF plot of the delta timing values of the Rubber Ducky attack. 39

7.1 Boxplots for typing timings in Problem 2. 43

List of Tables

6.1 Timeline of device 6 across the collected clients. 38

A.1 Number of devices per client with first and last plug time 61

Bibliography

[1] Hak5. https://hak5.org/. (Accessed: 03.10.2017).

[2] Payload OSX Root Backdoor. https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Payload---OSX-Root-Backdoor, April 2013. (Accessed: 03.10.2017).

[3] Android Studio Developers. https://developer.android.com/sdk/index.html, 2014. (Accessed: 05.10.2017).

[4] Duckyscript. https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Duckyscript, July 2014. (Accessed: 03.10.2017).

[5] blackhat. https://www.blackhat.com/, 2015. (Accessed: 05.10.2017).

[6] How To Be Sicher From USB Attacks. https://www.gdata.de/de-usb-keyboard-guard, 2015. (Accessed: 06.10.2017).

[7] Shmoocon. http://shmoocon.org/, 2015. (Accessed: 05.10.2017).

[8] USB-Rubber-Ducky Payloads. https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Payloads, October 2015. (Accessed: 03.10.2017).

[9] Livia CF Araujo, Luiz HR Sucupira, Miguel Gustavo Lizarraga, Lee Luan Ling, and Joao Baptista T Yabu-Uti. User authentication through typing biometrics features. *IEEE Transactions on Signal Processing*, 53(2):851–855, 2005.

[10] Paul Joseph Boehler. Apparatus and method for externally initiating automatic execution of media placed in basic removable disc drives, 2001. US Patent 6,282,710.

[11] Harlan Carvey. The windows registry as a forensic resource. *Digital Investigation*, 2(3):201–205, 2005.

[12] K Chen. Reversing and exploiting an apple firmware update. *Black Hat*, 2009.

[13] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf, December 2015. (Accessed: 15.10.2017).

[14] Paul Ducklin. BadBIOS is back – this time on your TV. https://nakedsecurity.sophos.com/2015/11/16/badbios-is-back-this-time-on-your-tv/, November 2015. (Accessed: 09.10.2017).

[15] Compaq et al. Universal Serial Bus Specification Revision 2.0. http://www.usb.org/developers/docs/usb20_docs/usb_20_100617.zip, April 2000. (Accessed: 15.10.2017).

[16] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[17] Stephen Gowdy. List of USB ID's. http://www.linux-usb.org/usb.ids, March 2016. (Accessed: 15.10.2017).

[18] Martin T Hagan, Howard B Demuth, Mark H Beale, et al. *Neural network design*. Pws Pub. Boston, 1996.

[19] Hak5. USB Rubber Ducky. https://github.com/hak5darren/USB-Rubber-Ducky, 2015. (Accessed: 03.10.2017).

[20] Marti A. Hearst, Susan T Dumais, Edgar Osman, John Platt, and Bernhard Scholkopf. Support vector machines. *IEEE Intelligent Systems and their Applications*, 13(4):18–28, 1998.

[21] Nicole Ibrahim. USB Device Research – USB Enumeration in Windows. http://nicoleibrahim.com/part-3-usb-device-research-windows-registry-enumerations/, October 2013. (Accessed: 08.10.2017).

[22] Imation Corp. Ironkey Secure USB Devices Protect Against BadUSB Malware. http://www.ironkey.com/en-US/solutions/protect-against-badusb.html, 2015. (Accessed: 02.10.2017).

[23] Texas Instruments. Vids, pids, and firmware: Design decisions when using ti usb device controllers, 2003.

[24] Samy Kamkar. USBdriveby. http://samy.pl/usbdriveby/, 2014. (Accessed: 02.10.2017).

[25] M Karnan, M Akila, and N Krishnaraj. Biometric personal authentication using keystroke dynamics: A review. *Applied Soft Computing*, 11(2):1565–1573, 2011.

[26] Ariane Keller. Kernel Space - User Space Interfaces. http://elk.informatik.hs-augsburg.de/tmp/elinux/Kernel_und_Treiber/kernel_user_space_howto.html, July 2008. (Accessed: 03.10.2017).

[27] James M Keller, Michael R Gray, and James A Givens. A fuzzy k-nearest neighbor algorithm. *IEEE Transactions on Systems, Man and Cybernetics*, (4):580–585, 1985.

[28] Greg Kroah-Hartman. udev–a userspace implementation of devfs. In *Proceedings of the Linux Symposium*, pages 263–271. Citeseer, 2003.

