FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A HARDWARE-ASSISTED INSIDER THREAT DETECTION AND PREVENTION FRAMEWORK

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Kyle Denney

2019

To: Dean John L. Volakis
College of Engineering and Computing

This thesis, written by Kyle Denney, and entitled A Hardware-Assisted Insider Threat Detection and Prevention Framework, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Kemal Akkaya

Alexander Perez-Pons

Michael Vai

A. Selcuk Uluagac, Major Professor

Date of Defense: June 28, 2019

The thesis of Kyle Denney is approved.

Dean John L. Volakis
College of Engineering and Computing

Andrés G. Gil
Vice President for Research and Economic Development
and Dean of the University Graduate School

Florida International University, 2019

DEDICATION

To the sunshine: I could not have done this without you.

ACKNOWLEDGMENTS

I would like to thank everyone at the Cyber-Physical Systems Security Lab (CSL) who have helped me learn and grow over my time here at Florida International University. Professors, faculty, advisors, and of course my peers, have acted as excellent mentors for me development in both knowledge and wisdom. I will take all that I have learned with me in my travels. This work is partially supported by the US National Science Foundation (Awards: NSF-CAREER-CNS-1453647, NSF-1663051) and Florida Center for Cybersecurity's Capacity Building Program.

ABSTRACT OF THE THESIS

A HARDWARE-ASSISTED INSIDER THREAT DETECTION AND PREVENTION FRAMEWORK

by

Kyle Denney

Florida International University, 2019

Miami, Florida

Professor A. Selcuk Uluagac, Major Professor

Today, the USB protocol is among the most widely used protocols. However, the mass-proliferation of USB has led to a threat vector wherein USB devices are assumed innocent, leaving computers open to an attack. Malicious USB devices can disguise themselves as benign devices to insert malicious commands to connected end devices. A rogue device appears benign to the average OS, requiring advanced detection schemes to identify malicious devices. However, using system-level hooks, advanced threats may subvert OS-reliant detection schemes. This thesis showcases USB-Watch, a hardware-based USB threat detection framework. The hardware can collect live USB traffic before the data can be altered in a corrupted OS. Behavioral analysis of USB devices allows for a generalizable anomaly detection classifier in hardware that can detect abnormal behavior from USB devices. The framework tested achieves an ROC AUC of 0.99 against a testbed of live USB devices.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF FIGURES

1. INTRODUCTION

USB devices are among the easiest to adopt into a computing system thanks to the number of devices which support USB protocols and the protocol's plug-and-play nature. However, this same ease of use allows for a wide attack vector for malicious parties to steal or modify sensitive data in computing systems [NL14]. For this reason, sensitive computing environments (e.g., government buildings, military bases, research labs, etc.) limit the use of USB devices in order to remove the threat of malicious USB devices [Duc18].

Methods to detect and prevent USB-based attacks can be categorized as static analysis or dynamic analysis. Static analysis methods detect potential malicious activity before the USB is inserted. While there are a handful of these methods in use, they are not sufficient. Should an attacker know which methods are implemented, they could circumvent them. Dynamic analysis methods are needed in the case of such attacks. Dynamic analysis methods operate once a USB device has already been inserted.

With this thesis, we utilize a hardware-based dynamic analysis framework, called *USB-Watch*, to detect and prevent USB-based insider threats. This framework aims to analyze unknown USB devices introduced to a computing environment and, through the use of machine learning, determine the behavior of the device before it can potentially cause harm to the computing environment.

All interconnected devices need a data protocol in which to effectively communicate. Likewise, each device needs a piece of hardware (commonly referred to as a 'bus') in which to transmit and receive communication utilizing such a data protocol. The USB-

Watch framework aims to improve upon these hardware buses to create a "smart bus" which can determine the nature of an unknown, connected USB device communicating with the host machine. If the hardware bus determines the USB device is malicious or abnormal, the bus can cease communication with the device until the user prompts the bus to reestablish communication.

Also considered in this thesis is the possibility of advanced adversaries which aim to subvert naive machine learning approaches. Such an adversarial model is simulated using live devices on our machine learning models. We then take the results of this simulation to motivate the necessity of specializing certain anomaly detection models and avoid generalization when an adversary can abuse this generalization to subvert detection.

1.1. Contributions

The contributions of this work are as follows:

- *USB-Watch*: We propose a hardware-based USB detection framework called USB-Watch to dynamically detect and prevent USB devices from injecting malicious keystrokes in a target computing environment.
- *Effective classification*: We perform an in-depth classification and feature analysis of typing dynamics to distinguish between benign and malicious typing dynamics with an ROC curve of 0.89.
- *Generalization of the design*: We extend the initial work done with keyboard-based devices to provide a generalized approach to detecting USB devices. With an improved dataset and training model, we are able to obtain a final ROC curve of 0.99.

2. BACKGROUND INFORMATION

In this chapter, we outline important background information which is needed for the remainder of this thesis. First, we overview the USB protocol and how human interface devices (e.g., USB keyboards and mice) interact with a computer. From there, we highlight how a malicious USB device can utilize these functionality to perform HID injection attacks.

2.1. USB Protocol

In this thesis, we utilize a hardware-based mechanism to collect incoming USB data; hence, we overview the basics of the USB protocol and how the operating system handles incoming USB data.

The USB protocol operates in a master-slave fashion [Cun17]. First, a device connects to a computing system via a USB host controller. The host controller (master) requests all data from the USB device (slave). A request for data is periodically sent to the USB device and, if the device has any, the data is placed on a USB buffer. A system interrupt is performed, the host controller reads the data on the buffer, and the communication process restarts from there.

Each USB transaction contains a token packet, data packet, and status packet. The token packet establishes what type of data flow (i.e., read, write, etc.) will occur. The data packet contains the actual USB data. The status packet reports if the prior packets were received correctly or if the end device is currently stalled or unable to receive packets.

2.1.1. USB Human Interface Device Reports

To ease functionality for users, technology commonly referred to a "plug and play" was developed. An unknown USB device has drivers loaded onto this device. If a computer does not know what a device does, it requests drivers from the USB device allowing for the device to operate.

Some of the USB devices the detection framework in this thesis attempts to distinguish are known as human interface devices (HIDs). Human Interface Devices (HIDs) are a subclass of USB devices which are designed for human input (e.g., keyboards, mice, gaming controllers, etc.) [Adm18]. Since HID functionality is built into every computer, it does not require the installation of drivers. This is part of why we chose to examine these devices specifically. They are already granted functionality and assumed to be benign.

To communicate, the host machine will periodically request input information from the HID. The HID will then produce a HID report and encapsulate it in a USB packet. The report format for a standard keyboard is shown in Figure 2.1a. As shown, there is a field in a keyboard report for a total of six concurrent key presses plus any modifier keys (i.e., SHIFT, CTRL, ALT, etc.). Note that the reserved bit is ignored in the reports. When a user presses a key, the HID report will continuously include the key being pressed until the user subsequently releases the key. Similarly, Figure 2.1b shows the report format for a standard USB mouse. The first byte describes if a button is pressed (with the remaining 5 bits reserved for no use). The next two bytes are for the X and Y axis, respectively. Finally, the mouse wheel information is stored in the fourth byte. In this thesis, we use these HID reports to collect data for our detection framework.

(a) Keyboard HID report structure.

(b) Mouse HID report structure

Figure 2.1: USB Human Interface Device (HID) report examples.

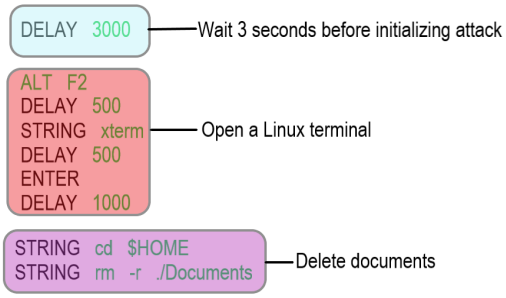


Figure 2.2: Rubber Ducky attack example.

