A file system for safely interacting with untrusted USB flash drives

Ke Zhong, Zhihao Jiang*, Ke Ma*, and Sebastian Angel
University of Pennsylvania **Shanghai Jiao Tong University*

Abstract

This paper introduces RBFuse, a system for interacting with USB flash drives safely in commodity OSes while bypassing the complex and bug-prone USB stack on the host. RBFuse prevents attacks in which malicious USB flash drives exploit low-level USB driver vulnerabilities to compromise the host machine. The simple idea behind RBFuse is to remap the USB stack to a virtual machine and export the flash drive's file system as a mountable virtual file system. The result of this decomposition is that the host can access all the files in the flash drive without speaking USB. This is beneficial from a security standpoint, since the VFS interface is small, has well-defined semantics, and can be formally verified. RBFuse requires no hardware modifications and introduces low overhead.

1 Introduction

When users connect their peripheral devices (keyboards, flash drives, chargers) into their machine, they expect them to work right away. This convenience is built deep into the fabric of the Universal Serial Bus (USB), which is the de facto protocol (and nowadays also connector type) for peripheral devices. USB has for decades assumed, even if tacitly, that any device physically attached to a user's machine has been vetted by the user and is therefore trustworthy. Consequently, USB stacks in existing OSes lack authentication mechanisms, blindly trust device payloads, and grant devices direct memory access (DMA). Such state of affairs has led to a myriad of attacks in which maliciously crafted devices abuse this trust to compromise low-level drivers [62], masquerade as other devices [21–23], and manipulate kernel memory via DMA [8, 55, 63, 65]; the academic community has followed with proposals to address many of these shortcomings [28, 31, 47, 49, 54, 66, 68–70].

While there is reason to be optimistic that newer versions of USB (connectors and protocol) will take security more seriously [24], safely interacting with the existing *tens of billions* of devices that lack the proposed capabilities remains challenging. A particular concern are flash drives, which, even in today's well-connected and cloud-powered world, remain widely popular for storing and sharing data (e.g., bitcoin keys, medical records, old photos, music). To support safe interactions with untrusted flash drives that may conduct the aforementioned attacks, we propose a lasting remedy: *instead of searching for (and patching) vulnerabilities across the complex USB stack, we arrange for the host to speak to the flash drive exclusively over a virtual file system (VFS) interface, avoiding the host's USB stack in its entirety*. This approach has the benefit that VFS designs have simple and narrow interfaces, can be tested using ideas borrowed from existing fuzzing [48, 71] and crash frameworks [57], and actually stand a chance of being formally verified to ensure the absence of memory errors and the presence of crash consistency [33, 34]. The key challenge is achieving the proposed architecture without hardware changes and with low overhead.

To tackle the above challenge we introduce *RBFuse*, which is the first system that efficiently allows interactions between the host and USB flash drives through a VFS interface, bypassing the USB stack on the host machine without the need for new hardware. To do so, RBFuse adapts and specializes the architecture proposed in Cinch [31] and Qubes OS [19]. Specifically, RBFuse sets up a virtual machine (VM), remaps the USB host controller to the VM's address space, mounts the flash drive, and mediates access to it via a VFS interface. A Fuse-based file system at the host proxies all operations to the VFS via RPCs. From the perspective of the host, when a flash drive is attached to the machine it appears as a new directory in the VFS that can be accessed by the kernel and all applications. From the perspective of the device, it talks USB to the VM; attacks are contained within the VM and only propagate to the kernel via the (hardened) VFS interface.

As one may expect, a straightforward implementation of RBFuse's architecture leads to unreasonably low performance (up to 310× higher completion time compared to the baseline of directly connecting the flash drive via the host's USB stack). To reduce costs, we adapt optimizations that have been proposed for networked file systems to aggressively cache metadata, batch operations, and prefetch files. The resulting implementation achieves comparable performance to the baseline on workloads that operate on a large single file, and less than 4× higher completion time on workloads that operate on thousands of small files. Additionally, we use formally verified serializers and parsers [58] to reduce the attack surface of the RPC layer used by the VFS and the proxy. Finally, inspired by Cinch [31] and SandUSB [54], RBFuse can leverage an optional hardware adapter that encrypts files and metadata between the flash drive and the Fuse-based proxy (kernel) with low overhead, which prevents malicious USB hubs or promiscuous devices from inspecting the content of files transferred.

While RBFuse is presently tailored to flash drives, we intend to accommodate other file-based devices such as USB SD card readers and USB SSDs; this last one is particularly challenging due to high USB 3 speeds. In the near future, we plan to borrow ideas from formally verified file systems [33, 34] to prove the correctness of our optimizations (particularly with respect to crash consistency) and to further reduce the attack surface. A major limitation of RBFuse is supporting multiple USB

devices: all devices attached to the same host controller are remapped to the same VM, so a compromised VM can harm other devices or take over for them. While desktops and newer laptops have multiple host controllers that could allow users to separate devices into untrusted (e.g., newly found flash drive) and trusted (e.g., keyboard owned for years) groups, this remains a major pain point.

