# USBeSafe: An End-Point Solution to Protect Against USB-Based Attacks

Amin Kharraz[†‡] Brandon L. Daley [◇‡] Graham Z. Baker[◇] William Robertson[‡] Engin Kirda[‡]

[◇]*MIT Lincoln Laboratory* [†]*University of Illinois at Urbana-Champaign* [‡]*Northeastern University*

## Abstract

Targeted attacks via transient devices are not new. However, the introduction of BadUSB attacks has shifted the attack paradigm tremendously. Such attacks embed malicious code in device firmware and exploit the lack of access control in the USB protocol. In this paper, we propose USBeSafe as a mediator of the USB communication mechanism. By leveraging the insights from millions of USB packets, we propose techniques to generate a protection model that can identify covert USB attacks by distinguishing BadUSB devices as a set of *novel* observations. Our results show that USBeSafe works well in practice by achieving a true positive [TP] rate of 95.7% with 0.21% false positives [FP] with latency as low as *three* malicious USB packets on USB traffic. We tested USBeSafe by deploying the model at several end-points for 20 days and running multiple types of BadUSB-style attacks with different levels of sophistication. Our analysis shows that USBeSafe can detect a large number of mimicry attacks without introducing any significant changes to the standard USB protocol or the underlying systems. The performance evaluation also shows that USBeSafe is transparent to the operating system, and imposes no discernible performance overhead during the enumeration phase or USB communication compared to the unmodified Linux USB subsystem.

## 1 Introduction

Transient devices such as USB devices have long been used as an attack vector. Most of these attacks rely on users who unwittingly open their organizations to an internal attack. Instances of security breaches in recent years illustrate that adversaries employ such devices to spread malware, take control of systems, and exfiltrate information.

Most recently, researchers have shown that despite several warnings that underscore the risk of malicious peripherals, users are still vulnerable to USB attacks [27, 28]. To tackle this issue, antivirus software is becoming increasingly adept at scanning USB storage for malware. The software automatically scans removable devices including USB sticks, memory cards, external hard drives, and even cameras after being plugged into a machine. Unfortunately, bypassing such checks is often not very difficult as the firmware of USB devices cannot be scanned by the host. In fact, the introduction of BadUSB attacks has shifted the attack paradigm tremendously as adversaries can easily hide their malicious code in the firmware, allowing the device to take covert actions on the host [9]. A USB flash drive could register itself as both a storage device and a Human Interface Device (HID) such as a keyboard, enabling the ability to inject surreptitious keystrokes to carry out malice.

Existing defenses against malicious USB devices have resulted in improvements in protecting end-users, but these solutions often require major changes in the current USB protocol by introducing an access control mechanism [26], modifying the certificate management [20], or changing the user experience (i.e., a user-defined policy infrastructure) [3, 24]. Our goal is different in a sense that we seek to improve the security of USB devices while keeping the corresponding protection mechanism completely in the background. The immediate benefit of such a solution is flexibility, allowing: (1) organizations to use standard devices, (2) manufacturers to avoid changing how their hardware operates, and (3) users to continue using their current USB devices.

In this paper, we propose USBeSafe, a system to identifying BadUSB-style attacks, which are probably the most prominent attack that exploits the USB protocol. Our approach relies upon analyzing how benign devices interact with the host and the operating system. By leveraging the insights from millions of USB Request Blocks (URBs) collected over 14 months from a veriety of USB devices such as keyboards, mouses, headsets, mass storage devices, and cameras, we propose classification techniques that can capture how a benign USB device interacts with a host by monitoring URBs as they traverse the bus. Starting with a wide range of classification features, we carefully analyze the labeled data and narrow down to three feature categories: content-based, timing-based, and type-based features. We train several different machine learning techniques including SVM [14], Nearest Neighbor [13], and Cluster-based Techniques [12] to find the most accurate algorithm for building our detection model. Our analysis showed that One-Class SVM achieved the highest detection results with a low false positive rate (a true positive [TP] rate of 95.7% with 0.21% false positives [FPs]) on the labeled dataset. The constructed model allows us to identify covert USB attacks by distinguishing BadUSB devices as *novel* observations for the trained dataset.

To test USBESAFE, we deployed the constructed model as a service on end-user machines for 20 days. Our analysis shows that USBESAFE is successful in identifying several forms of BadUSB attacks with a low false positive rate on live, unknown USB traffic. For a real-world deployment, we also performed a training/re-training analysis to determine how USBESAFE should be deployed on new machines to keep the detection rate constantly high with under a 1% false positive rate. We show that training USBESAFE with as low as two training days and re-training it every 16 days for 82 seconds are sufficient to maintain the detection rate over 93% across all the machines.

The most important finding in this paper is practical evidence that shows it is possible to develop models that can explain the benign data in a very precise fashion. This makes anomaly detection a promising direction to defend against BadUSB-style attacks without performing any changes to the standard USB protocol or underlying systems. We ran multiple forms of adversarial scenarios to test USBESAFE's resilience to evasion with the assumption that adversaries have significant freedom to generate new forms of BadUSB-style attacks to evade detection. Our analysis shows that USBESAFE can successfully detect mimicry attacks with different levels of sophistication without imposing a discernible performance impact or changing the way users interact with the operating system. We envision multiple potential deployment models for USBESAFE. Our detection approach can be incorporated as a light-weight operating system service to identify BadUSB attacks and disable the offending port or an early-warning solution to automatically identify the attacks and notify system administrators.

## 2 Background, Threat Model, and Related Work

A Universal Serial Bus (USB) device can be a peripheral device such as a Human Interface Device (HID), printer, storage, or a USB transceiver. An attached USB device can have multiple functionalities where each functionality is determined by its interfaces. The host controller interacts independently with these interfaces by loading a device driver for each interface. When a USB device is attached, the USB controller in the host issues a set of control requests to obtain the configuration parameters of the device in order to activate the supported configuration. The host parses the configuration, and reads the device descriptor which contains the information about the functionality of device. This information allows the host to load a driver based on the configuration information. This procedure is called *enumeration* phase. In the enumeration phase of the USB protocol, the endpoints are addressed as `IN` and `OUT` to manage the USB traffic. The `IN` endpoint stores the data coming to the host, and the `OUT` endpoint receives the data from the host. After the enumeration phase, the host loads the USB interfaces which allow a device to operate.

### 2.1 Threat Model

In our threat model, we assume that a connecting device can report any capabilities to the bus, and the host machine trusts the information that it receives from the device. Similar to BadUSB attacks [9], an adversary can use this capability by rewriting the firmware of an existing device to hide malware in the code that communicates with a host. More specifically, upon insertion into a host USB port, a mass storage device – i.e., a USB flash drive (with capabilities for Windows and Linux) – covertly performs keyboard actions to open a command prompt, issue a shell command to download malicious code from the Internet, and execute the downloaded malware.

We should mention that classic USB attacks, for example using the autorun capabilities of USB devices to distribute malware, are out of the scope of the paper as these attacks can be detected by most of malware scanners. Similar to prior work [24], we try to address the advanced persistent threat (APT) scenario where an adversary is attempting to expand its presence in a network by distributing USB devices with malicious firmware as described above. We assume that the malicious USB device is capable of generating new device identities during the enumeration phase by providing varying responses in each enumeration to evade potential device identification mechanisms. We also assume that there exists no USB-level authentication mechanism between the device and the target host. The OS simply acts on information provided by the device and will load a driver to accept the USB drive as, e.g., an HID device. We assume that once the device has been connected, the adversary can use any technique to expand her presence. For example, the malicious firmware can open a command prompt to perform privilege escalation, exfiltrate files, or copy itself for further propagation. Finally, in this work, we also assume that the trusted computing base includes the display module, OS kernel, and underlying software and hardware stack. Therefore, we consider these components of the system free of malicious code, and that normal user-based access control prevents attackers from running malicious code with superuser privileges.

