Journal of Hardware and Systems Security (2020) 4:136–149
https://doi.org/10.1007/s41635-020-00092-z

USB-Watch: a Generalized Hardware-Assisted Insider Threat Detection Framework

Kyle Denney[1] · **Leonardo Babun**[1] · **A. Selcuk Uluagac**[1]

Received: 20 September 2019 / Accepted: 13 February 2020 / Published online: 2 March 2020

Abstract

Today, the USB protocol is among the most widely used protocols—mostly due to its plug-and-play nature and number of supported devices. 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 are able to disguise themselves as benign devices to insert malicious commands to connected end devices. Currently, a rogue device appears as a normal USB device to the average OS, requiring advanced detection schemes (i.e., classification) to identify malicious behaviors from the devices. However, using system-level hooks, an advanced threat may subvert OS-reliant detection schemes. This paper showcases USB-Watch, a hardware-based USB threat detection framework. The use of hardware allows the framework to collect live USB traffic before advanced threats may alter the data in a corrupted OS. Through analyzing the behavioral dynamics of USB devices, a decision tree anomaly detection classifier can be placed into hardware—allowing for the detection of abnormal USB device behavior from connected USB devices. The framework tested achieves an ROC AUC of 0.99 against a testbed of live USB devices acting both normally and maliciously.

Keywords USB · Hardware security · Machine learning

1 Introduction

Sensitive computing environments (e.g., government laboratories, corporate offices, military institutions) that harbor critical infrastructure and highly confidential information [2] are key targets to cyber threats. Although research institutions invest considerate amounts to protect their data from possible outsider attacks [3, 6, 7, 17, 19, 32], insiders are still a source of potential attacks. There is currently little to be done should a malicious insider insert a device (e.g., USB device) with the intent to steal valuable data from or cause harm to the sensitive computing environment.

✉ Kyle Denney
kdenn016@fiu.edu

Leonardo Babun
lbabu002@fiu.edu

A. Selcuk Uluagac
suluagac@fiu.edu

1 Department of Electrical and Computer Engineering, Florida International University, Miami, FL, USA

Methods to detect and prevent USB-based attacks can be categorized as static analysis–based or dynamic analysis–based. Static analysis methods [9] detect potential malicious activity before the USB is operating [25]. While there are a handful of these methods in use [33, 35], 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 [4, 5] operate once a USB device has already been inserted [11, 26, 29].

With this paper, we introduce 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 algorithms, determine the behavior of the device before it can potentially cause harm to the computing environment.

All interconnected devices need a data protocol to effectively communicate. Likewise, each device needs a piece of hardware (commonly referred to as a "bus") 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 paper is the possibility of advanced adversaries which aim to subvert naive machine learning approaches. Such an adversarial model is investigated using live devices on our machine learning models. We then take the results of this to motivate specialization over generalization when using certain anomaly detection models so that an adversary has a more limited potential to mimic a target class.

Contributions: The contributions of this work are as follows:

- *USB-Watch*: We introduce a novel hardware-based USB detection framework called USB-Watch to dynamically detect and prevent USB devices from injecting malicious commands in a target computing environment.
- *Effective classification*: We perform an in-depth analysis of timing-based features on a wide variety of USB devices (e.g., mouse, keyboard, communication “dongles”) to create both a generalized model for detecting simple malicious USB devices and a specialized model for detecting advanced devices which mimic benign behavior.

Organization: The rest of this paper is organized as follows: In Section 2, we briefly overview the concepts of the USB protocol and USB-based command injection attacks. In Section 3, we discuss the state-of-the-art approaches in malicious USB detection, showing how USB-Watch offers similar results while also being tamper-proof and OS-independent. Section 4 introduces the real-world threat model that our framework prevents. The architecture of the proposed framework and our implementation of the architecture in this work is described in Section 5. The results of our experiments are outlined in Section 6 along with an analysis of the proposed framework against existing works and lists key benefits of USB-Watch. Concluding thoughts and future work are provided in Section 8.

2 Background

In this section, we outline important background information which is needed for the remainder of this paper. 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 this functionality to perform HID injection attacks.

2.1 USB Protocol

In this paper, we utilize a hardware-based mechanism to collect incoming USB data, and 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 [10]. 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.