2.1.2. USB HID Injection Attacks

A common attack with embedded USB devices we refer to as HID injection attacks [NL14]. In these attacks, the embedded device mimics a USB HID to inject malicious commands. Two of the most common devices used to implement HID injection attacks are the *Rubber Ducky* [Hak] and the *BadUSB* [Smi16]. Both come with a pre-defined language for a user to easily create an attack script to embed in his/her USB device. First, the malicious actor designs the attack s/he wants performed. The attacker determines which commands are needed to carry out the attack, for instance opening a terminal and deleting specific files or directories (Figure 2.2). From there, the attacker writes a script which iterates each HID command as text strings. To make sure the command is performed at the correct moment, the attacker introduces delays between major events (e.g., opening terminal and sequentially typing in command). With the script created, the malicious actor must compile the malicious commands ("payload") into an embedded USB device disguised as a common device (e.g., flash-drive, keyboard, etc.) [Hak].

To perform the attack, the attacker must have the malicious USB device inserted into the target computer. There are two common methods to do so: (1) a malicious insider plugs the device into the target machine when the user walks away or (2) the malicious device is dropped near the target premises, leading to an unwitting insider mistakenly plugging the device into a target machine [Red17, Bur16]. Once inserted, the device begins performing the attack by sending a HID report with the current command in the payload. To the computer, this HID report simply looks like it comes from benign devices. To prevent onlooker detection, intricate attacks may wait for onlookers to disappear by delaying for an arbitrary time or waiting for the computer to enter rest-mode. Since the payload is embedded in a USB device, this attack is difficult to detect through normal means, as a common anti-virus cannot simply scan the file to determine if it is malware.

2.1.3. USB Command Injection Attacks

We can further extrapolate the HID injection attack described above to any generic USB device that is capable of sending commands via a USB interface, such as communication "dongles" (i.e., devices which enable Bluetooth, ZigBee communications). Such a device may be infected with malicious hardware designed to send falsified or malicious commands to the host computer. We call this form of attack "command injection attacks." Through generalizing features across multiple device types, USB-Watch can detect malicious activity across both HID devices and any generic USB device which inputs commands via the USB protocol.

2.2. Machine Learning and Anomaly Detection

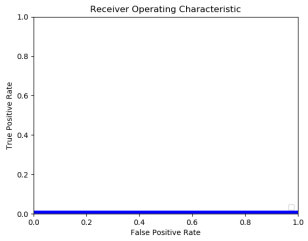
2.2.1. Anomaly Detection

In this thesis, we attempt to detect malicious USB devices by analyzing their typing behaviors. To do this, we utilize machine learning algorithms to classify benign and malicious samples. Here we introduce what methodology and metrics we will be using in this thesis to evaluate our anomaly detection model.

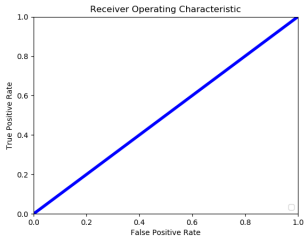
2.2.2. ROC Curves

Receiver operating characteristic (ROC) curves are the de-facto tool to analyze the efficacy of an anomaly detection model [MR04]. At a high level, this curve scores the model on its ability to differentiate proper signals (true-positives) from noise (false-positives) in the sampled data.

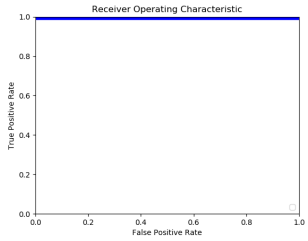
Any anomaly detection goal, and the overall goal of this thesis, is to maximize the rate that true-positives are detected and minimize the rate that false-positives occur. Indeed, it is good for a model to correctly label true-positives 100% of the time, but that result is meaningless if the model also labels noise as positives 100% of the time. Our metric for determining the worth of our model will be evaluating the area under the curve (AUC) of the ROC curve produced from the classification model. An ideal model produces an AUC of 1.00, while a model which completely fails produces an AUC of 0.00. Figure 2.3a shows an ROC curve where the predictive model completely fails to predict the target class. Figure 2.3b shows an ROC curve where the the model does little better than a coin-flip, wherein the model guesses the predictive class 50% of the time. The ideal model is shown in Figure 2.3c, where the model can predict the class everytime while also producing no errors.



(a) ROC Curve with an AUC of 0.0



(b) ROC Curve with an AUC of 0.5



(c) ROC Curve with an AUC of 1.0

Figure 2.3: Example ROC curves of increasing performance.

In this thesis, we prioritize our work on minimizing the rate of false-positives over maximizing true-positive detection rates. We choose this because no one will implement an intrusion detection model if it repeatedly claims that benign data is malicious in nature.

2.2.3. Classification Algorithms

To produce a model that best fits our data, we analyze a number of common machine learning algorithms and evaluate which performs the best for prediction. We give a brief overview of the algorithms used in this thesis including benefits and drawbacks to using them with our data.

Support Vector Machines – Support vector machines attempt to plot the training data in a means such that there is a split between different classed data. Through the use of a kernel algorithm, the data is mapped to planes in high-dimensional space so that there is good separation. Through careful pre-processing and data analysis, an SVM can be used

quite effectually to construct a generalized classifier. This entails constructing features and choosing a kernel so that the data is easily separable.

It is beneficial to use an SVM classifier when optimizing the kernel algorithm to fit the data is possible. This allows the classifier to avoid local minima and find the ideal solution quickly. However, it can still be the case that the data is not separable and the SVM will not converge in the first place.

Decision Trees – This algorithm goes through each feature and splits the data at various intervals in the data to form a tree until all the leaf nodes have 1 or 2 data points. From there, new data is classified depending on which node it falls under in the tree. To improve upon the optimization, decision trees use entropy and information gain to determine which feature to split on first. The features with the highest information gain will give the most data to split upon.

Decision trees are one of the easier classification algorithms to utilize as its core concept of splitting on features is easy to follow and replicate. It also allows the user to understand what the final classification model has learned by looking at which features the model has chosen to split on.

Random Forests – Decision trees have a tendency to overfit the data. To improve upon that, Random Forests are an ensemble learning approach to decision trees. The idea is to create many decision trees with slight variance in their splits. From there, the aggregate classification from all the decision trees is used to classify new data.

The obvious advantage random forest classification has over decision trees is the ability to aggregate multiple decision trees together. As the number of trees with slight variance in splits increase, the lower there is a chance for error. However, this incolves training and fitting multiple classifiers and may not be worth it if there is little decrease in error rate.

K-Nearest Neighbors – This algorithm maps the features to axes in Euclidean space.

From there, the data is given coordinates respective to these axes. Any new data is classified by taking a distance calculation from k neighboring points. The classes of the data closest to the new data is summed and the class with the highest total is given to the new data point.

k-Nearest Neighbors classification surprisingly robust given its simple design. There is no training stage like other classification models and the algorithm can effectively cluster large datasets to find patterns. However, as the dataset increases, the computational efficiency of the algorithm starts to decrease as the algorithm must compare new data entries with all existing points in the data set.

Naive Bayes – Naive Bayes classifiers utilize Bayes' rule in statistics and a 'naive' assumption that each feature extracted is conditionally independent as related to the classification. Using these two assumptions allows the classifier to predict a class for a given entry by calculating probability of the input's feature set given one of the classes.

The main benefit of using a Naive Bayesian classifier is that it needs significantly less training data than other classifiers to produce a model. However, the assumption that each feature is conditionally independent is not likely (hence 'naive'). Given that our dataset include a number of correlated features on USB packet timings, it is unlikely they are independent.

3. RELATED WORK

In this chapter, we detail related work in the field of malicious USB detection. Upon describing these works, we highlight how our work is different from or improves upon these prior works.

3.1. Embedded Device Threats

The rise of small, embedded devices (i.e., USB devices) brought a new attack vector through these devices acting maliciously when inserted to a computing system [Smi16, NL14, Mam14]. Traditional intrusion detection systems are not suited to detect these threats as, to the computing system, the behavior appears to come from a legitimate user. This requires new solutions to be added to effectively detect these embedded device threat vectors. We overview two categories of detection models: static and dynamic.

