Making USB Great Again with USBFILTER

Dave (Jing) Tian and Nolen Scaife, *University of Florida;* Adam Bates, *University of Illinois at Urbana–Champaign;* Kevin R. B. Butler and Patrick Traynor, *University of Florida*

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tian

This paper is included in the Proceedings of the
25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

Open access to the Proceedings of the
25th USENIX Security Symposium
is sponsored by USENIX

Making USB Great Again with USBFILTER

Dave (Jing) Tian⋆, Nolen Scaife⋆, Adam Bates†, Kevin R. B. Butler⋆, and Patrick Traynor⋆
⋆ University of Florida, Gainesville, FL
† University of Illinois, Urbana-Champaign, IL
{daveti,scaife,adammbates,butler,traynor}@ufl.edu

Abstract

USB provides ubiquitous plug-and-play connectivity for a wide range of devices. However, the complex nature of USB obscures the true functionality of devices from the user, and operating systems blindly trust any physically-attached device. This has led to a number of attacks, ranging from hidden keyboards to network adapters, that rely on the user being unable to identify all of the functions attached to the host. In this paper, we present USBFILTER, which provides the first packet-level access control for USB and can prevent unauthorized interfaces from successfully connecting to the host operating system. USBFILTER can trace individual USB packets back to their respective processes and block unauthorized access to any device. By instrumenting the host's USB stack between the device drivers and the USB controller, our system is able to filter packets at a granularity that previous works cannot — at the lowest possible level in the operating system. USBFILTER is not only able to block or permit specific device interfaces; it can also restrict interfaces to a particular application (e.g., only Skype can access my webcam). Furthermore, our experimental analysis shows that USBFILTER introduces a negligible (3-10μs) increase in latency while providing mediation of all USB packets on the host. Our system provides a level of granularity and extensibility that reduces the uncertainty of USB connectivity and ensures unauthorized devices are unable to communicate with the host.

1 Introduction

The Universal Serial Bus (USB) provides an easy-to-use, hot-pluggable architecture for attaching external devices ranging from cameras to network interfaces to a single host computer. USB ports are pervasive; they can often be found on the front, back, and inside of a common desktop PC. Furthermore, a single USB connector may connect multiple device classes. These composite devices allow disparate hardware functions such as a microphone and speakers to appear on the same physical connector (e.g., as provided by a headset). In the host operating system, technologies such as USBIP [21] provide the capability to remotely connect USB devices to a host over a network. The result is a complex combination of devices and functionalities that clouds the user's ability to reason about what is actually connected to the host.

Attacks that exploit this uncertainty have become more prevalent. Firmware attacks such as BadUSB [27] modify benign devices to have malicious behavior (e.g., adding keyboard emulation to a storage device or perform automatic tethering to another network). Hardware attacks [1] may inject malware into a host, provide RF remote control capabilities, or include embedded proxy hardware to inject and modify USB packets. Attackers may also exfiltrate data from the host by leveraging raw I/O (e.g., using libusb [14]) to communicate with the USB device directly, or bypass the security mechanism employed by the USB device controller by sending specific USB packets to the device from the host USB controller [4]. Unfortunately, the USB Implementers Forum considers defending against malicious devices to be the responsibility of the user [44], who is unlikely to be able to independently verify the functionality and intent of every device simply by its external appearance, and may just plug in USB devices to take a look [43].

Modern operating systems abstract USB authorization to physical control, automatically authorizing devices connected to the host, installing and activating drivers, and enabling functionality. We believe that a finer-grained control over USB is required to protect users. In this paper, we make the following contributions:

- **Design and develop a fine-grained USB access control system**: We introduce USBFILTER, a packet-level firewall for USB. Our system is the first to trace individual USB packets back to the source or destination process and interface. USBFILTER rules can stop attacks on hosts by identifying and dropping unwanted USB packets before they reach their destination in the host operating system.

- **Implement and characterize performance**: We demonstrate how USBFILTER imposes minimal overhead on USB traffic. As a result, our system is well-suited for protecting any USB workload.

- **Demonstrate effectiveness in real-world scenarios**: We explore how USBFILTER can be used to thwart attacks and provide security guarantees for benign devices. USBFILTER can pin devices (e.g., webcams) to approved programs (e.g., Skype, Hangouts) to prevent malicious software on a host from enabling or accessing protected devices.

USBFILTER is different from previous works in this space because it enables the creation of rules that explicitly allow or deny functionality based on a wide range of features. GoodUSB [41] relies on the user to explicitly allow or deny specific functionality based on what the device reports, but cannot enforce that the behavior of a device matches what it reports. SELinux [35] policies and PinUP [13] provide mechanisms for pinning processes to filesystem objects, but USBFILTER expands this by allowing individual USB packets to be associated with processes. This not only allows our system to permit pinning devices to processes, but also individual interfaces of composite devices.

Our policies can be applied to differentiate individual devices by identifiers presented during device enumeration. These identifiers, such as serial number, provide a stronger measure of identification than simple product and vendor codes. While not a strong authentication mechanism, USBFILTER is able to perform filtering without additional hardware. The granularity and extensibility of USBFILTER allows it to perform the functions of existing filters [41] while permitting much stronger control over USB devices.

The remainder of this paper is structured as follows: In Section 2, we provide background on the USB protocol and explain why it is not great anymore; in Section 3, we discuss the security goals, design and implementation of our system; in Section 4, we discuss how USBFILTER meets our required security guarantees; in Section 5, we evaluate USBFILTER and discuss individual use cases; in Section 6, we provide additional discussion; in Section 7, we explore related work; and in Section 8, we conclude.

2 Background

A USB device refers to a USB transceiver, USB hub, host controller, or peripheral device such as a human-interface device (*HID*, e.g., keyboard and mouse), printer, or storage. However, the device may have multiple functions internally, known as *interfaces*. An example device with three interfaces is shown in Figure 1. USB devices with more than one interface are known as composite devices. For example, USB headsets often have at least three interfaces: the speaker, the microphone, and the volume control functionalities. Each interface is treated as an independent entity by the host controller. The operating system loads a separate device driver for each interface on the device.

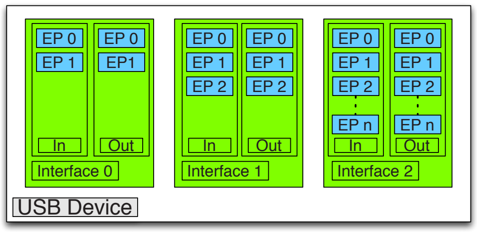

Figure 1: A detailed view of a generic USB device. Similar to a typical USB headset, this device has three interfaces and multiple endpoints.

The USB protocol works in a master-slave fashion, where the host USB controller is responsible to poll the device both for requests and responses. When a USB device is attached to a host machine, the host USB controller queries the device to obtain the *configurations* of the device, and activates a single configuration supported by the device. For instance, when a smartphone is connected with a host machine via USB, users can choose it to be a storage or networking device. By parsing the current active configuration, the host operating system identifies all the *interfaces* contained in the configuration, and loads the corresponding device drivers for each interface. This whole procedure is called *USB enumeration* [10]. Once a USB device driver starts, it first parses the *endpoints* information embedded within this interface as shown in Figure 1.

While the interface provides the basic information for the host operating system to load the driver, the *endpoint* is the communication unit when a driver talks with the USB device hardware. Per specification, the endpoint 0 (EP0) should be supported by default, enabling *Control* (packet) transfer from a host to a device to further probe the device, prepare for data transmission, and check for errors. All other endpoints can be optional though there is usually at least EP1, providing *Isochronous*, *Interrupt*, or *Bulk* (packet) transfers, which are used by audio/video, keyboard/mouse, and storage/networking devices respectively. All endpoints are grouped into either *In* pipes, where transfers are from the device to the host, or *Out* pipes, where transfers are from the host to the device. This in/out pipe determines the transmission direction of a USB packet. With all endpoints set up, the driver is able to communicate with the device hardware by submitting USB packets with different target endpoints, packet types, and directions. These packets are delivered to the host controller, which calls the controller hardware to encode USB packets into electrical signals and send them to the device.