2 Background and related work

The Universal Serial Bus (USB) is the de facto stack for interacting with peripheral devices such as flash drives. On the hardware side, it consists of the devices themselves, hubs like monitors and adapters (e.g., Apple "dongles"), and the host controller, which sits on the motherboard and is typically connected via PCIe. On the software side, USB consists of the host controller interface (HCI), which manages the host controller; USB core drivers that handle device enumeration and power management; and USB class drivers that manage devices (e.g., mass storage drivers manage flash drives).

While threats such as USB devices carrying malware have been around for decades (e.g., Stuxnet [42]), recent years have seen an influx of devices that are themselves malicious [3–5, 7, 13, 17, 18, 43, 56]. The difference between these two threat models is that in the former (malware) the device itself is entirely benign and follows the prescribed protocol. In the latter, the device itself exploits the USB protocol. For example, a flash drive could lie during the device enumeration process, pretend to be a keyboard, and issue arbitrary key presses [18]. As another example, the USB device could issue malformed packets to exploit vulnerabilities in USB's core or class drivers (which run inside the kernel), thereby gaining control over the machine with root privileges [2, 9–12, 15, 32, 38, 39, 41, 62].

2.1 Existing efforts

To prevent the above attacks, existing efforts for commodity OSes fall into three categories: (1) filtering USB packets; (2) adding authentication to the USB protocol; (3) sandboxing the device and monitoring its behavior. We discuss each of these.

Packet filtering. USBFilter [70] and Cinch [31] work by intercepting USB packets and allowing the user to specify rules to block devices or payloads (similar to specifying rules for a network firewall). USBFilter operates within the kernel similar to iptables. Cinch forces the USB device to connect to a VM, converts the USB packets into IP packets, uses existing network middleboxes to process the IP packets, and re-injects the "cleaned" USB packets into the host's USB subsystem.

A limitation of filtering is that devices with polymorphic attacks (i.e., attacks that change payloads) are hard to prevent with rules, so malicious devices could still exploit driver vulnerabilities. This is a key difference between RBFuse and these efforts: while Cinch goes from USB to IP, and then back to USB (which means that the host still has a full USB stack that can be exploited by polymorphic attacks), RBFuse goes from USB directly to VFS RPCs.

Device authentication. Several efforts [28, 49, 66, 69] extend USB devices and drivers to allow the device to cryptographically authenticate to the host (or vice versa). This allows the devices of trusted manufacturers to continue to operate, while preventing risky devices from being used. Similar efforts are being discussed for devices that use the USB-C connector [24]. A downside of this approach is that it prevents third-party manufacturers and custom hardware, and raises concerns that a small set of companies can dictate what devices people can use. These works differ from RBFuse in that they require new devices and protocols, whereas RBFuse's objective is to support existing devices—albeit only flash drives. In this light, RBFuse is complementary: should the above approaches be widely deployed, RBFuse can allow users to plug in old USB flash drives that lack the necessary extensions.

Sandbox the device. Some works [47, 54, 68] redirect devices to a sandbox rather than connecting them directly to the host. The sandbox can be a separate device or a virtual machine; while in the sandbox, the user (or an anomaly detection algorithm) can monitor the behavior of the device. If the device is deemed safe, it is then connected to the host machine. A challenge with these works is that it is hard for users to determine whether a device is "safe"; further, malicious devices could detect that they are running in a sandbox and change their behavior accordingly (VW has taught us a lesson here [1]). Unlike RBFuse, these works reconnect the device into the USB subsystem.

3 Design

We approach the problem of safely supporting flash drives from a pragmatic perspective: since fixing all bugs in the USB stack is difficult and formal verification requires a clear and narrow interface with easy to specify semantics (which is not the case in USB since its specification contains a lot of ambiguity [31]), we focus on avoiding USB at the host altogether. In the following sections, we describe (1) our high level architecture; (2) our concrete threat model; (3) how to remap the USB's host controller away from the kernel and onto a VM; (4) how to export flash drives from this VM to the host via a mountable virtual file system (VFS); (5) how to notify the host that a new flash drive has been plugged in; (6) an optional hardware adapter that encrypts file contents and metadata; and (7) how to make this architecture efficient.