3.2. Static Detection Methods

Static analysis methods aim to analyze a potential threat by examining it before execution [MKK07]. When detecting threats from embedded devices, static analysis methods include: disallowing unregistered devices from communicating [SSS17], require devices to request functionality permission [TSB+16], or simply disabling unnecessary USB ports [SSS17]. Assuming a trusted device contains maliciously embedded circuitry [RR],

these static frameworks are not enough to detect and eliminate all threats from embedded devices.

3.3. Dynamic Detection Methods

Dynamic analysis methods, on the other hand, analyze the potential threat as it is operating and examines the performance behavior of the device. Dynamic analysis today tends to involve the use of machine learning algorithms and classification models [Dal16, MW18, RGC18]. In the case of malicious threat detection, this entails using binary classification schemes to differentiate benign and malicious behavior on the system. Current approaches for dynamic threat detection in embedded devices use software on the host system to collect and process data. It has been shown that it is possible for advanced threats to spoof or alter software-based collection approaches by altering OS-level code [LBAU17]. Transferring the dynamic analysis to hardware placed between an unknown embedded device and the host system would ensure that the data collection is unalterable through these means.

3.4. Differences from Existing Work

Our approach aims to develop a *dynamic* detection framework to detect and prevent insider threats through embedded devices. Our proposed approach differs from prior work through three distinct characteristics stemmed from the use of a hardware mechanism. First, an advanced threat to the computing system may leave software-based detection approaches vulnerable to spoofing. The use of a hardware mechanism between the embedded device and the internal computing system ensures the capturing of raw, unaltered behavior of the device – allowing the detection of any malicious behavior performed by the embedded device. Second, any software-based approach to intrusion detection con-

sumes resources of the computing system. Segmenting the intrusion detection into dedicated hardware removes computation cost that would otherwise needed to be performed on the host machine. Finally, assuming a malicious device is embedded in an authenticated USB device, the attack can bypass static systems such as [TSB+16] whereas our dynamic approach can still detect. Hence, this work aims to provide a dynamic, hardware-based intrusion detection framework to mitigate USB-based attacks.

4. THREAT MODEL

In this chapter, we define the threat model which we aim to prevent with our detection framework. We first define an attacker's motivations through a real-world scenario. Then, we state assumptions that effectively subvert previous work in USB detection.

4.1. Subverting Prior Works

There has been prior work in detecting malicious USB devices through the aid of classification. However, we identify two key threat vectors that can subvert prior attempts: *kernel-level USB-trace hooking* and *mimicry attacks*. These threat vectors can cause prior works to fail. However, we account for both of these in our work (through hardware-assistance and feature selection, respectively).

4.1.1. Kernel-Level USB-Trace Hooking

Hooking is the process to alter the normal behavior of an operating system. This is typically done by intercepting operating system calls or events to output custom code (e.g., output "Hello World!" every time a device is inserted). A common method for sampling USB-traces is through the `USBmon` tool. In the Linux Kernel, USB traces are sent through a buffer which `USBmon` hooks into this buffer and outputs relevant trace information [Lin] (i.e., timestamp, device ID, device bus, etc.).

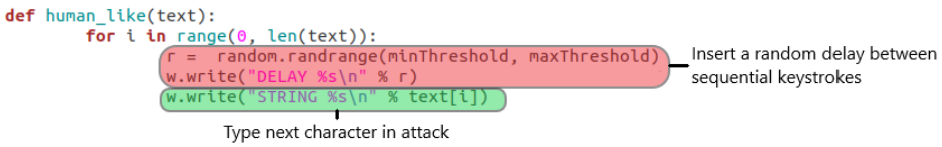


Figure 4.1: Python script to develop Rubber Ducky mimicry attack.

Through the use of system hooks, it is possible that an attacker may maliciously alter `USBmon` or other related hooks to not output specific USB device information. Should such an attack take effect, prior work discussed in Chapter 3 becomes effectively useless as the attacker can simply hide his USB device from detection schemes which utilize `USBmon`.

With novel hardware-based mechanism called USB-Watch, we overcome this limitation. Since our data is collected in hardware, it is unfeasible for an insider threat to alter incoming USB data packets from a malicious device, thereby obtaining the true output from a device.

4.1.2. Mimicry Attacks

Prior work in classification of malicious USB devices specify that the malicious USB device is assumed to act distinctly different from normal human behavior (i.e., vastly different typing patterns). Mulliner et. al. establish a simple method to prevent malicious USB devices by adding a block for any unknown device typing faster than 80ms [MW18] as no human can type faster than that. This work shows it is still possible for an attacker to develop a smarter USB device which subverts these prevention schemes. *No known work has been done at the hardware level to determine effective methods to detect a USB device which attempts to mimic human behavior.*

To show the simplicity of such an attack, we developed a malicious device which aims to subvert prior works. The outline of this attack is described in Figure 4.1. We developed a simple Python script which writes the Rubber Ducky attack file that the attacker can use

to inject the payload. The script takes in the text used for the attack and implements a delay between each typed character. The delay can have a custom lower/upper bound to more effectively mimic human behavior. For the purposes of our analysis, we used a lower bound delay of 100ms and an upper bound of 150ms. This ensures the USB device does not type faster than 100ms per keystroke so that the device types slow enough to subvert static threshold blocks. The 150ms upper bound is chosen so that the typing speed is not too slow as to impact the attack itself (i.e. being detected by a human observer, text being interspersed with other keystroke input).

We implemented this random delay using Python's random number generator, which utilizes the Mersenne Twister (MT) algorithm [Pyt]. We used this algorithm as it comes with a variety of advantages an attacker would find beneficial: (1) MT is used in most modern programming languages, making it ubiquitous to implement, (2) it utilizes a long period ($2^{19937} - 1$) which is beneficial to not repeat cycles, and (3) MT passes many statistical tests of randomness (e.g., birthday spacings, random spheres, etc.). With these 3 benefits, an attacker can easily implement the mimicry attack with more than sufficient assumptions his/her mimic can deceive simple detection schemes.

As we show later in Chapter 6, implementing this simple delay method is enough to fool other detection works. The only possible flaw with this method is the increased possibility of interference between the malicious device and the human typing on the target machine. However, we believe that the attack would still be able to perform properly if done within a reasonable time frame (e.g., 30 seconds) to minimize the likelihood of interference.

4.1.3. Subverting Anomaly Detection

There arises an issue in classification when mimicry attacks are introduced. If a mimic is able to replicate features of a target class, then the classifier will incorrectly label the

mimic as that target class. Here, we provide a general proof to showcase this issue followed by details on how this specifically impacts an anomaly detection model.

An inherent problem with using anomaly detection (or any machine-learning approach) comes from the possibility of an adversary who attempts to deceive the machine learning model [XCL+17]. We specifically look at an adversary model for an anomaly detection model in this work due to the adversary's ability to infer features and behaviors of an anomaly detection model over a more granular, multi-class machine-learning model.

Let us assume there is an anomaly detection model (M). M has a feature set X with features $(x_1, x_2, ...x_i)$. The anomaly detection model uses X to identify patterns in sampled data (D) and anomalous data (A) such that $D \equiv D'$ and $D \not\equiv A$. With patterns identified, the model can then identify new sampled data Y and apply it to the normal or anomalous class.

If an adversary were to know (or infer) X, he/she may construct malicious data (M) such that the data confuses the model to mistakenly classify M. In the case of this work, an adversary may build a malicious USB device which, when inserted, appears as a benign device to a machine learning model. For example, a device can be built which has typing dynamics similar to a human, therefore confusing an anomaly detection model designed to distinguish human and machine typing dynamics.

Therefore, when constructing machine learning models for threat prevention, we must consider the weaknesses in the model an adversary may exploit (e.g., ability to falsify features, low-specificity between classes, etc.). In Chapter 6, we show the feasibility of constructing such an adversary and steps done to mitigate this threat.

