LBM: A Security Framework for Peripherals within the Linux Kernel

Dave (Jing) Tian*, Grant Hernandez*, Joseph I. Choi*, Vanessa Frost*, Peter C. Johnson†, Kevin R. B. Butler*

*University of Florida

{daveti, grant.hernandez, choijoseph007, vfrost, butler}@ufl.edu

†Middlebury College

pete@middlebury.edu

Abstract—**Modern computer peripherals are diverse in their capabilities and functionality, ranging from keyboards and printers to smartphones and external GPUs. In recent years, peripherals increasingly connect over a small number of standardized communication protocols, including USB, Bluetooth, and NFC. The host operating system is responsible for managing these devices; however, malicious peripherals can request additional functionality from the OS resulting in system compromise, or can craft data packets to exploit vulnerabilities within OS software stacks. Defenses against malicious peripherals to date only partially cover the peripheral attack surface and are limited to specific protocols (e.g., USB). In this paper, we propose Linux (e)BPF Modules (LBM), a general security framework that provides a unified API for enforcing protection against malicious peripherals within the Linux kernel. LBM leverages the eBPF packet filtering mechanism for performance and extensibility and we provide a high-level language to facilitate the development of powerful filtering functionality. We demonstrate how LBM can provide host protection against malicious USB, Bluetooth, and NFC devices; we also instantiate and unify existing defenses under the LBM framework. Our evaluation shows that the overhead introduced by LBM is within 1 μs per packet in most cases, application and system overhead is negligible, and LBM outperforms other state-of-the-art solutions. To our knowledge, LBM is the first security framework designed to provide comprehensive protection against malicious peripherals within the Linux kernel.**

I. Introduction

Computer peripherals provide critical features to facilitate system use. The broad adoption of computers can be traced not only to the reduction in cost and size from mainframe to microcomputer, but to the interactivity afforded by devices such as keyboards and mice. Displays, printers, and scanners have become integral parts of the modern office environment. Nowadays, smartphones and tablets can not only act as peripherals to a host computer, but can themselves support peripherals that attach to them.

The scope of functionality that peripherals can contain is almost limitless, but the methods of connecting them to host computers have converged to a few select standards, such as USB [10] for wired connections and Bluetooth [15] for wireless. As a result, most modern operating systems provide support for these standards (and the peripherals that use them) by default, implementing the respective software stacks inside the kernel and running different device drivers to support various classes of peripherals.

However, with this virtually unconstrained functionality comes the threat of malicious devices that can compromise computer systems in myriad ways. The BadUSB attack [62] allows attackers to add functionality allowed by the USB protocol to device firmware with malicious intent. For example, a BadUSB flash drive presents not only expected behavior of a storage device when plugged into a computer, but also registers keyboard functionality to allow it to inject malicious keystrokes with the aim of gaining administrative privilege. Other examples of malicious USB functionality include chargers that can inject malware into iOS devices [51], or take control of Android devices via AT commands [78]. Bluetooth peripherals are also vulnerable: the BlueBorne attack [11] allows remote adversaries to craft Bluetooth packets that will cause a kernel stack overflow and enable privilege escalation, while BleedingBit [12] exploits a stack overflow within the Texas Instruments Bluetooth Low Energy (BLE) stack. We observe that malicious peripherals launch attacks in one of two ways, either by *(1)* sending unexpected packets (I/O requests or responses) to activate extra functionality enabled by the operating system, or by *(2)* crafting specially formed packets (either legitimate or malformed) to exploit vulnerabilities within the operating system's protocol software stack.

Current defenses against malicious peripherals are not comprehensive and are limited in scope. USBFILTER [79] applies user-defined rules to USB packet filtering within the Linux kernel, but fails to prevent exploitation from malformed packets. USBFirewall [43], on the other hand, provides bit-level protection by parsing individual incoming USB packets, but offers limited support for user-defined filtering rules. Apple recently added USB restricted mode in iOS 11.4, shutting down USB data connections after the device stays locked for an hour [84], but this restriction can be bypassed [2]. Not only do these defenses lack comprehensive coverage, but they often focus primarily or solely on USB, providing no protection against peripherals using other interfaces.

In this paper, we propose *Linux (e)BPF Modules* (LBM), a general security framework that provides a unified API for enforcing protection against malicious peripherals within the Linux kernel. LBM requires only a single hook for incoming and outgoing peripheral data to be placed in each peripheral subsystem, and modules for filtering specific peripheral packet types (e.g., USB request blocks or Bluetooth socket buffers)

can then be developed. Importantly for performance and extensibility, we leverage the Extended BSD Packet Filter (eBPF) mechanism [25], which supports loading of filter programs from user space. Unlike previous solutions, LBM is designed to be a general framework suitable for any peripheral protocol. As a result, existing solutions such as USBFILTER and USBFirewall can be easily instantiated using LBM. Moreover, new peripherals can be easily supported by adding extensions into the LBM core framework. To demonstrate the generality and flexibility of LBM, we have fully instantiated USBFILTER and USBFirewall using the LBM framework, developed hooks for the Bluetooth Host Control Interface (HCI) and Logical Link and Adaptation Protocol (L2CAP) layers, and demonstrated a hook mechanism for the Near-Field Communication (NFC) protocol. Our evaluation shows that the general overhead introduced by LBM is within 1 μs per packet across different peripherals in most cases; the application and system benchmarks demonstrate a negligible overhead from LBM; and LBM has a better performance when compared to other state-of-the-art solutions.

We summarize our contributions[1] below:

- *Design and implement LBM as a general security framework to defend against malicious peripherals.* The LBM core is designed as a high-performance packet filtering framework based on eBPF. LBM hooks are provided to extend support for different peripheral subsystems.
- *Develop a high-level filter language to facilitate writing LBM rules.* Users can write LBM rules in a high-level, PCAP-like language to apply different policies to peripheral data packets, to avoid having to write filters in the complex, low-level BPF assembly language. Our user-space LBMTOOL utility translates LBM rules into eBPF instructions and loads them into the LBM core.
- *Develop support for USB, Bluetooth, and NFC in LBM.* We extend LBM to support multiple peripheral protocols by exposing useful protocol fields to the user space and extending LBMTOOL to recognize LBM rules for different peripherals. We demonstrate LBM's extensibility by unifying and fully implementing the USBFILTER and USBFirewall defenses under the LBM framework.
- *Evaluate performance and analyze coverage against peripheral attacks.* By applying the appropriate LBM rules, we are able to defend against all known peripheral attacks. Our micro-benchmark shows that the general overhead introduced by LBM is within 1 μs in most cases, and the macro-benchmark shows that LBM has better performance than other solutions, with negligible impact on application throughput.

The remainder of the paper is structured as follows: Section II provides background on peripheral security and BPF; Section III presents our security model and goals alongside the design of our solution; Section IV details the implementation of our design in both kernel and user spaces; Section V evaluates LBM through case studies and benchmarks; Section VI discusses additional dimensions of our work; Section VII explains limitations of our work; Section VIII summarizes related work; and Section IX concludes.

II. BACKGROUND

A. Peripheral Security

USB. The Universal Serial Bus (USB) has been around since 1996 with the release of the version 1.0 specification [23]. USB emerged to provide a single, ubiquitous means to connect peripherals that would support a variety of applications with different performance requirements. Since its inception, USB has undergone many revisions (1.1, 2.0, 3.0, 3.1, and most recently 3.2 and Type-C). The set of supported peripheral devices expanded with each version, and the current USB version 3.2 [10] supports a data transfer rate of 20 Gbits per second, much improved over the 12 Mbits per second of v1.0.

Numerous attacks have been demonstrated by vulnerable or malicious USB peripherals. BadUSB [62] attacks work by altering the firmware of USB devices so they register as deceptive device types when plugged into a machine. For example, a USB mass storage device could masquerade as a keyboard to gain the ability to inject malicious keystrokes. A malicious USB charger can inject malware into iOS devices [51] or take full control of Android devices via AT commands [78]. MouseJack [61] affects wireless mice and keyboards that communicate with a computer through a USB dongle. An adversary may inject keystrokes by spoofing either a mouse or keyboard, and in some cases may even pair a fake keyboard with a victim's dongle.

More vulnerabilities with the USB protocol stack and device drivers have been identified with the help of tools such as FaceDancer [30] and syzkaller [31]. On one hand, these vulnerabilities are mostly implementation bugs within the software stack. On the other hand, malicious USB devices can exploit these vulnerabilities to compromise the whole system by sending out specially-crafted USB packets. For a comprehensive exploration of the variety of available USB attack vectors, we refer readers to Tian et al.'s study [80].

Bluetooth. Just as USB dominates wired connections for peripherals, Bluetooth [15] is the de facto standard for connecting peripherals wirelessly. Being a short-distance Radio Frequency (RF) technology, Bluetooth usually allows data transmission within 10 meters. After Bluetooth 4.0, Bluetooth Low Energy (BLE) and Bluetooth Mesh were introduced to support lower-power consumption devices (e.g., IoT) and sensor networks.

Bluetooth, like USB, is also susceptible to a wide variety of attacks [81] due to software implementation vulnerabilities and malicious Bluetooth peripherals. BlueBug [52] allows an attacker to send AT commands to take control of the victim's phone, from e.g., a malicious Bluetooth headset. Blueprinting [38] and BlueBag [21] identify and collect statistics on all discoverable devices in the area. BlueSnarf and BlueSnarf++ [52] allow an adversary to acquire files from a victim device without being authenticated. BlueDump [53] causes a victim device to dump its stored link keys associated with

connection events. CarWhisperer [37] allows an adversary to eavesdrop on and inject audio into a car over Bluetooth. Blue-Borne [11] attacks craft specially-formed Bluetooth packets to exploit certain vulnerabilities within the software stack implementation, causing e.g., privilege escalation. BleedingBit [12] attacks exploit another stack overflow within TI's BLE stack.

While pairing is used to prevent unidentified devices from being connected via Bluetooth, many attacks happen before the pairing procedure. Also, pairing does not work for simple devices without a means to input PINs. Unlike the case for USB, there is no available systematic solution that defends against malicious Bluetooth peripherals at all. The most effective defense seems to be turning off Bluetooth or physically unplugging the Bluetooth module.

NFC. Near Field Communication (NFC) [60] is another short-range wireless communication protocol based on RFID technology. The operation range is usually within 4 to 5 centimeters. Smartphones (e.g., Androids and iPhones) commonly use NFC as a quick means to exchange information, such as when downloading a poster or making a payment.

Similarly, these NFC software stacks are also vulnerable. A NFC feature that unknowingly invokes a Bluetooth connection can install malware on phones [83]. "Exploring the NFC Attacking Surface" [56] lists four possible attacks enabled by bugs within the Android and N9 software stacks. A recent bug within the Linux kernel NFC software stack [13] allows a malicious NFC device to inject a malformed packet to launch out-of-bounds writes in kernel memory.

In Summary. Regardless of wireline or wireless, these peripheral communication protocols often refer to their communication unit as a "packet" (e.g., USB packets or Bluetooth packets). The OS further instantiates the abstraction of these "packets" within the context of a given I/O subsystem. This provides us an opportunity to treat these peripheral security issues as we would treat networking security issues: by building firewalls for these peripherals and applying rules to filter unwanted (malicious) packets.