3.1 RBFuse's architecture

RBFuse's architecture, depicted in Figure 1, is inspired by Rushby's separation kernels [59, 60], Lampson's idea of freedom and accountability (Red/Green) [52], and the recent instantiation of these notions in Cinch's Red and Blue machines [31]. At a high level, the system is divided into two components: a "Red" machine that is untrusted (free to do what it wants), and a "Blue" machine that constitutes the trusted computing base (accountable). In RBFuse, the Red machine is a VM that runs one or more VFS servers and a

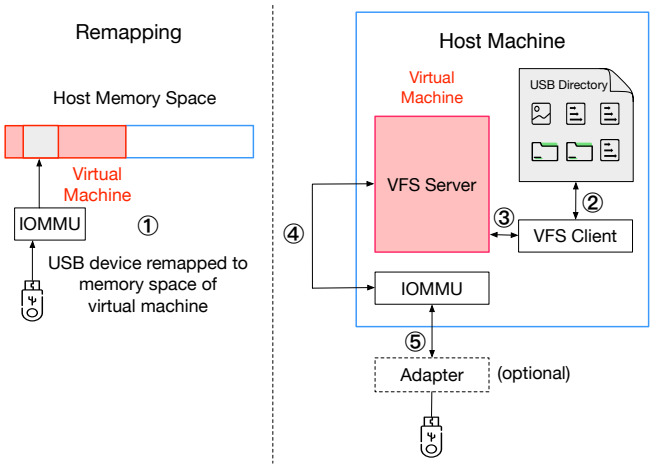

FIGURE 1—RBFuse's high-level architecture. RBFuse remaps the host controller to a deprivileged process, which exports the device to the host via a networked file system. This interface is the only communication between the device and the host. An optional adapter can be added to further provide authentication and file-level encryption.

USB stack (HCI, USB core, and mass storage drivers); the Blue machine consists of the host's kernel, other processes, and one or more VFS client instances. Following Rushby's methodology, the interactions between these two machines should be done through a narrow and simple interface that, in an ideal world, stands a chance of being formally specified.

3.2 Threat model

RBFuse's goal is to prevent the following types of attacks.

Attack on the host's USB stack. RBFuse should eliminate attacks on the host's USB stack and prevent DMA attacks on the host. Note that RBFuse does not prevent a malicious USB device from attacking the VM's USB stack. As a result, we assume that the attacker can corrupt the VM at any point, and can use it to issue malformed requests to the host via any available communication channels. We assume that virtualization software is not vulnerable to VM escapes [50], and that the machine's IOMMU prevents unauthorized memory accesses.

Attack on the VFS client. RBFuse should prevent attacks on the VFS client stack that runs inside the host's kernel. Since the VFS server runs on the untrusted VM, the adversary is free to issue malformed responses to exploit vulnerabilities (e.g., buffer and integer overflows) in the VFS client implementation. Note that attacks caused by the *content* of an otherwise correctly behaving flash drive (e.g., malware) are out of scope.

Limitations. RBFuse cannot prevent a malicious USB device from harming other USB devices connected to the same host controller. For example, a malicious USB can compromise the VM and then use it to misconfigure, snoop on, or impersonate other connected USB devices (our adapter in Section 3.6 prevents snooping and spoofing). This limitation is shared by other systems with similar architectures [16, 31]. However, most desktops and newer laptops include multiple USB host controllers. It is therefore possible to plug "trusted" USB devices to one host controller, and dedicate another to RBFuse. Note that RBFuse (and the VM) need only be active when the user wishes to plug in an untrusted USB flash drive.

3.3 Remapping USB devices

RBFuse leverages the machine's IOMMU (e.g., Intel VT-d) to remap the host controller to the address space of the Red machine (Step ① in Figure 1). This operation prevents the host controller (which has DMA capabilities) from writing anywhere else in memory. It also binds the host controller to the HCI driver running on the Red machine; this machine is now in charge of managing the host controller and any connected device. In our implementation we use a VM running Linux as the Red machine since it already has the necessary USB stack and can run our VFS servers.

3.4 Interactions between VFS server and client

In RBFuse, all file system operations such like `open`, `read`, `write` on the USB directory are captured by the running VFS client (step ② in Figure 1). The VFS client converts these operations into RPCs to the VFS server and supplies the required parameters (step ③). The VFS server parses the request and executes it on the USB device (step ④). After the execution, the VFS server sends back the response to the VFS client, specifying whether the operation succeeded, any error number, and other messages like attributes (for `getattr` requests) and file contents (for `reads`). RBFuse also has an optional step ⑤ in which an adapter (§3.6) provides encryption and authentication capabilities.

Since the entire USB stack is isolated in the Red machine, the only communication between the host and the USB device is via RPCs. However, if this RPC interface is not designed carefully, a compromised Red machine could exploit memory vulnerabilities on the VFS client implementation. To ensure that the RPC layer is free of memory bugs, we leverage a recent framework [58] to automatically generate formally verified code for serializing and parsing all VFS RPCs. While this is a far cry from formally verifying all of RBFuse (as we explain in Section 6), it is a promising start.