2.1 Why USB Was Great

Prior to USB's introduction in the 1990s, personal computers used a number of different and often platform-specific connectors for peripherals. Serial and parallel ports, PS/2, SCSI, ADB, and others were often not hot-pluggable and required users to manually set configuration options (such as the SCSI ID). The widespread industry adoption of USB fixed many of these issues by providing a common specification for peripherals. Hardware configuration is now handled exclusively by the host, which is able to manage many devices on a single port. The relative ease with which a USB peripheral can be installed on a host is simultaneously its greatest and most insecure property.

The USB subsystem has been expanded in software as well, with Virtio [30] supporting I/O virtualization in KVM, enabling virtual USB devices in VMs, and passing through the physical devices into VMs. USBIP [21] transfers USB packets via IP, making remote USB device sharing possible. Wireless USB (WUSB) [19] and Media Agnostic USB (MAUSB) [16] promote the availability of USB devices by leveraging different wireless communication protocols, making the distinction among local USB devices, virtual ones, and remote ones vanish.

Overall, the utility and complexity of USB has been steadily increasing in both hardware and software. Advances in circuit and chip design now allow hidden functionality to be placed inside the USB plug [1]. The ease-of-use that made USB great now threatens users by obscuring the individual interfaces in a USB device.

2.2 How USB Lost its Greatness

Attacks on USB prey on the fundamental misunderstanding of how devices are constructed from interfaces. Attacks such as BadUSB [27] and TURNIPSCHOOL [1] (itself designed on specifications from nation-state actors) use composite devices to present multiple interfaces to a host. Often these include one benign or expected interface and one or more malicious interfaces, including keyboards [9, 27] and network interfaces [27, 1]. Without communicating with the host operating system, a malicious USB device can only obtain power from the host. While it may be possible to perform power analysis attacks without sending USB packets, we focus on the problem of connecting malicious devices to the host's operating system. All of these attacks share a common thread: they attach an unknown interface to a host without the user's knowledge. Since operating systems implicitly trust any device attached, these hidden functions are enumerated, their drivers are loaded, and they are granted access to the host with no further impediment.

Data exfiltration from host machines may be the main reason why USB storage is banned or restricted in enterprise and government environments. Current secure storage solutions rely on access control provided by the host operating system [23] or use network-based device authentication [22]. While access controls can be bypassed by raw I/O, which communicates to the device directly from userspace (e.g., using libusb [14]), network-based methods are vulnerable to network spoofing (e.g., ARP spoofing [32] and DNS spoofing [36]). It is thus unclear whether data exfiltration has occurred or not until the USB port is glued or locked [39]. The remainder of this paper will show how a packet-level filter for USB permits fine-grained access controls, eliminating the implicit trust model while providing strong guarantees.

3 USB Access Control

The complex nature of the USB protocol and the variety of devices that can be attached to it makes developing a robust and efficient access control mechanism challenging. Layers in the operating system between the process and the hardware device create difficulties when identifying processes. Accordingly, developing a system such as USBFILTER is not as simple as intercepting USB packets and dropping those that match rules. In this section, we discuss our security goals, design considerations, and implementation of USBFILTER while explaining the challenges of developing such a system.

3.1 Threat and Trust Models

We consider an adversary against our system who has restricted external physical or full network access to a given host. The adversary may launch physical attacks such as attaching unauthorized USB devices to the host system or tampering with the hardware of previously-authorized devices to add additional functionality. The physically-present adversary may not open the device or tamper with the internal storage, firmware, or any other hardware. This type of adversary might (for example) be present in an data center or retail location, where devices have exposed USB ports, but tampering with the chassis of the device would raise suspicion or sound alarms. The adversary may also launch network attacks in order

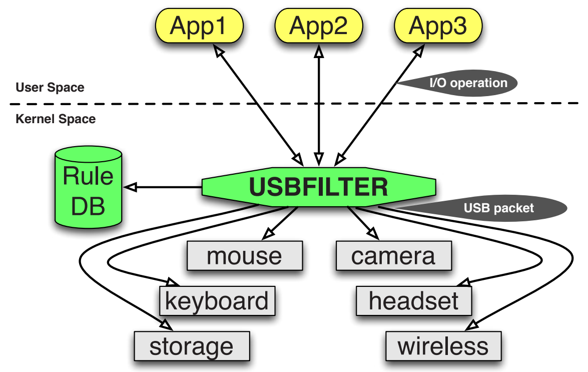

Figure 2: USBFILTER implements a USB-layer reference monitor within the kernel, by filtering USB packets to different USB devices to control the communications between applications and devices based on rules configured.

to enable or access authorized devices from unauthorized processes or devices. In either case, the adversary may attempt to exfiltrate data from the host system via both physical and virtual USB devices.

We consider the following actions by an adversary:

- **Device Tampering**: The adversary may attempt to attach or tamper with a previously-authorized device to add unauthorized functionality (e.g., BadUSB [27]).

- **Unauthorized Devices**: Unauthorized devices attached to the system either physically or virtually [21] can be used to discreetly interact with the host system or to provide data storage for future exfiltration.

- **Unauthorized Access**: The adversary may attempt to enable or access authorized devices on a host (e.g., webcam, microphone, etc.) via unauthorized software to gain access to information or functionality that would otherwise inaccessible.

We assume that as a kernel component, the integrity of USBFILTER depends on the integrity of the operating system and the host hardware (except USB devices). Code running in the kernel space has unrestricted access to the kernel's memory, including our code, and we assume that the code running in the kernel will not tamper with USBFILTER. We discuss how we ensure runtime and platform integrity in our experimental setup in Section 3.4.

3.2 Design Goals

Inspired by the Netfilter [40] framework in the Linux kernel, we designed USBFILTER to enable administrator-defined rule-based filtering for the USB protocol. To achieve this, we first designed our system to satisfy the concept of a reference monitor [2], shown in Figure 2. While these goals are not required for full functionality of USBFILTER, we chose to design for stronger security guarantees to ensure that processes attempting to access hardware USB devices directly would be unable to circumvent our system. We define the specific goals as follows:

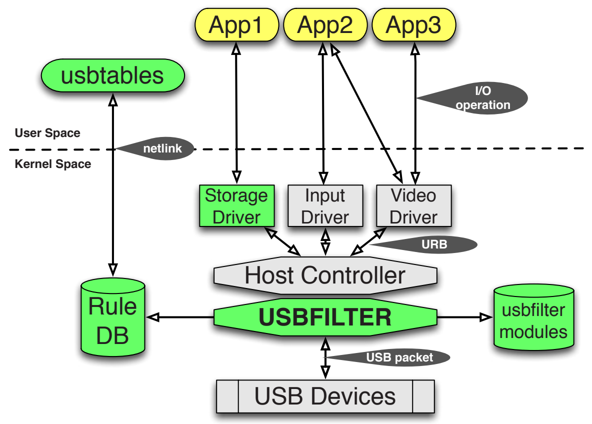

Figure 3: The architecture of USBFILTER.

G1 Complete Mediation. All physical or virtual USB packets must pass through USBFILTER before delivery to the intended destination.

G2 Tamperproof. USBFILTER may not be bypassed or disabled as long as the integrity of the operating system is maintained.

G3 Verifiable. The user-defined rules input into the system must be verifiably correct. These rules may not conflict with each other.

While the above goals support the security guarantees that we want USBFILTER to provide, we expand upon these to provide additional functionality:

G4 Granular. Any mutable data in a USB packet header must be accessible by a user-defined rule. If the ultimate destination of a packet is a userspace process, USBFILTER must permit the user to specify the process in a rule.

G5 Modular. USBFILTER must be extensible and allow users to provide submodules to support additional types of analysis.

3.3 Design and Implementation

The core USBFILTER component is statically compiled and linked into the Linux kernel image, which hooks the flow of USB packets before they reach the USB host controller which serves the USB device drivers, as shown in Figure 3. Like Netfilter, this USB firewall checks a user-defined rule database for each USB packet that passes through it and takes the action defined in the first matching rule. A user-space program, USBTABLES, provides mediated read/write access to the rule database. Since USBFILTER intercepts USB packets in the kernel, it can control access to both physical and virtual devices.

3.3.1 Packet Filtering Rules

To access external USB devices, user-space applications request I/O operations which are transformed into USB request blocks (URBs) by the operating system. The communication path involves the process, the device, and the I/O request itself (*USB packet*). Similarly, a USBFILTER rule can be described using the process information, the device information, and the USB packet information.

