ProvUSB: Block-level Provenance-Based Data Protection for USB Storage Devices

Dave (Jing) Tian
University of Florida
daveti@ufl.edu

Adam Bates
University of Illinois at Urbana-Champaign
batesa@illinois.edu

Kevin R. B. Butler
University of Florida
butler@ufl.edu

Raju Rangaswami
Florida International University
raju@cs.fiu.edu

ABSTRACT

Defenders of enterprise networks have a critical need to quickly identify the root causes of malware and data leakage. Increasingly, USB storage devices are the media of choice for data exfiltration, malware propagation, and even cyber-warfare. We observe that a critical aspect of explaining and preventing such attacks is understanding the *provenance* of data (i.e., the lineage of data from its creation to current state) on USB devices as a means of ensuring their safe usage. Unfortunately, provenance tracking is not offered by even sophisticated modern devices. This work presents *ProvUSB*, an architecture for fine-grained provenance collection and tracking on smart USB devices. ProvUSB maintains data provenance by recording reads and writes at the block layer and reliably identifying hosts editing those blocks through attestation over the USB channel. Our evaluation finds that ProvUSB imposes a one-time 850 *ms* overhead during USB enumeration, but approaches nearly-bare-metal runtime performance (90% of throughput) on larger files during normal execution, and less than 0.1% storage overhead for provenance in real-world workloads. ProvUSB thus provides essential new techniques in the defense of computer systems and USB storage devices.

1. INTRODUCTION

When securing computer systems and data, a great deal of effort is put into ensuring that the perimeter of an organization is secure. For example, firewalls and DMZs are designed to keep malicious outsiders from gaining access to, and exfiltrating, sensitive information. However, it is substantially more difficult to ensure that information is secure from a trusted insider within an organization. In some cases, the insider may be an active adversary, as with the Manning case where classified data was exfiltrated on portable media [37]. In other cases, the insiders may be unwitting vectors for attack, such as in 2008 when CENTCOM employees inadvertently loaded malware on to SIPRnet through an insecure USB drive [34].

USB devices are ubiquitous within organizations and enterprises, but because of their potential for propagating malware and becoming a medium for data exfiltration, their use has been curtailed or altogether banned [1, 73, 57]. While enterprise-level solutions to USB device security can potentially detect malware contained within these devices [49], they require centralized authentication mechanisms in addition to hardware crypto coprocessors [32]. Most important, these solutions are *incomplete*. No signature-based malware scheme is perfect: new and previously-undisclosed *zero-day* attacks, such as those used by the Stuxnet worm [18] and the Havex trojan [55], render these defenses moot. Furthermore, there are no mechanisms to determine *how* malicious software was loaded onto the USB device, nor any means of determining where this data has been disseminated; consequently, there is no method for understanding the *spread* of malware after an attack. After storage is compromised, it is non-trivial to determine even with hardware forensics where malware has originated and migrated [9]. Finally, there are minimal protections for data *integrity* within the storage. While filesystem encryption provides confidentiality, data integrity can still be violated by malicious but authorized I/O operations as the result of compromising or bypassing the authentication mechanisms.

Data *provenance* represents a powerful technique allowing us to address these shortcomings. The provenance of a data object characterizes its lineage from the time it was generated, describing all modifications made that result in the object's current state. With provenance mechanisms incorporated into storage devices, we would possess the means to carry out forensic analysis and even to thwart attacks based on information collected within the device itself.

In this paper, we propose *ProvUSB*, block-level provenance for USB storage devices. When plugged into a machine, the ProvUSB device uniquely identifies the host through trusted hardware, then generates provenance records describing each I/O operation that is performed by the host on the device storage partition. Extending this mechanism, we present a provenance-based integrity

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978398

protection scheme, providing fine-grained authorization of access attempts based on the Biba [11] and Low Water-Mark Access Control (LOMAC) [19, 20] integrity models. We fully implement the ProvUSB device using an embedded USB development board and perform a thorough evaluation covering enumeration overhead, I/O micro benchmarks, realistic workloads, storage overheads, and two forensic investigation case studies. The results show that ProvUSB imposes a once-per-session overhead of less than a second during device enumeration (the USB protocol handshake), but throughput and latency for normal device usage is close to bare metal performance (90% of the original throughput with a 17% latency overhead) when considering a mean file size of 10 MB. The overhead of provenance storage is less than 0.1% in real-world workloads.