B. BPF/eBPF

The BSD Packet Filter (BPF) [54] is a high-performance RISC-based virtual machine running inside the OS. Since its creation, it has been used as a standard way for packet filtering in the kernel space. The most well-known BPF customer might be `tcpdump`, which compiles filtering rules into BPF instructions and loads them into the kernel via socket APIs. Extended BPF (eBPF) [25], [45] is a new ISA based on the classic BPF. Compared to the old ISA, eBPF increases the number of registers from 2 to 10 and register width from 32-bit to 64-bit. eBPF also introduces a JIT compiler to map eBPF instructions to native CPU instructions, including x86, x86-64, ARM, PowerPC, Sparc, etc. A new syscall `bpf`, added since Linux kernel 3.18, supports loading eBPF programs from the user space.

Besides the ISA extensions, eBPF provides new ways to communicate between user and kernel spaces, and to call kernel APIs within BPF programs [67]. eBPF maps are a generic data structure to share data between the user/kernel spaces. A typical usage is to have the kernel update certain values (e.g., the number of IP packets received) inside the map with the user space program picking up the change. BPF helpers are a special call to bridge the eBPF programs and kernel APIs. The newly added CALL instruction can be used to trigger predefined BPF helpers, which usually wrap up kernel APIs to implement some functionalities that cannot be achieved by eBPF instructions themselves. eBPF also includes a verifier, which checks the safety of a given eBPF program via a directed acyclic graph (DAG) check (to ensure bounded execution) and by checking for memory violations. The purpose of this verifier is to make sure that an eBPF program cannot affect the kernel's integrity.

III. DESIGN

We first describe the security model we consider, outline the goals we set for our solution, and finally show how we achieve these goals through different aspects of the design.

A. Security Model

We consider attacks from peripherals to require physical access to the host machine (e.g., plugging into the USB port) or use wireless channels to connect with the host (e.g., over Bluetooth). These malicious peripherals usually try to achieve privilege escalation by claiming unexpected functionalities (e.g., BadUSB [62]), or exploiting the kernel protocol stack via specially crafted packets (e.g., BlueBorne [11]). Note that we do not consider DMA-based attacks [74], where IOMMU [41] is needed to stop arbitrary memory writes from the peripheral.

Our Trusted Computing Base (TCB) is made up of the Linux kernel and the software stack down below. We assume trusted boot or measured boot, such as Intel TXT [32], is deployed to protect system integrity. We also assume Mandatary Access Control (MAC), such as SELinux [69], is being enforced across the whole system.

B. Goals: Beyond A Reference Monitor

The first three goals we set (**G1** through **G3**) are drawn from the classic reference monitor concept [7], and are needed to build a secure kernel. The remaining goals (**G4** through **G7**) draw inspiration from existing security frameworks, such as Linux Security Modules (LSM) [86], and consider practical issues surrounding usage and deployment.

G1 Complete Mediation – For each kind of supported peripheral, we need to guarantee that all inputs from the device and all outputs from the host are mediated.

G2 Tamper-proofness – Assuming the system TCB is not compromised, we need to defend against any attacks originating from outside the TCB.

G3 Verifiability – While a whole-system formal verification may be infeasible, we should mandate formal guarantees for security-sensitive components.
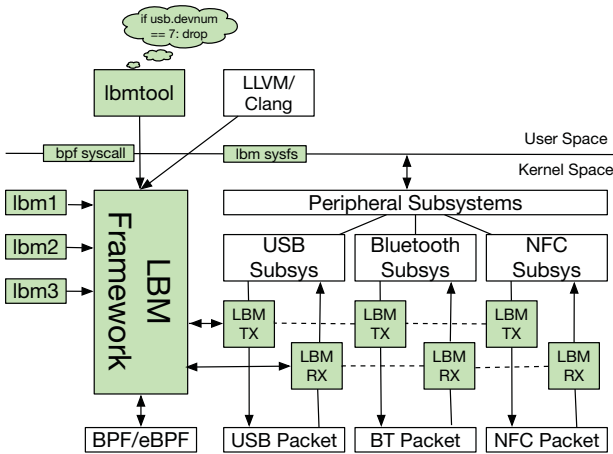
Figure 1: LBM Architecture.

Figure 2: LBM hooks inside the USB subsystem.

G4 Generality – The solution should provide a general framework that seamlessly incorporates the features of existing security solutions.

G5 Flexibility/Extensibility – The addition of support for new kinds of peripherals should be a straightforward and non-intrusive process.

G6 Usability – The solution should be easy to use.

G7 High Performance – The solution should introduce minimal overhead.

Bearing these goals in mind, we design the *Linux (e)BPF Module (LBM)*, as shown in Figure 1.

Within the kernel space, LBM interposes different peripheral subsystems (such as USB, Bluetooth, and NFC) at the bottom level, covering both TX and RX paths. Before a packet can be sent out or reach the corresponding protocol stack for parsing, LBM applies filtering rules (eBPF programs) and loaded LBM kernel modules to the packet for filtering. In the user space, we introduce a new filter language for peripherals. Filters written in this language are compiled into eBPF programs and loaded into the kernel by LBMTOOL.

In short, LBM provides a general peripheral firewall framework, running eBPF instructions as the packet filtering mechanism. We instantiate LBM on USB, Bluetooth, and NFC to cover the most common peripherals.

C. LBM Kernel Infrastructure

We design LBM as a standalone kernel component/subsystem statically linked into the kernel image. We rely on TPM and IMA [65] to guarantee the boot time integrity of the kernel and load time integrity of user-space dependencies. We further use MAC such as SELinux [69] to make sure LBM cannot be disabled without root permission. Since LBM cannot be unloaded/reloaded as a kernel module, disabled, or bypassed from the user space, we achieve **G2** – tamper-proofness.

For each kind of peripheral that LBM supports, we need to place "hooks" on both the TX and RX paths to mediate each packet being sent to and received from the peripheral. While
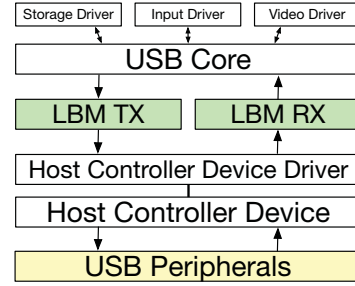
different peripheral subsystems may have different structuring of their software stack architectures within the kernel, we follow two general rules for the placement of LBM hooks. First, these hooks should be placed as close as possible to the real hardware controlling the corresponding peripherals. This helps reduce the potential impact from vulnerabilities within the upper layer of the software stack (e.g., by packets bypassing the hooks). Second, these hooks should be general enough without relying on the implementation of certain hardware. As a result, we place LBM hooks beneath the core implementation of a peripheral's protocol stack, and above a specific peripheral controller driver.

Take USB as an example. As shown in Figure 2, LBM hooks are deployed just above the host controller device and its driver, which communicates with USB peripherals directly. At the same time, the hooks are deployed below the USB core and other USB device drivers, preventing third-party USB drivers from bypassing these hooks. Through this careful placement of LBM hooks, we achieve **G1** – complete mediation.

Since LBM allows the loading of eBPF programs into the kernel space and executing of these programs for peripheral packet filtering, special care is needed to make sure these programs are not introducing new vulnerabilities into the kernel or bypassing security mechanisms enforced by the kernel. We leverage the eBPF verifier [70] to examine each eBPF program before it can be loaded. Unlike normal eBPF programs (mainly used by the networking subsystem) loaded by the bpf syscall, we forbid both bounded loop [26] and packet rewriting (e.g., changing the port number of a TCP packet) in LBM. Once a program passes verification, we can be sure that the program halts after a limited number of state transitions, that each program state is valid (e.g., no stack overflow occurs), and that no instruction changes the kernel memory (besides the program's own stack). We achieve **G3** – verifiability for programs executed by LBM.

LBM draws inspiration from state-of-the-art solutions including USBFILTER [79] and USBFirewall [43], and improves on them, as shown in Table I. Similarly to USB-FILTER, LBM supports kernel module plugin. As depicted in Figure 1, different LBM kernel modules (e.g., lbm1-lbm3) can be plugged into the LBM framework and essentially hook into the TX and/or RX paths for different peripherals. As we will later show in Section V-A, it takes less than 20 lines of

Table I: LBM compared to USBFILTER and USBFirewall. LBM unifies USBFILTER and USBFirewall, providing a superset their properties via extensible protocol support.

Feature	USBFILTER	USBFirewall	LBM
Plugin Modules	✓		✓
Stack Protection		✓	✓
User-defined Rules	✓		✓
TX Path Mediation	✓		✓
RX Path Mediation		✓	✓
Multiple Protocols			✓

Table I: LBM compared to USBFILTER and USBFirewall. LBM unifies USBFILTER and USBFirewall, providing a superset their properties via extensible protocol support.

Feature	USBFILTER	USBFirewall	LBM
Filter Mechanism	C	C	eBPF
User-space DSL	CNF	N/A	PCAP DSL
Acceleration	Short Circuit	N/A	JIT

Table II: LBM vs. USBFILTER vs. USBFirewall, specifically with respect to filter design of each.

change to convert a LUM (Linux USBFILTER Module) into an LBM module. To protect protocol stacks from malformed packets, we derive packet field constraints from specifications. Rather than translating these constraints into C and compiling them into the kernel image like USBFirewall does, we transform them into eBPF programs and load them on the RX paths for malformed packet filtering. In short, we achieve **G4** – generality, by incorporating all the features provided by existing solutions. Additionally, we extend support beyond USB to other peripherals, such as Bluetooth and NFC.

To ease support for a new kind of peripheral, we design a unified API used by different subsystems to hook into LBM:

```
int lbm_filter_pkt(
        int subsys, int dir, void *pkt)
```

subsys determines the index of a certain peripheral subsystem (e.g., 0 for USB and 1 for Bluetooth); dir specifies the direction of the I/O path: TX or RX; and pkt points to the core kernel data structure used to encapsulate the I/O packet depending on different subsystems, (e.g., urb for USB and skb for Bluetooth). Once this LBM hook is placed into a peripheral subsystem, developers can write an LBM module to filter packets using typical C programming, by implementing the TX and/or RX callbacks:

```
int (*lbm_ingress_hook)(void *pkt)
int (*lbm_egress_hook)(void *pkt)
```

A more useful extension is to expose some packet fields to the user space, and implement BPF helpers as backends to provide data access to these fields if needed (as we have done for USB and Bluetooth). As a result, LBMTOOL can generate a new dialect for the new peripheral based on a PCAP-like packet filtering language. Users can then write filtering rules as they would for tcpdump instead of directly crafting eBPF instructions. Through the design of the LBM framework and the introduction of a domain specific language (DSL), we achieve **G5** – flexibility/extensibility.