### 2.2 Related Work

A wide range of attacks have been introduced via USB including malware, data exfiltration on removable storage [8, 16, 17, 22], and tampered device firmware [5, 9]. These cases show that defending against USB attacks is often not straightforward as these attacks can be tailored for many scenarios. In the remainder of this section, we explore existing solutions for this class of attack vector and their limitations.

One approach to defend against attacks involving subverted firmware is to hardwire USB microcontrollers to only allow firmware updates that are digitally signed by the manufacturer. Currently, the de facto technology to protect against malicious data residing on and executing from a device exists in IEEE Standard 1667 [20]. The standard seeks to create a means for bidirectional authentication via an X.509 certificate infrastructure between hosts and devices. Unfortunately, the adoption of IEEE 1667 has been slow, and USB devices do not possess any entity authentication mechanism as a means of vouching for the safety of data residing on the device.

One of the first research efforts to secure the USB protocol was conducted by Bates et al. [4, 15] where they measured the timing characteristics during USB enumeration to infer characteristics of host machines. Another class of work focuses on proposing access control mechanisms on USB storage devices [6, 19, 23, 30]. While these approaches can lead to better defense mechanisms, recent studies [21,24] have shown that these approaches are coarse and cannot distinguish between desired and undesired usage of a particular interface. Very recently, Hernandez et al. [7] introduced FirmUSB, a firmware analysis framework, to examine firmware images using symbolic analysis techniques. By incorporating the tool, the authors identified the malicious activity without any source code analysis while decreasing the analysis time. In fact, the proposed technique is very effective in addressing some of the increasing concerns on the trustworthiness and integrity of USB device firmwares.

One other approach to mitigating such attacks is to minimize the attack surface without changing the fundamentals of USB communication or patching major operating systems. Recently, Tian et al. [24] have proposed GoodUSB which has similar goals to ours. Their approach is based on constructing a policy engine that relies on virtualization and a database that consists of already seen USB devices and reporting unknown USB devices to the user. The proposed solution mediates the enumeration phase, and verifies what the device claims as its functionality by consulting to a policy engine. GoodUSB shifts the burden of responsibility to the user to decide whether a USB device is malicious or benign.

In another work, Tian et al. [26] proposed USBFilter, a packet-driven access control mechanism for USB, which can prevent unauthorized interfaces from connecting to the host operating system. USBFilter traces individual USB packet, and blocks unauthorized access to the device. Tian et al. [25] complemented their previous work by introducing ProvUSB which incorporated provenance-based data forensics and integrity assurance to defend against USB-based threats. Angel et al. [3] uses a different approach and leverages virtualization to achieve the same goal. We posit that a solution such as the one described in this paper that introduces as little change as possible to the user operational status quo is more likely to prevent exploitation in practice, given that the underlying detection mechanism is reliable. That being said, these approaches are fundamentally orthogonal and could be composed to obtain the benefits of both.

## 3 Overview of The Approach

In this section, we provide more details on USBESAFE components and the model we use to detect BadUSB attacks. Figure 1 shows the pipeline used by USBESAFE to identify BadUSB-style attacks.

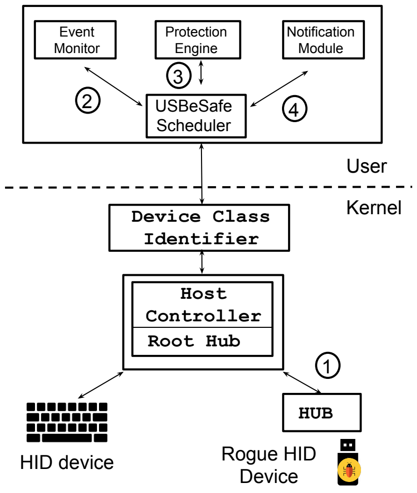


Figure 1: A high level view of a USBeSafe-enabled machine.

### 3.1 System Design

The architecture of a USBESAFE-enhanced system requires interactions among multiple components of the operating system. In this section, we describe the abstract design of USBESAFE, independent of the underlying OS. Later, we will demonstrate how our design can be realized in a prototype running on Linux. USBESAFE's components are mostly managed by a user space daemon. The daemon includes three main subsystems as shown in Figure 1: First, a lightweight user space module that processes transaction flows between the host and the connected device; second, a detection module that implements the USB mediator logic; and third, a user interface that generates alerts and notifies the user. When a USB device is connected to the host (1), USBESAFE collects and preprocesses the URBs (2). The protection engine utilizes the preprocessed data to construct the feature vector, and test whether the incoming USB packets are in fact new observations (3). In cases where the system detects a novel sequence of USB packets, it creates a notification, and sends an alert to the user (4). In the following, we provide more details on each proposed module.

### 3.1.1 USB Event Monitor

The ultimate goal of the event monitor is to analyze URBs and transform them to an appropriate format that can be used in the protection engine. To this end, the USB event monitor detects a connected device, and processes the transaction flows in the form of URBs which contain USB packets during a USB connection lifecycle from the enumeration, to configured communication, to termination. To store and analyze USB packets, we implemented a set of data objects. The module parses each URB, extracts the USB packet, and generates a *TraceEvent* containing the USB header information and payload. In fact, each TraceEvent is a tuple that contains the host bus ID as well as the assigned device ID on the bus. Each TraceEvent, representing a single USB packet, is appended to a *Trace* – a list of TraceEvents. USBESAFE generates a single Trace file for each USB device from the enumeration to disconnection phase of the connected device. TraceEvents in each Trace are sorted according to their timestamp, from earliest to most recent. For each Trace, we identify the device and configuration descriptor responses, storing them as auxiliary information for the Trace.

The root of the data structure is called the *TraceLibrary* which contains all Traces in the dataset. For each trace in the TraceLibrary, we sort existing traces into *TraceList*s which contain all the traces with the same (`busID`, `deviceID`) tuple. USBESAFE uses the class codes field in the device and interface descriptors to determine the device type and expected functionality. While USBESAFE has an extensible design, we focus specifically on USB HIDs including USB keyboard traffic and the features that characterize such traffic as benign. This focus stems from our goal which is to determine whether a covert HID configuration is present and active on a device.

### 3.1.2 Protection Engine

The protection engine is central to the security model proposed in USBESAFE which decides whether a new, previously unseen set of USB packets is potentially malicious. In the following, we explain the features we employed to characterize the USB packets, and train the detection model in USBESAFE.

**Packet Interarrival Times** Packet interarrival times characterize the USB keyboard traffic across a bus and, specifically, the timing information of packets. The timing information can help reveal user's typing patterns by serving as a proxy for inter-keystroke times, or how the bus manages the URBs of different kinds. Interarrival time values are measured in milliseconds, between one packet and the next for all the TraceEvents. Note that a user may enter a keystroke after a longer pause, ranging from a few seconds to hours. To tackle the problem of potentially unbounded interarrival time values in these situations, we explicitly define an upper bound for the interarrival time value between two interrupt packets. We explain this procedure and the selection of specific threshold values in more details in Section 5.

**Event Type** USBESAFE monitors the value that defines the type of each USB packet. More precisely, a URB event type can take two values which indicate whether there is an onging transaction (`URB_SUBMIT (`$0\times53$`)`) or if a transaction is complete (`URB_COMPLETE (`$0\times43$`)`).

**Transfer Type** USBESAFE also monitors the value of URB transfer type between the host and the USB device. The transfer type can take four possible values which are `URB_INTERRUPT`, `URB_CONTROL`, `URB_BULK`, and `URB_ISOCHRONOUS`. This value is selected for a USB device according to the requirements of the device and the software which is determined in the endpoint descriptor.

**Post-enumeration Time** USBESAFE monitors when the post-enumeration activity starts. For example, in a normal scenario, a user connects a USB keyboard to the host, and starts interacting with it. Along with other features, looking at millions of URBs allows USBESAFE to observe the normal start of post-enumeration activity of a HID device after attaching the device to the host. The system incorporates this feature as a numerical value by calculating the time period between a successful enumeration and the start of data packet transfer.