3.5 Bootstrapping device connections

A key aspect of our design is figuring out when to spawn VFS clients and servers. One possibility is to spawn as many client-server pairs as there are USB ports in the machine. However, USB hubs can dynamically change the port count. To address this, we rely on a monitoring program (in addition to VFS) that runs on both the Red and Blue machines. On the Red machine, the monitoring program detects connect/disconnect events and responds as follows: on a connect event, it spawns a VFS server on an unused port, and sends the port number to the Blue machine (it shuts down the server on a disconnect event). On the Blue machine, the monitoring program receives

a port number, spawns a VFS client that connects to it.

We note that if the Red machine is compromised, this additional channel allows an attacker to spawn an indefinite number of VFS clients at the Blue machine. We prevent this with a limit on the number of allowed concurrent flash drives.

3.6 Data encryption

An optional adapter can be used for data encryption and authentication, thereby preventing a compromised Red machine (or a malicious USB hub) from stealing or modifying file content or metadata. When using the adapter, instead of running the VFS server on the Red machine, we run it on the adapter itself; the Red machine merely forwards requests back and forth. This is possible because, as demonstrated by Cinch [31], the adapter can pretend to be a USB network card to the Red machine and can send Ethernet frames directly.

The adapter performs the following operations: (1) it encrypts file content and metadata with an authenticated encryption scheme to ensure confidentiality and integrity; and (2) it changes the key every time the device is plugged in to ensure *forward secrecy* (i.e., if a key is compromised it cannot be used to recover the content of data exchanged prior to the compromise). For this to work, we assume that the adapter knows the host's public verification key and the host knows the adapter's (there are many ways to accomplish this, including using QR codes). This assumption is crucial, as otherwise a compromised Red machine can act as a man-in-the-middle.

Our protocol is based on *mutual authentication TLS* (mTLS); in standard TLS, the application (e.g., the browser) authenticates the Web server, but not the other way around since the Web server does not care who is connecting to it. Mutual authentication prevents a compromised Red machine from pretending to be the flash drive to the host, or pretending to be the host to the adapter.

Protocol. When a flash drive is plugged in and the monitor spawns a new VFS client (§3.5), the VFS client sends a new Diffie-Hellman [40] key share and a random string, both signed with the host's private signing key, to the VFS server (running in the adapter). The adapter verifies the signature using the host's public verification key and responds with its own key share and random string, signed with the adapter's private signing key. Finally, the host verifies the adapter's message. Both the host and the adapter can derive a shared symmetric key by first deriving a shared secret from their key shares (standard Diffie-Hellman), and then deriving the symmetric secret key using a standard key derivation function [51] with the shared secret and both of the random strings. This essentially mirrors the TLS 1.3 handshake [26].

Once the adapter and the host agree on the same symmetric key *sk*, they can encrypt all operations with an authenticated encryption scheme, ensuring that the Red machine learns only traffic patterns (file sizes, access frequency, etc.). While this leakage could be prevented by adapting existing techniques [30, 36], it is likely not warranted in this case.

4 Reducing costs

As one would expect, a straightforward implementation of a VFS that proxies all operations through the Red machine on their way to the flash drive brings high overheads. This is specially concerning since every file system operation, including traditionally light-weight ones such as `mknod` and `close`, now take a long time to be processed. Even worse, when we build the VFS client on top of Fuse (as opposed to running it as a native kernel module), we encounter a few implementation issues. First, Fuse makes an excessive number of calls to functions like `getattr` to read file and directory metadata. As an example, when reading 1,000 different files from a USB flash drive, Fuse issues roughly 3,000 `getattr` calls. Second, the maximum size of read and write payloads in Fuse is 128 KB, causing operations with data larger than 128 KB to be split into multiple read/write RPCs in our VFS. This introduces significant latency, since RPCs are synchronous.

The above suggests that making RBFuse practical requires drastically reducing the number of unnecessary RPCs. A similar observation has guided decades of optimization in networked file systems (NFS), though our setting is different and allows certain actions that would not be appropriate in traditional NFS deployments. We nevertheless adapt many of these techniques, including caching metadata [20, 37, 45, 53, 72, 73], batching operations [35, 64], and prefetching files [44, 74].

We note that we can address the data size issue in Fuse by recompiling the Fuse kernel module with a higher `FUSE_MAX_PAGES_PER_REQ` value, but this is not ideal for several reasons. First, requiring users to recompile Fuse, which often comes statically linked with their OS, is burdensome. Second, other applications or components in the system may be using Fuse, and changing this value could have unintended consequences. Last, our optimizations provide benefits beyond increasing this global value and do not require recompilation.

4.1 Caching metadata

In our setting, while many applications can concurrently access the USB's file system via RBFuse, there is only one VFS client per VFS server. This is a key difference from NFS deployments that must support multiple clients. This gives rise to a simple proposition: file and directory metadata will not change once read, unless the VFS client itself performs an operation that changes this metadata. Consequently, RBFuse's VFS client caches metadata when files are first opened, and keeps this cache up to date (i.e., RBFuse treats this as a write-trough cache). This significantly cuts down on the number of RPCs that need to be issued by RBFuse's VFS client.