Besides the verifiability of eBPF programs, we choose eBPF as the filtering mechanism in LBM to strike a balance between
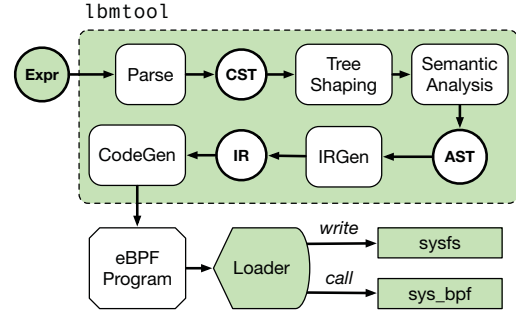

Figure 3: The flow of LBMTOOL in compiling LBM rules to eBPF programs and loading them into the running kernel.

performance and programmability. As shown in Table II, both USBFILTER and USBFirewall rely on hardcoded C compiled into the kernel to implement the filter mechanism. Although USBFirewall leverages the Haskell description of the specification to generate the C code, it lacks support for a user-space DSL. USBFILTER only supports a limited DSL following the conjunctive normal form (CNF). As we will elaborate in the following section, LBM DSL is more expressive and powerful. Instead of implementing a filtering mechanism directly, LBM builds an eBPF running environment for peripherals and executes eBPF programs as filters. Thanks to JIT compilation of eBPF code, LBM is able to run filters as fast as native instructions; thus, we achieve **G7** – high performance.

D. LBM User Space

To interact with an LBM-enabled kernel we design LBM-TOOL, a frontend utility to interact with the LBM kernel space. Its primary purpose is to compile, load, and manage LBM programs resident in the kernel. To create a unified, simple, and expressive way of describing peripheral filtering rules, we develop a custom Domain Specific Language (DSL) modeled on Wireshark and tcpdump filter expressions. These LBM rules are processed by LBMTOOL using a custom compiler that outputs eBPF filter programs, as shown in Figure 3. Compiled filters are loaded into the LBM framework via an extension to the sys_bpf syscall. Programs are then loaded into a specific subsystem: USB, Bluetooth, or NFC.

The filter syntax we develop is concisely described by the grammar shown in Appendix A. Filter rules are effectively stateless expressions that abstract away from the eBPF language syntax. For example, if we want to match on a specific USB device's vendor and product ID, such as a Dell optical mouse, we would write:

```
usb.idVendor == 0x413c && usb.idProduct == 0x3010
```

If we want to include more than one Dell product, we could write multiple rules, or we could consolidate them into a larger expression. To match on a Dell mouse, keyboard, printer, and Bluetooth adapter, we would write:

```
usb.idVendor == 0x413c && (
  usb.idProduct == 0x3010 || // Mouse
```

```
    usb.idProduct == 0x2003 || // Keyboard
    usb.idProduct == 0x5300 || // Printer
    usb.idProduct == 0x8501    // Bluetooth adapter
)
```

The LBMTOOL compiler supports multi-line nested sub-expressions while following the C 89 Standard operator precedence rules [5].

LBMTOOL is able to load a compiled LBM program into a target subsystem TX (OUTPUT) or RX (INPUT) path and specify a match action (i.e., ACCEPT or DROP). The following usage has LBMTOOL compile and load a filter rule:

```
lbmtool --expression "usb.idProduct == 0x3010"
    -o mouse.lbm
lbmtool --load mouse.lbm -t usb -A INPUT -j
    ACCEPT
```

By providing descriptive error-checking in LBMTOOL and developing a custom DSL that is easy to write in and reason about, we achieve **G6** – usability.

IV. IMPLEMENTATION

A. LBM Kernel Space

We divide the implementation of the LBM kernel space into three parts: core, USB implementation, and Bluetooth implementation. All LBM-specific code is located under the `security/lbm` directory of the Linux kernel source tree, as a new security component for the Linux kernel.

LBM Core: To load an eBPF program into LBM, we extend the existing bpf syscall, `sys_bpf`. We define a new program type `BPF_PROG_LOAD_LBM` to distinguish LBM calls from other typical BPF usage. Unlike typical eBPF programs, which normally only persist for the lifetime of the loading process, LBM filters must persist after LBMTOOL exits. To extend the lifetime of these programs, we pin them using the BPF filesystem [17], essentially using the filesystem to increase the reference count of the object.

Before a program is saved by the LBM core, the eBPF verifier checks every instruction of the program for any security violations. Depending on the subsystem (USB or Bluetooth) of the program, LBM provides different verifier callbacks, such as LBM USB or LBM Bluetooth (as we will detail later), thus making sure every memory access of the program is meaningful, aligned, and safe.

Inside LBM, all eBPF programs are organized based on the relevant subsystem and the direction of the filtering path (i.e., TX or RX). We allow the same program to apply for both the TX and RX paths when it is loaded using the BPF syscall, and duplicate the program on TX and RX queues, respectively. The separation of TX and RX paths is mainly for performance, since it allows us to bypass programs that do not interpose on a certain path during filtering. Additionally, to avoid expensive locking, each program is protected by the read-copy-update (RCU) [34] mechanism to enable concurrent reads by different LBM components. LBM modules are also organized according to subsystem and filter path, and protected

```
1  int lbm_filter_pkt(int subsys, int dir, void *pkt)
2  {
3    check_subsystem(subsys);
4    check_path(dir);
5    check_pkt(pkt);
6    res = ALLOW;
7    if (dir == TX) {
8      for_each_ebpf_in_db[subsys][dir] {
9        if (ebpf(subsys, dir, pkt) == DROP) {
10         res = DROP;
11         goto RET;
12       }}
13     for_each_kmod_in_db[subsys][dir] {
14       if (kmod(subsys, dir, pkt) == DROP) {
15         res = DROP;
16         goto RET;
17       }}
18   } else { /* Ditto for the RX */ }
19 RET:
20   return res; }
```

Figure 4: Pseudo-code of `lbm_filter_pkt`.

Subsystem	# of Fields	# of BPF-helpers	# of Lines
USB	34	31	621
Bluetooth-HCI	30	29	683
Bluetooth-L2CAP	28	27	744
TOTAL	92	87	2048

Table III: LBM statistics per subsystem, including # of fields exposed to the user space, # of BPF helpers implemented, and # of lines of code changes.

by RCU. The pseudo code of `lbm_filter_pkt`, previously mentioned in Section III-C, is presented in Figure 4.

To ease the management of LBM filters and modules, we expose ten entries under `/sys/kernel/security/lbm/`, including a global switch to enable/disable LBM; per-subsystem switches to enable/disable debugging, profiling, and statistics; and per-subsystem-per-path controls to view/remove loaded filters and modules. The whole implementation of LBM core is around 1.6K lines of code.

LBM USB: As shown in Figure 2, LBM hooks into the Host Controller Device (HCD) core implementation to cover both TX and RX paths. These hooks eventually call `lbm_filter_pkt` before the packet reaches the USB core, as demonstrated below:

```
lbm_filter_pkt(LBM_SUBSYS_INDEX_USB, LBM_DIR_TX,
    (void *)urb);
lbm_filter_pkt(LBM_SUBSYS_INDEX_USB, LBM_DIR_RX,
    (void *)urb);
```

Every USB packet (`urb`) then needs to go through the LBM core for filtering before being sent to or received from USB peripherals.

To support writing rules in LBMTOOL, we expose packet metadata maintained by the kernel and packet fields defined by the USB specification to the user space. To achieve this, a naive approach would be to mirror the `urb` structure to the userspace, while providing every field explicitly in the filter DSL. Unfortunately, exposing raw kernel structures

to the userspace is a security risk, as doing so will leak sensitive kernel pointer values, which can be used to break KASLR [24]. Explicitly supporting every field is infeasible as well, given the complexity of the protocol suites. As a trade-off, we expose the most commonly recognized and used fields, while providing special BPF helpers for accessing the rest of the fields. These helpers allow LBM filters to support array accesses to `urb` structures, thus enabling them to access every field within a USB packet.

As shown in Table III, we expose 34 fields and implement 31 BPF helpers for the USB subsystem. Besides the special BPF helpers mentioned above for accessing packet fields, additional helpers are implemented for returning the length of a buffer or string, or for providing access to the indirect members of the `urb` structure. For fields that are direct members, no helper is needed since we can access them using an offset from within the `urb`. We group these fields together in a struct and expose it to the user space, as listed below:

```
struct __lbm_usb {
  __u32 pipe;
  __u32 stream_id;
  __u32 status;
  __u32 transfer_flags;
  __u32 transfer_buffer_length;
  __u32 actual_length;
  __u32 setup_packet;
  __u32 start_frame;
  __u32 number_of_packets;
  __u32 interval;
  __u32 error_count; };
```

Instead of exposing `urb` itself to the user space and using the corresponding offsets, LBMTOOL only needs to know the `__lbm_usb` struct and use offsets against it to directly access these fields. LBM handles the translation of struct member access within `__lbm_usb` into one within the kernel `urb`.

To help the BPF verifier understand the security constraints of LBM and the scope of the USB subsystem, we implement three callbacks within the `bpf_verifier_ops` struct used by the verifier. We first explicitly enumerate all legal BPF helpers for the verifier, including the 31 LBM USB BPF helpers mentioned above as well as other common BPF map helpers. We exclude any existing BPF helpers designed for the networking subsystem. Therefore, the verifier would reject any LBM USB filters that use BPF helpers besides the ones specified. We then validate every member access of `__lbm_usb` within the range, and forbid any memory write operations. Finally, we rewrite the instructions accessing `__lbm_usb` and map them into corresponding `urb` accesses.

LBM Bluetooth: The implementation for Bluetooth follows the same procedure as for USB. We place hooks into the Host Control Interface (HCI) layer of the Bluetooth subsystem, as HCI talks to the Bluetooth hardware directly. While HCI provides the lowest-level of packet abstraction for the upper layers, it is not easy for normal users to interact with this layer since it lacks support for high-level protocol elements, such as connections and device addresses, which are better known to Bluetooth users. To bridge this semantic gap, we add another
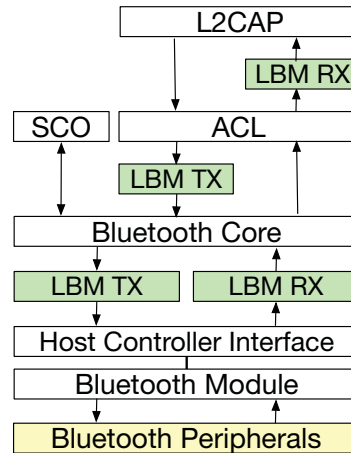

Figure 5: LBM hooks inside the Bluetooth subsystem.

set of hooks into the Logical Link Control and Adaptation Protocol (L2CAP) layer right above HCI, as shown in Figure 5. These hooks are effectively calls to `lbm_filter_pkt`, as demonstrated below:

```
lbm_filter_pkt(LBM_SUBSYS_INDEX_BLUETOOTH,
    LBM_DIR_TX, (void *)skb);
lbm_filter_pkt(LBM_SUBSYS_INDEX_BLUETOOTH,
    LBM_DIR_RX, (void *)skb);
lbm_filter_pkt(LBM_SUBSYS_INDEX_BLUETOOTH_L2CAP,
    LBM_DIR_TX, (void *)skb);
lbm_filter_pkt(LBM_SUBSYS_INDEX_BLUETOOTH_L2CAP,
    LBM_DIR_RX, (void *)skb);
```

The Bluetooth packet is encapsulated in a socket buffer, or `skb` in kernel parlance, for both the HCI and the L2CAP layers. During development, we encountered two challenges while hooking the TX path of L2CAP. Unlike for the RX path, the L2CAP layer provides multiple functions for sending out L2CAP packets. Even worse, because of different Maximum Transmission Unit (MTU) sizes between HCI and L2CAP, an L2CAP packet is usually fragmented during packet construction before being sent to the lower layer. One possible solution would be to place LBM hooks inside every function on the TX path and reassemble the packet there. Besides the resulting code duplication, the major fault in this solution is the maintenance burden of adding hooks to new TX functions.