2.2 USB Human Interface Device Reports

Some of the USB devices the detection framework in this paper 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) [1]. 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 Fig. 1a. As shown, there is 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, Fig. 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 axis and Y axis, respectively. Finally, the mouse wheel information is stored in the fourth byte. In this paper, we use these HID reports to collect data for our detection framework.

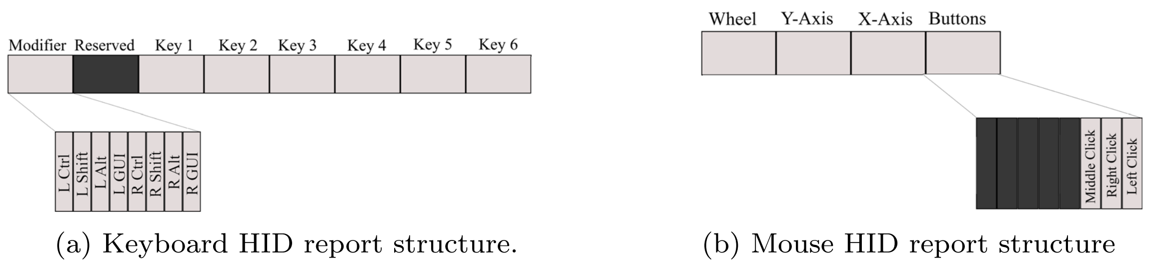


Fig. 1 **a**, **b** USB human interface device (HID) report examples

2.3 USB HID Injection Attacks

A common attack with embedded USB devices we refer to as HID injection attacks [27]. 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* [15] and the *BadUSB* [34]. 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. 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) [15].

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 [8, 30]. 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.

2.4 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 activities across both HID devices and any generic USB device which inputs commands via the USB protocol.

3 Related Work

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 [12–14, 22, 27, 34]. 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

Access control or policy-based methods to detect compromised USB devices aim to analyze a potential threat by examining it before being executed [25]. These methods normally focus on statically analyzing relevant USB controller and device information right before the data request process between both parties is established and initiated (see Section 2). When detecting threats from embedded devices, static detection methods include disallowing unregistered devices from communicating [33], require devices to request functionality permission [35], formal verification of the USB protocol [16], or simply disabling unnecessary USB ports [33]. Assuming a trusted device contains maliciously embedded circuitry [31], 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 [11, 26, 29]. 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 [21]. 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 consumes 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 [35] 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.

We have established prior work in regards to hardware-assisted dynamic detection models. The prior work was done to establish a classification model to detect malicious keyboard-based attacks. In this work, we extend upon our prior work by (1) generalizing the proposed framework to include any class of USB device evaluating the generalized framework's performance and (2) evaluating the framework's performance against advanced adversaries which can mimic 1:1 behaviors of benign devices and discussing means to mitigate this threat.

4 Threat Model

In this section, 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

As stated in Section 3, 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*. As shown in Section 6, 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 [20] (i.e., timestamp, device ID, device bus, etc.).

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 Section 3 becomes effectively useless as the attacker can simply hide his USB device from detection schemes which utilize `USBmon`.

With a novel hardware-based mechanism called USB-Watch, we overcome this limitation. Since our data is collected in hardware, it is not feasible 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). 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 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. We developed a simple Python script which writes the attack file to inject the payload. The script takes in the text used for the attack and implements a delay between each USB command. The delay can have a custom lower/upper bound to more effectively mimic benign behavior.

As we show later in our analysis, 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 interacting 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 s) 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 [36]. 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 this 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). In Section 6, we show the feasibility of constructing such an adversary and steps done to mitigate this threat.

5 Overview of Architecture

In this section, we discuss the architecture 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.

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 and 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.

The overall architecture for the detection framework is described in Fig. 2. 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,