Our contributions can be summarized as follows:

- **Block-level provenance**: ProvUSB leverages Trusted Platform Modules (TPMs) to authenticate host machines and collect provenance automatically for each I/O operation at the block level. To the best of our knowledge, ProvUSB is the first USB storage solution to support provenance natively.
- **Block-level policy control**: We extend ProvUSB with a policy mechanism that enforces integrity protections for USB storage at the block layer. This unprecedented granularity allows ProvUSB to remain useful in MLS environments in which devices may pass between machines of different integrity levels.
- **Comprehensive evaluation and analysis**: we evaluate ProvUSB's TPM attestation, USB enumeration, and I/O overhead, using micro and macro benchmarks, and real-world workloads, demonstrating that overhead is sufficiently low for practical usage.

The paper is organized as follows: Section 2 provides background on USB storage, data provenance and TPM attestation. Section 3 describes how ProvUSB works in general, identifies key challenges of block-level provenance and policy control, and presents our solution. Section 4 presents technical details of ProvUSB's implementation. We evaluate ProvUSB in Section 5. Section 6 discusses the trade-offs and limitations, as well as potential mitigations, of ProvUSB. Section 7 provides an overview of related work, and Section 8 concludes.

2. BACKGROUND

USB Storage: The USB (Universal Serial Bus) specification [16] defines protocols used in communication between a host machine and a device across a serial bus. When a USB device is plugged into a host machine, the host initiates an *enumeration* process that identifies the device and loads drivers on its behalf. During this procedure, the device reports hardware information (e.g., product ID, vendor ID) and supported configurations, and requests a specific configuration and set of device interfaces (e.g., Storage, Human Interface, etc.). Based on this information, the host USB controller loads and configures the appropriate drivers for the device to function. To further regulate the standards of USB devices, the USB DWG (Device Working Group) defines sub standards, categorizing USB devices into different classes, promoting increased interoperability and reliability among devices in the same class [66]. After the normal USB enumeration, the corresponding storage class driver, e.g., `usb-storage` in the Linux kernel, is loaded as a glue layer between the lower USB transportation layer and the higher block layer. The USB mass storage protocol provides support for a subset of SCSI commands, and every device implementing the standard supports these. When a USB mass storage device is connected, the host starts scanning the SCSI LUNs (logical unit numbers) on the device, which collects the corresponding filesystem information before the device or partition can be mounted correctly.

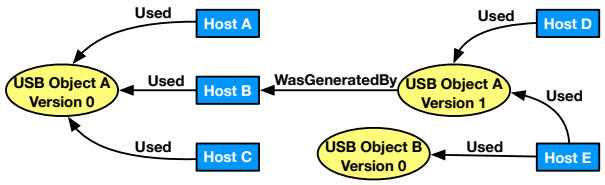


Figure 1: An example provenance graph from the perspective of a USB device capture agent. As the device lacks insight into the internal state of the connected host, hosts are represented as opaque, monolithic activities.

Data Provenance: Data provenance describes the history of manipulations performed on a data object from its creation up to the present. As such, provenance is a powerful means of reasoning about the value of a piece of data. Traditionally, provenance capture has been proposed on hosts at the system call layer, which requires changes to system libraries [27] or the operating system [8, 22, 46]. Such systems require provenance to be saved in the host machine or relayed to a remote server, placing trust in the software running on the host. Due to the fine granularity of provenance collection, these approaches are also prone to requiring storage overheads of gigabytes per day [8, 47].

In this work, we conceptualize data provenance from the perspective of a USB storage device in order to overcome the obstacles of storage overhead and trust in the host platform. A visualization of USB storage provenance is shown in Figure 1, shown here as a directed acyclic graph that is compliant with the W3C PROV data model [17]. The principals we consider are USB storage data objects and host machines, treating the internal state of the host machine as opaque. These entities are represented as vertices in Figure 1, while edges encode relationships between the principals and flow backwards into the history of system execution. Here, "Used" corresponds to object reads, and "WasGeneratedBy" to object writes. To prevent cycles from forming in the graph, a versioning system is used in which a new version node is created every time an object is written to. In this case, we see that *Host B* used *USB Object A, Version 0* and subsequently generated *USB Object A, Version 1.*

TPM Attestation: A Trusted Platform Module (TPM) is a tamper-resistant cryptographic module embedded in the motherboards of many commodity systems. It provides a hardware root of trust for storing cryptographic keys and *measurements* that represent the current system state. These measurements begin by hashing over the system BIOS and then successively storing hashes of all software in the stack, including boot loaders, operating systems and ap-

plications [33, 36, 51, 56]. All measurements are stored in dedicated Platform Configuration Registers (PCRs) that reside inside the TPM. The TPM enables third parties to remotely validate a system's state using an *attestation* protocol. A host's TPM enrolls keys with a Privacy Certification Authority (PCA) or Attestation Certification Authority (ACA), and acquires an Attestation Identity Key (AIK) credential key pair. To start TPM attestation, the challenger sends a nonce (to counter replay attacks) to the target TPM. The target TPM executes a *quote* command, which returns a report of the current PCR values signed with the AIK. Upon receiving the quote result, the challenger verifies it using the public AIK and compares the PCR values within the quote against known good values, which can be obtained from a factory installation or PCA. A successful attestation implies that the challenger is communicating with the correct host and that the system is in a known good state. While TPM attestations are usually performed remotely over a network, in this work we enhance the host operating system to support TPM attestations over the USB channel.