To solve these challenges, we deploy only one LBM hook at the Asynchronous Connection-Less (ACL) layer within HCI and reassemble the original L2CAP packet there, while fully covering all TX cases used by the L2CAP layer. Note that the RX path still has the LBM hook inside the L2CAP layer, as the kernel has already handled the packet reassembly.

As shown in Table III, we expose 30 and 28 protocol fields from the HCI and L2CAP layers, respectively. Note that both layers share the same 12 fields related with connections. For a HCI packet, a BPF helper is provided to check if a connection is established (indicated by the availability of these fields). For L2CAP, a connection is always established. We also implement 29 and 27 BPF helpers for HCI and L2CAP, respectively, which can retrieve the value of exposed fields. As with the

USB subsystem, we enumerate all the legal BPF helpers that can be called within the Bluetooth subsystem, and restrict the memory write operations in the verifier.

B. LBM User Space

LBMTOOL is responsible for compiling LBM rules to eBPF programs and loading them into the kernel. Rules/filters pass through standard compilation stages before ending up in the kernel as compiled eBPF. To begin, we tokenize and parse the input LBM filter. To simplify these initial steps we use *Lark*, a dependency-free Python library that supports LALR(1) grammars written in EBNF syntax. *Lark* processes our LBM rule grammar and creates a working standalone parser. Once filters are lexed, they are parsed into a Concrete Syntax Tree (CST), also known as a parse tree [4]. The raw parse tree is then shaped and canonicalized over multiple steps into a friendlier representation known as an Abstract Syntax Tree (AST). These steps include symbol (e.g., usb.idProduct) resolution, type checking, and expression flattening. After processing, the AST more accurately represents the LBM language semantics and is flattened into a low-level Intermediate Representation (IR) for backend processing.

Our IR is modeled on Three-Address Code (TAC) [4], and it has a close mapping to the DSL semantics. Additionally, we ensure that our IR conforms to Static Single Assignment (SSA) form to simplify register allocation and any late IR optimization passes. Once we have optimized our IR, it moves to the eBPF instruction generator. There, we allocate registers and translate each IR instruction into corresponding eBPF instructions. Our register allocator maps an infinite number of virtual registers from our SSA IR to a fixed number of eBPF physical registers. To do this, it builds an interference graph [22] of the IR statements in the program. This graph encodes the lifetime of each virtual register throughout the program and aids in quickly selecting appropriate physical registers during the allocation process. With registers allocated, each IR statement is processed in order by the eBPF instruction generation backend to produce assembly instructions. With machine code produced, any remaining control transfer labels are resolved by a final two-pass assembly step. The resulting eBPF instructions are packaged into a LBM object file with metadata for loading into the kernel. For an example of the compiler's output at each stage, visit Appendix B.

V. EVALUATION

To evaluate LBM, we first demonstrate how users can write simple LBM rules to protect protocol stacks and defend against known attacks through case studies. These case studies center around the USB and Bluetooth stacks, ending with a proof-of-concept implementation of NFC support in LBM. We divide the cases between specific attacks from malicious peripherals and general host system hardening against potential peripheral threats. The next part of our evaluation focuses on benchmarking the performance of LBM. We divide the benchmarking into our testing setup, micro-benchmark, (providing LBM overhead per packet), macro-benchmark (showing LBM overhead on the application and system level), and scalability (covering 100 LBM rules and comparing LBM with previous solutions).

A. Case Studies

Kernel Protocol Stack Protection: To protect the kernel's USB protocol stack similar to USBFirewall, we extract protocol constraints from the USB specification and translate them to LBM rules for loading via LBMTOOL. For example, to ensure the response of a Get_Descriptor request is well-formed during the enumeration phase, we write:

```
((usb.setup_packet != 0) && /* For enumeration */
 (usb.request[0] == 0x80) && /* Get_Descriptor */
 (usb.request[1] == 0x06) &&
 /* Make sure response contains at least 2 bytes
    */
 ((usb.actual_length < 2) ||
  /* Make sure the descriptor type matches */
  ((usb.request[3] != usb.data[1]) ||
   /* Device descriptor */
   ((usb.request[3] == 1) && ((usb.data[0] != 18)
      || (usb.actual_length != 18))) ||
   /* Configuration descriptor */
   ((usb.request[3] == 2) && ((usb.data[0] < 9)
      || (usb.actual_length < 9))) ||
   /* String descriptor */
   ((usb.request[4] == 3) && ((usb.data[0] < 4)
      || (usb.actual_length < 4)))))))
```

We first make sure the response has at least 2 bytes, for extracting the length (usb.data[0]) and type (usb.data[1]) of the response. We reject the packet if there is a type mismatch between request and response. Depending on the descriptor type, we then make sure the response has the minimum length required by the specification. To fully cover all the responses during USB enumeration, we also check the response returned by Get_Status in a similar fashion. We use FaceDancer [30] and umap2 [57] to emulate a malicious hub device fuzzing the host USB stack. Our stack protection filters are able to drop all malformed packets during USB enumeration.

To protect the Bluetooth stack within the kernel, we extract the constraints from the Bluetooth specification and rewrite them using LBMTOOL as follows:

```
/* HCI-CMD */
((bt.hci.type == 1) && (bt.hci.len < 3)) ||
/* HCI-ACL */
((bt.hci.type == 2) && (bt.hci.len < 4)) ||
/* HCI-SCO */
((bt.hci.type == 3) && (bt.hci.len < 3)) ||
/* HCI-EVT */
((bt.hci.type == 4) && (bt.hci.len < 2)))
```

This rule provides basic protection for the HCI layer. Depending on the packet type, we make sure the response has the minimum length required by the specification. We also implemented similarly styled protection for the L2CAP layer.

Preventing Data Leakage: In addition to propagating malware, USB storage devices are also used to steal sensitive information from a computer. To tackle this threat, USBFILTER implemented a plugin to drop the SCSI write command

on the TX path, thus preventing any data from being written into a connected USB storage device; this plugin mechanism is referred to as Linux USBFILTER Module (LUM).

Recall LBM is designed to support the features of existing solutions. We are able to port the SCSI-write-drop LUM to LBM with only around 10 lines of code changes (primarily adjusting naming of callbacks and header files). In fact, any LUM can be ported to LBM with similarly minimal changes, because LUMs can be treated as a special case on USB in LBM. As they are essentially kernel modules, neither LUMs nor LBM module extensions are as constrained as the LBM filter DSL, given that they are written in C and call kernel APIs directly.

Trusted Input Devices: One of the most common BadUSB attacks relies on the Human Interface Device (HID) class, in which a malicious USB device behaves like a keyboard, injecting keystrokes into the host machine. With LBM, we can write a rule specifying a trusted input device, such that keystrokes from all other input devices are dropped, as follows:

```
((usb.pipe == 1) && /* INT (Keystroke) */
 ((usb.manufacturer != "X") ||
  (usb.product != "Y") ||
  (usb.serial != "Z") ||
  (usb.plugtime != 12345)))
```

For all keystrokes, we check against the expected manufacturer, product, and serial number of the trusted input device. This rules out any devices from different vendors or different device models, and only permits keystrokes from the trusted input device without completely disabling the USB keyboard functionality. Similarly to writing udev rules, system administrators can plug in their trusted input devices to collect the device information before writing and loading LBM filters into the kernel. In case of a BadUSB device spoofing its identity, we extend the USB hub thread to report the initial timestamp when a device was plugged in, and expose this field to user space. Sysadmins can discover this timestamp in dmesg and include it as part of a LBM rule.[2] As such, even if a malicious device were able to mimic the identity of the trusted input device, the malicious keystrokes would be dropped because the initial timestamp would differ.

Securing USB Charging: A well-known defense against BadUSB attacks by USB chargers is the "USB condom" [75], which effectively physically disconnects the USB data pins (D+/-) from the USB bus. Unfortunately, this prevents phones that support USB Battery Charging [63] from drawing extra power via the data wires. As a result, fully charging a phone may take 15 times as long due to the lower amperage. Additionally, a comparable device is not available for USB Type-C. Using LBM, we could instead implement a software USB data blocker:

```
((usb.busnum == 1) && (usb.portnum == 1))
```

After applying this LBM rule to the RX path, we are able to drop any data transmission from the physical USB port 1 under bus 1, thus making the port charge-only for any connections. This LBM rule does not interfere with USB Battery Charging, since the data wires are still physically connected, and can be applied to any physical USB port, regardless of whether or not it is Type-C.

Securing Bluetooth Invisible Mode: To prevent a Bluetooth device from being scanned by another (potentially) malicious device, such as during a Blueprinting [38] or BlueBag [21] attack, Bluetooth introduces discoverable and non-discoverable modes to devices. A device in non-discoverable mode does not respond to inquires from other devices, thus hiding its presence from outsiders. On one hand, the toggling of this mode can be controlled from the user space, (e.g., using bluetoothctl, which should require root permission). On the other hand, any vulnerabilities within these user-space daemons and tools, once exploited, might put the device into discoverable mode again. To prevent this, we could define a LBM rule as follows:

```
((bt.hci.type == 1) && /* HCI-CMD */
 (bt.hci.command.ogf == 3) && /* Discoverable */
 (bt.hci.command.ocf == 58))
```

This rule detects the HCI command used to enable the discoverable mode on the device. Once applied to the TX path, the rule drops any request from the user space attempting to put the device into discoverable mode. We could write a similar rule to enforce non-connectable mode, which is used to prevent any Bluetooth connection to the device, even if its MAC address is known beforehand.

Controlling Bluetooth/BLE Connections: Along with the rise of IoT devices, which often rely on Bluetooth Low Energy (BLE), Android devices began to support BLE since version 4.3 [8], with iOS adding support from the iPhone 4S forward. The Linux kernel Bluetooth stack (BlueZ [48]) also supports both classic Bluetooth and BLE at the same time. Although it is not uncommon to see a dual-mode device supporting both classic Bluetooth and BLE, it is surprisingly challenging (if not impossible) to enable only one of them while disabling the other. [16] With LBM, enabling/disabling Bluetooth or BLE connections is just a one-liner:

```
((bt.hci.conn == 1) && /* A link exists */
 (bt.hci.conn.type == 0x80)) /* BLE link */
```

This LBM rule checks the connection type for each Bluetooth or BLE packet, and drops the packet if the connection is BLE, thus preventing unfamiliar IoT devices from establishing a connection while still allowing classic Bluetooth connections. It also provides a quick workaround for BleedingBit attacks [12] without waiting for firmware updates. Simply changing == 0x80 to != 0x80 achieves the opposite effect, only permitting BLE connections and thus providing a temporary defense against BlueBorne attacks [11].