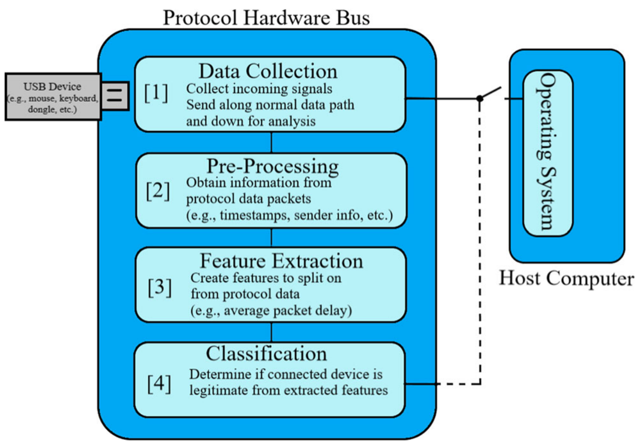

Protocol Hardware Bus
USB Device (e.g., mouse, keyboard, dongle, etc.)
[1] Data Collection
Collect incoming signals
Send along normal data path and down for analysis
[2] Pre-Processing
Obtain information from protocol data packets (e.g., timestamps, sender info, etc.)
[3] Feature Extraction
Create features to split on from protocol data (e.g., average packet delay)
[4] Classification
Determine if connected device is legitimate from extracted features
Operating System
Host Computer

Fig. 2 USB-Watch architecture to classify malicious USB attacks

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). 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.

5.1 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 simulation purposes, we used a field-programmable gate array (FPGA) board to emulate a modified USB host controller capable of detecting malicious USB devices. 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 Pre-Processing

The data pre-processor takes incoming binary from the data collection to collect information 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) for further behavioral analysis, but we also demonstrate that analysis can be done on any USB-based device.

5.3 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 [24]. 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.

$$\mathrm{ctt}_i = t_i - t_{i-1}, \tag{1}$$

4. *Duration held* (D)—This feature defines how long a command was held. To obtain this, the framework scans the incoming USB packets for the first instance of a specific command. Then, the framework scans all sequential USB packets until the command is removed from the report. The difference in timestamps between

the packet with the command removed and initial packet gives the duration the command was held. The equation is provided below.

$$d_i = t_{\text{Released}} - t_{\text{Start}} \quad (2)$$

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, the duration is 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 for efficient and effective performance of the framework.

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:

$$\|x_i\| = (x_i - \mu)/\sigma. \quad (3)$$

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 different 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.

5.4 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 should 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 Section 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) 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 below.

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 timing 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 benign device dynamics so as to appear normal 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 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 100 ms and an upper bound of 150 ms. This ensures the USB device does not type faster than 100 ms per keystroke so that the device types slow enough to subvert static threshold blocks. The 150-ms 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 [28], 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). 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 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 Fig. 3. 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.

6.2.1 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 [18] 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 variability of the USB-Watch hardware over multiple device types.

With the USB data collected, the features defined in Section 5 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.

6.2.2 Classification Performance

With the device signatures created, the benign data was used to train the anomaly detection classifier. To train the classification model, we used a 10-fold cross-validation

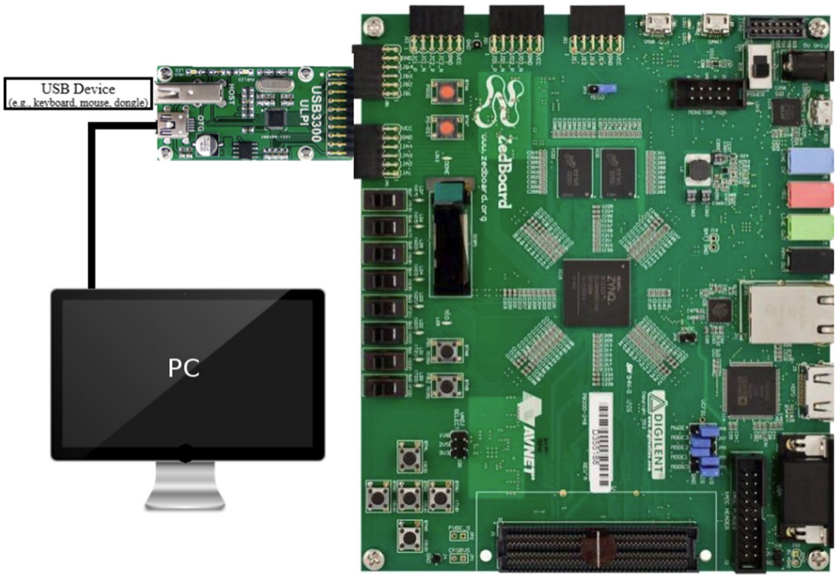


Fig. 3 Zedboard FPGA used to implement USB-Watch's testbed

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 as anomalous.