3. DESIGN

3.1 Threat Model & Assumptions

This work considers an enterprise environment that wishes to protect itself from USB-based threats. For portable storage, we assume that such an organization is willing to make exclusive use of advanced USB technologies such as IronKeys [32], which is already a common practice in many high-security environments. We assume that these devices are distributed within the organization similarly to the way that "loaner" devices are distributed for the purposes of business travel: in order to to check-out a device, an employee registers with a technology office; after a pre-defined length of time (*epoch*), the employee is required to return the equipment to the technology office; upon receiving the piece of equipment, the technology office will perform forensic analysis and then re-image the device.

Although firewalls, IDS, and security training seminars for employees may be extensively deployed in such an organization, it is only a matter of time before powerful attackers (e.g., APTs [45]) gain a foothold within the network. We thus assume that any manner of software compromise is possible on hosts in the network. Therefore, our solution cannot trust any software running on the host machine, including the kernel and the entire USB stack. However, we do assume that all hosts contain a TPM, and that because the attacker does not have physical access, hardware or firmware attacks against the TPM are not possible.[1]

USB storage is a known delivery mechanism for malware, e.g., Stuxnet [18]. To mitigate the threat of USB-based malware propagation, we present a provenance-based integrity protection scheme that prevents data originating within low integrity sources from flowing to high-integrity targets. To facilitate this, we make the assumption that all hosts in the enterprise have been assigned integrity levels by a system administrator. This process would be intuitive to administrators, as each host's integrity level would likely correspond

[1]While any mechanism that ensures the security of the host and underlying firmware and hardware may be used, we focus discussion here on the TPM due to its wide-spread deployment and well-understood integrity semantics [52].

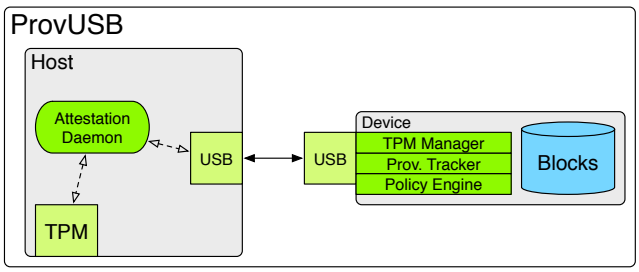


Figure 2: The ProvUSB device contains several components: a TPM manager, Provenance Tracker, and Policy Engine. When plugged into the host, the device issues a TPM challenge to the Attestation Daemon over the USB channel, which then responds with the signed quote.

to the privilege level of its users. Examples of Low Integrity (LI) hosts might include unprivileged employee workstations and machines outside of the demilitarized zone, while a High Integrity (HI) level might be assigned to administrator hosts or workstations handling highly sensitive data.

Attack Scenarios: Our architecture is designed to provide provenance for two notorious forms of USB attack, malware propagation and data exfiltration.

Malware Propagation. For malware, we specifically focus on tracking infected hosts that inject payloads onto storage devices, which are later read by other machines. This pattern of behavior is consistent with live malware samples, including ZeuS [64] and Stuxnet [18]. Antivirus software may eventually detect such malware after new signatures are generated and pushed to the client, but offer little forensic insight into when and how the malware was initially injected. This is usually true because most forensics rely on the MAC (Modification, Access, Change) time provided by the filesystem, which only provides the latest timestamps.

Data Exfiltration. In the case of data exfiltration, insider attacks are particularly dangerous. With a USB storage device, an attacker can easily exit the physical premises with large amounts of sensitive data secreted away in their pockets. Most secure USB storage devices constrain the I/O operations via authenticating the user, either using centralized authentication servers or local passwords [31], but do not address the problem of malicious users with legitimate access credentials. We aim to ensure that exfiltration events can be traced back to the host that acted as a source of the data leak, even if that host has been compromised to wipe traces of the exfiltration. Additionally, device policy can be configured such that the storage device erases all of its blocks (while maintaining provenance metadata) if it is plugged into an unrecognized or unauthorized host, substantially limiting the damage from a malicious insider absconding with the device from an enterprise. In both of these scenarios, forensic analysis requires that the attacker's actions be tracked prior to detection and without requiring trust in software running on the host.