Defending Against BlueBorne: BlueBorne attacks exploit vulnerabilities within Bluetooth protocol stack implementa-

tions, by sending either malformed or specially crafted Bluetooth packets. Within the Linux kernel, this vulnerability resulted from a missing check before using a local buffer. As a result, a crafted packet could cause a kernel stack overflow, potentially leading to remote code execution. Although the fix was a straightforward one, adding the missing checks [68], and applying patches to existing devices still requires additional steps of rebuilding the kernel and flashing new firmware. With LBM, we can write a simple rule to properly defend against the potential kernel stack overflow:

```
((bt.l2cap.cid == 0x1) && /* L2CAP Signaling */
/* Configuration Response */
(bt.l2cap.sig.cmd.code == 0x5) &&
(bt.l2cap.sig.cmd.len >= 66))
```

We first pinpoint where the vulnerability was triggered, which is at the L2CAP layer during configuration response. Because the local buffer is 64 bytes and the first 4 bytes are used for the header, the actual data buffer to hold configuration options is 60 bytes. In the rule above, `bt.l2cap.sig.cmd.len` denotes the total length of a L2CAP command packet. Without counting the 6-byte header, the actual payload size of a command packet is `cmd.len – 6`. To defend against BlueBorne attacks, all we need is to make sure `(cmd.len – 6) < 60`. Therefore, our rule, which is written to drop any configuration response larger than 66 bytes, will put a stop to BlueBorne. The above two rules demonstrate that LBM provides a dynamic patching capability to protocol stacks within the kernel, without waiting for official kernel patches or firmware updates to be upstreamed.

NFC Support: To further show the generality of LBM, we extend LBM to support NFC. Unlike Bluetooth, NFC has three different standards (software interfaces) for communicating with NFC modules, including HCI [28], NCI [59], and Digital [58]. As a proof-of-concept, we focus on NCI, exposing two protocol fields and implementing one BPF helper. The number of additional lines of code added to the kernel and LBMTOOL to make LBM support NFC is shown in Table IV.

Step 1: Placing LBM hooks. NCI provides unique interfaces to cover both TX and RX transmission: `nci_send_frame` and `nci_recv_frame`. As for other networking subsystems, `skb` is used to carry NFC packets. We place the following LBM hooks at the two interfaces:

```
lbm_filter_pkt(LBM_SUBSYS_INDEX_NFC, LBM_DIR_TX,
    (void *)skb);
lbm_filter_pkt(LBM_SUBSYS_INDEX_NFC, LBM_DIR_RX,
    (void *)skb);
```

Step 2: Exposing protocol fields. We expose the packet length (`nfc.nci.len`) and message type (`nfc.nci.mt`) fields to the user space. The packet length is a member of the `struct __lbm_nfc` exposed in the LBM user-space header file. The message type is implemented as a BPF helper calling other NCI APIs.

Step 3: Enhancing lbmtool. LBMTOOL is easily extensible for new protocols, as we do for NFC. The internal LBM-

NFC	Kernel	lbmtool	Total
# of lines	85	12	97

Table IV: The number of lines added to support NFC.

LBM Rule	Purpose	# of Insn	Scope
USB-1	Stack Protection	72	Micro/Macro BM
USB-2	Stack Protection	25	Micro/Macro BM
USB-3	User Defined	22	Scalability BM
HCI-1	Stack Protection	81	Micro/Macro BM
L2CAP-1	Stack Protection	76	Micro/Macro BM

Table V: Details about the five LBM rules used during the benchmarks.

Subsystem	Min	Max	Avg	Med	Dev
USB	0.29	11.18	**1.26**	1.83	0.44
	0.12	8.87	**0.55**	0.28	0.33
Bluetooth-HCI	1.16	17.87	**2.81**	2.70	0.62
	0.27	15.67	**0.98**	0.77	0.47
Bluetooth-L2CAP	1.32	25.87	**2.93**	2.99	0.67
	0.44	23.76	**1.15**	1.26	0.53

Table VI: LBM overhead in μs based on processing 10K packets on the RX path. For each subsystem, the 1st row is for normal LBM and the 2nd row is for LBM-JIT. In most cases, the overhead of is within 1 μs when JIT is enabled.

rule code generation backend is abstracted from the specific subsystem the rules will apply to. As such, the only changes required to support NFC are to include a symbol descriptor table for each variable exposed to the user space by the kernel. Once these changes are incorporated, LBMTOOL accepts LBM filters with NFC protocol fields and compiles them into eBPF instructions.

B. Benchmark Setup

We performed all of our benchmarks on a workstation with a 4-core Intel i5 CPU running at 3.2 GHz and 8 GB memory. The peripheral used during testing include a 300 Mbps USB 2.0 WiFi adapter, a Bluetooth 4.0 USB 2.0 adapter, and a 500 GB USB 3.0 external storage device. Depending on the benchmark, some subset of devices were connected.

We list all the LBM rules used during the benchmarks in Table V. We deploy all the rules on the RX path, since our protection target is the host machine. In addition to the "Stack Protection" rules mentioned in the case studies, we include "USB-3", a user defined rule similar to `usb.serial == "7777"` which drops the USB packet if the sending device's serial number is 7777. As no devices that we test have a serial number matching this pattern, we mainly use this rule for the scalability benchmark.

C. Micro-Benchmark

For USB testing, we load LBM rules "USB-1" and "USB-2" into the system. We then capture 10K USB packets on the RX path from the WiFi adapter. As shown in the first two rows of Table VI, the average overhead is 1.26 μs per packet. When JIT is enabled, the overhead is reduced to 0.55 μs.
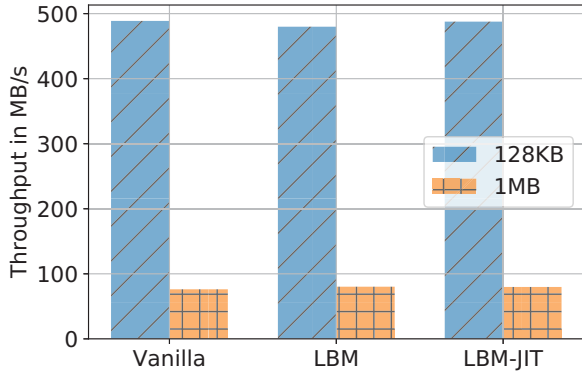
Figure 6: `filebench` across different kernel configurations. All configurations achieve similar throughputs, meaning a minimum performance impact from LBM.
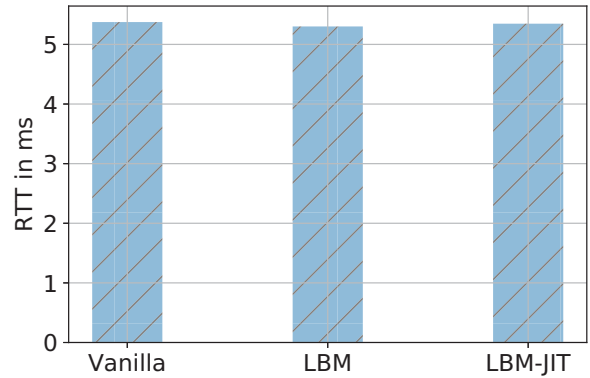

Figure 7: RTT of `l2ping` in milliseconds (lower is better) based on 10K pings, across different kernel configurations. All configurations achieve similar throughputs, meaning a minimal performance impact from LBM.

For Bluetooth testing, we load LBM rules "HCI-1" and "L2CAP-1" into the system. We implement a simple L2CAP client/server protocol based on PyBluez [1] to generate 10K packets on the RX path for the HCI and L2CAP layers, respectively. As shown in the last four rows of Table VI, the average overheads are 2.81 μs for HCI and 2.93 μs for L2CAP. Again, with the help of JIT, we can reduce the overhead to around 1 μs.

Takeaway: the general overhead introduced by LBM is around 1 μs for most cases.

D. Macro-Benchmark

For USB, we load the rules "USB-1" and "USB-2" and use `filebench` [50] to measure the throughput of the USB 3.0 external storage device. We chose the "fileserver" workload model with 10K files, 128KB and 1MB mean file sizes, 10 working threads, and 10-min running time. This workload generates roughly 1GB and 10GBs of files, respectively, within the storage device. As shown in Figure 6, all kernel configurations achieve similar throughput during our testing. When the mean file size is 128KB, the total file size (1 GB) can easily fit into the system page cache. Thus, we are able to achieve close to 500 MB/s throughput (faster than the hard drive's maximum speed of 150 MB/s). When the mean file size is 1MB, the total file size (10 GB) cannot completely fit into the page cache, thus resulting in much lower throughput.

For Bluetooth, we load the rules "HCI-1" and "L2CAP-1" and use `l2ping` [49] to benchmark the Round-Trip-Time (RTT) for 10K pings. As with the USB testing, all kernel configurations achieve similar RTTs of around 5 ms, as shown in Figure 7. Because the overhead of LBM is under 1 μs in general (Section V-C), the overhead contributed to the RTT measurement is negligible.

To double-check that LBM introduces a minimal overhead across the whole system, we use `lmbench` [55] to benchmark the whole system across different kernel configurations. The complete summary is available in Appendix C. In short, LBM achieves comparable performance with the vanilla kernel.
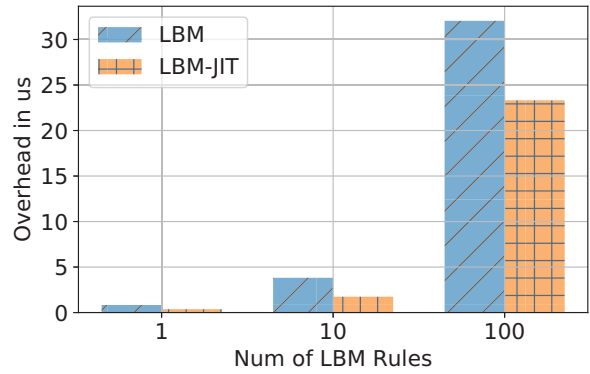

Figure 8: LBM overhead in μs based on varying numbers of rules. While the general overhead increases as the number of rules increases, the overhead of going through each individual rule decreases, thus the total overhead is essentially amortized.

Takeaway: the overhead introduced by LBM is negligible for applications and for the system as a whole.

E. Scalability

To understand the scalability of LBM, we load the rule "USB-3" into the RX path once, 10 times, and 100 times. As in the micro-benchmark, we record 10K USB packets generated by the USB WiFi adapter and compute the overhead of LBM going through these rules for each packet. As shown in Figure 8, while the total overhead increases as the number of rules increases, the average overhead of checking individual rules decreases. The average overhead was 0.83 μs when there was only one rule loaded. It decreased to 0.32 μs when there were 100 rules loaded. Under JIT, the overhead was further reduced to 0.23 μs. This might be the result of increased cache hits from accessing the same rule in a loop. Even for different rules, it is possible to observe this amortization effect, as long as each rule occupies a different cache line. Also, in
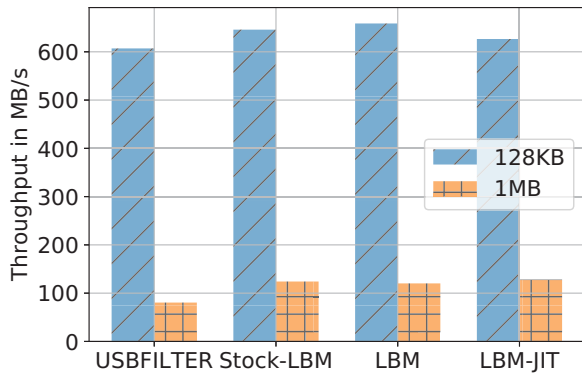
Figure 9: LBM vs. USBFILTER benchmark using `filebench` with 10 same rules loaded respectively. LBM introduces a minimum overhead comparing to the stock kernel and performs better than USBFILTER in general.
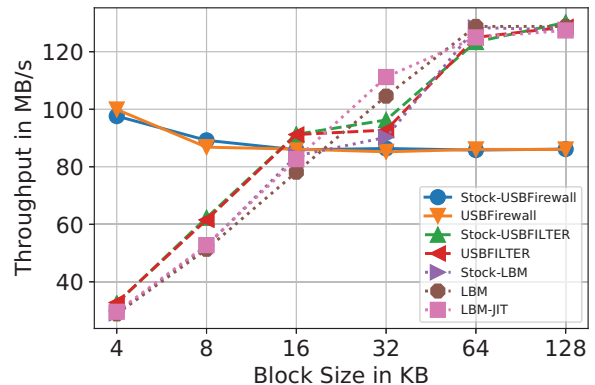

Figure 10: LBM vs. USBFILTER vs. USBFirewall benchmark using `dd` with 10 same rules loaded respectively. Comparing to their stock versions, all the solutions show minimum overhead. USBFirewall does not vary much based on the block size. LBM performs better than USBFirewall and USBFILTER when block size is beyond 16 KB in general.

general, more complicated rules will also induce more runtime overhead.