First, 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. 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 positives (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 gives 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). These three metrics are defined in Eqs. 4, 5, and 6, respectively.

$$\text{Accuracy} = \frac{T_P + T_N}{T_P + T_N + F_P + F_N} \tag{4}$$

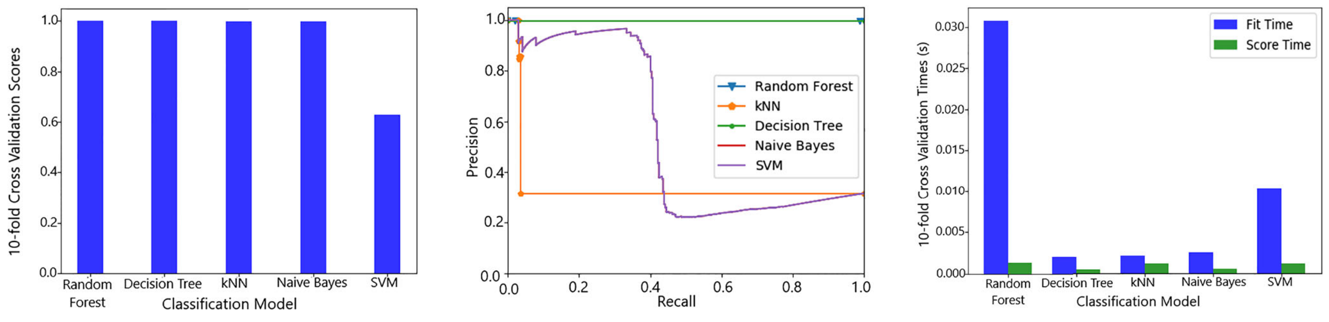
$$\text{Precision} = \frac{T_P}{T_P + F_P} \tag{5}$$

$$\text{Recall} = \frac{T_P}{T_P + F_N} \tag{6}$$

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 Fig. 4a. Figure 4a 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 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 and malicious samples. Therefore, the SVM cannot successfully converge all of the data on a single dimension. From there, the precision and recall of the classifiers were 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 Fig. 4b. As stated in our motivation, we prioritize classification models which produce a high precision as we do not want models believing normal behavior is malicious. As shown, the decision tree and random forest classifiers produce high results, whereas 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 4c 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



(a) 10-fold cross-validation scores. (b) Precision vs. recall scores across varying thresholds. (c) Fit and score times.

Fig. 4 **a–c** Model evaluations for decision tree, random forest, naive Bayes, k-nearest neighbor, and support vector machine classifiers 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 with 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.

With the final 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 5a 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, the better the classification model.

Figure 5a 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 area under the curve (AUC) of over 0.90, which indicates an excellent model [23].

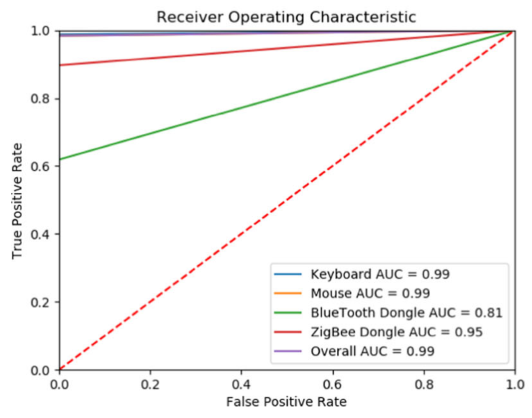
6.3 Hardware Latency Analysis

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

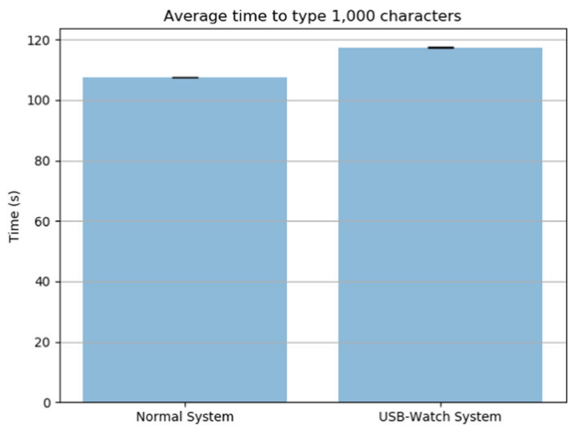
To test for added latency, we created a benign Rubber Ducky script which typed 1000 characters with a delay of 100 ms between each character. We plugged this device into both (1) a normal computer system and (2) a computer system with our USB-Watch hardware placed between it and the Rubber Ducky device. 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 Fig. 5b. As shown, the average time to complete the script for the normal and USB-Watch systems were 113 s and 119 s respectively. When considering each individual keystroke, that adds an increased latency of about 6 ms 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.