Additionally, since the number of RPCs (and their synchronous nature) is the key contributor to latency overhead in our system, RBFuse goes one step further: during initialization (when the device is first plugged in), RBFuse fetches the metadata of all files and directories, and caches this information (we set a timeout to account for cases where the user plugs a

device with millions of files). This yields significant speedups in the common case. For example, if an application asks for a non-existent file, RBFuse can consult its local cache and respond to the `getattr` request without issuing any RPCs.

4.2 Prefetching

RBFuse also exploits the spatial locality of files, taking into account that a common workload in flash drives is to copy an entire directory to or from the drive, or to copy one big file. When an application reads a file, RBFuse reads other small files in the same directory (up to a configurable total size limit), and sends them back to the VFS client to be cached locally. When the application reads a large file, since Fuse issues requests at the granularity of 128 KB chunks, RBFuse's VFS client prefetches subsequent chunks. In this way, if the application requests the next chunk, the VFS client can immediately respond from its local cache, avoiding RPCs.

4.3 Batching operations

In our initial implementation, each file system operation was translated to an RPC. To reduce the number of RPCs, RBFuse batches many operations together. For `mknod`, `open`, `write`, and `close` operations, RBFuse does not immediately issue the corresponding RPCs to the VFS server. Instead, RBFuse stores these operations for each file in a pending request queue (writes get their own pending request queue). When a pending queue exceeds a threshold, the VFS client tags the corresponding requests as a `compound` request and issues them to the VFS server in a single RPC.

Batching operations has significant benefits when copying files from the host to the device. For large files, batching operations allows RBFuse to combine many writes together to mask the limit imposed by Fuse. For small files, the improvements are even more significant. Typically, writing a small file requires only one write request if the content is smaller than 128 KB. However, Fuse first calls `getattr` (to see if the file exists), `mknod`, `getattr` again (to see if the file was created), `open`, `write`, and `close` sequentially. Due to our caching optimization (§4.1), `getattr` is done locally. And with batching, the other four operations are combined into one.

In order to support batching, RBFuse must speculatively respond to Fuse with the outcome of an operation so that Fuse may issue the next operation. For `write` and `mknod`, RBFuse keeps track of the space remaining in the flash drive; if there is enough space to perform the operation, RBFuse optimistically returns "success" or the number of bytes written. When `fsync` and `umount` are issued, RBFuse flushes all requests from its pending queues. This kind of lightweight speculation is similar in flavor to the I/O buffering and disk write caching mechanisms that OSes often do. An obvious drawback is that if the drive is unplugged before `fsync` and `umount` are called, or the queues reach the threshold, data may be lost.

5 Preliminary evaluation

We now discuss preliminary results of evaluating the following prototype implementation.

Implementation. We build RBFuse's VFS client using Fuse 2.9.4 [29]. Fuse mounts the USB directory on the host, and all file system operations on the USB directory are forwarded to the VFS server via RPCs. To instantiate the Red machine we run Ubuntu 16.04 (Linux 4.15.0-45) on QEMU, and the Blue machine is also Ubuntu 16.04 with KVM. The adapter for authentication and data encryption is built on a BeagleBone Black [6], which is 32-bit single-board computer with 1 GHz ARM Cortex-A8 processor and 512 MB RAM. It runs Debian 9.1 (Linux 4.4.88-ti-r125) and performs all cryptographic operations using the OpenSSL library [14]. We evaluate RBFuse using a Kingston Digital DataTraveler SE9 Flash Drive with the FAT32 file system.

5.1 Experiments and results

In this evaluation, we are interested in answering one simple question: what is the overhead introduced by RBFuse for common operations such as reading or writing a file to the USB flash drive over a baseline that connects the flash drive directly to the machine without our framework. To put the performance results in context, we note that a straightforward implementation of RBFuse without any optimizations resulted in up to 310× higher response time for reading or writing a large number (1,000) of small files.

We explore two common use cases: reading or writing a single large file, and copying many small files (from device to host or vice versa). Considering that in RBFuse flash drives are (optionally) attached to our adapter (§3.6), we run all tests on the host alone (this is the baseline), the adapter alone (since it runs Debian, it can be thought of as another machine), RBFuse without the adapter, and RBFuse with the adapter. We use filebench [27, 67] for all of our experiments to drive the copying of files back and forth, and measure the mean read/write completion time over 10 runs. The only caveat is that filebench does not work on the adapter (when tested in a standalone way) when the file is large due to limited memory. In this case, we run `cp` and measure its completion time.