[29] David Kushner. The real story of stuxnet. *IEEE Spectrum*, 50(3):48–53, 2013.

[30] Mario Lamberger and Florian Mendel. Higher-order differential attack on reduced sha-256. *IACR Cryptology ePrint Archive*, 2011:37, 2011.

[31] Terran Lane and Carla E Brodley. An application of machine learning to anomaly detection. In *Proceedings of the 20th National Information Systems Security Conference*, volume 377, pages 366–380. Baltimore, USA, 1997.

[32] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[33] Robert Love. *Linux Kernel Development Second Edition.* Novell Press: Sams Publishing, 2005.

[34] Chen Change Loy, W Lai, and C Lim. Development of a pressure-based typing biometrics user authentication system. *ASEAN Virtual Instrumentation Applications Contest Submission*, 2005.

[35] Victor Chileshe Luo. Tracing usb device artefacts on windows xp operating system for forensic purpose. In *Australian Digital Forensics Conference*, page 23, 2007.

[36] Jacob Maskiewicz, Benjamin Ellis, James Mouradian, and Hovav Shacham. Mouse trap: exploiting firmware updates in usb peripherals. In *Proceedings of the 8th USENIX conference on Offensive Technologies*, pages 12–12. USENIX Association, 2014.

[37] Jamie McQuaid. How to Analyze USB Device History in Windows. https://www.magnetforensics.com/computer-forensics/how-to-analyze-usb-device-history-in-windows/, July 2014. (Accessed: 03.10.2017).

[38] Microsoft. System Center Configuration Manager Datasheet. http://download.microsoft.com/download/5/D/B/5DBEBA38-8D5D-4119-B2E8-B8369B74BF43/system_center_configuration_manager_and_microsoft_intune_datasheet.pdf. (Accessed: 03.10.2017).

[39] Microsoft. USB Device Registry Entries. https://msdn.microsoft.com/en-us/library/windows/hardware/jj649944(v=vs.85).aspx. (Accessed: 15.10.2017).

[40] John V Monaco, Md Liakat Ali, and Charles C Tappert. Spoofing key-press latencies with a generative keystroke dynamics model. In *IEEE 7th International Conference on Biometrics Theory, Applications and Systems (BTAS)*, pages 1–8. IEEE, 2015.

[41] Paul Mueller and Babak Yadegari. The stuxnet worm. *Compuer science department University of Arizona, http://www.cs.arizona.edu/~collberg/Teaching/466-566/2012/Resources/presentations/2012/topic9-final/report.pdf*, 2012.

[42] Collin Mulliner and Benjamin Michéle. Read it twice! a mass-storage-based tocttou attack. In *WOOT*, pages 105–112, 2012.

[43] Pablo Neira-Ayuso, Rafael M Gasca, and Laurent Lefevre. Communicating between the kernel and user-space in linux using netlink sockets. *Software: Practice and Experience*, 40(9):797–810, 2010.

[44] Karsten Nohl and Jakob Lell. BadAndroid . https://opensource.srlabs.de/attachments/download/107/BadAndroid-v0.2.zip, 2014. (Accessed: 03.10.2017).

[45] Karsten Nohl and Jakob Lell. BadBIOS . https://srlabs.de/blog/wp-content/uploads/2014/11/SRLabs-BadUSB-Pacsec-v2.pdf, 2014. (Accessed: 03.10.2017).

[46] Karsten Nohl and Jakob Lell. BadUSB - On Accessories that turn Evil. https://srlabs.de/badusb/, 2014. (Accessed: 03.10.2017).

[47] Linux Kernel Organization. USBMon Kernel Documentation. https://www.kernel.org/doc/Documentation/usb/usbmon.txt, May 2015. (Accessed: 15.10.2017).

[48] overwraith. VID_PID_SWAPPER.exe easily swap random VID/PID numbers. https://forums.hak5.org/index.php?/topic/29804-infoexecutablevid-pid-swapperexe-easily-swap-random-\vidpid-numbers/, July 2013. (Accessed: 15.10.2017).

[49] Mike Owens and Grant Allen. *SQLite*. Springer, 2010.

[50] Inverse Path. USB armory – Open Source Flash-Drive Sized Computer. https://inversepath.com/usbarmory, 2015. (Accessed: 02.10.2017).

[51] Alen Peacock, Xian Ke, and Matthew Wilkerson. Typing patterns: A key to user identification. *IEEE Security & Privacy*, (5):40–47, 2004.