6.4 Comparative Analysis

In this section, the viability of the proposed USB-Watch framework is discussed. An analysis is performed



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



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

Fig. 5 **a**, **b** Performance analysis of the final USB-Watch model to compare USB-Watch against other works. Here, we construct a 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.

6.4.1 Static Detection Models

There has been prior work to prevent malicious USB devices from being authenticated to operate on a computer [35]. 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 [31]. Assume a malicious device is 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.

6.4.2 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 [26] of 80 ms.

Prior 2 —A one-class SVM is used for this model and is trained using CTT and ||CTT|| features, similar to USBeSafe [11].

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` [20]).

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

6.4.3 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 attacks 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.

Attack 2 —To subvert human detection, this attack waits for 1 min 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 100 ms between each typed character.

Attack 4 —This attack uses random delays between an interval of 100 ms and 150 ms 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`. Although an attacker has access to other targets when corrupting an OS, this is still a valid attack vector. We highlight this attack vector to show the importance of isolation in secure designs—specifically with isolated hardware in the USB-Watch approach.

6.4.4 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 1 provides the complete results of our findings.

Table 1 Each detection framework's ability to detect different attacks

	Prior 1 [26]	Prior 2 [11]	USB-Watch 1	USB-Watch 2
Attack 1	✓	✓	✓	✓
Attack 2	✓	✓	✓	✓
Attack 3	X	✓	✓	✓
Attack 4	X	X	✓	✓
Attack 5	X	X	X	✓

Prior 1 —This framework fails after attack 3 is introduced. Since the CTT 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 100 ms exceeds the CTT check of 80 ms, so the malicious device is allowed to operate.

Prior 2 – The classifier based on CTT also begins to weaken at attack 3, but since attack 3 has a static CTT, 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 due to a larger feature selection to properly detect malicious behaviors. 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.

7 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.

Performance: The final USB-Watch classification model uses a binary decision tree classifier to distinguish between benign and malicious USB device behaviors. To do so, a feature set of device type, packet size, command transition time, command duration, and command frequency is used. We further show that mimicking benign 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. Multiple device types were introduced in this work to show the ability to generalize our approach beyond just HID-based devices. Figure 5a discusses the impact on classification performance when introducing multiple device types. We assume the performance is a result of the data collected and specific device behaviors, rather than the capabilities of the classifier itself. We believe that developing and honing the training data would be a wonderful compliment to this work. When considering the amount of packets needed to confidently identify a malicious actor, we calculated the confidence interval of our model's ability to determine if the device is malicious. Given our model's performance, it can be 90% confident of the attacker in 40 packets or less. This is well within an acceptable range to identify attacks before they can finish.

Scalability: The prototype implemented in this work is meant to mimic the behavior of a sole USB port (though full production may emulate any number of USB ports). With the addition of dongles, bridges, etc., the single USB port can be capable of supporting up to 127 distinct USB devices, which is the limitation of the USB protocol itself. Classification performance would not be impacted as each device is uniquely identified in order to effectively communicate on the channel—allowing the classifier to similarly uniquely identify specific malicious actors. On average, the additional latency introduced per keystroke is 6 ms per keystroke—an overall low latency. This latency is largely due to the USB-Watch prototype acting as a second USB host controller between the USB device and the host OS. This is supported when testing the prototype's scalability. When plugging in multiple devices, the latency remained a constant. 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: An advanced adversary may be able to mimic behavior dynamics used to train the classifier. We