**Packet Payload** USBESAFE monitors the payload of individual USB packets. USBESAFE examines the payload to determining patterns in data by using a byte histogram to measure value frequencies within each TraceEvent. The histogram represents a space of 256 values ($[0,255]$) by bucketizing values into 16 equal intervals or bins. For each TraceEvent, the system generates a feature vector which contains *interarrival_time, event_type, transfer_type, post_enumeration, data_histogram*. The feature vector is then used to construct a detection model which will be used as an augmented service in the operating system.

To analyze the URB payloads, we extracted all the n-grams of EventTraces that appeared in a sliding window of the length $n$ where the value of $n$ varied from 2 to 4. Each unique sequence of length $n$ is added to the detection model for the USB HID class. The intuition here is that those n-grams are characteristics of benign USB packets, and any traffic that does not follow similar patterns compared to the extracted model for a given user is a novel observation, and with high likelihood correspond to a new typing pattern. In Section 5.3, we provide more details on our model searching process since we should take into account several configuration parameters to achieve the highest detection rate and the lowest false positive rate.

## 4 Implementation

In this section, we provide more details on the implementation of USBESAFE's prototype which relies on the Linux USB stack. The implementation of USBESAFE consists of three independent modules which were discussed in Section 3.1: (1) a USB event monitor which interposes the bus transactions, (2) a protection module which constructs the feature vector and validates whether the incoming USB packets comply with the generated model, and (3) a notification module which produces an alert and notifies the user if a novel traffic pattern is detected. In the following, we provide more details on the implementation details of each module.

### 4.1 USB Event Monitor

We used the `usbmon` Linux kernel module, as a general USB layer monitor, to capture all the URBs transmitted across the monitored USB bus. In user space, USBESAFE implements the transaction flow introspection module by extracting device information using `sysfs, lsusb` and device activities using `usbmon` and `tcpdump`. Monitoring the USB devices starts at the boot time. USBESAFE collects self-reported device information as well as actions taken by associated drivers during the normal usage. The USB event monitor module is a user space program which is developed in Python. This module is loaded prior to the device's enumeration phase by updating the `udev` database. We defined a set of datastructure to collect interface information (e.g., Descriptor Type, Interface Number, Interface Class and protocol), configuration information (e.g., max power), and device information (e.g., manufacturer) for each connected device.

### 4.2 Protection Engine

As mentioned earlier in Section 3.1, the protection engine in USBESAFE is responsible for determining whether a set of USB packets from a connected device to the host are, in fact, new observations and whether the USB device should be disabled or not. In cases where the USBESAFE identifies a set of USB packets as new observations, it notifies the user as well as kernel space components to block the corresponding interface. As the protection engine is the core part of the USBESAFE, we want to make sure that the model is constructed based on a suitable algorithm. To this end, we evaluated multiple machine learning algorithms by measuring their detection accuracy on the labeled dataset. In Section 5, we provide more details on the detection accuracy of the algorithms as well as the parameter configurations. The results of our analyses revealed that one-class SVM [14] achieved the highest detection rate with a very low false positive rate. In our detection model, the one-class SVM can be viewed as a regular two-class SVM where all the training data is benign and lies in the first class, and the unseen data by a large margin from the hyperplane is taken as the second class. In fact, the constructed model in USBESAFE solves an optimization problem to find a term with maximal geometric margin. Therefore, if the geometric margin is less than zero, the test sample is reported as a novel observation. As USBESAFE has a high privilege, it automatically unbinds the offending USB port by calling `/sys/bus/usb/drivers/usb/unbind` without involving the user.

### 4.3 Notification Module

The notification module is deployed as a user space daemon which produces alerts whenever the protection module identifies a novel observation. We should mention that there are several design choices for implementing the notification module. However, the core requirement of the module is that the notification should be always stacked on top of the screen contents, and cannot be obscured, interrupted, or interfered with by other processes. We achieve this by leveraging `libnotify-bin` module which is usually used to send desktop notifications to users. To prevent the notification module from being killed programmatically by a potentially malicious process with the same user ID, we recommend creating another user ID to run the process. Consequently, only root could kill the process. As the notification module is not the core part of our contributions, we do not explore the notification module further in this paper.

## 5 Evaluation

To test USBESAFE we conducted two experiments. In the first experiment, we train and test USBESAFE with a labeled dataset, and in the second experiment, we test the derived model on a previously unseen dataset to evaluate the detection capability of the system in a real-world deployment. Although our design is sufficiently general to be applied to different operating systems, we built our prototype for Ubuntu 14.04 LTS with the Linux kernel 3.19. In the following, we first describe how we created our dataset, and then provide the details of evaluation and benchmarks.

### 5.1 Data Collection

In order to create the training dataset, we monitored USB packet exchanged between a set of devices and five machines. Each time a USB device was connected, USBESAFE generated a new trace, named it based on the bus and device ID of the USB device, and created logs for real-time USB packets across the monitored USB bus. On the system shutdown, the module saved the generated trace file to the disk. We sorted each TraceEvent based on USB device class code which was extracted from the interface descriptor. The HID class, which keyboards, mouses, headsets, and game joysticks fall under, is defined by the class code 3. During 14 months of data collection, several types of USB devices such as different keyboards, storage devices, cameras, and headsets were connected to the machines. We considered a connected USB device as a HID

| Machine | URBs | No. of Traces |
|---|---|---|
| Machine1 | 3,385,445 | 124 |
| Machine2 | 2,394,345 | 90 |
| Machine3 | 2,884,345 | 101 |
| Machine4 | 943,984 | 50 |
| Machine5 | 1,620,265 | 58 |
| Total | 11,228,384 | 423 |

Table 1: The collected USB Packets over 14 months. We collected 423 HID trace files which contained more than 11 million USB Packets. The trace files were collected from several types of keyboards, mouses, headsets. The dataset also includes traces of other USB devices such as cameras, storage devices which are not HID devices, but registered themselves as a HID device in addition to their main driver.

device, if it requested HID driver from the operating system. For example, we found a benign USB printer that registered itself both as a printer and a HID device to enable the touchscreen. Although standard USB printers are not considered as a HID device, we considered the corresponding traces in the training phase as the device potentially had the capability to run commands.

In total, 423 trace files were collected from HID devices, consisting of 11,228,384 URBs. Note that the actual number of USB packets that crossed the bus was greater than this value as any usbmon-based packet actually represented two or three USB packets on the bus, depending on whether the interaction included a payload or not. Table 1 illustrates a summary of the data collected over 14 months from five different machines.

Note that our approach is not an outlier detection method, but a novelty detection technique. This means that we need to have a clean training dataset representing the population of regular observations for building a model and detecting anomalies in new observations. Therefore, the malicious data used in our experiments was solely collected for testing purposes. To generate this malicious dataset, we used a *Rubber Ducky* USB drive [2], updated the firmware, and generated a set of scripts that establish covert channels, inject code for data exfiltration, and connect to a remote server. Such attacks have several forms, and an adversary has significant freedom to generate such attacks. For example, it is quite feasible to write a malicious script that checks an active session and verifies whether a user is logged in or not before launching an attack. In Section B, we provide some case studies, and show how USBESAFE identifies these attacks as a set of novel observations.

For our experiments, we created eight realistic attack scenarios. In each experiment, we connected the device to the measurement machine and made sure the attack executed while logging the USB packets from the device enumeration to device termination. The malicious dataset contains 202,394 USB packets, which was significantly smaller than the benign dataset, reflecting the expected low base rate of BadUSB-style attacks.