3.2 Security Goals

Critically, in both of the above attacker scenarios, the ability to reliably track read and write operations to the USB storage device is sufficient to explain the actions of the

attacker. With the above considerations in mind, ProvUSB sets out to provide the following capabilities:

G1 Minimal Trusted Computing Base. ProvUSB's TCB must be limited to software running on the device. ProvUSB must verify the identity of host machines prior to granting access to the storage partition. Additionally, ProvUSB must verify both the identity and the software configuration of privileged machines prior to exposing provenance information.

G2 Forensic Validity. ProvUSB must provide a complete and exhaustive provenance description of the device's interactions with host machines. If provenance loss occurs it must be detectable by the administrator.

G3 Tamperproof. The host machine should not be able to disable ProvUSB's provenance collection logic.

G4 Track Malware Propagation. Given out-of-band knowledge that host A is known to be infected at time t (e.g., Antivirus alert), ProvUSB must 1) identify the hosts that contributed to A's internal state prior to time t, and 2) identify the hosts that used A's internal state between time t and the present.

G5 Explain Data Leaks. Given out-of-band knowledge that file f was leaked prior to time t (e.g., f's contents were published online), ProvUSB must identify the hosts that had knowledge of f prior to time t.

G6 Integrity Assurance. For data in the storage partition, ProvUSB must prevent all integrity violations in which data from a lower integrity host flows to a higher integrity host.

3.3 Design Overview

An overview of the ProvUSB architecture is shown in Figure 2. ProvUSB requires host machines to be equipped with a *TPM* chip, an *Attestation Daemon* in user space to communicate with the TPM, and an enhanced USB storage driver in the kernel space to support TPM attestations. Several components are introduced on the device side: a *TPM Manager*, *Provenance Tracker*, and *Policy Engine*. The host machine and the device communicate exclusively over the USB channel. When a ProvUSB device is connected to the host, the host initiates a TPM attestation following normal enumeration. The *TPM Manager* verifies the attestation on the device side, then permits the host to perform storage operations. During normal usage, the *Provenance Tracker* generates provenance for all I/O events performed on the device. An optional *Policy Engine* can be enabled to authorize access attempts. During device usage, all provenance is created and managed within the ProvUSB device. While the software stack within ProvUSB is included in the TCB, no software running in the host system is trusted.

3.4 Identifying Host Machines

Before we can track device provenance, we need a means of authenticating the hosts to which ProvUSB connects. The identifier should be persistent over the host's lifetime, meaning volatile or spoofable values, such as the MAC address of a NIC, will not work. The device must also be able to obtain the identifier over the USB channel. While other approaches including host fingerprinting over USB exist [6], our approach is to leverage TPMs, found in most current commodity machines.

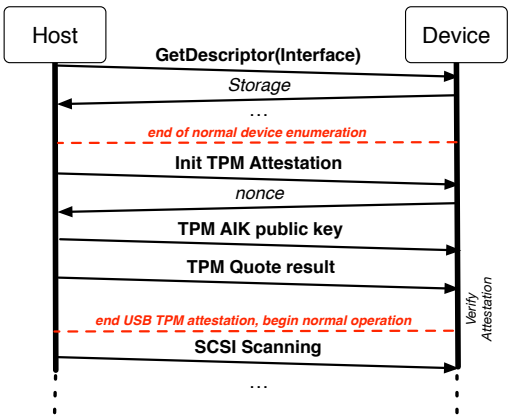


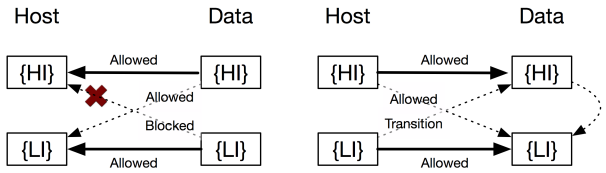
Figure 3: In device enumeration with ProvUSB, the host initiates an attestation procedure over the USB channel to allow the device to authenticate the host.

To allow TPM attestation over USB, we make use of a technique introduced in the Kells framework [12]. Kells uses TPM attestations to authenticate hosts prior to exposing sensitive storage partitions. In the event that attestation fails, Kells only exposes the public partition instead of the private one. It accomplishes this functionality through the introduction of a USB-based TPM attestation, as shown in Figure 3. In ProvUSB, the TPM Manager is able to verify the identity of the host using the following procedure: *(1)* During normal device enumeration, the host USB storage driver recognizes the device as ProvUSB-enabled based on its vendor ID. *(2)* Following enumeration, the host sends out a USB Accept Device Specific Command (ADSC) [72] to the device asking for a 20 byte nonce, which marks the beginning of the TPM attestation. *(3)* The device's TPM Manager sends a nonce to the host. *(4)* The host USB storage driver sends the nonce to the Attestation Manager, which generates a TPM quote and returns the TPM AIK public key and quote to the storage driver. The host storage driver sends the public key and quote to the device over ADSC. *(5)* On the device, the TPM Manager validates the TPM AIK public key, using it to verify the TPM quote.