A USBFILTER rule R can be expressed as a triple $(N, \mathscr{C}, A)$ where N is the name of the rule, $\mathscr{C}$ is a set of conditions, and $A \in \{ALLOW, DROP\}$ is the action that is taken when all of the conditions are satisfied. As long as the values in conditions, action, and name are valid, this rule is valid, but may not be correct considering other existing rules. We discuss verifying the *correctness* of rules in Section 4.

3.3.2 Traceback

USB packets do not carry attribution data that can be used to determine the source or destination process of a packet. We therefore need to perform traceback to attribute packets to interfaces and processes.

Interfaces. As discussed in Section 2, a USB device can have multiple interfaces, each with a discrete functionality served by a device driver in the operating system. Once a driver is bound with an interface, it is able to communicate with that interface using USB packets.

Determining the driver responsible for receiving or sending a given USB packet is useful for precisely controlling device behaviors. However, identifying the responsible driver is not possible at the packet level, since the packets are already in transit and do not contain identifying information. While we could infer the responsible driver for simple USB devices, such as a mouse, this becomes unclear with composite USB devices with multiple interfaces (some of which may be served by the same driver).

To recover this important information from USB packets without changing each driver and extending the packet structure, we save the interface index into the kernel endpoint structure during USB enumeration. This reverse mapping of interface to driver needs to be performed only once per device. The interface index distinguishes interfaces belonging to the same physical device and USB packets submitted by different driver instances. Once the mapping has been completed, the USB host controller is able to easily trace the originating interface back to the USB packets.

Processes. Similarly, tracking the destination or source process responsible for a USB packet is not trivial due to the way modern operating systems abstract device access from applications. For example, when communicating with USB storage devices, the operating system provides several abstractions between the application and the raw device, including a filesystem, block layer, and I/O scheduler. Furthermore, applications generally submit asynchronous I/O requests, causing the kernel to perform the communications task on a separate background thread.

This problem also appears when inspecting USB network device packets, including both wireline (e.g., Ethernet) dongles and wireless (e.g., WiFi) adapters. It is common for these USB device drivers to have their own RX/TX queues to boost the system performance using asynchronous I/O. In these cases, USB is an intermediate layer to encapsulate IP packets into USB packets for processing by the USB networking hardware.

These cases are problematic for USBFILTER because a naïve traceback approach will often only identify the kernel thread as the origin of a USB packet. To recover the process identifier (PID) of the true origin, we must ensure that this information persists between all layers within the operating system before the I/O request is transformed into a USB packet.[1]

USBFILTER instruments the USB networking driver (*usbnet*), the USB wireless driver (*rt2x00usb*), the USB storage driver (*usb-storage*), as well as the block layer and I/O schedulers. Changes to the I/O schedulers are needed to avoid the potential merging of two block requests from different processes. By querying the rule database and USBFILTER modules, USBFILTER sets up a filter for all USB packets right before being dispatched to the devices.

3.3.3 Userspace Control

USBTABLES manages USBFILTER rules added in the kernel and saves all active rules in a database. Using `udev`, saved rules are flushed into the kernel automatically upon reboot. USBTABLES is also responsible for verifying the correctness of rules as we will discuss in Section 4. Once

[1] USBFILTER does not overlap with Netfilter or any other IP packet filtering mechanisms which work along the TCP/IP stack.

verified, new rules will be synchronized with the kernel and saved locally.

If no user-defined rules are present, USBFILTER enforces default rules that are designed to prevent impact on normal kernel activities (e.g., USB hot-plugs). These rules can be overridden or augmented by the user as desired.

3.4 Deployment

We now demonstrate how we use existing security techniques in the deployment of USBFILTER. Attestation and MAC policy are necessary for providing complete mediation and tamperproof reference monitor guarantees, but not for the functionality of the system. The technologies we reference in this section are illustrative examples of how these goals can be met.

3.4.1 Platform Integrity

We deployed USBFILTER on a physical machine with a Trusted Platform Module (TPM). The TPM provides a root of trust that allows for a measured boot of the system and provides the basis for remote attestations to prove that the host machine is in a known hardware and software configuration. The BIOS's core root of trust for measurement (CRTM) bootstraps a series of code measurements prior to the execution of each platform component. Once booted, the kernel then measures the code for user-space components (e.g., provenance recorder) before launching them using the Linux Integrity Measurement Architecture (IMA)[31]. The result is then extended into TPM PCRs, which forms a verifiable chain of trust that shows the integrity of the system via a digital signature over the measurements. A remote verifier can use this chain to determine the current state of the system using TPM attestation. Together with TPM, we also use Intel's Trusted Boot (tboot)[2]

3.4.2 Runtime Integrity

After booting into the USBFILTER kernel, the runtime integrity of the TCB (defined in Section 3.1) must also be assured. To protect the runtime integrity of the kernel, we deploy a Mandatory Access Control (MAC) policy, as implemented by Linux Security Modules. We enable SELinux's MLS policy, the security of which was formally modeled by Hicks et al. [20]. We also ensure that USBTABLES executes in a restricted environment and that the access to the rules database saved on the disk is protected by defining an SELinux Policy Module and compiling it into the SELinux Policy.

[2] See http://sf.net/projects/tboot

4 Security

In this section, we demonstrate that USBFILTER meets the security goals outlined in Section 3 using the deployment and configurations described in that section.

Complete Mediation (G1). As we previously discussed, USBFILTER must mediate all USB packets between devices and applications on the host. In order to ensure this, we have instrumented USBFILTER into the USB host controller, which is the last hop for USB packets before leaving the host machine and the first when entering it. Devices cannot initiate USB packet transmission without permission from the controller.

We also instrument the virtual USB host controller (*vhci*) to cover virtual USB devices (e.g., USB/IP). To support other non-traditional USB host controllers such as Wireless USB [19] and Media Agnostic USB [16], USBFILTER support is easily added via a simple kernel API call and the inclusion of a header file.

Tamperproof (G2). USBFILTER is statically compiled and linked into the kernel image to avoid being unloaded as a kernel module. The integrity of this runtime, the associated database, and user-space tools is assured through the SELinux policy as described in Section 3.4.2. Tampering with the kernel or booting a different kernel is the only way to bypass USBFILTER, and platform integrity measures provide detection capabilities for this scenario (Section 3.4.1).

Formal Verification (G3). The formal verification of USBFILTER rules is implemented as a logic engine within USBTABLES using GNU Prolog [11]. Instead of trying to prove that an abstract model of rule semantics is correctly implemented by the code, which is usually intractable for the Linux kernel, we limit our focus on rule correctness and consistency checking. Each time USBTABLES is invoked to add a new rule, the new rule and the existing rules are loaded into the logic engine for formal verification. This process only needs to be performed once when adding a new rule and USBFILTER continues to run while the verification takes place.

The verification checks for rules with the same conditions but different actions. These rules are considered conflicting and USBTABLES will terminate with error when this occurs. We define the correctness of a rule:

$$\begin{aligned}
&is\_correct(R,\mathbb{R}) \leftarrow \\
&\qquad is\_name\_unique(R)\wedge \\
&\qquad are\_condition\_values\_in\_range(R)\wedge \\
&\qquad has\_no\_conflict\_with\_existing\_rules(R,\mathbb{R}).
\end{aligned}$$

where R is a new USBFILTER rule and $\mathbb{R}$ for all other existing rules maintained by USBFILTER. If the new rule has a unique name, all the values of conditions are in range, and it does not conflict with any existing rules, the rule is correct.

While the name and the value checks are straightforward, there are different conflicting cases between the conditions and the action, particularly when a rule does not contain all conditions. For example, a rule can be contradictory with, a sub rule of, or the same as another existing rule. As such, we define the general conflict between two rules as follows:

$$\begin{aligned} & general\_conflict(R_a, R_b) \leftarrow \\ & \quad \forall C_i \ni \mathscr{C} : \\ & \qquad (\exists C_i^a \ni R_a \land \exists C_i^b \ni R_b \land value(C_i^a) \neq value(C_i^b)) \lor \\ & \qquad (\exists C_i^a \ni R_a \land \nexists C_i^b \ni R_b) \lor \\ & \qquad (\nexists C_i^a \ni R_a \land \nexists C_i^b \ni R_b). \end{aligned}$$

A rule R_a is generally conflicted with another rule R_b if all conditions used by R_a are a subset of the ones specified in R_b. We consider a *general conflict* to occur if the new rule and an existing rule would fire on the same packet.