One large file. We depict the results of reading and writing a single large file (500 MB) in Figure 2. The key take away is that with our optimizations, writing a file under RBFuse takes 2% longer than the baseline without the adapter. With the adapter, RBFuse takes 3× longer to complete the same task. The majority of the overhead stems from the double copying of data (from device to adapter and from adapter to host). The results for reading a large file are much worse: RBFuse incurs a 70% increase in completion time without the adapter, and 10× with the adapter. While these results are slightly discouraging, we discuss several potential enhancements in Section 6.

Many small files. We perform a similar experiment, but this time target the copying of 1,000 small files (16 KB each).

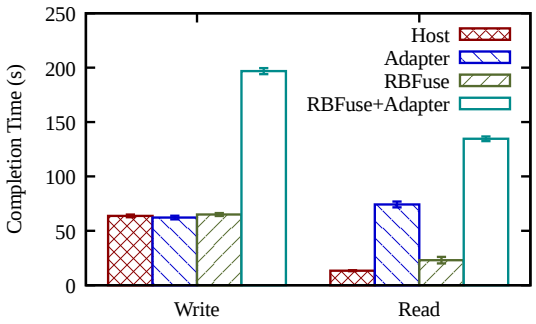

FIGURE 2—Mean response time for writing/reading one large file (500MB) to/from a USB flash drive with different mechanisms. Error bars depict one standard deviation.

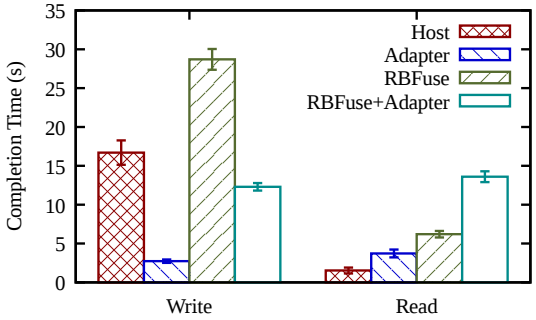

FIGURE 3—Mean response time for writing/reading 1,000 small files (16KB) to/from a USB flash drive with different mechanisms. Error bars depict one standard deviation.

The results are given in Figure 3. Unlike the previous experiment where the results were consistent with our expectations, this experiment yields some unusual results. We find that RBFuse with the adapter actually *outperforms* the baseline when copying files from the host to the device ("Writes") by 35%. Note that this is not the result of caching or prefetching, since caching metadata plays a small role here, and prefetching helps reads but not writes. Furthermore, batching does not fully explain this difference since we would otherwise had seen similar improvement when copying large files (previous experiment). Most importantly, RBFuse with the adapter also outperforms RBFuse without the adapter!

Instead, we find that this performance difference is due to the OS on which the VFS server runs. When RBFuse uses the adapter, the VFS server runs on the adapter instead of on the Red machine (the Red machine simply acts as a network proxy). Prior discussions [25] have noted that Ubuntu has high overhead when copying files to flash drives compared to Windows; we are somewhat surprised to see such a big difference between different Linux kernel versions. In our case, the adapter runs 32-bit Linux 4.4.88; on its own, it can copy 1,000 files in 2.73 seconds (6× faster than the 64-bit Linux 4.15.0-45 running on the Red machine). We have also reproduced this result with RBFuse without the adapter by running different 64-bit Linux kernel versions on the Red machine; performance varies substantially between versions.

The results for reading many small files are less interesting and mirror those of the large file experiment: RBFuse without the adapter has 4× higher request completion time than the baseline; with the adapter the difference is 8.8×.

6 Discussion

This paper introduces RBFuse a framework for interacting with USB flash drives while bypassing the USB stack. The driving philosophy of this work is that getting the virtual file system interface "right" is much easier than getting the USB stack right. We think this view is justified since there are numerous frameworks for fuzzing different aspects of file systems [48, 57, 71] and for reasoning formally about their operations [33, 34, 46]. Indeed, we take a tiny step in the latter direction by using formally verified serializers and parsers for our RPC layer (§3.4). Meanwhile, we are not aware of any efforts of equivalent scope in the context of the USB stack (although driver synthesis [61] is a promising direction).

Crash consistency. USB flash drives are notorious for being unplugged at arbitrary times. It is therefore important that RBFuse achieves crash consistency. We have adapted CrashMonkey [57] to support the vfat file system and used it to fuzz RBFuse and our flash drives. A preliminary finding is that the flash drive's vfat file system loses empty files after a crash; we have found no issues with RBFuse or its optimizations.

Possible extensions. In this work we have focused exclusively on USB flash drives. However, other storage devices such as USB SD cards and SSDs could likely be supported by this same architecture. One challenge with SuperSpeed USB SSDs is that they achieve order-of-magnitude higher throughput than traditional USB 2.0 flash drives by leveraging the host controller's ability to perform DMA. However, RBFuse limits the host controller to writing only in the Red machine's address space, and forces all data between the host and the Red machine to be exchanged via message passing. It is unclear how to retain the isolation benefits of our approach while leveraging the performance benefits of DMA—especially when the DMA device could be malicious.