5. ARCHITECTURE

In this chapter, we define the ultimate USB-Watch architecture framework and how it can be applied in the full realization. We explain the steps taken to build a USB host controller with smart functionality to adequately determine if USB devices are benign or malicious.

5.1. Architecture Overview

As stated, the ultimate goal is a USB host controller with smart functionality. The host controller should be able to differentiate between normal USB performance a potential USB-based attack. If the host controller identifies a potential attack, communication between the device and host computer is severed.

For the USB Watch hardware to understand normal behavior, a machine learning technique can be utilized to teach a model the difference between benign and malicious USB device behaviors. Samples from both malicious and benign USB devices can be used to train and fit a model, such that it accurately distinguishes the two types of behavior.

The final model can be placed on the USB Watch hardware so that when a USB device is inserted it can properly infer the device's intent. If the hardware determines the device is malicious, it can cease communication with the device before an attack can occur. The hardware would be capable of retraining in a live setting to fit user needs.

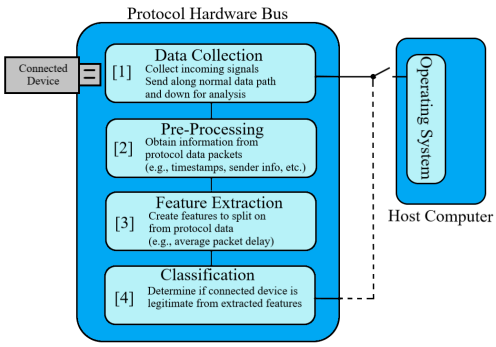
5.2. Implementation

In this section, we discuss the implementation of the proposed detection framework. We start with a custom-built hardware tool which we use to collect USB data. Then, we discuss components of the classifier: what pre-processing was required on unlabeled data, the features extracted, and what algorithms were tested for classification. Finally, we discuss how these two components work together to detect and classify malicious USB behavior in a live system.

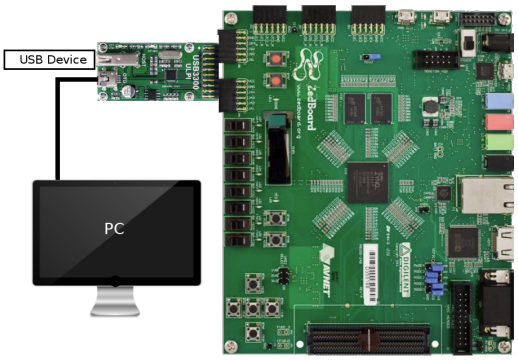
The overall architecture for the detection framework is described in Figure 5.1a. First, a USB device is inserted into a hardware mechanism located between the device and the host computer. Through the hardware module, the USB device establishes a connection to the host OS as normal. When the USB device communicates with the host machine, the raw USB signals are both sent to the host machine and collected for analyzing if the device is a potential threat (1). In (2), the hardware processes the USB packets to further extract relevant information (e.g., packet timestamps, keys being pressed, mouse movements, etc.). With the captured USB packets, the hardware can extract behavioral features which can be used to identify if the device is acting maliciously (3). Finally, the features are sent to a machine-learning classifier which determines the behavior of the unknown USB device (i.e., benign vs. malicious) (4). If the classifier deems the USB device as acting maliciously, it terminates the device's connection to the host machine.

5.2.1. Data Collection

We utilize a hardware-based mechanism between the USB-port and the computer to collect USB traffic. Basically, the hardware provides similar functionality to a software-based USB sniffer. For the testbed, we used a field-programmable gate array (FPGA) board to emulate a modified USB-host controller capable of detecting malicious USB de-



(a) USB-Watch architecture to classify malicious USB attacks



(b) Zedboard FPGA used to implement USB-Watch's testbed

Figure 5.1: USB-Watch architecture (a) and implemented testbed for evaluation of USB-Watch (b).

vices. First, an unknown USB device is connected to the FPGA board which is further connected to the user's machine. From there, normal USB traffic flows from the USB device through the FPGA and into the user's machine. Utilizing a hardware mechanism such as this provides two distinct advantages. First, the hardware cannot be spoofed as it collects data at the USB physical layer. As discussed, it is possible to subvert or tamper data collected by software sniffers. However, we collect data from the physical layer signals created by the USB device, which is not susceptible to OS-level obfuscation means. Additionally, by using a separated piece of hardware for data collection and pre-processing, an entire segment of computation is removed that would be normally performed on the host computer – saving resources for classification and detection.

5.2.2. Pre-Processing

The data pre-processor takes incoming binary information from the data collection about the connected USB device. USB packets are analyzed and relevant information about the device (e.g., device ID, frequency of packets sent, the data of each packet) is gathered for further analysis. For the purposes of this work, we analyze two common human interface

devices (keyboard and mouse) in Chapter 6 for further behavioral analysis, but we also demonstrate that analysis can be done on any USB-based device in Chapter 7.

5.2.3. Feature Extraction

Once the USB packets are collected, further analysis is done to construct features which can be used to determine the behavior of an unknown USB device. In this work, we demonstrate this by creating a command time-based feature set which, when generalized, can be applied to any number of USB devices.

The feature set is created by demonstrating reasonable variance in device behaviors in its uses. For instance, with keyboards, it has been shown that every user has a uniquely distinguishable typing behavior [MR97]. This fact can be used to create a feature set aimed to classify specific users on a machine and only permit authorized entities from using a keyboard on the host machine. To show variance in the USB-watch framework, we demonstrate:

1. Device Type – The device type is simply whatever function the device is intended to form. For instance, a keyboard or mouse would be classified as such.

2. Packet Size – A packet is a collection of data sent by a device. Packet size is the size, in bits, of that data collection. Packet size is standard by the type of device. This feature is included in this classifier to protect potential buffer-overflow style attacks that a device may use by sending incorrect packet sizes.

3. Command Transition Time (CTT) - The command transition time (CTT) is calculated for every command (i) by taking the difference of the time stamp (t_i) of the command prior (t_{i-1}). Since there is no prior command to compare for the first sampled command, ctt_0 is given a value of 0. The equation for extracting the CTT feature is provided below.

$$ctt_i = t_i - t_{i-1}, \tag{5.1}$$

4. Duration Held (D) - This feature defines how long a command was held. To obtain this, we scan the incoming USB packets for the first instance of a specific command. Then, we scan all sequential USB packets until the command is removed from the report. The difference in time stamps between the packet with the command removed and initial packet gives the duration the command was held. The equation is provided below.

$$d_i = t_{Released} - t_{Start} \tag{5.2}$$

Note that this feature does not apply to all command types. For instance, duration held makes sense for a keyboard or mouse where buttons may be pressed down. However, commands such as mouse movement or devices establishing communication do not have a command duration. For these command types, we set the duration held to a normalized average value so it does not impact the performance of the machine learning model.

5. Command Frequency (F) - This feature is defined simply as how often a command occurs over a sliding window of time. For instance, clicking the left-click button multiple times would report a high command frequency.

Note that this feature also does not apply to all command types. Getting a command frequency for a keyboard device would entail keeping track of what is being typed by the user of the keyboard – something we do not consider in this work. To relieve this, we simply leave command frequency to 0 for devices where this feature is not applicable.

Standardized Features - Since we analyze multiple device types with widely variant behaviors, we normalize features defined above. This is done by taking the mean (μ) and standard deviation (σ) of a feature vector ($X = < x_1, x_2, ... >$). Each value in the normalized feature vector is then calculated through the standardization equation:

$$\|x_i\| = (x_i - \mu)/\sigma. \tag{5.3}$$

Standardizing the features allows USB-Watch to still identify malicious behavior across multiple device types. Not standardizing them would result in USB-Watch requiring a dif-

ferent classification model for every device type analyzed. We can subvert this through standardization to get a generalized sense of malicious activity in the features USB-Watch uses.