[52] Craig Peacock. USB Protocols. http://www.beyondlogic.org/usbnutshell/usb3.shtml, 2010. (Accessed: 15.10.2017).

[53] Niels Provos and David Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.

[54] D Raghu, CH Raja Jacob, and YVKD Bhavani. Neural network based authentication and verification for web based key stroke dynamics. *dimension*, 2:1, 2011.

[55] Kenneth Revett, Florin Gorunescu, Marina Gorunescu, Marius Ene, Sergio Magalhaes, and Henrique Santos. A machine learning approach to keystroke dynamics based user authentication. *International Journal of Electronic Security and Digital Forensics*, 1(1):55–70, 2007.

[56] Chris Sanders. *Practical packet analysis: Using Wireshark to solve real-world network problems.* No Starch Press, 2011.

[57] Juergen Schmidt. Kostenloses G-Data-Tool schuetzt vor BadUSB-Angriffen. http://www.heise.de/security/meldung/Kostenloses-G-Data-Tool-schuetzt-vor-BadUSB-Angriffen-2329545.html, September 2014. (Accessed: 06.10.2017).

[58] Sergej Schumilo, Ralf Spenneberg, OpenSource Training Ralf Spenneberg, and Hendrik Schwartke. Don't trust your usb! how to find bugs in usb device drivers. *Blackhat EU*, 2014.

[59] Nir Sofer. USBDeview v2.52. http://www.nirsoft.net/utils/usb_devices_view.html, 2016. (Accessed: 03.10.2017).

[60] Dawn Song, Peter Venable, and Adrian Perrig. User recognition by keystroke latency pattern analysis. 19, 1997.

[61] Andy Spruill and Chris Pavan. Tackling the u3 trend with computer forensics. *Digital Investigation*, 4(1):7–12, 2007.

[62] Paul Stoffregen and Robin Coon. Teensy USB Development Board. https://www.pjrc.com/teensy/, 2015. (Accessed: 02.10.2017).

[63] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery.* Pearson Education, 2007.

[64] OS TEZER. Sqlite vs mysql vs postgresql: A comparison of relational database management systems. *Digital Ocean*, 2014.

[65] Dave Tian, Adam Bates, and Kevin Butler. Defending against malicious usb firmware with goodusb. In *31st Annual Computer Security Applications Conference*, 2015.

[66] USB Device Working Group. Universal Serial Bus Still Image Capture Device Definition. http://www.usb.org/developers/docs/devclass_docs/usb_still_img10.zip, July 2000. (Accessed: 15.10.2017).

[67] USB Implementers Forum, Inc. USB Class Codes. http://www.usb.org/developers/defined_class, January 2016. (Accessed: 15.10.2017).

[68] Zhaohui Wang and Angelos Stavrou. Exploiting smart-phone usb connectivity for fun and profit. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 357–366. ACM, 2010.

[69] Ian Wienand. What actually happens when you plug in a USB device? https://www.technovelty.org/linux/what-actually-happens-when-you-plug-in-a-usb-device.html, 2007. (Accessed: 15.10.2017).

[70] Brandon Wilson. Making BadUSB Work for You - DerbyCon. https://adamcaudill.com/2014/10/02/making-badusb-work-for-you-derbycon/, October 2014. (Accessed: 03.10.2017).

[71] Lih Wern Wong. Forensic analysis of the windows registry. *Forensic Focus*, 1, 2007.

[72] Bo Yang, Dengguo Feng, Yu Qin, Yingjun Zhang, and Weijin Wang. *TMSUI: A Trust Management Scheme of USB Storage Devices for Industrial Control Systems*, pages 152–168. Springer International Publishing, 2016.

[73] JR Young and RW Hammon. Method and apparatus for verifying an individual's identity. *United States Patent US*, 4(805):222, 1989.

[74] Enzhe Yu and Sungzoon Cho. Ga-svm wrapper approach for feature subset selection in keystroke dynamics identity verification. In *Proceedings of the International Joint Conference on Neural Networks*, volume 3, pages 2253–2257. IEEE, 2003.

[75] Fei Yu and Yulei Huang. An overview of study of passowrd cracking. In *International Conference on Computer Science and Mechanical Automation (CSMA), 2015*, pages 25–29. IEEE, 2015.

[76] Pete Zaitcev. The usbmon: Usb monitoring framework. In *Linux Symposium*, page 291, 2005.