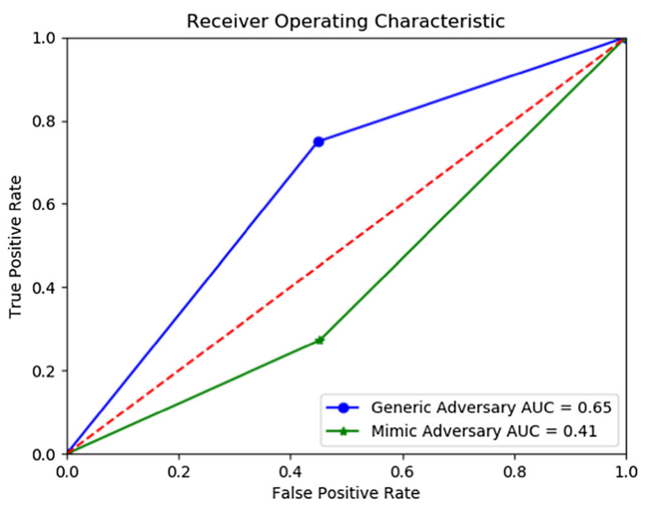


Fig. 6 ROC curve of the USB-Watch classifier when against an unknown and mimicked adversary

tested this possibility of (1) an adversary who uses their own behaviors to initiate the attack and (2) an attacker who mimicked the target's exact behaviors against our model. As shown in Fig. 6, the model performs weaker against both types of adversaries. However, the model is not fully broken unless the adversary can truly mimic the target's behaviors (which is unrealistic in most cases). We consider this attack vector unlikely as the adversary must install keylogging capability in the target machine. If the adversary can do this, then they may install another, more advanced attack on the target machine. Nonetheless, this result implies limiting the training data to as few users as possible so that it becomes more difficult for an adversary to effectively mimic exact user behaviors. The method described in this work dynamically detects malicious USB devices. While it has been shown to be effective, it does not cover all attack vectors. Static detection methods as described in the related work can be implemented in congruence with USB-Watch to cover a wider range of threats.

8 Conclusions and Future Work

In this work, 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 study that USB-Watch hardware can be used to replicate a smart USB host controller. When a USB device is inserted into the USB-Watch hardware, 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 analyses into the USB-Watch framework would create a truly smart USB host controller that can prevent malicious USB behavior.

Funding Information 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. The views expressed are those of the authors only, not of the funding agencies.

References

1. Admin: Tutorial about usb hid report descriptors. https://eleccelerator.com/tutorial-about-usb-hid-report-descriptors/ (2018). Accessed: 16 Sept 2018
2. Aksu H, Babun L, Conti M, Tolomei G, Uluagac AS (2018) Advertising in the iot era: Vision and challenges. IEEE Commun Mag 56(11):138–144. https://doi.org/10.1109/MCOM.2017.1700871
3. Babun L, Aksu H, Uluagac AS (2019) A system-level behavioral detection framework for compromised cps devices: Smart-grid case. ACM Trans Cyber-phys Syst 4(2). https://doi.org/10.1145/3355300
4. Babun L, Celik ZB, McDaniel P, Uluagac AS (2019) Real-time analysis of privacy-(un)aware iot applications
5. Babun L, Sikder AK, Acar A, Uluagac AS (2018) Iotdots: A digital forensics framework for smart environments. CoRR arXiv:abs/1809.00745
6. Babun L, Aksu H, Uluagac SA (2018) Detection of counterfeit and compromised devices using system and function call tracing techniques. http://www.freepatentsonline.com/10027697.html
7. Babun L, Aksu H, Uluagac SA (2019) Method of resource-limited device and device class identification using system and function call tracing techniques, performance, and statistical analysis. http://www.freepatentsonline.com/10242193.html
8. Bursztein E (2016) Does dropping usb drives really work? Blackhat, Tech. Rep. Accessed: 16 Sept 2018
9. Celik ZB, Babun L, Sikder AK, Aksu H, Tan G, McDaniel P, Uluagac AS (2018) Sensitive information tracking in commodity iot. In: 27Th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore, pp 1687–1704
10. Cunningham A (2017) How usb became the undefeated king of connectors. https://www.wired.co.uk/article/usb-history. Accessed: 25 Nov 2018
11. Daley BL (2016) Usbesafe: Applying one class svm for effective usb event anomaly detection. Tech. rep., Northeastern University, College of Computer and Information Systems Boston United States. Accessed: 04 Oct 2018