5.2.4. Classification

The extracted features are then placed into a anomaly detection classifier to determine if the device is benign or potentially malicious. If the device is considered an anomaly, communication between the USB device and the host system is discontinued – preventing a potential attack from occurring.

During data collection, we sampled benign and malicious devices using the FPGA board and labeled the samples as such. To train the anomaly detection model, we used 90% of the collected benign data so that the model can understand how normal USB devices should behave. To test the model, the remaining 10% of benign data and all samples of malicious data were fed to the model. The model would be able to identify which of the samples are anomalous. If the model is working correctly, it should flag those anomalous samples as potential malicious and cut off communication from them. The best performing model was chosen to conduct live testing. We discuss the results of the model performance in Chapter 6 below.

For practicality, this model was constructed in software, using Python machine learning libraries (Sci-Kit Learn). The best performing model was re-written into hardware logic (VHDL). The hardware logic was then placed onto the FPGA board to test the model on live performance. We plugged in a sample of benign and malicious USB devices (e.g., keyboards, mice, etc.) into the FPGA board to test (1) if the model works and (2) if there are any performance overhead costs from introducing a piece of hardware between the USB device and host computer. The results of this testing are discussed in Section 6.

6. Performance Evaluation

This section aims to evaluate the proposed framework model on a live test bed. First, we evaluate the prototype hardware device used in this work by showing the overhead introduced into the system. Then, we aim to establish a final classification model to evaluate against other works. Feature analysis is performed on the proposed feature set of typing dynamics. Once a feature ranking is obtained, model selection is performed on a sample of classification algorithms (e.g., Decision Tree, Random Forest, Naive Bayes, k-nearest neighbors, and support vector machine). With the best overall model chosen, a comparative analysis is performed where the proposed USB-Watch detection framework and prior works are tested against real-world attack scenarios to evaluate which detection framework performs the best.

6.1. Attack Implementation

As mentioned in Section 4, this work considers the possibility that a malicious USB device may intentionally mimic human-like typing dynamics so as to appear human to an onlooker. To show the simplicity of such an attack, we developed a malicious device which aims to subvert prior works. We developed a simple Python script which writes the Rubber Ducky attack file that the attacker can use to inject the payload. The script takes in the text used for the attack and implements a delay between each typed character. The delay can have a custom lower/upper bound to more effectively mimic human behavior. For the purposes of our analysis, we used a lower bound delay of 100ms and

an upper bound of 150ms. This ensures the USB device does not type faster than 100ms per keystroke so that the device types slow enough to subvert static threshold blocks. The 150ms upper bound is chosen so that the typing speed is not too slow as to impact the attack itself (i.e. being detected by a human observer, text being interspersed with other keystroke input).

We implement this random delay using Python's random number generator [Pyt], which utilizes the Mersenne Twister (MT) algorithm. We use this algorithm as it comes with a variety of advantages an attacker would find beneficial: (1) MT is used in most modern programming languages, making it ubiquitous to implement, (2) it utilizes a long period ($2^{19937} - 1$) which is beneficial to not repeat cycles, and (3) MT passes many statistical tests of randomness (e.g., birthday spacings, random spheres, etc.). With these 3 benefits, an attacker can easily implement the mimicry attack with more than sufficient assumptions his/her mimic can deceive simple detection schemes.

6.2. Testbed and Data Acquisition

To perform our evaluations, we used a combination of a Zynq ZedBoard and a USB3300 evaluation board for a hardware data collector as shown in Figure 5.1b. We tested both benign and malicious key presses with several hundred keystrokes collected for our samples. We collected 180 total samples from both varying users and a malicious device (90 samples for each class) performing an equal variation of three behaviors: (1) typing as fast as possible for 30 seconds, (2) typing a predefined paragraph as fast as possible, and (3) typing the predefined paragraph while trying to limit typing errors. Behaviors (1) and (2), represent anomalous human behavior so that the model does not incorrectly label a human as a malicious actor. Behavior (3) captures normal human typing behaviors and malicious devices attempting to mimic human behavior. While a 1:1 ratio of benign and malicious samples is not completely representative of real-life scenarios (malicious

to benign data ratio would be considerable lower in real-life setups), we challenge our classifier with added malicious data for training and evaluation purposes.

6.3. Feature Analysis

An analysis of each proposed feature is performed in order to obtain information on how well each feature provides information to a classifier. With this, any unnecessary feature can be removed to minimize both classification time and model overfit. To do so, features were run through a recursive feature elimination which produces ranks of which features to prioritize. We detail which features were removed/prioritized as well as provide analysis to back up these findings below. The only feature removed was normalized key press duration. All of the malicious devices we tested simply send a single HID report of which ever character is next in the sequence. There is no known ability to enforce a key to be held down. This means that the duration the key is pressed is simply how frequent the device is polled. When the feature is standardized, it loses all relevant information if compared with the original key press duration. This is due to there being very little standard deviation between malicious sampled packets which all produced a duration of about 2ms. When standardized, the malicious key press duration lies squarely in the middle of a standardized benign key press, which is difficult to produce meaningful information. The remaining features are: *duration*, *KTT*, and *normalized KTT*. The most information gained comes from duration as, through analysis, it is shown that existing devices used to inject malicious keystrokes cannot alter the duration a key is held.

As shown in Figure 6.1, this inability to modify the key press duration is what leads to our classification model being so successful. Even when our malicious device produced a human-like key transition times, it was unable to mimic human behavior for key press duration. We assume this behavior occurs from malicious USB devices loading the USB buffer with new information as soon as it is requested and then moving on to the next

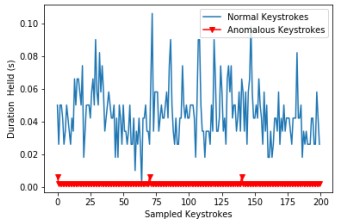


Figure 6.1: Keypress duration (s) from benign and malicious samples.

instruction in the payload. There is no known mechanism in these devices to simulate a keystroke being pressed. Even if an attacker gave a sequence of repeated characters, the USB host controller will read the characters as individual keystrokes. This fact allows us to successfully classify USB-based mimicry attacks that other classification models fail to predict.

However, we cannot simply classify on this feature alone as there are a few overlaps in duration when a human releases a key faster than 2ms. If we classify on this feature alone, this would produce a false-positive in the model. Therefore, we include the KTT and normalized KTT features for better classification scores.

6.4. Classification Algorithms

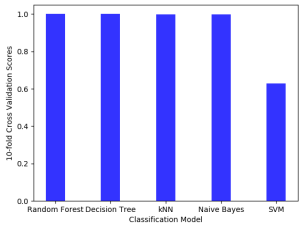
With features selected, an analysis of classification models is performed. The models chosen are: Decision Trees, Random Forest Ensembles, k-Nearest Neighbors, Naive Bayes, and Support Vector Machines. The best performing model in all criteria is chosen at the end.

To train the classification model, we used a 10-fold cross-validation approach on 90% of the collected benign samples. This is done so that the classification models may understand what is expected of USB device behavior. To test the model, the remaining 10% of the benign samples and all malicious samples were used. The classification model label the benign samples as such and the malicious samples and anomalous.

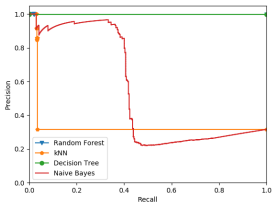
An initial parameter optimization was done to ensure well-performing models using the various machine-learning algorithms. For kNN, a k-size of 3 was used and no weights were put on the data points. For the Decision Tree and Random Forest classifiers, a tree-depth of 4 was established using entropy for information gain. 10 decision trees (default value) were used in the random forest (further evaluation in this section details why this parameter had little effect on the final model). A linear-kernel is used for the Support Vector Machine model using the hinge loss function. Finally, the Complement Naive Bayes algorithm was implemented for the Naive Bayes classifier due to the unbalanced nature of our dataset (i.e., vastly more benign samples than malicious).