Based on the general conflict, we define *weak conflict* and *strong conflict* as follows:

$$\begin{aligned} & weak\_conflict(R_a, R_b) \leftarrow \\ & \quad general\_conflict(R_a, R_b) \land action(R_a) = action(R_b). \\ & strong\_conflict(R_a, R_b) \leftarrow \\ & \quad general\_conflict(R_a, R_b) \land action(R_a) \neq action(R_b). \end{aligned}$$

While weak conflict shows that the new rule could be a duplicate of an existing rule, strong conflict presents that this new rule would not work. The weak conflict, however, depending on the requirement and the implementation, may be allowed temporarily to shrink the scope of an existing rule while avoiding the time gap between the old rule removed and the new rule added. For instance, rule A drops any USB packets writing data into any external USB storage devices. Later on, the user decides to block write operations only for the Kingston thumb drive by writing rule B, which is weak conflicted with rule A, since both rules have the same destination and action. When the user wants to unblock the Kingston storage by writing rule C, rule C is strong conflicted with both rule A and B, since rule C has a different action, and will never work as expected because of rule A/B. By relying on the logic reasoning of Prolog, we are able to guarantee that a rule before added is formally verified no conflict with existing rules [3].

[3]Note that all rules are monotonic by design, which means rules to be added cannot override existing ones. Future work will add general rules, which can be overwritten by new rules.

```
-d|--debug    enable debug mode
-c|--config   path to configuration file (TBD)
-h|--help     display this help message
-p|--dump     dump all the rules
-a|--add      add a new rule
-r|--remove   remove an existing rule
-s|--sync     synchronize rules with kernel
-e|--enable   enable usbfilter
-q|--disable  disable usbfilter
-b|--behave   change the default behavior
-o|--proc     process table rule
-v|--dev      device table rule
-k|--pkt      packet table rule
-l|--lum      LUM table rule
-t|--act      table rule action
--------------------------------
proc: pid,ppid,pgid,uid,euid,gid,egid,comm
dev: busnum,devnum,portnum,ifnum,devpath,product,
     manufacturer,serial
pkt: types,direction,endpoint,address
lum: name
behavior/action: allow|drop
```

Figure 4: The output of "`usbtables -h`". The permitted conditions are divided into 4 tables: the process table, the device table, the packet table, and the Linux USBFILTER Module (LUM) table.

Granular (G4). A USBFILTER rule can contain 21 different conditions, excluding the name and action field. We further divide these conditions into 4 tables, including the process, device, packet, and the Linux USBFILTER Module (LUM) table, as shown in Figure 4. The process table lists conditions specific to target applications; the device table contains details of USB devices in the system; the packet table includes important information about USB packets; and the LUM table determines the name of the LUM to be used if needed. Note that all LUMs should be loaded into the kernel before being used in USBFILTER rules.

Module Extension (G5). To support customized rule construction and deep USB packet analysis, USBFILTER allows system administrators to write Linux USBFILTER Modules (LUMs), and load them into the kernel as needed. To write a LUM, developers need only include the <*linux/usbfilter.h*> header file in the kernel module, implement the callback *lum_filter_urb()*, and register the module using *usbfilter_register_lum()*. Once registered, the LUM can be referenced by its name in the construction of a rule. When a LUM is encountered in a rule, besides other condition checking, USBFILTER calls the *lum_filter_urb()* callback within this LUM, passing the USB packet as the sole parameter. The callback returns 1 if the packet matches the target of this LUM, 0 otherwise. Note that the current implementation supports only one LUM per rule.

5 Evaluation

The USBFILTER host machine is a Dell Optiplex 7010 with an Intel Quad-core 3.20 GHz CPU with 8 GB memory and is running Ubuntu Linux 14.04 LTS with kernel version 3.13. The machine has two USB 2.0 controllers and one USB 3.0 controller, provided by the Intel 7 Series/C210 Series chipset. To demonstrate the power of USBFILTER, we first examine different USB devices and provide practical use cases which are non-trivial for traditional access control mechanisms. Finally we measure the overhead introduced by USBFILTER.

The default behavior of USBFILTER in our host machine is to allow the USB packet if no rule matches the packet. A more constrained setting is to change the default behavior to drop, requiring each permitted USB device to need an allow rule. In this setting, malicious devices have to impersonate benign devices to allow communications, which are still regulated by the rules, e.g., no HID traffic allowed for a legit USB storage device. All tests use the same front-end USB 2.0 port on the machine.

5.1 Case Studies

Listen-only USB headset. The typical USB headset is a composite device with multiple interfaces including speakers, microphone, and volume control. Sensitive working environments may ban the use of USB headsets due to possible eavesdropping using the microphone [17]. Physically disabling the headset microphone is often the only mechanism for permanently removing it, as there is no other way to guarantee the microphone stays off. Users can mute or unmute the microphone using the desktop audio controls at any time after login. However, with USBFILTER, the system administrator can guarantee that the headset's microphone remains disabled and cannot be enabled or accessed by users.

We use a Logitech H390 Headset to demonstrate how to achieve this guarantee on the USBFILTER host machine:

```
usbtables -a logitech-headset -v ifnum=2,product=
      "Logitech USB Headset",manufacturer=Logitech -k
      direction=1 -t drop
```

This rule drops any incoming packets from the Logitech USB headset's microphone. By adding the interface number (`ifnum=2`), we avoid breaking other functionality in the headset.

Customizing devices. To further show how USBFILTER can filter functionalities provided by USB devices, we use Teensy 3.2 [29] to create a complex USB device with five interfaces including a keyboard, a mouse, a joystick, and two serial ports. The keyboard continually types commands in the terminal, while the mouse continually moves the cursor. We can write USBFILTER rules to completely shutdown the keyboard and mouse functionalities:

```
usbtables -a teensy1 -v ifnum=2,manufacturer=
            Teensyduino,serial=1509380 -t drop
usbtables -a teensy2 -v ifnum=3,manufacturer=
            Teensyduino,serial=1509380 -t drop
```

In these rules, we use condition "manufacturer" and "serial" (serial number) to limit the Teensy's functionality. Different interface numbers represent the keyboard and the mouse respectively. After these rules applied, both the keyboard and the mouse return to normal.

Default-deny input devices. Next, we show how to defend against HID-based BadUSB attacks using USBFILTER. These types of devices are a type of *trojan horse*; they appear to be one device, such as a storage device, but secretly contain hidden input functionality (e.g., keyboard or mouse). When attached to a host, the device can send keystrokes to the host and perform actions as the current user.

First, we create a BadUSB storage device using a Rubber Ducky [18], which looks like a USB thumb drive but opens a terminal and injects keystrokes. Then we add following rules into the host machine:

```
usbtables -a mymouse -v busnum=1,devnum=4,portnum=2,
      devpath=1.2,product="USB Optical Mouse",
      manufacturer=PixArt -k types=1 -t allow
usbtables -a mykeyboard -v busnum=1,devnum=3,
      portnum=1,devpath=1.1,
      product="Dell USB Entry Keyboard",
      manufacturer=DELL -k types=1 -t allow
usbtables -a noducky -k types=1 -t drop
```

The first two rules whitelist the existing keyboard and mouse on the host machine; the last rule drops any USB packets from other HID devices. After these rules are inserted into the kernel, reconnecting the malicious device does nothing. Attackers may try to impersonate the keyboard or mouse on the host machine. However, we have leveraged information about the physical interface (`busnum` and `portnum`) to write the first two rules, which would require the attacker to unplug the existing devices, plug the malicious device in, and impersonate the original devices including the device's VID/PID and serial number. We leave authenticating individual USB devices to future work, however USBFILTER is extensible so that authentication can be added and used in rules.