A more immediate question is whether we can reduce the number of memory copies. We posit that since everything is running on the same machine (ignoring the adapter for a moment), it might be possible to maintain the isolation provided by NFS, but with zero-copy data movement between the VFS server and client.

Acknowledgments

We thank Riad Wahby and Michael Walfish for discussions that inspired this work. We also thank the anonymous HotStorage reviewers, and in particular our shepherd Malte Schwarzkopf, for their thorough comments and helpful advice that significantly improved our content and presentation.

References

[1] https://en.wikipedia.org/wiki/Volkswagen_emissions_scandal.

[2] AnywhereUSB/5 integer overflow. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4459.

[3] BadUSB—now with do-it-yourself instructions. https://nakedsecurity.sophos.com/2014/10/06/badusb-now-with-do-it-yourself-instructions/.

[4] BadUSB: Big, bad USB security problems ahead. http://www.zdnet.com/article/badusb-big-bad-usb-security-problems-ahead/.

[5] BadUSB: what you can do about undetectable malware on a flash drive. http://www.pcworld.com/article/2840905/badusb-what-you-can-do-about-undetectable-malware-on-a-flash-drive.html.

[6] BeagleBone Black. http://beagleboard.org/BLACK.

[7] Hubs—BadUSB exposure. https://opensource.srlabs.de/projects/badusb/wiki/Hubs.

[8] Inception. https://github.com/carmaa/inception.

[9] Linux audio driver dereferences null pointer under invalid device. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2184.

[10] Linux serial driver dereferences null pointer under device with no bulk-in or interrupt-in endpoints. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2782.

[11] Linux hid-picolcd_core.c buffer overflow. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3186.

[12] Linux report_fixup HID functions out-of-bounds write. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3184.

[13] Only half of USB devices have an unpatchable flaw, but no one knows which half. http://www.wired.com/2014/11/badusb-only-affects-half-of-usbs/.

[14] OpenSSL. https://www.openssl.org.

[15] OS X USB hub descriptor memory corruption. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3723.

[16] Qubes OS project. https://www.qubes-os.org.

[17] This thumbdrive hacks computers. http://arstechnica.com/security/2014/07/this-thumbdrive-hacks-computers-badusb-exploit-makes-devices-turn-evil/.

[18] USB Rubber Ducky. http://usbrubberducky.com.

[19] Using and Managing USB devices. Qubes OS Project. https://www.qubes-os.org/doc/usb/.

[20] The vfs inode cache. http://www.science.unitn.it/~fiorella/guidelinux/tlk/node110.html, 1997.

[21] Episode 709: Usb rubber ducky part 1. https://www.hak5.org/episodes/episode-709, 2013.

[22] Usb rubber ducky payloads. https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Payloads, 2013.

[23] Usbdriveby. http://samy.pl/usbdriveby/, 2014.

[24] USB 3.0 promoter group defines authentication protocol for USB Type-C. https://usb.org/sites/default/files/article_files/USB_Type-C_Authentication_PR_FINAL.pdf, 2016.

[25] Why does my usb flash drive write performance drop by more than 50% when running ubuntu vs windows 10? https://askubuntu.com/questions/853736/why-does-my-usb-flash-drive-write-performance-drop-by-more-than-50-when-running, 2016.

[26] The transport layer security (tls) protocol version 1.3. https://tools.ietf.org/html/rfc8446, 2018.

[27] Filebench. https://github.com/filebench/filebench, 2020.

[28] Kanguru flashtrust USB 3.0 flash drive with secure firmware | kanguru solutions. https://www.kanguru.com/storage-accessories/kanguru-flashtrust-secure-firmware.shtml, 2020.

[29] Libfuse. https://github.com/libfuse/libfuse, 2020.

[30] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[31] 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 USENIX Security Symposium*, 2016.

[32] D. Barrall and D. Dewey. "Plug and Root," the USB key to the kingdom. In *Proceedings of the Black Hat USA Conference*, July 2005.

[33] H. Chen, T. Chajed, A. Konradi, S. Wang, A. undefinedleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[34] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[35] M. Chen, G. B. Bangera, D. Hildebrand, F. Jalia, G. Kuenning, H. Nelson, and E. Zadok. Vnfs: Maximizing nfs performance with compounds and vectorized i/o. *ACM Trans. Storage*, Sept. 2017.

[36] W. Chen and R. A. Popa. Metal: A metadata-hiding file-sharing system. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.

[37] M. D. Dahlin, C. J. Mather, R. Y. Wang, T. E. Anderson, and D. A. Patterson. A quantitative analysis of cache policies for scalable network file systems. *SIGMETRICS Perform. Eval. Rev.*, May 1994.