12. Denney K, Erdin E, Babun L, Uluagac AS (2019) Dynamically detecting usb attacks in hardware: Poster. In: Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks, WiSec '19. Association for Computing Machinery, New York, pp 328–329. https://doi.org/10.1145/3317549.3326315
13. Denney K, Erdin E, Babun L, Vai M, Uluagac S (2019) Usb-watch: a dynamic hardware-assisted usb threat detection framework. In: International Conference on Security and Privacy in Communication Systems, Springer, pp 126–146
14. Ducklin P, Parkes M, James T, Pottage D (2016) Sidestepping your lockscreen with an innocent-looking usb stick. https://nakedsecurity.sophos.com/2016/09/09/sidestepping-your-lockscreen-with-an-innocent-looking-usb-stick/
15. Hak5: Looks like a flash drive. types like a keyboard. https://www.hak5.org/gear/usb-rubber-ducky. Accessed: 28 Aug 2018
16. Johnson PC, Bratus S, Smith SW (2017) Protecting against malicious bits on the wire: automatically generating a usb protocol parser for a production kernel. In: Proceedings of the 33rd Annual Computer Security Applications Conference, ACM, pp 528–541
17. Kaygusuz C, Babun L, Aksu H, Uluagac AS (2018) Detection of compromised smart grid devices with machine learning and convolution techniques. In: 2018 IEEE International Conference on Communications (ICC), pp 1–6. https://doi.org/10.1109/ICC.2018.8423022
18. Killourhy K, Maxion R Keystroke dynamics - benchmark data set. https://www.cs.cmu.edu/keystroke/. Accessed: 25 Mar 2019
19. Babun L. author=Aksu, H (2017) Identifying counterfeit smart grid devices: a lightweight system level framework. In: Proceedings of the IEEE ICC Intern Conf on Communications. IEEE, Paris, France
20. Linux: Usbmon documentation. https://www.kernel.org/doc/Documentation/usb/usbmon.txt. Accessed: 04 Oct 2018
21. Lopez J, Babun L, Aksu H, Uluagac AS (2017) A survey on function and system call hooking approaches. Journal of Hardware and Systems Security 1(2):114–136. Accessed: 17 Nov 2018
22. Mamiit A (2014) How bad is badusb? security experts say there is no quick fix. Retrieved November 18, 2014. Accessed: 19 Oct 2018
23. Maxion RA, Roberts RR (2004) Proper use of ROC curves in Intrusion/Anomaly Detection. University of Newcastle upon Tyne, Computing Science. Accessed: 05 Nov 2018
24. Monrose F, Rubin A (1997) Authentication via keystroke dynamics. In: Proceedings of the 4th ACM Conference on Computer and Communications Security, pp 48–56. Citeseer. Accessed: 30 Oct 2018
25. Moser A, Kruegel C, Kirda E (2007) Limits of static analysis for malware detection. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp. 421–430. IEEE. Accessed: 08 Dec 2018
26. Mulliner C, Weippl ER (2018) 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, vol 10980. Springer, p 278. Accessed: 16 Sept 2018
27. Nohl K, Lell J (2014) Badusb–on accessories that turn evil. Black Hat USA. Accessed: 19 Aug 2018
28. Python: Python 9.6. random - generate pseudo-random numbers. https://docs.python.org/2/library/random.html
29. Raval MS, Gandhi R, Chaudhary S (2018) Insider Threat Detection: Machine Learning Way. Springer International Publishing, Cham, pp 19–53. Accessed: 16 Oct 2018
30. 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/ (2017). Accessed: 25 Jan 2019
31. Robertson J, Riley M 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: 04 Mar 2019
32. Rondon LP, Babun L, Akkaya K, Uluagac AS (2019) Hdmi-walk: Attacking hdmi distribution networks via consumer electronic control protocol. In: Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC '19. Association for Computing Machinery, New York, pp 650–659. https://doi.org/10.1145/3359789.3359841
33. Sikka S, Srivastva U, Sharma R (2017) A review of detection of usb malware. International Journal of Engineering Science 14283. Accessed: 14 Sept 2018
34. 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 (2016). Accessed: 09-16-2018
35. Tian DJ, Scaife N, Bates A, Butler K, Traynor P (2016) Making USB great again with USBFILTER. In: 25th USENIX Security Symposium (USENIX Security 16), pp 415–430. Accessed: 15 Mar 2019
36. Xu X, Chen X, Liu C, Rohrbach A, Darell T, Song D (2017) Can you fool ai with adversarial examples on a visual turing test. arXiv preprint arXiv:1709.08693. Accessed: 15 Feb 2019