Data exfiltration. To prevent data exfiltration from the host machine to USB storage devices, we write a LUM (Linux USBFILTER Module) to block the SCSI write command from the host to the device, as shown in Figure 9 in the Appendix. The LUM then registers itself with USBFILTER and can be referenced by its name in rule constructions. In this case study, we use a Kingston DT 101 II 2G USB flash drive, and insert the following rule:

```
usbtables -a nodataexfil -v manufacturer=Kingston
      -l name=block_scsi_write -t drop
```

This rule prevents modification of files on the storage device. Interestingly, `vim` reports files on the device to be read-only, despite the filesystem reporting that the files are read-write. Since USBFILTER is able to trace packets back to the applications initiating I/O operations at the Linux kernel block layer, we are able to write rules blocking (or allowing) specific users or applications from writing to flash drive:

```
usbtables -a nodataexfil2 -o uid=1001
      -v manufacturer=Kingston
      -l name=block_scsi_write -t drop
usbtables -a nodataexfil3 -o comm=vim
      -v manufacturer=Kingston
      -l name=block_scsi_write -t drop
```

The first rule prevents the user with `uid=1001` from writing anything to the USB storage; the second blocks `vim` from writing to the storage. We can also block any writes to USB storage devices:

```
usbtables -a nodataexfil4
      -l name=block_scsi_write -t drop
```

USBFILTER logs dropped USB packets, and these logs can easily be used in a centralized alerting system, notifying administrators to unauthorized access attempts.

Webcam pinning. Webcams can easily be enabled and accessed by attackers from exploiting vulnerable applications. Once access has been established, the attacker can listen or watch the environment around the host computer. In this case study, we show how to use USBFILTER to restrict the use of a Logitech Webcam C310 to specific users and applications.

```
usbtables -a skype -o uid=1001,comm=skype -v
      serial=B4482A20 -t allow
usbtables -a nowebcam -v serial=B4482A20 -t drop
```

The serial number of the Logitech webcam is specified in the rules to differentiate any others that may be attached to the system as well as to prevent other webcams from being attached. The first rule allows USB communication with the webcam only if the user is `uid=1001` and the application is Skype. The following `nowebcam` rule drops other USB packets to the webcam otherwise. As expected, the user can use the webcam from his Skype but not from Pidgin, and other users cannot start video calls even with Skype.

USB charge-only. Another form of BadUSB attacks is DNS spoofing using smartphones. Once plugged into the host machine, the malicious phone automatically enables USB tethering, is recognized as a USB NIC by the host, then injects spoofed DNS replies into the host. The resulting man-in-the-middle attack gives the attacker access to the host's network communications without the authorization of the user. To prevent this attack, we use USBFILTER to prevent all USB packets from a Google Nexus 4 smartphone:

```
usbtables -a n4-charger -v product="Nexus 4" -t drop
```

This rule rule drops any USB packets to/from the phone, which enforces the phone as a pure charging device without any USB functionality. The phone is unable to be used for storage or tethering after the rule is applied.

We can construct a more specific charge-only rule:

```
usbtables -a charger -v busnum=1,portnum=4 -t drop
```

This rule specifies a specific physical port on the host and this port can only be used for charging. This type of rule is useful where USB ports may be exposed (e.g., on a point of sale terminal) and cannot be physically removed. It is also vital to defend against malicious devices whose firmware can be reprogrammed to forge the VID/PID such as BadUSB, since this type of rule only leverages the physical information on the host machine. USBFILTER can partition all physical USB ports and limit the USB traffic on each port.

Prolog Engine	Min	Avg	Med	Max	Dev
Time (20 rules)	128.0	239.8	288.0	329.0	73.2
Time (100 rules)	132.0	251.7	298.0	485.0	78.6

Table 1: Prolog reasoning time (μs) averaged by 100 runs.

5.2 Benchmarks

We first measure the performance of the user-space tool, USBTABLES. We then measure the overhead imposed by USBFILTER.

The measurement host is loaded with the rules mentioned in the case studies above before beginning benchmarking. When coupled with the default rules provided by USBFILTER, there are 20 total rules loaded in the kernel. We chose 20 because we believe that a typical enterprise host's USB devices (e.g., keyboard, mouse, removable storage, webcam, etc.) will total less than 20. Then we load 100 rules in the kernel to understand the scalability of USBFILTER.

5.2.1 Microbenchmark

USBTABLES Performance. We measure the time used by the Prolog engine to formally verify a rule before it is added into the kernel. We loaded the kernel with 20 and

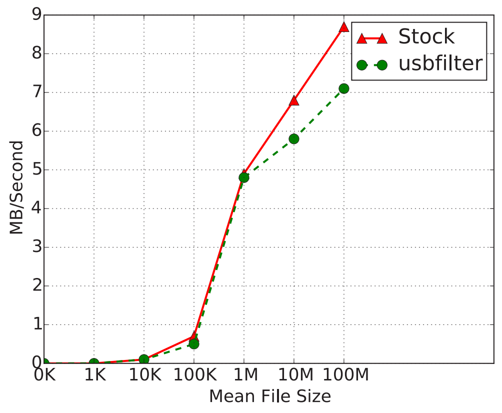

Figure 5: Filebench throughput (MB/s) using `fileserver` workload with different mean file sizes.

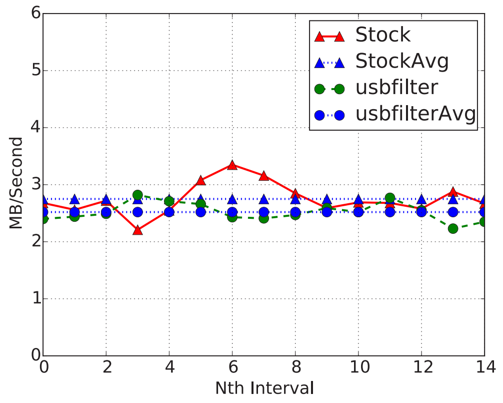

Figure 6: Iperf bandwidth (MB/s) using TCP with different time intervals.

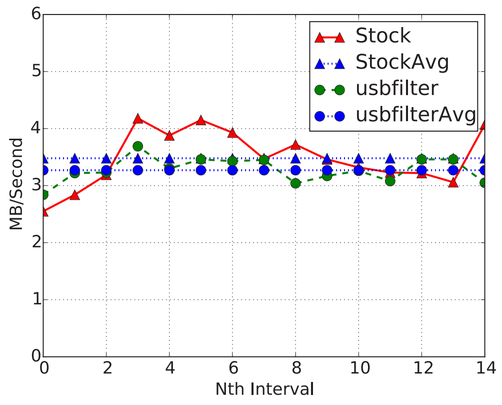

Figure 7: Iperf bandwidth (MB/s) using UDP with different time intervals.

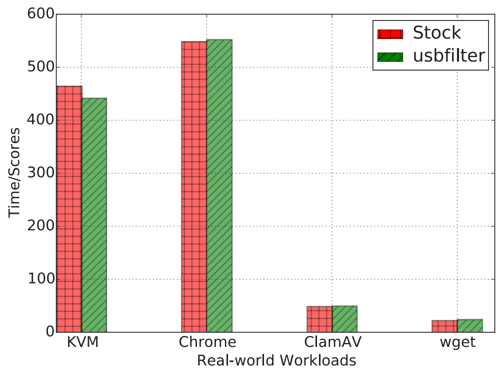

Figure 8: Performance comparison of real-world workloads.

Rule Adding	Min	Avg	Med	Max	Dev
Time (20 rules)	5.1	5.9	6.1	6.6	0.3
Time (100 rules)	4.9	5.9	6.1	6.8	0.4

Table 2: Rule adding operation time (ms) averaged by 100 runs.

USB Enumeration	Min	Avg	Med	Max	Dev	Cost
Stock Kernel	32.0	33.9	34.1	34.8	0.6	N/A
USBFILTER (20 rules)	33.2	34.4	34.3	35.8	0.7	1.5%
USBFILTER (100 rules)	33.9	34.8	34.6	36.0	0.5	2.7%

Table 3: USB enumeration time (ms) averaged by 20 runs.

100 rules and measured the time to process the rules. For each new rule, the Prolog engine needs to go through the existing rules and check for conflicts.