[38] A. Davis. Lessons learned from 50 bugs: Common USB driver vulnerabilities. Technical report, NCC Group, Jan. 2013.

[39] A. Davis. USB attacks need physical access right? Not any more.... In *Proceedings of the Black Hat Asia Conference*, Mar. 2014.

[40] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 1976.

[41] R. Dominguez Vega. USB attacks: Fun with Plug and 0wn. In *Proceedings of the DEF CON Hacking Conference*, Aug. 2009.

[42] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet dossier. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[43] T. Goodspeed. Facedancer21. http://goodfet.sourceforge.net/hardware/facedancer21/.

[44] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson. Informed mobile prefetching. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2012.

[45] D. Howells. Fs-cache: A network filesystem caching facility. 01 2006.

[46] A. Ileri, T. Chajed, A. Chlipala, F. Kaashoek, and N. Zeldovich. Proving confidentiality in a file system using DISKSEC. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[47] M. Kang and H. Saiedian. Usbwall: A novel security mechanism to protect against maliciously reprogrammed usb devices. *Information Security Journal: A Global Perspective*, 26(4), 2017.

[48] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[49] D. Kopeek. USB guard: USB device authorization policies. https://usbguard.github.io/, 2020.

[50] K. Kortchinsky. CLOUDBURST: A VMware guest to host escape story. In *Proceedings of the Black Hat USA Conference*, 2009.

[51] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2010.

[52] B. Lampson. Accountability and freedom. http://research.microsoft.com/en-us/um/people/blampson/slides/accountabilityandfreedomabstract.htm, 2005.

[53] E. Lim, S. Ahn, Y. Kim, G. Cha, and W. Choi. Design of cache backend using remote memory for network file system. In *2017 International Conference on High Performance Computing Simulation (HPCS)*, July 2017.

[54] E. L. Loe, H. Hsiao, T. H. Kim, S. Lee, and S. Cheng. SandUSB: An installation-free sandbox for USB peripherals. In *Proceedings of the IEEE World Forum on Internet of Things (WF-IoT)*, Dec. 2016.

[55] F. Lone Sang, V. Nicomette, and Y. Deswarte. I/O attacks in Intel PC-based architectures and countermeasures. In *Proceedings of the SysSec Workshop*, July 2011.

[56] A. Mamiit. How bad is BadUSB? security experts say there is no quick fix. http://www.techtimes.com/articles/17078/20141004/how-bad-is-badusb-security-experts-say-there-is-no-quick-fix.htm.

[57] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2018.

[58] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko. Everparse: Verified secure zero-copy parsers for authenticated message formats. In *Proceedings of the USENIX Security Symposium*, Aug. 2019.

[59] J. Rushby. The design and verification of secure systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Dec. 1981.

[60] J. M. Rushby. Proof of separability—a verification technique for a class of security kernels. In *Proceedings of the International Symposium on Programming*, Apr. 1982.

[61] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-guided device driver synthesis. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.

[62] S. Schumilo, R. Spenneberg, and H. Schwartke. Don't trust your USB! How to find bugs in USB device drivers. In *Proceedings of the Black Hat Europe Conference*, Oct. 2014.

[63] R. Sevinsky. Funderbolt: Adventures in Thunderbolt DMA attacks. In *Proceedings of the Black Hat USA Conference*, July 2013.

[64] H. Shim, B. Seo, J. Kim, and S. Maeng. An adaptive partitioning scheme for dram-based cache in solid state drives. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2010.

[65] P. Stewin. A primitive for revealing stealthy peripheral-based attacks on the computing platform's main memory. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Oct. 2013.

[66] K. Suzaki, Y. Hori, K. Kobara, and M. Mannan. Deviceveil: Robust authentication for individual usb devices using physical unclonable functions. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2019.

[67] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[68] D. Tian, A. Bates, and K. Butler. Defending against malicious USB firmware with GoodUSB. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2015.

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

[70] J. Tian, N. Scaife, A. Bates, K. R. B. Butler, and P. Traynor. Making USB great again with USBFILTER. In *Proceedings of the USENIX Security Symposium*, Aug. 2016.

[71] W. Xu, H. Moon, S. Kashyap, P. Tseng, and T. Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[72] J. Yang, J. Izraelevitz, and S. Swanson. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2019.

[73] J. Zhang, G. Wu, X. Hu, and X. Wu. A distributed cache for hadoop distributed file system in real-time cloud services. In *2012 ACM/IEEE 13th International Conference on Grid Computing*, Sep. 2012.

[74] Zijian Liu, Fang Dong, Junxue Zhang, Pengcheng Zhou, Zhuqing Xu, and Junzhou Luo. A client-side directory prefetching mechanism for glusterfs. In *2016 IEEE*

International Conference on Systems, Man, and Cybernetics (SMC), Oct 2016.