If the verification succeeds, the device permits the SCSI scanning request, allowing the host machine to mount the partition, and the device moves into the provenance tracking phase. If verification fails, the storage partition is not exposed to the host. As described above, depending on how device policy is configured, a failed verification can also result in the device erasing all of its blocks to prevent exfiltration attempts. Furthermore, when ProvUSB devices are deployed in enterprise environments, host machines should also reject non-ProvUSB storage devices to avoid attacks from uncertified devices.

3.5 Block-level Tracking and Protection

Provenance Tracking: When designing ProvUSB, we initially considered instrumenting a specific filesystem, as was done in the past for host system provenance [27]. However, this would constrain our approach to a particular filesystem format, which was in conflict with the Plug and Play



(a) Read Operations (b) Write Operations

Figure 4: ProvUSB's Integrity Model. High Integrity (HI) hosts are not permitted to read Low Integrity (LI) data blocks from USB storage. HI data blocks transition to an LI state if written to by an LI host.

nature of USB devices. Even worse, filesystem layer monitoring could be bypassed using raw I/O [58, 70], leading to incomplete provenance. Additionally, we wished to be able to track storage operations at the finest possible granularity.

Instead, ProvUSB integrates provenance capture into the lower layers of the storage device; provenance is generated for all I/O operations at the SCSI layer. By designing the provenance at the block level, we are able to make ProvUSB filesystem agnostic. This means, although ProvUSB requires specialized firmware or software running on the device, it can be adopted without requiring any changes to the filesystem format of the storage partition.

One consequence of this approach is that, in many cases, block-level provenance will need to be translated into human-readable filesystem semantics before it can be interpreted by an investigator. Auxiliary tools that are compatible with ProvUSB's provenance store will therefore be necessary to translate block-level activity to file level information. Such tools can be built as straightforward extensions of block reverse engineering mechanisms such as filesystem checkers or even built as file-to-block mapping services embedded within existing filesystems. In Section 4, we implement a tool that correlates files with their blocks for the FAT filesystem. By relegating this translation to a post-processing step, we decouple provenance collection from interpretation and ensure that our design is independent of actual filesystem implementation.

Block-level Integrity Protection: While a variety of USB devices appearing on the market [32, 38] and in the literature [12] provide data confidentiality, less attention has been paid to integrity assurance. We now discuss an extension to ProvUSB that provides provenance-based enforcement of an MLS integrity lattice. For clarity, we introduce simple lattice featuring Low Integrity (LI) and High Integrity (HI) levels, although our system could accommodate arbitrarily many integrity levels. Although we choose to focus on integrity assurance, the following could also be easily adapted to provide fine-grained confidentiality in an MLS model that provides Bell-LaPadula guarantees [10].

To provide a realistic operating environment for USB storage, ProvUSB's integrity model borrows from both Biba [11] and LOMAC [19, 20] in that it assumes immutable host labels and mutable data labels. Like Biba, hosts are assigned an integrity level at creation time and subsequently have a null transition state. Blocks on the storage device have a stateful transition between integrity levels, like LOMAC. At creation time, all blocks are in an HI state. Their current level is equal to the lowest integrity class found in the object's provenance ancestry (e.g., the lowest integrity writer).

On access attempts, ProvUSB's *Policy engine* applies the enforcement rules pictured in Figure 4. For `read` operations, ProvUSB grants the request only if the host machine's integrity level is less than or equal to the data block's integrity level. For `write` operations, ProvUSB grants all requests. If an LI host writes to an HI block, ProvUSB transitions the block's level to LI. An additional transition (not pictured) is if ProvUSB detects that an entire block has been overwritten by an HI host, the block returns to an HI state.

Optimization: Provenance Filtering: The automatic capture of provenance is associated with high storage overheads. For example, several provenance-aware operating systems generate more than 1 GB of provenance during kernel compilation [8, 47]. While we expect the overall volume of activity to be less for a USB device than an operating system, the storage overhead imposed by provenance collection will nonetheless dictate the frequency with which devices need to be collected and re-imaged by the technology office (§3.1). We introduce the following filtering optimization to the ProvUSB device provenance daemon to ensure that provenance storage costs are minimized without loss of expressivity, considering the following three scenarios:

(1) Consecutive Reads: A host reads the same block multiple times prior to a new write. After the initial read, we assume this data already exists on the host.