We measured 100 trials of each test. The performance of the Prolog engine is shown in Table 1. The average time used by the Prolog engine is 239.8 μs with 20 rules and 251.7 μs with 100 rules. This fast speed is the result of using GNU Prolog (`gplc`) compiler to compile Prolog into assembly for acceleration. We also measure the overhead for USBTABLES to add a new rule to the kernel space. This includes loading existing rules into the Prolog engine, checking for conflicts, saving the rule locally, passing the rule to the kernel, and waiting for the acknowledgment. As shown in Table 2, the average time of adding a rule using USBTABLES stays at around 6 ms in both cases, which is a negligible one-time cost.

USB Enumeration Overhead. For this test, we used the Logitech H390 USB headset, which has 4 interfaces. We manually plugged the headset into the host 20 times. We then compare the results between the USBFILTER kernel with varying numbers of rules loaded and the stock Ubuntu kernel, where USBFILTER is fully disabled,

Packet Filtering	Min	Avg	Med	Max	Dev
Time (20 rules)	2.0	2.6	3.0	5.0	0.5
Time (100 rules)	2.0	9.7	10.0	15.0	1.0

Table 4: Packet filtering time (μs) averaged by 1500 packets.

Configuration	1K	10K	100K	1M	10M	100M
Stock	97.6	98.1	99.2	105.5	741.7	5177.7
USBFILTER	97.7	98.2	99.6	106.3	851.5	6088.4
Overhead	0.1%	0.1%	0.4%	0.8%	14.8%	17.6%

Table 5: Latency (*ms*) of the `fileserver` workload with different mean file sizes.

as shown in Table 3. The average USB enumeration time is 33.9 ms for the stock kernel and 34.4 ms and 34.8 ms for the USBFILTER kernel with 20 and 100 rules preloaded respectively. Comparing to the stock kernel, USBFILTER only introduces 1.5% and 2.7% overheads, or less than 1 ms even with 100 rules preloaded.

Packing Filtering Overhead. The overhead of USB enumeration introduced by USBFILTER is the result of packet filtering and processing performed on each USB packet, since there may be hundreds of packets during USB enumeration, depending on the number of interface and endpoints of the device. To capture this packet filtering overhead, we plug in a Logitech M105 USB Optical Mouse, and move it around to generate enough USB packets. We then measure the time used by USBFILTER to determine whether the packet should be filtered/dropped or not for 1500 packets, as shown in Table 4. The average cost per packet are 2.6 μs and 9.7 μs respectively, including the time to traverse all the 20/100 rules in the kernel, and the time used by the benchmark itself to get the timing and print the result. The 100-rule case shows that the overhead of USBFILTER is quadruped when the number of rule increases by one order of magnitude. As we mentioned before, most common USB usages could be covered within 20 rules. We assume it is rare for a system to have 100 rules for different USB devices. To search in hundreds of rules efficiently, we can setup a hash table using e.g., USB port numbers as keys to save rules instead of a linear array (list) currently implemented.

5.2.2 Macrobenchmark

We use filebench [37] and iperf [42] to measure throughputs and latencies of file operations, and bandwidths of network activities, under the stock kernel and the USBFILTER kernel, using different USB devices. The USBFILTER kernel is loaded with 20 rules introduced in the case studies before benchmarking.

Filebench. We choose the `fileserver` workload in filebench, with the following settings: the number of files in operation is 20; the number of working threads is 1; the run time for each test case is 2 minutes; the mean file size in operation ranges from 1 KB to 100 MB; all other settings are default provided by filebench. These settings emulate a typical usage of USB storage devices, where users plug in flash drives to copy or edit some files. All file operations happen in a SanDisk Cruzer Fit 16 GB flash drive. The throughputs under the stock kernel and the USBFILTER kernel are demonstrated in Figure 5. When the mean file size is less than 1 MB, the throughput of USBFILTER is close to the one of the stock kernel. Since there is at most 20×1 MB data involved in block I/O operations, both the stock kernel and USBFILTER can handle this data size smoothly. When the mean file size is greater than 1 MB, USBFILTER shows lower throughputs comparing to the stock kernel, as the result of rule matching for each USB packet. Compared to the stock kernel, USBFILTER imposes 14.7% and 18.4% overheads when the mean file sizes are 10 MB and 100 MB respectively. That is, when there is 20×100 MB (2 GB) involved in block I/O operations, the throughput decreases from 8.7 MB/s to 7.1 MB/s, when USBFILTER is enabled.

The corresponding latencies are shown in Table 5. The latency of USBFILTER is higher than the stock kernel. Following the throughput model, the latencies between the two kernels are close when the mean file size is less than 1 MB. The overhead introduced by USBFILTER is less than 1.0%. When the mean file sizes are 10 MB and 100 MB, USBFILTER imposed 14.8% and 17.6% overheads in latency. comparing to the stock kernel. That is, to deal with 20×100 MB data, users need one more second to finish all the operations with USBFILTER enabled, which is acceptable for most users.

iperf. We use iperf to measure bandwidths of upstream TCP and UDP communications, where the host machine acts as a server, providing local network access via a Ralink RT5372 300 Mbps USB wireless adapter. The time interval for each transmission is 10 seconds, and each test runs 5 minutes (30 intervals). For TCP, we use the default TCP window size 64 KB; for UDP, we use the default available UDP bandwidth size 10 MB. The TCP bandwidths of the two kernels are shown in Figure 6, where we aggregate each two intervals into one, reducing the number of sampling points from 30 to 15. and the average bandwidths are also listed in dot lines. Though having different transmission patterns, the average bandwidths of both are close, with the stock kernel at 2.75 Mbps and USBFILTER at 2.52 Mbps. Comparing to the stock kernel, USBFILTER introduces 8.4% overhead.

The UDP benchmarking result closely resembles TCP, as shown in Figure 7. Regardless of transmission patterns, average bandwidth of the two kernels is similar, with the stock kernel at 3.48 Mbps and USBFILTER at 3.27 Mbps. Comparing to the TCP transmission, UDP transmission is faster due to the simpler design/implementation of UDP, and USBFILTER introduces 6.0% overhead. In both cases, USBFILTER has demonstrated a low impact to the original networking component.

5.3 Real-world Workloads

To better understand the performance impact of USBFILTER, we generate a series of real-world workloads to measure typical USB use cases. In the KVM [24] workload, we create and install a KVM virtual machine automatically from the Ubuntu 14.04 ISO image file (581 MB) saved on USB storage. In the Chrome workload, we access the web browser benchmark site [5] via a USB wireless adapter. In the ClamAV [25] workload, we scan the unzipped Ubuntu 14.04 ISO image saved on the USB storage for virus using ClamAV. In the wget workload, we download the Linux kernel 4.4 (83 MB) via the USB wireless adapter using wget. The USB storage is the SanDisk 16 GB flash drive, and the USB wireless adapter is the Ralink 300 Mbps wireless card. All time measurements are in seconds except the Chrome workload, where scores are given, and are divided by 10 to fit into the figure. Figure 8 shows the comparison between the two kernels when running these workloads. In all workloads, USBFILTER either performs slightly better than the stock kernel, or imposes a small overhead compared to the stock kernel in our test. It is clear that USBFILTER approximates the original system performance.

5.4 Summary

In this section, we showed how USBFILTER can help administrators prevent access to unauthorized (and unknown) device interfaces, restrict access to authorized devices using application pinning, and prevent data exfiltration. Our system introduces between 3 and 10 μs of latency on USB packets while checking rules, introducing minimal overhead on the USB stack.

6 Discussion

6.1 Process Table

We have successfully traced each USB packet to its originating application for USB storage devices by passing the PID information along the software stack from the VFS layer, through the block layer, to the USB layer within the kernel. However, it is not always possible to find the PID for each USB packet received by the USB host controller. One example is HID devices, such as keyboards and mouses. Keystrokes and mouse movements happen in the interrupt (IRQ) context, where the current stopped process has nothing to do with this USB packet. All these packets are delivered to the Xorg server in the user space, which then dispatches the inputs to different applications registered for different events. USBFILTER is able to make sure that only Xorg can receive inputs from the keyboard and mouse. To guarantee the USB packet delivered to the desired application, we can enhance the Xorg server to understand USBFILTER rules.