#### 5.1.1 Preprocessing the Dataset

Over the course of the data collection, we found several unpredicted situations during the device enumeration phase. For example, a subset of USB keyboards used in our experiments were not reported as the USB class code 3, but were instead reported as the USB class code 0. Though this occurred, each observed instance of these event sequences yielded a successfully enumerated device, and the host accepted the keyboard input immediately after receiving the device descriptor. For this reason, we worked around the issue during class bucketization in the feature extraction phase, moving all the class code 0 traffic into the class code 3 bucket.

Another issue we had to account for in processing the trace files was to determine what action to take when encountering malformed packets. In some instances, when the host requested a device descriptor, the device would respond with a malformed descriptor packet, forcing the host to make the request again. For the purposes of prototype evaluation, we chose to ignore these request/response pairs when they occurred.

### 5.2 Model Selection

One of the first questions that arises is which machine learning algorithm achieves the highest detection results if it is trained with the labeled dataset. To this end, we used five different algorithms that are known to model anomaly detection problems. We also considered the *local* and *global* features of the anomaly detection approaches in order to determine whether the novelty score of an incoming URB should be determined with respect to the entire training data or solely based on a subset of previous URBs. To run the experiment, we used one-class SVM as a classifier-based approach [14], *k*-NN as a *global Nearest-neighbor* [13], and *Local Outlier Factor (LOF)* as a *local Nearest-neighbor*-based approach [13]. We also incorporated *Cluster-based Local Outlier Factor (CBLOF)* [12] as a *global Clustering*-based approach and *Local Density Cluster-based Outlier Factor (LDCOF)* [12] as a *local Clustering*-based approach.

To identify the best detection algorithm, we performed an analysis on the 423 traces we collected from five machines (see Section 5.1). More specifically, we split the USB traces of each machine to a training and a testing set using 4-fold cross-validation, and averaged the value of the detection rate and false positive rate for each algorithm. As shown in Table 2, the analyses reveal that LDCOF and LOF, which use local data points, produce lower false positive cases. However, the empirical evidence suggests that the one-class SVM achieves the best detection results among the selected algorithms on the same dataset. Based on our analysis, one likely reason is that the one-class SVM classifier maps the USB traffic to a high dimensional feature space more accurately. This results in producing less false positive cases by identifying the maximal margin hyperplane that best separates the new

| Metric | OCSVM | k-NN | LOF | CBLOF | LDCOF |
|---|---|---|---|---|---|
| TPR | 94.2% | 90.6% | 91.2 | 92.7% | 92.3% |
| FPR | 0.71% | 11.3% | 5.3% | 3.2% | 1.9% |

Table 2: The detection results of different machine learning algorithms on the labeled dataset. The analyses show that one-class SVM achieves the highest TPs with a very low FP rate on the same dataset.

observations. Based on this empirical analysis, we used the one-class SVM as our default machine learning algorithm for the rest of the paper.

### 5.2.1 Determining the Novelty Score

In the model testing process, USBESAFE applies the trained decision function to determine whether an input observation falls within the trained class or outside the trained class. We consider a URB as a *novel* observation, if the decision function assigns the input to the -1 class. In fact, the assigned value -1 implies that the input observation is outside the trained region. The novelty score is calculated as the ratio of inputs classified as novel observations over the total number of input observations. We ran an experiment by incorporating four different kernel functions to train the one-class SVM (see Appendix A), and understand what novelty score should be selected as the threshold at which we decide whether the observation is novel or not. Our analysis shows that the system produced less than 1% false positives when the threshold value = 13.2% was selected for all the four different kernel functions in the one-class SVM algorithm. In Section 5.3, we describe how we enhanced the detection model by empirically identifying a specific set of parameters for the kernel functions.

## 5.3 Optimizing the Model

Another question that we wanted to answer was whether we can improve the detection model by changing the required configuration parameters of the model on the same labeled dataset. After constructing possible n-grams (see Section 3.1.2), we performed a grid search [31] over the parameter space which consists of: (1) the one-class SVM model parameters (e.g., the polynomial degree), (2) the n-gram window size, and (3) the combinations of detection features.

Based on the resulting parameter space with 105 model parameter settings, 5 features and n-grams with window size 2 (see Table 6 in Appendix A), we generated 6,510 unique one-class SVM model instances. To test the accuracy of the models against BadUSB attacks, we created a set of attacks using a Rubber Ducky USB drive [2]. The attacks were designed to perform covert HID attacks which open a command prompt and execute a malicious code, or connect to a remote server. We elaborate on the malicious dataset later in Section 5. For each individual one-class SVM test, we logged the parameter setting used to generate the model, calculated the average accuracy across all the 4-fold cross validations, and

| Machine | No. of Traces | TPs | FPs |
|---|---|---|---|
| Machine1 | 124 | 97.4% | 0.16% |
| Machine2 | 90 | 95.6% | 0.23% |
| Machine3 | 101 | 96.7% | 0.15% |
| Machine4 | 50 | 94.0% | 0.31% |
| Machine5 | 58 | 94.3% | 0.28% |
| Per User Model (avg) | 423 | 95.7% | 0.21% |
| General Model | 423 | 94.9% | 0.93% |

Table 3: The detection results of USBESAFE on different machines. In this experiment, we used one-class SVM with the polynomial kernel with degree 3, $\gamma = 0.1$ and $\nu = 0.75$ using all the features. Our analysis shows that per user model is more effective in terms of producing lower false positive cases.

their corresponding standard deviations. We removed a model instance from our search space if the false positive rate of the model was more han 4.0%. Our assumption was that it is very unlikely that an end-point solution with a false positive rate more than 4.0% would be useful to be deployed on user machines. Our analysis shows that USBESAFE achieves the highest TP and FP rates (TP rate 95.7% at 0.21% FPs) when one-class SVM uses the polynomial kernel with degree 3, $\gamma = 0.1$ and $\nu = 0.75$ using all the features defined in Section 3.1. Table 3 shows the True Positives (TPs) and False Positives (FPs) for each user machine based on the derived model. The results show that it is possible to achieve even a higher detection rate at a lower false positive rate on the same dataset by tuning the detection model and incorporating appropriate configuration parameters.

A question that arises is whether a general multi-user model can achieve the same level of detection accuracy when being used on several machines. To test this, we incorporated all the 423 traces in the learning process and tested the detection results of USBESAFE. We observed that the general model, unsurprisingly, produced a higher false positive rate on the labeled dataset. Table 3 summaries the detection accuracy of USBESAFE in the per user and multi-user scenarios. While the general model achieved a lower detection rate with a higher false positive rate, we observed that it can be deployed temporarily on new machines while the per user model is in the training phase. We provide more details on the real-world deployment of USBESAFE in Section 6.

### 5.3.1 Feature Set Analysis

We also performed an experiment to measure the contribution of the proposed features by testing the model with the labeled datasets collected from all the five machines, and calculating the average of TPs and FPs. To this end, we used a *recursive feature elimination* (RFE) approach on the labeled dataset. We divided the feature set into three different categories: Type-based features which are transfer and event type of packets ($F_1$), Time-based features which are interarrival and post-enumeration time of the packets ($F_2$), and Content-based feature which is the payload of the packets ($F_3$). The procedure started by incorporating all the feature categories while measuring the FP and TP rates. Then, in each step, a feature set with the minimum weight was removed, and the FP and TP rates were calculated by performing 4-fold cross-validation to quantify the contribution of each feature in the proposed feature set. Table 4 provides the details of feature set evaluation using One-class SVM with the configuration parameters we tested in Section 5.3.