(2) Consecutive Writes: A host writes to the same block multiple times without intervening reads from another host. Since this block is already tagged with the host's integrity level, recording a second write event provides no new information except the latest timestamp.

(3) Write, then Read: A host reads from a block that it had previously written. Here, the consumed data was already assumed to be on the host, so recording the read event is not necessary.

Our versioning algorithm filters the number of provenance events recorded, maintaining an in-memory map structure for the current version of each block in use. Each version is a tuple consisting of:

$< BlockNum, VersionNum, H_{wGB}, TS_{wGB}, UsedBy >$

where $BlockNum$ is the block number, $VersionNum$ is a monotonically increasing integer representing the current version of $BlockNum$, H_{wGB} is the label of the host that most recently wrote to the block, and TS_{wGB} is the timestamp when H_{ID} first wrote to $BlockNum$. $UsedBy$ is a list of tuples (H_{UB}, TS_{UB}) where TS_{UB} is the time that H_{UB} first read the current version of $BlockNum$. The filtering algorithm works as follows:

Read from Host H_A to $BlockNum$: the device daemon updates $BlockNum$'s struct, appending $UsedBy$ iff $H_{wGB} \neq H_A$ and $\forall H_{UB} \in UsedBy, H_{UB} \neq H_A$.

Write from Host H_A to $BlockNum$: iff $H_{wGB} \neq H_A$, the device daemon writes $BlockNum$'s struct to flash storage, re-initializes the struct, sets $H_{wGB} = H_A$ and TS_{wGB} equal to the current timestamp.

The result is that flushes to storage never occur for consecutive reads, and only occur on writes when a new host writes to a given block. When the ProvUSB device receives an ejection signal, the in-memory structure is flushed to storage,

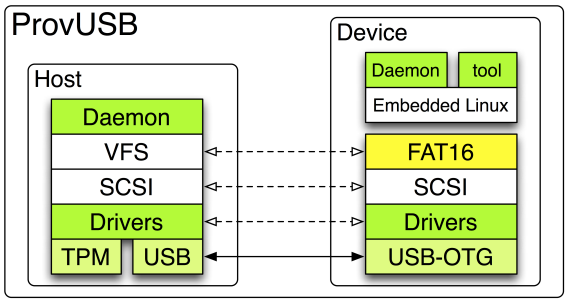


Figure 5: ProvUSB Software Stacks. The left figure shows a software stack of a USB storage in the view of a host machine. The right box displays the software stack of ProvUSB devices. All layers are mapped between the host machine and the storage device.

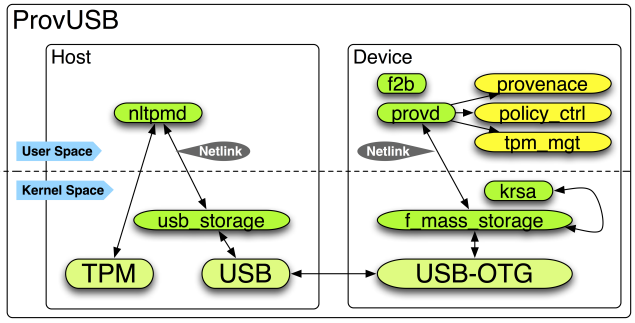


Figure 6: ProvUSB Architecture. In the host machine, the `nltpmd` communicates with the TPM and relays the TPM attestation results to the `usb-storage` driver. In the ProvUSB device, `f-mass-storage` driver verifies the TPM attestation using `krsa` and monitors each I/O operations at the block level, enforcing the policy control and relaying the provenance to `provd` in the user space.

saving provenance and block integrity information, which is read back into memory next time the device is plugged in.

4. IMPLEMENTATION

We implemented ProvUSB on a Gumstix COM (Computer-on-Module) [25] equipped with a USB OTG (On-The-Go [71]) interface, acting as a USB storage device formated in FAT16 filesystem, as shown in Figure 5. The host machine was a TPM-equipped desktop running a modified version of Linux. An overview of the architecture is shown in Figure 6.

Host Modifications: We modified the Linux `usb-storage` driver to support TPM attestation over USB. ProvUSB devices enumerating to the host signal their presence by using vendor ID `0xb000` to distinguish themselves from other USB storage devices. After the normal enumeration and before the driver begins scanning the storage device, the host initiates a TPM attestation procedure as outlined in Section 3.4, beginning with sending an ADSC over the USB control channel (EP0) to activate the TPM attestation procedure on the device side. Note that because EP0 is mandatory for all USB devices and ADSC uses EP0, ADSC is natively supported by the USB hardware.