The other example comes from USB networking devices. Though we have enhanced the general USB wireline driver `usbnet` to pass the PID information into each USB packet, unlike USB storage devices sharing the same `usb-storage` driver, many USB Ethernet dongles have their own drivers instead of using the general one. Even worse, there is no general USB wireless driver at all. Depending of the device type and model, one may need to instrument the corresponding driver to have the PID information, like what we did for `rt2800usb` driver. Future work will introduce a new USB networking driver framework to be shared by specific drivers, providing a unified interface for passing PID information into USB packets.

Another issue of using process table in USBFILTER rules is TOCTTOU (time-of-check-to-time-of-use) attacks. A malicious process can submit a USB packet to the kernel and exit. When the packet is finally handled by the host controller, USBFILTER is no longer able to find the corresponding process given the PID. Fortunately, these attacks does not impact rules without process tables. When process information is crucial to the system, we recommending using USBTABLES to change the default behavior to "drop", make sure that no packet would get through without an explicit matching rule.

6.2 System Caching

USBFILTER is able to completely shut down any write operations to external USB storage devices, preventing any form of data exfiltration from the host machine. Similarly, one can also write a "block_scsi_read" LUM to stop read operations from storage devices. Nevertheless, this LUM may not be desired or work as expected in reality. To correctly mount the filesystem in the storage device, the kernel has to read the metadata saved in the storage. One solution would be to delay the read blocking till the filesystem is mounted. However, for performance considerations, the Linux kernel also reads ahead some data in the storage, and brings it into the system cache (page cache). All following I/O operations will happen in the memory rather than the storage. While memory protection is out of scope for this paper, we rely on the integrity of the kernel to enforce the MAC model it applies. Write operations, even though in the memory, will be flushed into the storage, where USBFILTER is able to provide a strong and useful guarantee.

6.3 Packet Analysis From USB Devices

Because of the master-slave nature of the USB protocol, we do not setup USBFILTER in the response path, which is from the device to the host, due to performance considerations. However, enabling USBFILTER in the response path provides new opportunities to defend against malicious devices and users, since the response packet could be inspected with the help of USBFILTER. For example, one can write a LUM to limit the capability of a HID device, such as allowing only three different key actions from a headset's volume control button, which is implemented by GoodUSB as a customized keyboard driver, or disabling sudo commands for unknown keyboards. Another useful case is to filter the spoofing DNS reply message embedded in the USB packet sent by malicious smart phones or network adapters, to defend against DNS cache poisoning. We are planning to investigate these new case studies in future work.

6.4 Malicious USB Drivers and USB Covert Channels

While BadUSB is the most prominent attack that exploits the USB protocol, we observe that using USB communication as a side channel to steal data from host machines, or to inject malicious code into hosts, is another technically mature and plausible threat. On the Linux platform, with the development of libusb [14], more USB drivers run within user space and can be delivered as binaries. On Windows platform, PE has been a common format of device drivers. To use these devices, users have to run these binary files without knowing if these drivers are doing something else in the meantime.[4] For instance, USB storage devices should use bulk packets to transfer data per the USB spec. However, a malicious storage driver may use control packets to stealthily exfiltrate data as long as the malicious storage is able to decode the packet. This works because control transfers are mainly used during the USB enumeration process. With the help of USBFILTER, one can examine each USB packet, and filter unrecognized ones without breaking the normal functionality of the device.

[4]N.B. that there are ways to instrument DLL files on Windows platform, though this does not appear to be commonly done with drivers.

6.5 Usability Issues

To write USBFILTER rules, one needs some knowledge about the USB protocol in general, as well as the target USB device. The `lsusb` command under Linux provides a lot of useful information that can directly be mapped into rule construction. Another tool `usb-devices` also helps users understand USB devices. Windows has a GUI program `USBView` to visualize the hierarchy and configuration of USB devices plugged into the host machine. While users can write some simple rules, we expect that developers will provide useful LUMs, which may require deep understanding of the USB protocol and domain specific knowledge (e.g., SCSI, and will share these LUMs with the community. We wll also provide more useful LUMs in the future.

7 Related Work

Modern operating systems implicitly approve all interfaces on any device that has been physically attached to the host. Due to this, a wide range of attacks have been built on USB including malware and data exfiltration on removable storage [15, 34, 46], tampered device firmware [27, 7], and unauthorized devices [1]. These attacks fall into two major categories: those that involve data ingress and egress via removable storage and those that involve the attachment of unknown USB interfaces.

Proposals for applying access control to USB storage devices [12, 28, 38, 48] fall short because they cannot guarantee that the USB write requests are blocked from reaching the device. Likewise, defenses against unauthorized or malicious device interfaces [41, 33] and disabling device drivers are coarse and cannot distinguish between desired and undesired usage of a particular interface. Another solution employed by the Windows Embedded platform [26] binds USB port numbers with the VID/PID/CID (device class ID) information of devices to accept/reject the device plugged in. While CID helps limit the usage of the device, this solution does not work for composite devices equipped with multiple interfaces (with different CIDs). Besides, users may have to update the policy each time when different devices are plugged into the same port. Given the increasing ubiquity of USB, this is not a sustainable solution. Guardat demonstrates a means of expressing a robust set of rules for storage access but requires substantial new mechanisms for operation within a host computer, such as implementation within a hybrid disk microcontroller [45].

Netfilter [40] has become the de facto network firewall standard on Linux due to its ability to perform fine-grained filtering on network packets between applications and the physical network interface. Netfilter can prevent compromise of a program by preventing unwanted packets from reaching the process. Similarly, our system can defend processes by denying USB traffic before it reaches its destination.

Furthermore, fine-grained filtering has been applied to the usage of filesystem objects by applications [13, 35], however, these filters take place *after* the host and operating system have enumerated the device and loaded any device drivers. USBFILTER applies filtering at the USB packet layer, preventing unauthorized access to interfaces regardless of whether they have been approved elsewhere. Since our system operates between the device drivers and the USB host controller and traces packets back to their source or destination application, USBFILTER can uniquely filter access to any USB interface.

While USBFILTER working in the host operating system directly, other USB security solutions make use of virtualization. GoodUSB [41] leverages a QEMU-KVM as a honeypot to analyze malicious USB devices, while Cinch [3] separates the trusted USB host controller and untrusted USB devices into two QEMU-KVMs, between which a gateway is used to apply policies on USB packets. By mitigating the need for additional components for standard operation, be believe that USBFILTER is better suited for adoption within operating system kernels.

USBFILTER protects the host machine from malicious USB devices, but there are solutions as well for exploring the protection of devices from malicious hosts. USB fingerprinting [6] establishes the host machine identity using USB devices, while Kells [8] protects the USB storage device by attesting the host machine integrity.

Wang and Stavrou [47] suggest that a "USB firewall" might protect against exploitation attacks but do not discuss the complexities of how such a mechanism could be designed or implemented.

8 Conclusion

USB attacks rely on hosts automatically authorizing any physically-attached device. Attackers can discreetly connect unknown and unauthorized interfaces, causing device drivers to be automatically loaded and allowing malicious devices access to the host. In this paper, we prevent unauthorized devices from accessing a host with USBFILTER, the first packet-level access control system for USB. Through tracing each packet back to its associated process, our system can successfully block unauthorized interfaces and restrict access to devices by process. With a default deny policy for new devices, administrators can restrict connection of unknown devices using granular identifiers such as serial number. Our experiments test USBFILTER using a range of I/O benchmarks and find that it introduces minimal overhead. The result is a host that is unresponsive to attacks that may try to discreetly introduce unknown functionality via USB while maintaining high performance.

Acknowledgements

This work is supported in part by the US National Science Foundation under grant numbers CNS-1540217, CNS-1540218 and CNS-1464088.

References

[1] TURNIPSCHOOL - NSA playset. `http://www.nsaplayset.org/turnipschool`.

[2] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, 1972.

[3] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish. Defending against malicious peripherals. *arXiv preprint arXiv:1506.01449*, 2015.

[4] J. Bang, B. Yoo, and S. Lee. Secure usb bypassing tool. *digital investigation*, 7:S114–S120, 2010.

[5] Basemark, Inc. Basemark browsermark. `http://web.basemark.com/`, 2015.

[6] A. Bates, R. Leonard, H. Pruse, K. R. B. Butler, and D. Lowd. Leveraging USB to Establish Host Identity Using Commodity Devices. In *Proceedings of the 2014 Network and Distributed System Security Symposium*, NDSS '14, February 2014.