Our experiments show that the highest false positive rate is 43.4% and is produced when USBESAFE only incorporates type-based features. When time-based and content-based features were used together, USBESAFE achieved (1.8% FP with 94% TP). $F_{23}$ resulted in higher detection rate as USBESAFE was able to detect evasive scenarios where we intentionally imposed an artificial delay, similar to stalling code in malware attacks, before launching a command injection attack. When all the features were combined, USBESAFE achieved (0.21% FP with 95.7% TPs) on labeled dataset. Note that if USBESAFE uses a larger window size ($n = 3$), it is possible to achieve 100% TPs. However, it results in higher false positive cases as the number of suspicious sequence of USB packets also increases. Therefore, as a design decision, we decided to use the window size ($n = 2$). We provide more details in Section 5. The results clearly imply that USBESAFE achieves the highest accuracy by incorporating all the features.

| Feature Sets | FPs | TPs |
|---|---|---|
| $F_1$ | 43.4% | 54.7% |
| $F_2$ | 14% | 78% |
| $F_3$ | 16% | 69% |
| $F_{12}$ | 2.2% | 86.3% |
| $F_{13}$ | 5.6% | 65% |
| $F_{23}$ | 1.8% | 94% |
| **All Features** | 0.21% | 95.7% |

Table 4: The true positive and false positive rate for different combinations of features. The analysis shows that USBESAFE achieves the best results by incorporating all the features.

We performed another experiment to rank the relative contribution of each feature. We first incorporated all the features, and measured the FP and TP rates. Then, in each step, we removed the feature with the minimum weight, and calculated the FP and TP rates to quantify the contribution of each feature. Table 5 shows the results by ranking all the features with the most significant one at the top. For easier interpretation, we calculated the score ratio by dividing the score value of each feature with the most significant score value. The ratio of each feature simply tells how much the corresponding feature can contribute to identify novel observations.

### 5.3.2 Modeling the USB Traffic Pattern

A question that arises here is how URB arrivals can be modeled. This is an important question as we want to test the possibility of developing mimicry attacks where an adversary can bypass the proposed detection mechanism. For example, an attacker can create BadUSB attacks that generate URBs which are similar to a normal user typing pattern. Prior work revealed that user-generated traffic arrivals such as Telnet can be well modeled as Poisson distribution [18]. To test whether the URB arrivals follow Poisson distribution, we ran a simple statistical methodology where we tested whether the URB arrivals follow *exponentially distributed* and *independent* interarrivals – the two requirements for Poisson distribution.

| Rank | Category | Feature | Type | Score Ratio |
|---|---|---|---|---|
| 1 | Time | Packet Interarrival Times | Continuous | 100% |
| 2 | Content | Packet Payload | Ordinal | 83.2% |
| 3 | Time | Post-enumeration Time | Continuous | 35.6% |
| 4 | Type | Event Type | Categorical | 14.4% |
| 5 | Type | Transfer Type | Categorical | 12.1% |

Table 5: The rank of each feature in USBESAFE to detect novel observations.

To this end, we randomly selected 100 traces from the labeled dataset. Figure 2 represents the results of the analysis. The x-axis represents the percentage of the intervals in the traces that follow exponentially distributed interarrivals and the y-axis represents the percentage of the intervals that follow independent interarrivals. We used Anderson – Darling test [1] to verify whether the interarrivals follow an exponential distribution. To test the interarrivals for independence, one simple way is to check whether there is significant autocorrelation among URB arrivals in a given time lag. To this end, we used Durbin – Watson statistics [29] to test the autocorrelation among URBs. As shown in the figure, more than 95% of the intervals pass the test showing that the URB arrivals are truly Poisson. We use this finding to generate mimicry attacks and test whether the system can detect attacks that follow Poisson arrivals (see Appendix B).

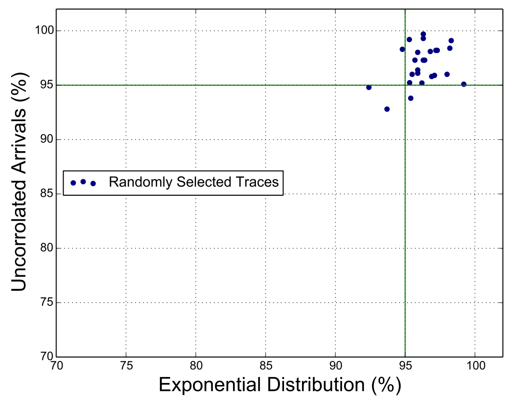
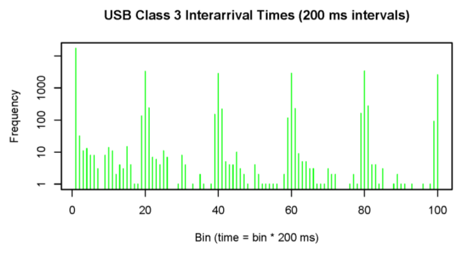


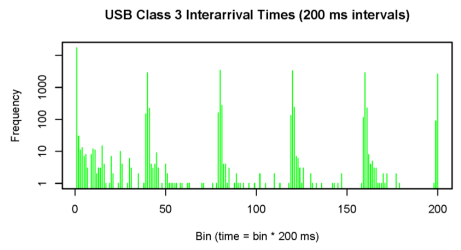
Figure 2: The result of statistical analysis on 100 randomly selected traces. Our analysis shows that the URB arrivals can be well modeled by Poisson arrivals.

### 5.3.3 Determining the Effect of Pause Time

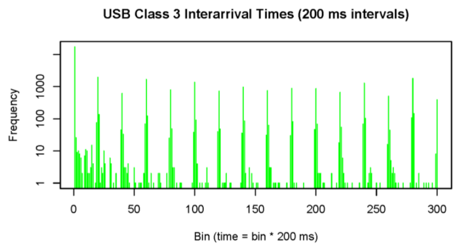
As mentioned in Section 3.1, to tackle the issue of the unbounded interarrival time value between two consequent USB



(a) Payload histogram on a 20,000 ms pause with bin intervals of 200 ms.



(b) Payload histogram on a 40,000 ms pause with bin intervals of 200 ms.



(c) Payload histogram on a 60,000 ms pause with bin intervals of 200 ms.

Figure 3: The effect of pause time value on payload histogram. The figures show localized modality occurs approximately every 4,000 ms with large spikes.

packets, we defined two configuration parameters: pause time and session. A session is a series of USB packets where the interarrival time value within the series does not exceed a specified pause length. To determine the impact of pause time value, we performed a set of experiments. The experiments had multiple goals: (1) to empirically characterize the distribution of the benign interarrival time values; (2) to find out whether varying the pause time value has any impact on the volume of information in each session as well as n-gram construction; and (3) to define the maximum interarrival time value between two TraceEvents before we consider the user to be starting a new typing session. To determine an optimal pause time, we examined three pause time candidates (in milliseconds): 20,000, 40,000, and 60,000 milliseconds, with a sampling period of 200 and 500 milliseconds. For each pause time value, we normalized the values for class codes 0 and 3. More specifically, when a raw interarrival time value $i$ was greater than the pause time, we reset $i$ to 0, thereby starting a new session.

We observed some common patterns by generating payload histograms while varying pause values and interval lengths. An interesting observation for the HID traffic was that, regardless of the pause time value, a localized modality occurs approximately every 4000 ms, or 4 seconds, with large spikes in the number of packets transmitted during these times. Ultimately, the results of this experiment revealed that there was minimal information payload differences among the pause time values used, indicating that the value we chose is not consequential to overall model performance. For this reason, we set the pause to our lowest value of 20,000 ms. Figure 3 shows the details of this experiment.

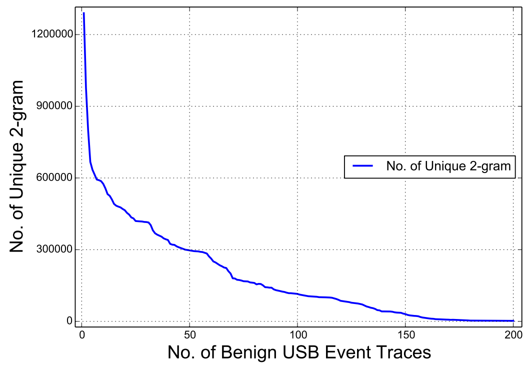


Figure 4: Unique 2-grams for the first 200 USB packet traces in our dataset. The number of unique sequences significantly decreases as USBeSafe observes more USB packets.