In user space, the NetLink TPM Daemon (`nltpmd`) listens on a netlink socket for a new TPM challenge (i.e., a nonce) from the `usb-storage` driver. Once the nonce arrives, it uses the TrouSerS API [69] to talk to the TPM hardware, then sends both the TPM quote and AIK public key back to the `usb-storage` driver. The driver delivers this response to the ProvUSB device over an ADSC message. Immediately after this, the driver requests SCSI scanning the device storage in order to mount the partition, which is permitted by ProvUSB only if it successfully authorizes the host.

Device Modifications: In the kernel space of the USB storage device, we modified the `f-mass-storage` gadget driver. We created a new device entry mapping for vendor ID `0xb000` and added it into the USB Mass Storage compliant devices list. To support TPM attestation over USB, we added new EP0 callbacks to process ADSCs from the host machine. To manage the TPM AIK public keys and maintain the policy control for each block, we added two linked lists to the gadget driver. We backported the RSA verification implementation from Linux Kernel 3.13 to directly support signature verification in kernel space. We also patched the crypto subsystem to make it callable in the EP0 handlers (IRQ context). Finally, we added a kernel timer to the gadget driver so TPM attestations can gracefully fail in the event of a timeout, which could occur if the host machine was not equipped with a TPM or the TPM attestation response was sent late. In the event of a failure, the gadget driver rejects subsequent requests from the host machine.

In the user space of the USB storage device, a provenance daemon (`provd`) communicates with the kernel gadget driver using a netlink socket. `Provd` performs three key tasks: 1) *TPM management*, 2) *policy configuration*, and 3) *provenance logging*. TPM management supports administrative tasks such as adding TPM AIK public keys to the gadget driver key store, or removing old ones. The policy configuration component allows administrators to assign integrity labels to new machines. It also maintains persistent storage of the integrity labels of different blocks on the storage partition for when the device is unmounted, which is necessary for bus-powered devices. This information is loaded automatically into the gadget driver when the device is plugged into a host machine again. The provenance logging component runs in two configurations, *ProvFlush* and *ProvSave*. The ProvFlush configuration synchronously flushes each new provenance record to persistent storage as soon as it is created. The ProvSave configuration implements the optimization described in Section 3.5 in which provenance is only flushed to persistent storage on write operations and initial read operations among consecutive ones. To bridge the semantic gap between human-interpretable file metadata and block provenance, we implemented a utility called file2block (`f2b`) that can identify the blocks corresponding to a given file name in the FAT16 filesystem. After recovering the blocks associated with a file, the provenance of each block can then be queried individually to discover the file's provenance.

5. SECURITY ANALYSIS

Minimal Trusted Computing Base (G1). To verify the identity and software configuration of the host, ProvUSB performs a TPM-based remote attestation over the USB interface. ProvUSB does not trust any configuration infor-

mation reported by the host during USB enumeration, as the host could spoof these messages. The host cannot lie about the operations it performs on the device's storage partition, as ProvUSB is able to record the ground truth of the hosts interactions with the USB storage. Therefore, ProvUSB only trusts software running onboard the device.

Forensic Validity (G2). ProvUSB provenance information is aggregated and analyzed by an enterprise's technology office periodically upon device check-in. The length of epochs is parametrizable, but we envision that device check-ins will occur on the order of days. Failure to return a ProvUSB device could lead to incomplete provenance, but in this instance the technology office is alerted to the possibility of nefarious behavior on the part of the employee that checked-out the device. From this, we can conclude that G2 is satisfied, as at the end of each epoch the administrator is guaranteed to have either a complete provenance history for the device or an alert to the possibility of misbehavior.

Tamperproof (G3). A violation of Goal G3 would occur if an employee is able to manipulate ProvUSB in order to insert or remove provenance records between epochs; through use of a modified version of the Yocto Linux kernel and USB OTG, the storage of provenance is separated from the storage partition exposed to the host, making it extremely difficult to view or modify provenance records without physically opening the device. The possibility of such attacks could be further mitigated through implementing ProvUSB on tamper-resistant hardware, in much the same way that IronKey ensures confidentiality guarantees [32]. Another possible attack that could lead to lost provenance would be to exploit the race condition between the times when a block operation occurs and when its provenance record is written to persistent storage. For example, immediately after a write operation the employee could physically unplug the device, causing the device to shut down before ProvUSB can record the operation's provenance. While our prototype implementation would be vulnerable to this attack, this threat could be remedied by extending ProvUSB with a capacitor-backed mechanisms to flush out such metadata quickly when disconnected from the host power source.