[7] M. Brocker and S. Checkoway. iseeyou: Disabling the macbook webcam indicator led. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 337–352, 2014.

[8] K. R. B. Butler, S. E. McLaughlin, and P. D. McDaniel. Kells: a protection framework for portable data. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 231–240. ACM, 2010.

[9] A. Caudill and B. Wilson. Phison 2251-03 (2303) Custom Firmware & Existing Firmware Patches (BadUSB). *GitHub*, 26, Sept. 2014.

[10] Compaq, Hewlett-Packard, Intel, Microsoft, NEC, and Phillips. Universal Serial Bus Specification, Revision 2.0, April 2000.

[11] D. Diaz et al. The GNU Prolog web site. `http://gprolog.org/`.

[12] S. A. Diwan, S. Perumal, and A. J. Fatah. Complete security package for USB thumb drive. *Computer Engineering and Intelligent Systems*, 5(8):30–37, 2014.

[13] W. Enck, P. McDaniel, and T. Jaeger. PinUP: Pinning user files to known applications. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 55–64. ieeexplore.ieee.org, Dec. 2008.

[14] J. Erdfelt and D. Drake. Libusb homepage. *Online, http://www.libusb. org*.

[15] N. Falliere, L. O. Murchu, and E. Chien. W32. Stuxnet Dossier. 2011.

[16] U. I. Forum. Media Agnostic Universal Serial Bus Specification, Release 1.0a, July 2015.

[17] D. Genkin, A. Shamir, and E. Tromer. RSA key extraction via Low-Bandwidth acoustic cryptanalysis. In *Advances in Cryptology – CRYPTO 2014*, Lecture Notes in Computer Science, pages 444–461. Springer Berlin Heidelberg, 17 Aug. 2014.

[18] Hak5. Episode 709: USB Rubber Ducky Part 1. `http://hak5.org/episodes/episode-709`, 2013.

[19] Hewlett-Packard, Intel, LSI, Microsoft, NEC, Samsung, and ST-Ericsson. Wireless Universal Serial Bus Specification 1.1, September 2010.

[20] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel. A Logical Specification and Analysis for SELinux MLS Policy. *ACM Trans. Inf. Syst. Secur.*, 13(3):26:1–26:31, July 2010.

[21] T. Hirofuchi, E. Kawai, K. Fujikawa, and H. Sunahara. USB/IP-A peripheral bus extension for device sharing over IP network. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 42–42, 2005.

[22] IronKey, Inc. Access Enterprise. `http://www.ironkey.com/en-US/access-enterprise/`, 2015.

[23] Jeremy Moskowitz. Managing hardware restrictions via group policy. `https://technet.microsoft.com/en-us/magazine/2007.06.grouppolicy.aspx`, 2007.

[24] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[25] T. Kojm. Clamav, 2004.

[26] Microsoft Windows Embedded 8.1 Industry. Usb filter (industry 8.1). `https://msdn.microsoft.com/en-us/library/dn449350(v=winembedded.82).aspx`, 2014.

[27] K. Nohl and J. Lell. BadUSB–On accessories that turn evil. *Black Hat USA*, 2014.

[28] D. V. Pham, M. N. Halgamuge, A. Syed, and P. Mendis. Optimizing Windows Security Features to Block Malware and Hack Tools on USB Storage Devices. In *Progress in Electromagnetics Research Symposium*, 2010.

[29] PJRC. Teensy 3.1. `https://www.pjrc.com/teensy/teensy31.html`, 2013.

[30] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[31] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. *Proceedings of the 13th USENIX Security Symposium*, 2004.

[32] SANS Institute. Real World ARP Spoofing. `http://pen-testing.sans.org/resources/papers/gcih/real-world-arp-spoofing-105411`, 2003.

[33] S. Schumilo, R. Spenneberg, and H. Schwartke. Don't trust your USB! How to find bugs in USB device drivers. In *Blackhat Europe*, Oct. 2014.

[34] S. Shin and G. Gu. Conficker and Beyond: A Large-scale Empirical Study. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 151–160, New York, NY, USA, 2010. ACM.

[35] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. *NAI Labs Report*, 1:43, 2001.

[36] J. Stewart. Dns cache poisoning–the next generation, 2003.

[37] Sun Microsystems, Inc. and FSL at Stony Brook University. Filebench. `http://filebench.sourceforge.net/wiki/index.php/Main_Page`, 2011.

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

[39] The Information Assurance Mission at NSA. Defense against Malware on Removable Media. `https://www.nsa.gov/ia/_files/factsheets/mitigation_monday_3.pdf`, 2007.

[40] The Netfilter Core Team. The Netfilter Project: Packet Mangling for Linux 2.4. http://www.netfilter.org/, 1999.

[41] D. J. Tian, A. Bates, and K. Butler. Defending against malicious USB firmware with GoodUSB. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 261–270, New York, NY, USA, 2015. ACM.

[42] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The tcp/udp bandwidth measurement tool. *htt p://dast. nlanr. net/Projects*, 2005.

[43] M. Tischer, Z. Durumeric, S. Foster, S. Duan, A. Mori, E. Bursztein, and M. Bailey. 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.

[44] USB Implementers Forum. USB-IF statement regarding USB security. `http://www.usb.org/press/USB-IF_Statement_on_USB_Security_FINAL.pdf`.

[45] A. Vahldiek-Oberwagner, E. Elnikety, A. Mehta, D. Garg, P. Druschel, R. Rodrigues, J. Gehrke, and A. Post. Guardat: Enforcing data policies at the storage layer. In *Proceedings of the Tenth European Conference on Computer Systems*, page 13. ACM, 2015.

[46] J. Walter. "Flame Attacks": Briefing and Indicators of Compromise. *McAfee Labs Report*, May 2012.

[47] Z. Wang and A. Stavrou. Exploiting Smart-phone USB Connectivity for Fun and Profit. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, 2010.

[48] B. Yang, D. Feng, Y. Qin, Y. Zhang, and W. Wang. TMSUI: A Trust Management Scheme of USB Storage Devices for Industrial Control Systems. Cryptology ePrint Archive, Report 2015/022, 2015. `http://eprint.iacr.org/`.

Appendix

```c
/*
 * lbsw - A LUM kernel module
 * used to  block SCSI write command within USB packets
 */
#include <linux/module.h>
#include <linux/usbfilter.h>
#include <scsi/scsi.h>

#define LUM_NAME        "block_scsi_write"
#define LUM_SCSI_CMD_IDX    15

static struct usbfilter_lum lbsw;
static int lum_registered;

/*
 * Define the filter function
 * Return 1 if this is the target packet
 * Otherwise 0
 */
int lbsw_filter_urb(struct urb *urb)
{
    char opcode;

    /* Has to be an OUT packet */
    if (usb_pipein(urb->pipe))
        return 0;

    /* Make sure the packet is large enough */
    if (urb->transfer_buffer_length <= LUM_SCSI_CMD_IDX)
        return 0;

    /* Make sure the packet is not empty */
    if (!urb->transfer_buffer)
        return 0;

    /* Get the SCSI cmd opcode */
    opcode = ((char *)urb->transfer_buffer)[LUM_SCSI_CMD_IDX];

    /* Current only handle WRITE_10 for Kingston */
    switch (opcode) {
    case WRITE_10:
        return 1;
    default:
        break;
    }

    return 0;
}

static int __init lbsw_init(void)
{
    pr_info("lbsw: Entering: %s\n", __func__);
    snprintf(lbsw.name, USBFILTER_LUM_NAME_LEN, "%s", LUM_NAME);
    lbsw.lum_filter_urb = lbsw_filter_urb;

    /* Register this lum */
    if (usbfilter_register_lum(&lbsw))
        pr_err("lbsw: registering lum failed\n");
    else
        lum_registered = 1;

    return 0;
}

static void __exit lbsw_exit(void)
{
    pr_info("exiting lbsw module\n");
    if (lum_registered)
        usbfilter_deregister_lum(&lbsw);
}

module_init(lbsw_init);
module_exit(lbsw_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("lbsw module");
MODULE_AUTHOR("dtrump");
```

Figure 9: An example Linux USBFILTER Module that blocks writes to USB removable storage.