### 5.3.4 Determining the Effect of N-Grams

To understand the diversity of the collected USB packets for the USB HID class, we performed an experiment on constructing n-grams by varying the value of $n$ from 2 to 3. First, we examined the number of unique 2-grams that can be found in the first 200 USB trace files which contained 5,938,492 USB packets. The number of unique 2-grams on labeled dataset is shown in Figure 4. As depicted, the number of unique sequences significantly decreases as USBESAFE observes more USB packets. The finding suggests that n-grams can closely capture the characteristics of the benign dataset. That is, if the model is deployed, it is unlikely that benign keyboard activities will not have been observed in our training phase, resulting in low false alarms.

To verify this, we performed an experiment that incorporated the entire labeled dataset that is a representative mix of possible BadUSB attacks as well as benign USB HIDs. We varied $n$ and the threshold $k$ of malicious n-grams that need to be observed before a USB device is flagged as malicious. The results for $n = 2$ and $n = 3$ and $k$ ranging over an interval from 1 to 50 are evaluated. Figure 5 shows the results of the analysis. As depicted, the detection rates are very high, specially for small values of $k$. The false positive rate is 0.21% for $k = 3$.

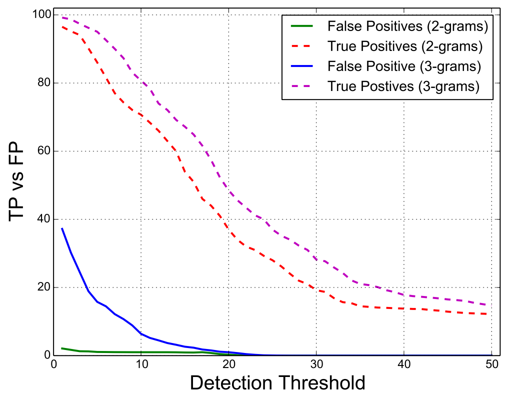


Figure 5: Detection results for 2-grams and 3-grams. The detection threshold $k$ is on the X-axis (e.g., $k = 2$ and $n = 3$ means that a USB trace must match two 3-grams to generate an alert).

## 6 Real-world Deployment

The main goal of this experiment is to evaluate the detection accuracy of USBESAFE by incorporating an unlabeled dataset which has not been observed during the training phase. We incorporated the results of our previous measurement on the labeled datasets by setting the window size and number of n-grams to established values ($n = 2$ and $k = 3$). For a real-world deployment, we first need to determine how much data is required for initial model training if a new user decides to use USBESAFE as a service. To answer this question, we ran an experiment on seven new machines for 20 days. Depending on the type of machines and their usage, multiple HID and non-HID devices were connected to the machines. This resulted in generating different numbers of trace files per machine. Therefore, for easier interpretation, we performed the tests by varying the number of days, referred to as the training window size, instead of the number of trace files as five out of seven machines had more than one trace file per day. We also generated five new BadUSB attacks (e.g., establishing covert channels, logging keyboard activity) for testing.

To run the test, we varied the training window size from 1 to 20 days for all the machines and computed TP and FP rates to determine the optimal training window size. The result of this experiment showed that USBESAFE requires two to four training days to keep the TP rate over 93% with a 0.9% FP rate in all the machines. Our analysis on the training window size also showed that the machines with two connected HID devices (a keyboard and a mouse) do not usually need more than three training days to reach that level of detection accuracy.

We ran another experiment to test whether the general model that was constructed based on our labeled 423 trace files would work well in the new machines. We observed that USBESAFE achieved on average 90% TP rate with a 2.2% FP rate across all the machines. In fact, per user deployment model achieved a higher detection rate with a significantly lower false positive rate (FP = 0.9%) at the cost of two to four training days. However, since the general model does not require any initial training for a large-scale deployment, we recommend temporarily activating the general model on a new machine while USBESAFE is in the training phase.

We also deployed the general model on three multi-user machines for five consecutive days. We did not receive any complaint from users during the test period. However, we cannot provide any strong security guarantees to protect multi-user machines or produce low false positive rate as we do not have enough data to make any statistically significant claim on the accuracy of USBESAFE for this deployment option. Furthermore, recall that one of the main design goals of USBESAFE was to reduce the risk of BadUSB attacks – a form of targeted attacks on end-users. Consequently, the architecture, feature selection, and implementation details make USBESAFE a more effective solution for single user machines. In fact, protecting multi-user machines has a different set of security and privacy requirements and is out of the scope of this paper.

### 6.1 Re-training the Detection Model

USBESAFE should counter the problem of *model drift*, in which the constructed model makes an assumption that the incoming USB packets will exhibit *new normal patterns* that have not been observed during the training phase. For example, users' typing patterns can change for various reasons (e.g., completing a specific task) or URBs interarrival rate might change across different devices which might affect the detection accuracy. Therefore, a practical deployment of USBESAFE requires periodically re-training the system. To simulate a practical deployment, we started an experiment by training USBESAFE on all the new machines and tested with the attack payloads we developed. Our analysis shows that, based on the labeled dataset and subsequent data collection, training USBESAFE with an initial dataset similar to ours and re-training every 16 days were sufficient to maintain the detection rate over 93% with less than 1% false positive cases across all the machines.

The re-training process, including the false positive and false negative analysis, usually took on average 2.1 hours each time during the course of experiment. More specifically, the time needed to re-train the model and address model drift was a function of the size of input data which took approximately 82 seconds on average every 16 days on normal PCs and laptops. However, the manual intervention for evaluating the results was almost inevitable. We had to verify *how* and *why* false positive or false negative cases occurred, and whether they were produced as a result of model drift or an evasive attack. In a real-world deployment, USBESAFE requires only the re-training schedule which is less than two minutes. In Section 7, we provide more details on the risk of adversaries' malicious influence during the re-training process.

### 6.2 Evaluating False Positives

During 20 days of experiment, the system processed 3,434,452 USB packets across seven machines. To speed up the false positive analysis, we asked the users to log the number of times, the exact time and date they received the system's alert. We received false positive reports on two machines. A more in-depth analysis revealed that all the false positive cases in one of the machines were produced in two consecutive days when the user was filling out a set of web forms with random data for research purposes. USBESAFE detected these USB packets as new observations because the payload histograms as well as the interarrival time values among the URBs were following a significantly different pattern with the novelty score 32%.

We observed that the false positive cases on the other machine was because of running a user study experiment in which several users were asked to run a test on the perceived functionality of websites by interacting with them while triggering their event listeners. A few users in that experiment were typing random characters in multiple fields of web forms in those websites. In fact, the false positive cases in both machines were very similar in a sense that they were flagged by USBESAFE when the users performed a set of activities that did not match with their normal interaction with the machines. We did not encounter any other cases of legitimate USB packets being incorrectly reported. These results are in fact quite encouraging as the experiment was performed on a set of new machines with relatively small training window size compared to our first experiment without imposing a discernible impact on the detection accuracy of USBESAFE.

## 7 Discussions and Limitations

Note that a fundamental design goals of USBESAFE is to keep the protection mechanism completely in the background. We assume that adversaries have significant freedom in providing varying responses for device identities to evade potential defense mechanisms. Furthermore, adversaries can convince users to connect seemingly benign devices to hosts for various reasons. Consequently, shifting the burden of responsibility to users to verify the reported identity, and decide on *unknown* devices is less likely to be a very reliable defense mechanism. In this section, we discuss the limitations of USBESAFE, and the implications of these limitations on the detection results.

First, recall that USBESAFE is an anomaly-based detection system where the detection results depend on the quality and volume of the trained dataset. If an attack occurs during the learning phase, USBESAFE accepts data or behavior that would otherwise be considered malicious. Therefore, an additional analysis should be performed on the authenticity of the new data for re-training purposes to prevent such malicious influences. This may increase the cost of the data collection as the proposed model is a per user solution. Furthermore, as mentioned earlier, an attacker can try to imitate benign USB traffic patterns and evade the detection mechanism. An attacker can be successful in running these attacks, if she is able to accurately learn the typing behavior of the target user. Our analysis shows that automatically injecting artificial delays (See Appendix B) can decrease the novelty score of USB traffic. However, it cannot entirely change the traffic patterns, or possibly adapt to each user typing pattern.