The metrics used to evaluate the models are: *accuracy*, *precision*, *recall*, *fit time*, and *score time*. The first three metrics characterize how well the classifier predicts the target class. Good models correctly predict the target class (True-Postives (T_P)) and differentiate from other classes (True-Negatives (T_N)). Conversely, poor models falsely classify the target incorrectly (False-Negatives (F_N)) or classify other data as the target class (False-Positives (F_P)). Accuracy is defined as the number of correct classifications over the total number of classifications. The addition of precision and recall give a sense of how often a classification model incorrectly classifies data as the target class (precision) and how often the target class goes undetected by the model (recall). The other two metrics *fit time* and *score time* tell how fast a classification model performs. Fit time is defined as the time it takes to construct a model given the set of training data. Similarly, score time is how long it takes the constructed model to predict new data.

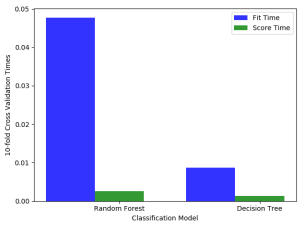
The first metric analyzed was the accuracy of the classification models. To analyze this, a 10-fold cross-validation, replicated 100 times, was conducted on each of the five tested classifiers. The results are shown in Figure 6.2a. Figure 6.2a first compares the overall score of each model in cross validation. As shown, the Decision Tree, Random Forest, k-Nearest Neighbor, and Naive Bayes classifiers performed the best overall with



(a) 10-fold cross-validation scores for each classification model.



(b) Precision vs. recall scores across varying thresholds for each classification model.



(c) Fit and score times for each classification model.

Figure 6.2: Model evaluations for decision tree, random forest, naive bayes, k-nearest neighbor, and support vector machine classifiers.

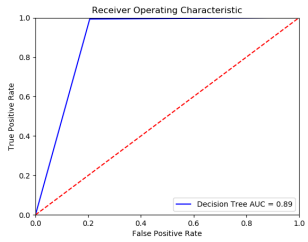
near perfect scores. The SVM classifier performs quite poorly at a rate of 0.6. As for the performance of the SVM classifier, the results relate back to the sample data. The data was intentionally constructed to produce overlaps in both benign (typing as fast as possible) and malicious (mimicked human behavior) samples. Therefore, the SVM cannot successfully converge all of the data on a single dimension. From there, the precision and recall of the classifiers was analyzed. To do this, newly generated test samples with similar behaviors as the training samples were collected. Each classifier attempted to classify the test data and returned the probability to predict either malicious or benign behavior. Precision and recall are calculated by taking the percentage of false-positives (human typing classified as a malicious actor) and false-negatives (malicious actor going undetected in the model). These precision and recall scores were calculated over a range of classification thresholds then plotted in Figure 6.2b.

As stated in our motivation, we prioritize classification models which produce a high precision first, as we do not want models believing normal behavior is malicious. As shown, the Decision Tree and Random Forest classifiers produce high results where as the kNN, Naive Bayes, and SVM classifiers produce quite poor results. For the latter classifiers, as the models try to capture all malicious devices, the model inevitably treats human behavior as malicious.

Figure 6.2c compares the time to fit and time to score the data the remaining models. The higher the bar, the more time it takes to fit/score. As shown, the Random Forest takes considerably longer to fit the data, but comparable time to score new data. This makes sense as a Random Forest ensemble creates many Decision Trees with slight variances in which data to split on – meaning that there is a linear increase in complexity to train on data compared to a Decision Tree. When considering all the evaluation metrics (accuracy, precision/recall, computation time), the *Decision Tree* is finally chosen for the classification model. This is chosen because while normally a Random Forest model removes bias and overfitting that comes from decision trees, this does not appear to be the case in fitting the sample data. Indeed, the two models perform comparably in accuracy, precision, and recall. However, the Random Forest model takes considerably longer to fit/score the data, so we choose instead to go with the Decision Tree. To test the viability of the Decision Tree model, a Receiver Operating Characteristic (ROC) curve is constructed. The ROC curve scores the model on its ability to differentiate proper signals (true-positives) from noise (false-positives) in the sampled data. Figure 6.3a shows the ROC curve of the proposed USB-Watch classifier. Note the dashed line shows a hypothetical model which simply guesses the output class. Any line above the diagonal indicates a predictive model which can properly infer the correct class from the input data – the higher the curve meaning a better classification model. As shown, the ROC area under the curve (AUC) of USB-Watch is 0.89, which indicates that the proposed framework produces near-excellent prediction results [MR04].

6.5. Comparative Analysis

In this section, the viability of the proposed USB-Watch framework is discussed. An analysis is performed to compare USB-Watch against other works. Here, we construct a



(a) ROC curve of the proposed USB-Watch decision tree classifier.

Figure 6.3: Performance analysis of the final USB-Watch model.

test suite of increasingly complex attack scenarios and simulate them against USB-Watch and other frameworks.

With the final detection model created, we need to show how effective it is. Here, simulated frameworks of prior works are created and tested against increasingly complex attack models to compare prior work with USB-Watch. We start by describing the frameworks evaluated, then detail the attack models, and conclude with results of how each framework performs against the attack models.

6.5.1. Static Detection Models

There has been prior work to prevent malicious USB devices from being authenticated to operate on a computer [TSB+16]. However, this approach does not guarantee mitigation of all attack vectors. It has been shown that malicious actors can embed malicious circuits within larger devices [RR]. Assume a malicious device in embedded within a trusted USB device and waits for specific keyphrases (e.g., wait for a user to type 'confidential', then execute an attack to collect the last typed document). To the computer, the device is still an authenticated device and is trustworthy – even when the embedded circuit is conducting the attack. USB-Watch would be able to detect this attack as it is a dynamic approach to detecting USB attacks.

6.5.2. Dynamic Detection Models

Here, we describe the dynamic detection models we recreated for evaluation. Each model is created using the same data and tested against the same attacks which are described in the section below.

Prior 1 – This framework blocks USB keyboard packets which do not exceed a minimum key transition time threshold. We use the same threshold as described in USBlock [MW18] of 80ms.

Prior 2 – A One-Class SVM is used for this model and is trained using KTT and $\|KTT\|$ features, similar to USBeSafe [Dal16].

USB-Watch 1 – This is our proposed classification model. However, we replace the hardware-based data collector with software-based tools used in other works (e.g., `usbmon` [Lin]).

USB-Watch 2 – This is our final proposed framework with the fully hardware-based detection scheme.

6.5.3. Attack Models

The frameworks were tested against a number of attacks with increased sophistication. Each attack model is described below. For each attack, the payload attempts to delete the user's Documents folder through the use of keyboard commands. Additionally, to increase sophistication, each subsequent attack model implements all the methods introduced in prior attack models (i.e. Attack 3 has all the subversion features of Attack 1 and 2).

Attack 1 – This is a simple keystroke injection attack. Once plugged in, the USB device immediately executes the payload and types it as fast as possible. This attack is used as a baseline to demonstrate all the frameworks operate correctly.

	Prior 1 [MW18]	**Prior 2 [Dal16]**	**USB-Watch 1**	**USB-Watch 2**
Attack 1	✓	✓	✓	✓
Attack 2	✓	✓	✓	✓
Attack 3	X	✓	✓	✓
Attack 4	X	X	✓	✓
Attack 5	X	X	X	✓

Table 6.1: Each detection framework's ability to detect different attacks.

Attack 2 – To subvert human detection, this attack waits for 1 minute before executing the payload. Again, the payload is typed as fast as possible.

Attack 3 – This attack attempts to mimic human typing dynamics by implementing a static delay of 100ms between each typed character.

Attack 4 – This attack uses random delays between an interval of 100ms and 150ms for each typed character.

Attack 5 – Here, we assume that the target computer is subject to an advanced threat which falsifies `usbmon` outputs. A rootkit introduces a system hook to the system which removes any USB packet which matches a Rubber Ducky or BadUSB vendor ID from displaying in `usbmon`.

6.5.4. Results

With the attack models and frameworks defined, we evaluate each attack model against each detection framework. Since each attack model increases in sophistication from the prior, we detail which attack causes the detection model to fail. Table 6.1 provides the complete results of our findings.