We then compare LBM with USBFILTER using `filebench`.[3] Except the difference in kernel versions[4], we ran LBM and USBFILTER on the same physical machine. To set up the benchmark, we load "USB-3" into the RX path 10 times on LBM and load an equivalent rule the same number of times into USBFILTER. As shown in Figure 9, both LBM and LBM-JIT show a minimum overhead comparing to the stock kernel, and provide better throughput than USBFILTER regardless the mean file size. This could be the result of both kernel code improvements across versions and the design of LBM (e.g., due to its use of eBPF). The throughput boost is even clearer when the mean file size is 1MB and JIT is enabled. Compared to USBFILTER, LBM-JIT improves the throughput by roughly 60%.

Finally, we compare LBM with USBFILTER and USBFirewall using `dd` on VFAT filesystem with direct I/O enabled to bypass the page cache. Since USBFirewall does not support loading rules from the user space directly, we statically built these 10 rules when compiling USBFirewall. As shown in Figure 10, comparing to their stock versions, all the solutions show minimum overheads. The throughput of USBFirewall does not vary much based on the block size. We tried both the native FreeBSD version of `dd` and the GNU version. Both demonstrate similar throughput regardless the block size. We double check this by increasing the block size to 1 MB. When the block size is beyond 16 KB, both LBM and USBFILTER show better throughput than USBFirewall. Similarly, both LBM and LBM-JIT have better throughput than USBFILTER.

Takeaway: compared to other state-of-the-art solutions, LBM provides better scalability and performance.

---

[3] Due to a kernel bug within USBFILTER, the front USB 3.0 ports could not support USB 3.0 devices. We switched to the rear USB 3.0 ports in this testing. We also tried to run USBFirewall. Unfortunately, FreeBSD does not support `filebench` or EXT4 filesystem used by our external drive.

[4] LBM is running Linux kernel 4.13 while USBFILTER runs 3.13.

VI. DISCUSSION

A. LBM vs. USBFILTER vs. USBFirewall

The LBM filter DSL is more expressive than the USB-FILTER policy, which only supports concatenating equality checks using logical AND. The LBM filter DSL supports different arithmetic and logical operations, as well as changing of operation precedence using parentheses. Compared to US-BFILTER, LBM USB also doubles the number of protocol fields exposed to the user space, although LBM does not support pinning applications to peripherals.[5] Nevertheless, LBM enables more complicated and powerful filtering rules than USBFILTER. Besides, any LUM can be converted into an LBM module without much hassle. LBM USB has also fully replicated functionality provided by USBFirewall, which required a kernel recompile and reboot to make any rule changes.

B. L2CAP Signaling in Bluetooth

Unlike L2CAP signaling in BLE, where each L2CAP packet only carries a single command, L2CAP signaling in the Bluetooth classic may have packets containing multiple commands. As we saw in the BlueBorne defense case study, if there is a malicious configuration response command contained in a L2CAP signaling packet, the entire payload will be dropped, including other "innocent" commands if they exist.

One possible solution to such coarse-grained drops is to separate each command from the same L2CAP signaling packet into standalone packets. This requires packet parsing and duplication in the early stage. Another solution is to add a new customized hook in the place where each command is extracted by the L2CAP stack. Our current implementation

---

[5]USBFILTER instrumented some USB device driver to support application pinning. It is ad-hoc, rather than a generic method.

does not apply either solution, for performance and simplicity considerations. From a security perspective, if one command from a certain device is recognized as malicious, it seems reasonable to drop other commands from the same device.

C. BPF Memory Write

For security considerations, we forbid memory writes in LBM eBPF programs. While this restriction improves the kernel's security posture towards user-loaded code, we also lose a powerful feature provided by eBPF and BPF helpers—packet mangling, which allows for fields to be changed on the fly. This feature has been employed by the networking subsystem, e.g., for changing the source IP address and/or the destination port number. For LBM, one potential use of memory write is removing only malicious commands while keeping others within the same L2CAP signaling packet intact. As an intermediate step to enable memory write in LBM programs, we can restrict the memory write ability to certain BPF helpers. As long as these BPF helpers are safe, the BPF verifier can still verify these programs by rejecting store instructions as before.

D. BPF Helper Kernel Modules

Ideally, we should allow BPF helpers for each subsystem to be implemented as a standalone kernel module, which can be plugged in when needed. Unfortunately, this is forbidden by the current eBPF design, and we follow the same design principles for similar reasons. First of all, BPF helpers are like syscalls in a system. The number of a BPF helper is like the syscall number, which is part of the Application Binary Interface (ABI) of the system. Although by introducing LBM, we have essentially namespaced LBM BPF helpers from other general and networking-specific helpers, these helpers still share the same LBM namespace regardless their respective subsystems. As a result, the number of a LBM BPF helper implemented within a kernel module cannot be decided until all used numbers are known, including the ones defined by LBM internals and those defined in other BPF helper modules. A possible solution here is to further namespace LBM BPF helpers per subsystem, e.g., have USB helpers always start with 100, Bluetooth helpers with 200, etc. Note that this solution would consequently limit the number of helpers each subsystem could have.

E. LLVM Support

LLVM began to support eBPF as an architectural backend in early 2015 [71]. A typical workflow involves writing an eBPF filter in C and compiling it using Clang. eBPF loaders such as `tc` are able to parse the generated ELF file and load it into the kernel [18]. While LLVM brings C into eBPF programming, easing filter writing for C developers, we realized that eBPF programming might still be challenging for sysadmins, who need an easy and intuitive way to write eBPF filters; we designed the LBM filter DSL with this in mind. We are planning to support LLVM as well by adding a new eBPF loader into LBM.

VII. LIMITATIONS

A. Stateless vs. Stateful Policy

LBM filters are designed to be policy-independent, although a large part of the case studies presented stateless polices. Whether the policy is stateless or stateful essentially depends on what protocol fields and packet data are exposed to the user space. For example, USB does not have a "session" concept, and we could write useful LBM filters based on just the device information (a.k.a., stateless policy). Bluetooth has the "connection" concept in the L2CAP layer (like TCP connections), so we could write LBM filters using this field (a.k.a., stateful policy). Besides protocols fields defined by standards, the Linux kernel also maintains some bookkeeping data structures, e.g., counters. Exposing these kernel fields would also help to create stateful polices.

The current LBM USB and Bluetooth implementations focus on exposing basic protocol fields rather than stateful variables. Nevertheless, we have noticed the potential of stateful policies. For instance, we could write a stateful policy to detect BleedingBit [12] attacks by observing a sequence of multiple BLE advertising packets with a certain bit off followed by another BLE advertising packet with that bit on.

B. DMA-Oriented Protocols

We have not instantiated LBM on Thunderbolt, HDMI, or DisplayPort, although it is indeed possible to support these DMA-oriented protocols using LBM.[6] Since LBM works at the packet layer, we are able to filter packets for these protocols as long as the concept of packet, given a protocol, is defined by the standard and implemented by the kernel. For example, DisplayPort defines different packets to carry different payloads such as stream and audio [46], implemented as such within the kernel. Thunderbolt, however, is a proprietary standard. It is not clear whether the protocol itself is packetized, and the only packet-level message available within the kernel is Thunderbolt control request/response instead of data transfer. Another challenge to supporting these protocols comes from determining the proper hook placement for complete mediation. DisplayPort is not a standalone subsystem but rather a component of Direct Rendering Manager (DRM) inside the kernel. Thunderbolt does not have a core layer but only provides few drivers due to the limited hardware devices.

C. Operating Systems Dependency

Although LBM is built upon the Linux kernel, it is possible to apply LBM to other operating systems. To achieve that, we need the target operating system to support a generic in-kernel packet filtering mechanism such as eBPF. The classic BPF is not enough because LBM relies on calling kernel APIs within filters to access different kernel data. While it is non-trivial to extend the classic BPF to eBPF, some porting effort has been done for FreeBSD to support eBPF [35]. The other requirement is a software architecture enabling complete-mediation hook placement for different peripherals. For instance, it is

---

[6]USB is also DMA-oriented.

possible to mediate all USB packets within the FreeBSD USB subsystem, as proven by USBFirewall. Nevertheless, it might be challenging to port LBM to Windows, since it has a different packet filtering mechanism [85] and it is closed-source.

D. lbmtool Limitations

LBMTOOL currently does not support LBM filter consistency checking, meaning it is possible to have two LBM filters conflict with each other. Regarding eBPF instruction generation, LBMTOOL does not support stack allocations when the return value of BPF helpers is beyond 8 bytes (width of an eBPF register). Manual assembly is needed to manipulate the stack for those BPF helpers. LBMTOOL also does not support lazy evaluation on BPF helpers. They are always called at first to retrieve all the values of protocols fields needed before the actual evaluation of the LBM filter DSL expression. These are merely the current limitations of the custom compiler itself and could be eliminated with additional code.

VIII. RELATED WORK

Peripheral Security Defenses: A number of solutions have considered aspects of defenses against malicious peripherals. By treating USB kernel drivers as capabilities, GoodUSB [76] asks for user's expectation about the device before loading the drivers. Cinch [9] interposes on the USB bus by isolating the suspicious USB device within a VM environment, with the help of IOMMU and hypervisors, but imposes substantial performance overhead and considerable architectural changes to systems it is deployed upon. USBFILTER [79] is a USB packet filtering mechanism built into the Linux kernel. Users can write simple filtering rules and pass them into the kernel space. USBFirewall [43] protects the USB protocol stack within FreeBSD from malformed packets by generating the USB packet parser from Haskell. Other solutions focus on developing more secure devices; for example Kells [20] and ProvUSB [77] protect USB devices from malicious hosts at the granularity of partitions and blocks, respectively, but require the deployment of new peripheral devices. Solutions such as FirmUSB [39] allow analysis of a device for malicious intent but require a means of accessing its firmware. For more details regarding other related defenses, we refer readers to a systematic study on USB security [80]. Thunderbolt 3 also introduced security levels, and boltctl [44] is used to set security levels for different peripherals on Linux. These security levels are designed to control the creation of PCIe channels from peripherals rather than high-level packets. As previously discussed, LBM is designed as a generic framework working at the packet layer, not only enabling existing solutions such as USBFILTER and USBFirewall, but also covering other peripherals such as Bluetooth and NFC.

eBPF-based Solutions: BPF Compiler Collections (BCC) [42] provides a Python interface for writing, compiling, and loading eBPF programs. Its backend is still LLVM and C programing. eXpress DataPath (XDP) [36] provides eBPF hooks within the NIC drivers, filtering packets before `skb` is created to store the packet. Network Flow Processor (NFP) [45] allows filtering packets within the NIC by JITing eBPF programs into native NIC instructions and running them on the NIC directly. eBPF tracing tools [33] provide an alternative for DTrace on Linux. Bpfilter [73] is an ongoing project trying to replace the `iptables` firewall. InKeV [3] is a network virtualization solution allowing inserting network functions dynamically using eBPF. Hyperupcalls [6] allows VMs to load eBPF programs and asks hypervisors to execute them. One can treat these hyperupcalls as another form of BPF helpers. On the security side, eBPF has been hardened against JIT spray attacks [64] and Spectre attacks [47], [72], [19]. New file mode and LSM hooks are also added for eBPF program permission control to remove the dependency on "CAP_SYS_ADMIN" [29]. LBM expands the scope of eBPF usage by exploring peripheral space.