Second, recall that one of the primary design decisions of USBESAFE is to treat existing operating systems in a black box fashion, and build a central security model of USBESAFE independent of the user's perception of malice. However, USBESAFE cannot provide strong protection guarantees against scenarios where an adversary attempts to trick users into voluntarily disabling USBESAFE, for instance, by mimicking the output of USBESAFE, and forcing the user to disable protection. We stress that these issues are fundamental to any host-based protection tool.

Third, USBESAFE cannot provide any security guarantees in scenarios where an adversary has a privilege to run code in the kernel. In fact, if an adversary can successfully run malicious code in the kernel, she can also disable all the possible defense mechanisms, including USBESAFE. For this reason, we explicitly consider kernel-level attacks outside the scope of USBESAFE's threat model. Despite all the limitations, USBESAFE provides important practical security benefits that complement the standard USB protocol employed in operating systems without any significant detriments to performance.

## 8 Conclusion

In this paper, we empirically show that it is possible to develop models that can accurately explain the the nature of USB traffic. We presented the design and implementation of USBESAFE, and demonstrated that it can successfully block modern BadUSB-style attacks without relying on end-user security decisions or requiring changes in the current USB protocol or the operating system. We hope that the concepts we propose will be useful for end-point protection providers and facilitate creating similar services on other platforms to enhance defense mechanisms against future malicious devices.

## Acknowledgements

This work was partially supported by the Office of Naval Research (ONR) under grant N00014-19-1-2364 award and the United States Air Force under Air Force Contract No. FA8702-

15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.
**DISTRIBUTION STATEMENT A.** Approved for public release. Distribution is unlimited.

## References

[1] *Anderson–Darling Test*. Springer New York, New York, NY, 2008, pp. 12–14.

[2] USB Rubber Ducky. https://hakshop.com/products/usb-rubber-ducky-deluxe, 2017.

[3] ANGEL, S., WAHBY, R. S., HOWALD, M., LENERS, J. B., SPILO, M., SUN, Z., BLUMBERG, A. J., AND WALFISH, M. Defending against malicious peripherals with cinch. In *USENIX Security Symposium* (2016).

[4] BATES, A. M., LEONARD, R., PRUSE, H., LOWD, D., AND BUTLER, K. R. B. Leveraging USB to establish host identity using commodity devices. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014* (2014).

[5] BROCKER, M., AND CHECKOWAY, S. iseeyou: disabling the macbook webcam indicator led. In *Proceedings of the 23rd USENIX conference on Security Symposium* (2014), USENIX Association, pp. 337–352.

[6] DIWAN, S., PERUMAL, S., AND FATAH, A. Complete security package for usb thumb drive. *Computer Engineering and Intelligent Systems 5*, 8 (2014), 30–37.

[7] HERNANDEZ, G., FOWZE, F., YAVUZ, T., BUTLER, K. R., ET AL. Firmusb: Vetting usb device firmware using domain informed symbolic execution. *ACM Conference on Computer and Communications Security* (2017).

[8] JIM WALTER. "Flame Attacks": Briefing and Indicators of Compromise. http://downloadcenter.mcafee.com/products/mcafee-avert/sw/old_mfe_skywiper_brief_v.1.pdf.zzz, 2012.

[9] KARSTEN NOHL, SACHA KRIBLER, JAKOB LELL. BadUSB–On accessories that turn evil. BlackHat, 2014.

[10] KHARAZ, A., ARSHAD, S., MULLINER, C., ROBERTSON, W., AND KIRDA, E. UNVEIL: A large-scale, automated approach to detecting ransomware. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 757–772.

[11] KOLBITSCH, C., KIRDA, E., AND KRUEGEL, C. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 285–296.

[12] LEARN, S. Anomaly detection with Local Outlier Factor (LOF). http://scikit-learn.org/stable/auto_examples/neighbors/plot_lof.html.

[13] LEARN, S. Nearest Neighbors. http://scikit-learn.org/stable/modules/neighbors.html.

[14] LEARN, S. One Class SVM. http://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html.

[15] LETAW, L., PLETCHER, J., AND BUTLER, K. Host identification via usb fingerprinting. In *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on* (2011), IEEE, pp. 1–9.

[16] NEUGSCHWANDTNER, M., BEITLER, A., AND KURMUS, A. A transparent defense against usb eavesdropping attacks. In *Proceedings of the 9th European Workshop on System Security* (2016), ACM, p. 6.

[17] NICOLAS FALLIERE, LIAM O MURCHU, ERIC CHIEN. W32. Stuxnet Dossier. http://www.bbc.com/news/technology-36478650, 2011.

[18] PAXSON, V., AND FLOYD, S. Wide area traffic: the failure of poisson modeling. *IEEE/ACM Transactions on Networking (ToN) 3*, 3 (1995), 226–244.

[19] PHAM, D. V., HALGAMUGE, M. N., SYED, A., AND MENDIS, P. Optimizing windows security features to block malware and hack tools on usb storage devices. In *Progress in electromagnetics research symposium* (2010), pp. 350–355.

[20] RICH, D. Authentication in transient storage device attachments. *Computer 40*, 4 (2007).

[21] SCHUMILO, S., AND SPENNEBERG, R. Don't trust your usb! how to find bugs in usb device drivers.

[22] SHIN, S., AND GU, G. Conficker and beyond: a large-scale empirical study. In *Proceedings of the 26th Annual Computer Security Applications Conference* (2010), ACM, pp. 151–160.

[23] TETMEYER, A., AND SAIEDIAN, H. Security threats and mitigating risk for usb devices. *IEEE Technology and Society Magazine 29*, 4 (2010), 44–49.

[24] Tian, D. J., Bates, A., and Butler, K. Defending against malicious usb firmware with goodusb. In *Proceedings of the 31st Annual Computer Security Applications Conference* (2015), ACM, pp. 261–270.

[25] Tian, D. J., Bates, A., Butler, K. R., and Rangaswami, R. Provusb: Block-level provenance-based data protection for usb storage devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS '16, pp. 242–253.

[26] Tian, D. J., Scaife, N., Bates, A., Butler, K., and Traynor, P. Making usb great again with usbfilter. In *Proceedings of the USENIX Security Symposium* (2016).

[27] Tian, J., Scaife, N., Kumar, D., Bailey, M., Bates, A., and Butler, K. Sok: "plug & pray" today - understanding usb insecurity in versions 1 through c. In *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, pp. 613–628.

[28] Tischer, M., Durumeric, Z., Foster, S., Duan, S., Mori, A., Bursztein, E., and Bailey, M. Users Really Do Plug in USB Drives They Find. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P '16)* (San Jose, California, USA, May 2016).

[29] Watson, G. S., and Durbin, J. Exact tests of serial correlation using noncircular statistics. *The Annals of Mathematical Statistics* (1951), 446–451.

[30] Yang, B., Qin, Y., Zhang, Y., Wang, W., and Feng, D. Tmsui: A trust management scheme of usb storage devices for industrial control systems. In *International Conference on Information and Communications Security* (2015), Springer, pp. 152–168.

[31] Zhuang, L., and Dai, H. Parameter optimization of kernel-based one-class classifier on imbalance learning. *Journal of Computers 1*, 7 (2006), 32–40.