Prior 1 – This framework fails after Attack 3 is introduced. Since the KTT threshold should not falsely claim human typing is malicious, the attacker can make an educated guess as to what delay he/she should implement in their device as to surpass the threshold

check. In the test results, the minimum delay of 100ms exceeds the KTT check of 80ms, so the malicious device is allowed to operate.

Prior 2 – The classifier based on KTT also begins to weaken at Attack 3, but since Attack 3 has a static KTT, the classifier still performs relatively well. However, with Attack 4 introducing a random delay interval, the classifier begins to classify the attack as human in nature.

USB-Watch 1 – This model performs well against all mimicry attacks. This is due to the fact that all known USB injection attacks cannot mimic human key press duration. However, once the rootkit is introduced in Attack 5, the detection framework is simply unable to collect data from the attack and therefore cannot even begin to classify the attack.

USB-Watch 2 – Our final proposed framework performs well against all of the simulated attack models. It properly classifies Attacks 1-4 like USB-Watch 1. However, because the hardware mechanism is used to collect/analyze the raw USB signals from the malicious device, this model can properly classify Attack 5 even when the operating system on the host machine is corrupted.

6.6. Discussion

In this section, we discuss key findings of USB-Watch and how the proposed framework may be implemented in real-world scenarios. From there, benefits and limitations of the framework are elaborated.

The final USB-Watch classification model uses a binary decision tree classifier to distinguish between human (benign) and machine (malicious) typing behaviors. To do so, a feature set of keystroke transition time (i.e., time between sequential keystrokes) and keypress duration (i.e., time an individual key is held) is used. Current USB devices which mimic human typing are not able to emulate a human-like keypress duration,

which allows for the classification model to have a high accuracy and precision. We further show that mimicking human typing dynamics or altering kernel modules may cause other dynamic detection frameworks to fail. However, USB-Watch is able to detect these advanced threats due to its feature set and hardware mechanism respectively.

This work describes the methodology to implement a USB-Watch hardware implementation to detect keyboard-based USB attacks, but the overall USB-Watch framework is designed to be generalized. By collecting generic USB device behaviors, one can train the decision tree model to understand device class 'signatures'. Since USB devices broadcast what kind of device they are, the decision tree can first split on which device class (e.g., mouse, keyboard, flash storage, etc.) the unknown USB device is, then analyze the timing-based behaviors defined by the features in this work. The classifier can then determine if the device is rogue dependent on the behaviors defined by known devices of that device class.

When considering the amount of keystrokes needed to confidently identify a malicious actor, we calculated the confidence interval of our model's ability to determine if the device is malicious. A simple attack to open a terminal and delete sensitive documents takes $\tilde{4}0$ keystrokes. Given our model's perfomance, it can be 90% confident of the attacker in 40 keystrokes or less – meaning it can identify the attacker before they can finish the attack.

Since USB-Watch is enacted in a hardware mechanism between an unknown USB device and the host system, latency will be introduced. On average, the additional latency introduced per keystroke is 6ms per keystroke – an overall low latency. This latency is largely due to the USB-Watch hardware acting as a second USB host controller between the USB device and the host OS. In a final realization of USB-Watch, the actual USB host controller of the host system would have the functionality of USB-Watch, further reducing the latency.

Benefits: The proposed USB-Watch architecture utilizes a hardware mechanism to dynamically collect and process incoming USB traffic which provides two distinct benefits. First, the use of a segmented piece of hardware ensures that the architecture is operating system independent. If a device supports the USB protocol and has a USB host controller, USB-Watch will work on the system. Second, data collection and processing take up computation time with any security mechanism on a computing device. Detaching these processes to a segmented hardware like USB-Watch frees up resources on the host machine. This provides a lightweight solution to detection and minimizes overhead on the host system. With the low cost of simple FPGA circuitry (<$20 USD for a simple programmable logic board), this solution can easily be implemented when fabricating a computer with minimal increase in price per unit.

Limitations: As shown in Figure 6.1, the occasional overlap in keypress duration exists between a malicious device and normal human typing behavior. However, an adversary may easily mimic this behavior and subvert the model. Given that, it is possible for this model to weaken and potentially fail. We therefore suggest that the classification model be trained based on the user's specific typing behaviors. We note that it is unrealistic for an attacker to truly mimic a target's exact typing behaviors as that entails the attacker using advanced threats to obtain (e.g., keylogging software). Nonetheless, our proposed system is built against such a keylogger to begin with.

7. GENERALIZING THE FRAMEWORK

Here, we analyze the results of implementing the USB-Watch framework. First, we analyze the performance of the anomaly detection classifier on a multitude of varying USB devices, both benign and malicious. We score the model on common machine learning metrics (e.g., accuracy, precision, recall, etc.) to get a sense of how well the framework performs. From there, we test the USB-Watch hardware in a live testbed to analyze performance metrics and discuss any added overhead to the final system.

7.1. Anomaly Detection Model Performance

To perform our evaluations, we used a combination of a Zynq ZedBoard and a USB3300 evaluation board for a hardware data collector as shown in Figure 5.1b. The process to construct a machine learning model is to (1) collect sample data from an available dataset and simulated attackers with real RubberDucky devices plugged into the ZedBoard, (2) train and test various models in Python from the available data, (3) choose the best performing model through performance metrics, and (4) program the Python model in hardware logic (i.e., VHDL) to place on the ZedBoard for validation in a real testbed.

7.1.1. Data Collection

To train and test the model, a number of public datasets and live-captured USB samples were used. The data collected is described in detail below.

A public keyboard sample dataset comes from Carnegie Mellon [KM] which contains 400 samples from 51 users each of their typing behaviors. From there, we collected samples from a RubberDucky USB device acting both as a simple, static typing behavior, and an advanced model which mimics one of the users sampled in the dataset. The latter is done to attempt and fool our anomaly detection model.

Additionally, we collected data from other live USB devices. USB mouse samples were collected over normal use. The device list includes: USB keyboards, USB mice, and dongles used to enable connection with BlueTooth and ZigBee devices over USB. This was done to showcase the varability of the USB-Watch hardware over multiple device types.

With the USB data collected, the features defined in Chapter 5.2 were used to create a signature of each device class (e.g., keyboard, mouse, BlueTooth dongle, ZigBee dongle). The collected data was processed to fit the defined features and then standardized to get a sense of normal behavior from the device. This creates a device class signature which can be used by the classifier to understand how a normal USB device should behave.

7.1.2. Classification Performance

With the device signatures created, the benign data was used to train the anomaly detection classifier. The classification algorithm used for this work is a decision tree classifier and is established as a binary classifier (i.e., benign or not benign USB device behavior). This algorithm goes through each feature and splits the data at various intervals in the data to form a tree until all the leaf nodes have 1 or 2 data points. From there, new data is classified depending on which node it falls under in the tree. This is ideal for the type of data we are using in this work as the features have a point to split where normal and abnormal behavior appear (e.g., a human-like keyboard should have typing speed above a certain threshold – anything faster is abnormal).

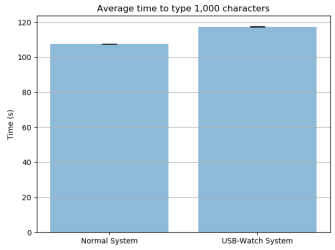
To train the classification model, we used a 10-fold cross-validation approach on 90% of the collected benign samples. This is done so that the classification models may understand what is expected of USB device behavior. To test the model, the remaining 10% of the benign samples and all malicious samples were used. Ideally, the classification model should label the benign samples as such and the malicious samples and anomalous.

With the model created, evaluation of the model can be performed. To test the validity of the anomaly detection model, the receiver operating characteristic (ROC) curve was calculated for each USB device analyzed and the accumulative data as a whole. The ROC curve scores the model on its ability to differentiate proper signals (true-positives) from noise (false-positives) in the sampled data. Figure 7.1b shows the ROC curve of the proposed USB-Watch classifier. Note the dashed line shows a hypothetical model which simply guesses the output class. Any line above the diagonal indicates a predictive model which can properly infer the correct class from the input data – the higher the curve meaning a better classification model.