Linux Kernel Security Frameworks: Linux Security Modules (LSM) [86] is a general framework to implement MAC on Linux, by providing hundreds of hooks for security-sensitive operations within the kernel. Integrity Measurement Architecture (IMA) [65] leverages TPM to measure the kernel image as well as user-space applications. Android Security Modules (ASM) [40] promotes security extensibility to the Android platform, by adding new authorization hooks within Android OS APIs. Linux Provenance Modules (LPM) [14] provides a whole-system provenance framework by mirroring LSM hooks. Seccomp [27] uses the classic BPF filter to limit the number of syscalls that can be invoked by a process or container. Landlock [66] controls how a process could access filesystem objects by writing polices in C within applications and compiling them into eBPF programs using LLVM. Guardat [82] presents a high-level policy language for mediating I/O events, but is implemented at the storage layer, above the peripheral layer, and would thus not provide defenses against protocol-level attacks. While we have seen kernel frameworks covering different aspects of security concerns, LBM is the first framework for unifying defenses across protocols against malicious peripherals.

IX. CONCLUSION

In this paper we described LBM, an extensible security framework for defending against malicious peripherals. LBM implements a high-level filtering language for creating peripheral policies, which compile into eBPF instructions for loading into the Linux kernel to provide performance and extensibility. Within this framework we added support for the USB, Bluetooth, and NFC protocols, described the design process of LBM, and demonstrated specific cases of how LBM could be leveraged to harden the operating system's protocol stacks. Our evaluation of LBM showed that it performs as well as or better than previous solutions, while only introducing overhead within 1 μs per packet in most cases. LBM is practical and to the best of our knowledge, is the first security framework designed to provide comprehensive protection within the Linux

kernel peripheral subsystem, covering different subsystems while supporting and unifying existing defensive solutions.

X. ACKNOWLEDGEMENTS

REFERENCES

[1] PyBluez: Bluetooth Python extension module. https://github.com/pybluez/pybluez, 2018.

[2] O. Afonin. This $39 Device Can Defeat iOS USB Restricted Mode. https://blog.elcomsoft.com/2018/07/this-9-device-can-defeat-ios-usb-restricted-mode/, July 2018.

[3] Z. Ahmed, M. H. Alizai, and A. A. Syed. Inkev: In-Kernel Distributed Network Virtualization for DCN. *ACM SIGCOMM Computer Communication Review*, 46(3), 2016.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers: principles, techniques, and tools. *Addison Wesley*, 7(8):9, 1986.

[5] American National Standards Institute (ANSI). ANSI X3.159-1989: Programming Language C. Technical report, 1989.

[6] N. Amit and M. Wei. The Design and Implementation of Hyperupcalls. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[7] J. P. Anderson. Computer Security Technology Planning Study, ESD-TR-73-51, Vol. 1. Technical report, Air Force Systems Command: Electronic Systems Division, Oct. 1972.

[8] Android Developers. Bluetooth low energy overview. https://developer.android.com/guide/topics/connectivity/bluetooth-le, Apr. 2018.

[9] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish. Defending against Malicious Peripherals with Cinch. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[10] Apple, Hewlett-Packard, Intel, Microsoft, Renesas, STMicroelectronics, and Texas Instruments. Universal Serial Bus 3.2 Specification: Revision 1.0. Technical report, Sept. 2017.

[11] Armis Inc. BlueBorne. https://www.armis.com/blueborne/, 2017.

[12] Armis Inc. Bleeding Bit. https://armis.com/bleedingbit/, 2018.

[13] S. Baghdasaryan. [v3,2/4] NFC: Fix possible memory corruption when handling SHDLC I-Frame commands. https://patchwork.kernel.org/patch/10378895/, May 2018.

[14] A. M. Bates, D. Tian, K. R. Butler, and T. Moyer. Trustworthy Whole-System Provenance for the Linux Kernel. In *Proceedings of the USENIX Security Symposium*, 2015.

[15] Bluetooth SIG, Inc. Bluetooth Core Specification v5.0. Technical report, Dec. 2016.

[16] A. Borg, S. N, and P. Uttarwar. Can BLE be turned on while Bluetooth Classic is off on an Android device? https://www.quora.com/Can-BLE-be-turned-on-while-Bluetooth-Classic-is-off-on-an-Android-device, 2016.

[17] D. Borkmann. [PATCH net-next 3/4] bpf: add support for persistent maps/progs, Oct. 2015. LKML Archive.

[18] D. Borkmann. On getting tc classifier fully programmable with cls bpf. *tc*, (1/23), 2016.

[19] D. Borkmann. [bpf] bpf: use array_index_nospec in find_prog_type. http://patchwork.ozlabs.org/patch/908385/, May 2018.

[20] 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 (ACSAC'10)*, 2010.

[21] L. Carettoni, C. Merloni, and S. Zanero. Studying Bluetooth malware propagation: The Bluebag project. *IEEE Security & Privacy*, 5(2), 2007.

[22] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, 1982.

[23] Compaq, Digital Equipment Corporation, IBM PC Company, Intel, Microsoft, NEC, and Northern Telecom. Universal Serial Bus Specification: Revision 1.0. Technical report, Jan. 1996.

[24] K. Cook. Linux kernel aslr (kaslr). *Linux Security Summit*, 69, 2013.

[25] J. Corbet. Extending extended BPF. *Linux Weekly News*, 2014.

[26] E. Cree. [RFC/PoC PATCH bpf-next 00/12] bounded loops for eBPF. https://www.mail-archive.com/netdev@vger.kernel.org/msg218182.html, Feb. 2018.

[27] W. Drewry. [RFC,PATCH 0/2] dynamic seccomp policies (using BPF filters). https://www.spinics.net/lists/linux-security-module/msg12441.html, Jan. 2012. Kernel Patch.

[28] ETSI Technical Committee Smart Card Platform (SCP). ETSI TS 102 622 V10.2.0: Smart Cards; UICC-Contactless Front-end (CLF) Interface; Host Controller Interface (HCI) (Release 10). Technical report, Mar. 2011.

[29] C. Feng. bpf: security: New file mode and LSM hooks for eBPF object permission control. https://lwn.net/Articles/737402/, 2017.

[30] GoodFET. Facedancer21. http://goodfet.sourceforge.net/hardware/facedancer21, 2018.

[31] Google, Inc. syzkaller - kernel fuzzer: Found Linux kernel USB bugs. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs_usb.md, 2018.

[32] J. Greene. Intel® Trusted Execution Technology. Technical report, Intel Corporation, 2012.

[33] B. Gregg. Linux Extended BPF (eBPF) Tracing Tools. http://www.brendangregg.com/ebpf.html, 2016.

[34] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, Dec. 2007.

[35] Y. Hayakawa. eBPF Implementation for FreeBSD. https://www.bsdcan.org/2018/schedule/track/Hacking/963.en.html, 2018.

[36] T. Herbert and A. Starovoitov. eXpress Data Path (XDP). https://github.com/iovisor/bpf-docs/blob/master/Express_Data_Path.pdf, 2016.

[37] M. Herfurt. Bluetooth Security. What the Hack Conference, 2005.

[38] M. Herfurt and C. Mulliner. Blueprinting: Remote Device Identification based on Bluetooth Fingerprinting Techniques. In *21st Chaos Communication Congress (21C3)*, Dec. 2004.

[39] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. B. Butler. FirmUSB: Vetting USB device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security (CCS'17)*, 2017.

[40] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. ASM: A Programmable Interface for Extending Android Security. In *Proceedings of the USENIX Security Symposium*, 2014.

[41] Intel Corporation. Intel® Virtualization Technology for Directed I/O: Architecture Specification. Technical report, June 2018.

[42] IO Visor Project. BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc, 2015.

[43] P. C. Johnson, S. Bratus, and S. W. Smith. Protecting Against Malicious Bits On the Wire: Automatically Generating a USB Protocol Parser for a Production Kernel. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, 2017.

[44] C. Kellner. Introducing bolt: Thunderbolt 3 security levels for GNU/Linux. https://christian.kellner.me/2017/12/14/introducing-bolt-thunderbolt-3-security-levels-for-gnulinux/, 2017.

[45] J. Kicinski and N. Viljoen. eBPF Hardware Offload to SmartNICs: cls bpf and XDP. *Proceedings of netdev*, 1, 2016.

[46] A. Kobayashi. Displayport (tm) ver. 1.2 overview.

[47] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[48] M. Krasnyansky and M. Holtmann. BlueZ: Official Linux Bluetooth protocol stack. http://www.bluez.org/, 2002.

[49] M. Krasnyansky and M. Holtmann. l2ping.c. https://github.com/pauloborges/bluez/blob/master/tools/l2ping.c, 2002.

[50] E. Kustarz, S. Shepler, and A. Wilson. The New and Improved FileBench File System Benchmarking Framework. In *Proceedings of the USENIX Conference and File and Storage Technologies (FAST)*, 2008. WiP.

[51] B. Lau, Y. Jang, C. Song, T. Wang, P. Chung, and P. Royal. Mactans: Injecting Malware into iOS Devices via Malicious Chargers. *Proceedings of the Black Hat USA Briefings, Las Vegas, NV, August 2013*, 2013.

[52] A. Laurie, M. Holtmann, and M. Herfurt. Hacking Bluetooth enabled mobile phones and beyond - Full Disclosure. BlackHat Europe, 2005.

[53] A. Laurie, M. Holtmann, and M. Herfurt. Bluetooth Hacking: The State of the Art. BlackHat Europe, 2006.

[54] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 93, 1993.

[55] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 1996.

[56] C. Miller. Exploring the nfc attack surface. *Proceedings of Blackhat*, 2012.

[57] NCCGROUP. Umap2. https://github.com/nccgroup/umap2, 2018.

[58] Near Field Communication Forum, Inc. NFC Digital Protocol: Digital 1.0. Technical report, Nov. 2010.

[59] Near Field Communication Forum, Inc. NFC Controller Interface (NCI) Specification: NCI 1.0. Technical report, Nov. 2012.

[60] Near Field Communication Forum, Inc. Core Protocol Technical Specifications. https://nfc-forum.org/our-work/specifications-and-application-documents/specifications/protocol-technical-specifications/, 2018.