## A Model Search Configurations

Table 6 represents all the possible configurations for training the detection model.

| Configuration Setting | Values | # |
|---|---|---|
| **Error Upper Bound ($\nu$)** | [0.01, 0.25, 0.5, 0.75, 1] | – |
| **Kernel Coefficient ($\gamma$)** | [0.1, 0.01, 0.001, 0.0001] | – |
| **Degree of Polynomial** | [1, 2, 3] | – |
| **Kernel Functions** | RBF, $\nu, \gamma$ | 20 |
| | Sigmoid, $\nu, \gamma$ | 20 |
| | Linear, $\nu$ | 5 |
| | Polynomial, $\nu, \gamma$, degree | 60 |
| **Total** | | 105 |

Table 6: All combinations of parameters for any applicable $\nu$, $\gamma$, and degree settings for each kernel option. Defining this parameter space results in 105 parameter settings to apply to SVM instances.

## B Case Studies

As mentioned earlier, an attacker has significant freedom in developing malicious code that can potentially bypass USBeSafe. Therefore, as an end-point solution, it is quite useful to study how the system responds to different levels of attack sophistication. To this end, we ran each of the following attacks, collected the corresponding USB traces, and measured the percentage of USB packets in each attack that was novel to the system based on the model learned on each machine.

**Attack No. 1: Running a Malicious Payload** By running a malicious payload, we specifically focus on executing commands to call a binary that downloads code from the Internet, and installs malware. Note that this attack can be designed to be as stealthy as possible. For example, the malicious code can start when the user is logged off with the assumption that the user is very likely not physically present.

Our analysis showed that this attack had an average novelty score of 47.9% when tested with different learned models. Our further investigation revealed that the USB packets received a relatively high novelty score compared to the learned model because the interarrival time values among URBs was

| Machine | Attack1 | Attack2 | Attack3 |
|---|---|---|---|
| **Machine6** | 63.2% | 58.2% | 37.3% |
| **Machine7** | 54.5% | 52.5% | 42.4% |
| **Machine8** | 49.8% | 40.8% | 19.2% |
| **Machine9** | 31.5% | 17.4% | 30.8% |
| **Machine10** | 41.2% | 42.8% | 30.6% |
| **Machine11** | 46.5% | 47.6% | 29.8% |
| **Machine12** | 49.1% | 49.3% | 33.6% |
| **Average** | 47.9% | 44.1% | 27.1% |

Table 7: the novelty score of the evasion tests in the real-world deployment. The novelty score of all the attacks are significantly higher than the threshold value ($t = 13.2\%$).

significantly smaller than most of real user typing behaviors. Furthermore, the 2-gram analysis shows that the average content histogram of the first 103 request packets were more than 195 during the command injection which was significantly higher than the content of the USB packets in the benign dataset.

**Attack No. 2: Adding Artificial Delays** A question that arises is that in the previous attack, the malicious code launched a list of commands immediately after the enumeration phase. We updated the code to wait for a random period of time similar to the stalling code in malware attacks [10,11], and then open a terminal to run the commands. Our analysis revealed that this attack could bypass the post-enumeration feature by waiting for a random period of time before running the commands. As shown in Table 7, compared to attack No. 1, the novelty score of the malicious code decreased in all the machines. However, USBESAFE reported this attack as a new observation as the interarrival of the packets was still too small.

**Attack No. 3: Manipulating the Interarrival Times** We enhanced the attack payload to be more stealthy by adding delays among the injected commands in order to simulate human typing patterns. The delays were injected such that the arrivals of URBs followed Poisson distribution. We used Poisson distribution as we observed in Section 5 that the URBs' interarrivals in our labeled dataset can be well-modeled using Poisson. While the novelty score of the USB traffic in attack 3 (see Table 7) is relatively lower than the novelty score of the other attacks, the attack is still detected since the novelty scores of the USB traffic in all the traces are significantly higher than the pre-defined threshold (t = 13.2%). Further analysis suggests that injecting artificial delays with Poisson distribution during the command injection phase is not sufficient to automatically generate very serious mimicry attacks that perfectly resemble users' typing patterns. In fact, we empirically found that to successfully run such attacks, the adversary needs a more precise mechanism to learn the normal typing behavior of individual users. This makes crafting mimicry attacks more complicated as the adversary has to incorporate other techniques to reliably hook certain OS functions in order to learn the typing pattern of each user. This particular area has been studied extensively in malware detection, i.e., spyware detection, and is out of the scope of this paper.

## C Benchmarks

Since USBESAFE is intended as an online monitoring system, it may impact the performance of other applications or the operating system. We expect USBESAFE's performance overhead to be overshadowed by I/O processing delays, but in order to obtain measurable performance indicators and characterize the overhead of USBESAFE, we ran experiments that exercised the critical performance paths of USBESAFE. Note that designing custom test cases and benchmarks require careful consideration of factors that might influence our runtime measurements. In these tests, we mainly focused on the core parts of the USB device communication which were the *USB device enumeration* and *data transfer* mechanisms. We explain each of these benchmarks in more details below.

**Device Enumeration** In the first experiment, we tested whether USBESAFE introduces any noticeable performance impact during the USB enumeration phase. The testing USB device was a headset which had a HID interface. We manually plugged the headset into the host 20 times and compared the results between USBESAFE-enhanced host and the standard machine. The average USB enumeration time was 37.4 ms for the standard system and 39.1 ms for the USBESAFE-enhanced host respectively. Comparing to the standard host, USBESAFE only introduced 4.5% or less than 2 ms. We created the same benchmark using a mouse, and repeated it for 20 times. The system imposed 4.1% or 1.4 ms for device enumeration. The measurement results imply that USBESAFE does not have a significant impact on the enumeration of USB devices. More details are provided in Table 8.

**USB Packet Inspection** In the second experiment, we created a benchmark to characterize the performance overhead of our system during a normal device use. To measure the overhead of the detection model, we plugged in a USB optical mouse and moved it around to generate USB traffic. We then measured the time used by USBESAFE to determine whether the incoming USB packets should be filtered or not. The required time is calculated from the time a URB is delivered to the packet inspection subsystem to the time the packet is analyzed by the protection engine. We tested the experiment on the first 2,000 URBs, and repeated the experiment 10 times as shown in Table 8. As shown, the average cost per URB is 12.7 $\mu s$, including the time used by the benchmark to get the timing and print the results.

**File System** We also created a benchmark to measure the latency of file operations under the baseline and USBESAFE-enabled machines. The goal of the experiment is to measure the performance overhead of the system during normal usage of a USB storage device, where users plug in flash drives to copy or edit files. We ran the experiments using a 16 GB USB flash drive and varied file sizes from 1 KB to 1 GB. Each test was done 10 times and the average was calculated. As shown, the throughput of USBESAFE is close to baseline when the file size is less than 100 MB (approximately 3.9%). When the mean file size becomes greater than 100 MB, USBESAFE shows lower throughput compared to the standard machine

| **Experiment** | **Device** | **Standard** | **USBESAFE** | **Overhead** |
|---|---|---|---|---|
| Enumeration | *Head Set* | 37.4 ms | 39.1 ms | 4.5% |
| | *Mouse* | 33.5 ms | 34.9 ms | 4.1% |
| | *Keyboard* | 34.2 ms | 35.6 ms | 4.2% |
| | *Mass Storage* | 36.6 ms | 38 ms | 3.9% |
| **Overhead Mean** | | 35.4 ms | 36.8 ms | 4.2% |
| Event Inspection | *Mouse* | - | 12.3 $\mu s$ | - |
| | *Keyboard* | - | 13.1 $\mu s$ | - |
| **Overhead Mean (per packet)** | | - | 12.7 $\mu s$ | - |

Table 8: USBESAFE's overhead on the USB communication protocol. USBESAFE imposes on average 4.2% overhead during the enumeration phase and 12.7 $\mu s$ per packet during USB packet inspection.

as a result of pattern monitoring on the bus. The results show that USBESAFE imposes 7.2% and 11.4% overhead when the mean file sizes are 100 MB and 1 GB respectively. For example, if a user wants to copy 10 100 MB files, throughput would drop from 8.9 MB/s to 8.26 MB/s when USBESAFE is enabled on the user's machine.