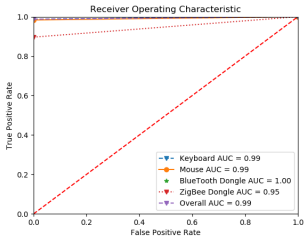
Figure 7.1b shows an ROC curve for each evaluated USB device type (e.g., keyboard, mice, and communication dongles) from the collected samples. Additionally, the ROC curve was collected from a random collection of the overall sample set. As shown, the model performs quite well when trying to distinguish benign vs. malicious USB behavior from the collected samples – each performing with an AUC of over .90, which indicates an excellent model [MR04].

7.1.3. Hardware Latency Analysis

Our proposed architecture utilizes hardware to detect and prevent malicious USB devices from injecting malicious commands to the host computer. Therefore, there is added latency since all USB traffic must go through an additional step before reaching the oper-



(a) Latency analysis of the USB-Watch hardware.



(b) Anomaly detection model ROC performance when given test-data of each device type and an aggregated overall performance.

Figure 7.1: Performance analysis of the final USB-Watch model.

ating system. Here, we analyze the additional latency introduced in the system and show its minimal effects on overall performance.

To test for added latency, we created a benign Rubber Ducky script which typed 1000 characters with a delay of 100ms between each character. We plugged this device into both (1) a normal computer system and (2) a computer system with our USB-watch hardware placed between it and the Rubber Ducky device. When the Rubber Ducky is inserted, we time how long it takes to complete the script. This process is performed 30 times for each system and then the results are all averaged. The results are shown in Figure 7.1a. As shown, the average time to complete the script for the normal and USB-Watch systems were 113s and 119s respectively. When considering each individual keystroke, that adds an increased latency of about 6ms per keystroke. This makes sense as the only increased latency the Rubber Ducky device sees is the USB-Watch hardware copying the USB packet during transit before sending the packet to the host OS – all other analysis is done concurrently by the hardware.

8. CONLCUSION

In this thesis, we use a model to dynamically analyze USB device behavior, using an anomaly detection classifier. Since this framework was created in hardware, we were able to accurately identify abnormal behaviors with an aggregate ROC AUC score of 0.99 from the collected data, even when the attacker uses advanced threats that may be able to circumvent currently established static and dynamic analysis methods used in software.

We show in this thesis that USB-Watch framework can be used to replicate a smart USB host controller. When a USB device is inserted into the USB-Watch framework, it can infer the behavior of the device and prevent malicious actions. This is done with the introduction of minimal performance overhead.

More work can be done to improve the USB-Watch framework. First, future work can be done to improve the dynamic detection model, teaching it to infer more information about the USB devices being plugged in (i.e., what is the device, what user is using the device, etc.) based on the security needs of the user. Second, further static analysis methods can be introduced to the USB-Watch framework, as dynamic methods cannot cover all potential threats. Merging both dynamic and static analysis into the USB Watch framework would create a truly smart USB host controller that can prevent malicious USB behavior.

BIBLIOGRAPHY

[Adm18] Admin. Tutorial about usb hid report descriptors. https://eleccelerator.com/tutorial-about-usb-hid-report-descriptors/, Jan 2018. Accessed: 09-16-2018.

[Bur16] E Bursztein. Does dropping usb drives really work? *Blackhat, Tech. Rep*, 2016. Accessed: 09-16-2018.

[Cun17] Andrew Cunningham. How usb became the undefeated king of connectors. https://www.wired.co.uk/article/usb-history, Oct 2017. Accessed: 11-25-2018.

[Dal16] Brandon L Daley. Usbesafe: Applying one class svm for effective usb event anomaly detection. Technical report, Northeastern University, College of Computer and Information Systems Boston United States, 2016. Accessed: 10-04-2018.

[Duc18] Paul Ducklin. Ibm bans usb drives – but will it work? https://nakedsecurity.sophos.com/2018/05/11/ibm-bans-usb-drives-but-will-it-work/, May 2018. Accessed: 09-28-2018.

[Hak] Hak5. Looks like a flash drive. types like a keyboard. https://www.hak5.org/gear/usb-rubber-ducky. Accessed: 08-28-2018.

[KM] Kevin Killourhy and Roy Maxion. Keystroke dynamics - benchmark data set. https://www.cs.cmu.edu/~keystroke/. Accessed: 03-25-2019.

[LBAU17] Juan Lopez, Leonardo Babun, Hidayet Aksu, and A Selcuk Uluagac. A survey on function and system call hooking approaches. *Journal of Hardware and Systems Security*, 1(2):114–136, 2017. Accessed: 11-17-2018.

[Lin] Linux. Usbmon documentation. https://www.kernel.org/doc/Documentation/usb/usbmon.txt. Accessed: 10-04-2018.

[Mam14] A Mamiit. How bad is badusb? security experts say there is no quick fix. *Retrieved November*, 18:2014, 2014. Accessed: 09-19-2018.

[MKK07] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007. Accessed: 12-08-2018.

[MR97] Fabian Monrose and Aviel Rubin. Authentication via keystroke dynamics. In *Proceedings of the 4th ACM conference on Computer and communications security*, pages 48–56. Citeseer, 1997. Accessed: 10-30-2018.

[MR04] Roy A Maxion and Rachel R Roberts. *Proper use of ROC curves in Intrusion/Anomaly Detection*. University of Newcastle upon Tyne, Computing Science, 2004. Accessed: 11-05-2018.

[MW18] Collin Mulliner and Edgar R Weippl. Usblock: Blocking usb-based keypress injection attacks. In *Data and Applications Security and Privacy XXXII: 32nd Annual IFIP WG 11.3 Conference, DBSec 2018, Bergamo, Italy, July 16–18, 2018, Proceedings*, volume 10980, page 278. Springer, 2018. Accessed: 09-16-2018.

[NL14] Karsten Nohl and Jakob Lell. Badusb–on accessories that turn evil. *Black Hat USA*, 2014. Accessed: 08-19-2018.

[Pyt] Python. Python 9.6. random - generate pseudo-random numbers. `https://docs.python.org/2/library/random.html`.

[Red17] RedTeam. Usb drop attacks: The danger of "lost and found" thumb drives. `https://www.redteamsecure.com/usb-drop-attacks-the-danger-of-lost-and-found-thumb-drives/`, Oct 2017. Accessed: 01-25-2019.

[RGC18] Mehul S. Raval, Ratnik Gandhi, and Sanjay Chaudhary. *Insider Threat Detection: Machine Learning Way*, pages 19–53. Springer International Publishing, Cham, 2018. Accessed: 10-16-2018.

[RR] Jordan Robertson and Michael Riley. The big hack: How china used a tiny chip to infiltrate u.s. companies. `https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies`. Accessed: 03-04-2019.

[Smi16] Smith. Say hello to badusb 2.0: A usb man-in-the-middle attack proof of concept. https://www.csoonline.com/article/3087484/security/say-hello-to-badusb-20-usb-man-in-the-middle-attack-proof-of-concept.html, Jun 2016. Accessed: 09-16-2018.

[SSS17] Sunil Sikka, Utpal Srivastva, and Rashika Sharma. A review of detection of usb malware. *International Journal of Engineering Science*, 14283, 2017. Accessed: 09-14-2018.

[TSB+16] Dave Jing Tian, Nolen Scaife, Adam Bates, Kevin Butler, and Patrick Traynor. Making USB great again with USBFILTER. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 415–430, 2016. Accessed: 03-15-2019.

[XCL+17] Xiaojun Xu, Xinyun Chen, Chang Liu, Anna Rohrbach, Trevor Darell, and Dawn Song. Can you fool ai with adversarial examples on a visual turing test. *arXiv preprint arXiv:1709.08693*, 2017. Accessed: 02-15-2019.