[61] M. Newlin. MouseJack: Injecting Keystrokes into Wireless Mice. Technical report, Bastille Threat Research Team, 2016.

[62] K. Nohl, S. Krißler, and J. Lell. BadUSB - On accessories that turn evil. BlackHat, 2014.

[63] T. Remple and A. Burns. Battery Charging Specification: Revision 1.2. Technical report, Dec. 2010.

[64] E. Reshetova, F. Bonazzi, and N. Asokan. Randomization Can't Stop BPF JIT Spray. In *Proceedings of the International Conference on Network and System Security (NSS)*, 2017.

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

[66] M. Salaün. File access-control per container with landlock, 2018.

[67] J. Schulist, D. Borkmann, and A. Starovoitov. Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt, 2018.

[68] B. Seri. Bluetooth: Properly check L2CAP config option output buffer length. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e860d2c904d1a9f38a24eb44c9f34b8f915a6ea3, Sept. 2017. Kernel Patch.

[69] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux Security Module. Technical report, Dec. 2001. NAI Labs Report 01-043.

[70] A. Staravoitov. [RFC,net-next,08/14] bpf: add eBPF verifier. https://lore.kernel.org/patchwork/patch/477364/, June 2014. Kernel Patch.

[71] A. Starovoitov. BPF in LLVM and kernel. Linux Plumbers Conference, 2015.

[72] A. Starovoitov. [PATCH v2 bpf] bpf: prevent out-of-bounds speculation. https://lwn.net/Articles/743288/, Jan. 2018.

[73] A. Starovoitov, D. Borkmann, and D. S. Miller. [Patch RFC 0/4] net: add bpfilter. https://www.mail-archive.com/netfilter-devel@vger.kernel.org/msg11127.html, Feb. 2018.

[74] P. Stewin and I. Bystrov. Understanding DMA Malware. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2012.

[75] SyncStop. The Original USB Condom. https://shop.syncstop.com/products/usb-condom?variant=35430087052, 2018.

[76] 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.

[77] D. J. Tian, A. Bates, K. R. B. Butler, and R. Rangaswami. ProvUSB: Block-level provenance-based data protection for USB storage devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, 2016.

[78] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, C. Ruales, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, M. Grace, and K. R. B. Butler. ATtention spanned: Comprehensive vulnerability analysis of AT commands within the Android ecosystem. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[79] D. J. Tian, N. Scaife, A. Bates, K. R. B. Butler, and P. Traynor. Making USB Great Again with USBFILTER. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[80] D. J. Tian, N. Scaife, D. Kumar, M. Bailey, A. Bates, and K. R. B. Butler. SoK: "Plug & Pray" Today - Understanding USB Insecurity in Versions 1 through C. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[81] trifinite.group. trifinite. https://trifinite.org/, 2004.

[82] 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 ACM European Conference on Computer Systems (EuroSys'15)*, 2015.

[83] R. Verdult and F. Kooman. Practical Attacks on NFC Enabled Cell Phones. In *Proceedings of the 3rd International Workshop on Near Field Communication (NFC)*, 2011.

[84] C. Welch. Apple's USB Restricted Mode: how to use your iPhone's latest security feature. https://www.theverge.com/2018/7/10/17550316/apple-iphone-usb-restricted-mode-how-to-use-security, July 2018.

[85] Windows Dev Center. Windows Filtering Platform. https://docs.microsoft.com/en-us/windows/desktop/fwp/windows-filtering-platform-start-page, 2018.

[86] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

A. Frontend Grammar

⟨*expr*⟩ ::= ⟨*logical-or*⟩

⟨*logical-or*⟩ ::= ⟨*logical-and*⟩ ('||' ⟨*logical-and*⟩)*

⟨*logical-and*⟩ ::= ⟨*comparison*⟩ ('&&' ⟨*comparison*⟩)*

⟨*comparison*⟩ ::= ⟨*atom*⟩ (⟨*comparison-op*⟩ ⟨*atom*⟩)*

⟨*comparison-op*⟩ ::= '<' | '>' | '<=' | '>=' | '==' | '!='

⟨*access*⟩ ::= '[' ⟨*number*⟩ ':' ⟨*number*⟩ ']'

⟨*attribute*⟩ ::= '.' ⟨*IDENTIFIER*⟩

⟨*struct*⟩ ::= ⟨*IDENTIFIER*⟩ ⟨*attribute*⟩* ⟨*access*⟩?

⟨*number*⟩ ::= ⟨*DEC_NUMBER*⟩ | ⟨*HEX_NUMBER*⟩

⟨*string*⟩ ::= ⟨*STRING*⟩

⟨*atom*⟩ ::= ⟨*number*⟩ | '−' ⟨*number*⟩
  | ⟨*struct*⟩
  | ⟨*string*⟩
  | '(' ⟨*expr*⟩ ')'

⟨*DEC_NUMBER*⟩ ::= ⟨*DIGIT*⟩+

⟨*HEX_NUMBER*⟩ ::= '0x' ⟨*HEXDIGIT*⟩+

⟨*LETTER*⟩ ::= 'a' ... 'z' | 'A' ... 'Z'

⟨*STRING*⟩ ::= '"' ('\"' | /[^'"']/)* '"'

⟨*DIGIT*⟩ ::= '0'...'9'

⟨*HEXDIGIT*⟩ ::= 'a' ... 'f' | 'A' ... 'F' | ⟨*DIGIT*⟩

⟨*IDENTIFIER*⟩ ::= ('_' | ⟨*LETTER*⟩) ('_' | ⟨*LETTER*⟩ | ⟨*DIGIT*⟩)*

Figure 11: The Extended Backus-Naur Form (EBNF) of our constructed LBM expression grammar.

B. Compiler Example

LBM Program

```
usb.idVendor == 0x413c && usb.idProduct == 0x3010
```

Intermediate Representation

```
0: t1 := call(lbm_usb_get_idVendor)
1: t0 := binop(EQ, t1, 16700)
2: t3 := call(lbm_usb_get_idProduct)
3: t2 := binop(EQ, t3, 12304)
4: t4 := binop(AND, t0, t2)
```

eBPF Assembly

```
LSTART:
            MOV64_REG(REG_9, REG_1)
            MOV64_REG(REG_1, REG_9)
            CALL_FUNC(FUNC_lbm_usb_get_idVendor)
            MOV64_REG(REG_1, REG_0)
            MOV64_IMM(REG_6, 1)
            JMP_IMM(JEQ, REG_1, 16700, L1_)
            MOV64_IMM(REG_6, 0)
L1_:
            MOV64_REG(REG_1, REG_9)
            CALL_FUNC(FUNC_lbm_usb_get_idProduct)
            MOV64_REG(REG_2, REG_0)
            MOV64_IMM(REG_3, 1)
            JMP_IMM(JEQ, REG_2, 12304, L2_)
            MOV64_IMM(REG_3, 0)
L2_:
            JMP_IMM(JEQ, REG_6, 0, L3_)
            JMP_IMM(JEQ, REG_3, 0, L3_)
            MOV64_IMM(REG_4, 1)
            JMP_A(L4_)
L3_:        MOV64_IMM(REG_4, 0)
L4_:
            JMP_IMM(JNE, REG_4, 0, L5_)
L6_:        MOV64_IMM(REG_0, 0)
            EXIT_INSN()
L5_:        MOV64_IMM(REG_0, 1)
LEND:       EXIT_INSN()
```

Figure 12: The compilation stages of an LBM expression.

C. lmbench

Table VII presents the complete summary of `lmbench` results from Section V-D We use `lmbench` to benchmark the whole system across different kernel configurations and demonstrate that LBM does indeed introduce minimal overhead across the whole system.

	Processor & Processes (ns)									
	Null call	Null I/O	Stat	Open/Close	Select TCP	Signal install	Signal Handle	Fork	Execute	Exec. Shell
Vanilla	0.23	0.32	0.65	1.39	6.26	0.27	0.81	151.	497.	1425
LBM	0.22	0.32	0.66	1.38	5.65	0.27	0.80	141.	400.	1411
LBM-JIT	0.22	0.32	0.66	1.38	5.65	0.27	0.80	92.6	415.	1446

	Basic integer operations (ns)			
	bit	add	div	mod
Vanilla	0.2800	0.1400	6.1100	6.5700
LBM	0.2800	0.1400	6.0200	6.4900
LBM-JIT	0.2800	0.1400	6.0300	6.5300

	Basic uint64 operations (ns)		
	bit	div	mod
Vanilla	0.280	12.0	11.7
LBM	0.280	12.1	11.7
LBM-JIT	0.280	12.1	11.7

	Basic float operations (ns)			
	add	mul	div	bogo
Vanilla	0.8400	1.3900	3.7800	1.9500
LBM	0.8400	1.3900	3.6800	1.9500
LBM-JIT	0.8400	1.3900	3.6800	1.9600

	Basic Double Operations (ns)			
	add	mul	div	bogo
Vanilla	0.8400	1.3900	5.6200	3.9000
LBM	0.8400	1.3900	5.6300	3.9000
LBM-JIT	0.8400	1.3900	5.6500	3.9100

	Context Switching (ns)						
	2p/0K	2p/16K	2p/64K	8p/16K	8p/64K	16p/16K	16p/64K
Vanilla	1.7300	1.6600	2.4000	4.2000	5.0700	4.24000	5.79000
LBM	1.6500	1.5800	2.1900	3.3800	4.9100	4.11000	7.77000
LBM-JIT	1.6100	1.5000	2.2600	3.2200	7.5000	3.28000	7.55000

	Local Communication Latencies (us)					
	2p/0K context switch	Pipe	AF UNIX	UDP	TCP	TCP/connection
Vanilla	1.730	5.028	6.97	9.127	11.5	17.
LBM	1.650	4.998	6.31	8.973	11.3	17.
LBM-JIT	1.610	5.068	7.27	8.966	11.4	17.

	File & VM system latencies (us)							
	0K File Cre.	0K File Del.	10K File Cre.	10K File Del.	Mmap Latency	Prot. Fault	Page Fault	100 FD Select
Vanilla	5.7323	3.8630	13.3	6.8787	6493.0	0.501	0.22380	1.609
LBM	5.7247	3.8566	13.2	7.0278	6518.0	0.502	0.22080	1.602
LBM-JIT	5.7531	3.8511	13.7	6.8543	6523.0	0.500	0.22310	1.613

	Local Communication bandwidths (MB/s), Larger is better								
	Pipe	AF UNIX	TCP	File Reread	Mmap Reread	Bcopy (libc)	Bcopy (custom)	Memory Read	Memory Write
Vanilla	5597	12.K	7539	7455.9	15.0K	8126.0	5886.8	14.K	8528.
LBM	5606	12.K	7365	7473.6	15.0K	8193.2	5911.6	14.K	8535.
LBM-JIT	5686	12.K	7466	7494.9	15.0K	8169.2	5909.9	14.K	8542.

	Memory latencies (ns)				
	Mhz	L1 Cache	L2 Cache	Main memory	Random memory
Vanilla	3192	1.1140	3.3420	15.2	84.1
LBM	3192	1.1140	3.3420	14.6	84.9
LBM-JIT	3192	1.1140	3.3430	15.2	83.9

Table VII: `lmbench` results for a Vanilla kernel, LBM, and LBM-JIT.