Track Malware and Explain Data Leaks (G4,G5). Given Goals G2 and G3, it logically follows that G4 and G5 are satisfied. This is because previous work has already demonstrated that data provenance can be used to track the propagation of malware [5, 21, 54, 65, 74] and detect data exfiltration [8, 35, 40, 41, 43]. We provide specific examples of how ProvUSB can detect malware propagation in §6.6.

Integrity Assurance (G6). Given Goals G2 and G3, it logically follows that G6 is satisfied. This is because previous work has demonstrated that provenance histories can be used to provide fine-grained access control enforcement [7, 48, 50]. We provide specific examples of how ProvUSB can prevent integrity violations in §6.6.

6. EVALUATION

We now evaluate the performance of the various components of ProvUSB. The host machine used in our tests is a Dell Optiplex 7010 desktop with a quad-core Intel i5-3470 3.20 GHz CPU, 8 GB memory, standard USB 2.0 ports and an STM TPM (version 1.2 and firmware 13.12), running Ubuntu LTS 14.04 (x86-64) with Linux kernel version 3.13.11 and TSS API 1.2 rev 0.3. The ProvUSB device is a Gumstix Overo COM with an ARMv7 600 MHz CPU, 256 MB flash memory and a USB OTG port, running Yocto 1.6 with Linux kernel version 3.5.7. The device uses an 8 GB class 10 microSDHC flash memory card; the USB storage partition is 1 GB in size, formatted as a FAT16 filesystem with the logical block size of 512 bytes. We compare the ProvUSB storage with two commercial USB storage devices: a Kingston DataTraveler 2 GB USB thumb drive, and a SanDisk Cruzer Fit 16 GB USB stick. All devices were plugged into the same USB 2.0 port throughout the evaluation.

TPM Operation	Min	Avg	Med	Max	Dev
PCR Read	10.633	12.188	11.978	23.199	1.709
AIK Read	48.220	57.244	60.273	60.491	5.218
Quote	323.647	344.028	347.858	360.209	6.100

Table 1: TPM operation time (ms) averaged by 1000 runs.

Enumeration	Min	Avg	Med	Max	Dev
Kingston2G	338.446	343.003	340.682	389.194	10.630
SanDisk16G	119.962	122.583	122.550	125.309	1.383
Original	40.246	65.382	50.857	97.724	22.286
ProvUSB	839.120	850.743	851.300	851.766	2.670

Table 2: USB enumeration time (ms) averaged across 20 plugging operations for different devices.

6.1 TPM Operations

One source of overhead imposed by our system is the introduction of a TPM-based attestation procedure during device enumeration. We determined the cost of host attestation by measuring the completion time of each TPM operation used by ProvUSB. Averaged across 1000 iterations, we found that reading the system measurement from PCRs takes 12.19 ms, while retrieving the AIK public key takes 57.24 ms. The cost of attestation is dominated by the quote operation, which required 344.03 ms to complete on average, with a standard deviation of 6.10 ms, as shown in Table 1. Fortunately, *this overhead is a one-time effort during USB enumeration, and does not affect the runtime performance of the device.* The speed of TPM operations is constrained by the performance of the TPM hardware. Though TPM are usually low-speed chips, the enumeration overhead could be brought to minimum through the help of high-end TPM chips that uses the 400 KHz I2C bus [28].

6.2 USB Enumeration

To show the overhead of USB enumeration using ProvUSB compared to other commercial devices, we measured the complete procedure from the host side, starting from the beginning of the device probing, to the TPM attestation protocol (for ProvUSB), and including the SCSI scanning for the USB storage, by manually plugging each storage device into the host machine 20 times. As shown in Table 2, Kingston 2G takes 343.00 ms on average while SanDisk 16G needs 122.58 ms. Compared to these products, the Gumstix board we use for ProvUSB is much faster, spending only 65.38 ms to finish the enumeration. This is due to the host kernel quickly recognizing the device as a USB storage gadget and loading the corresponding driver immediately. The time required by ProvUSB is 850.74 ms on average. This is high compared to commercial products, but *this is a one-time effort per session or plugging.* Less than one second is required to mount a partition.

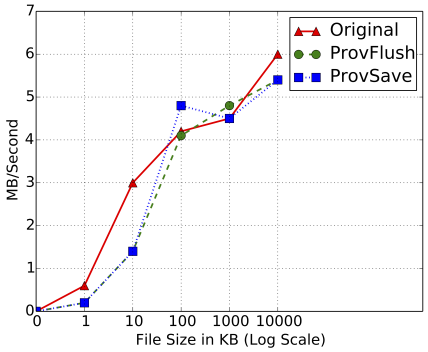


Figure 7: ProvUSB throughput (MB/s) using `fileserver` workload with different file sizes.

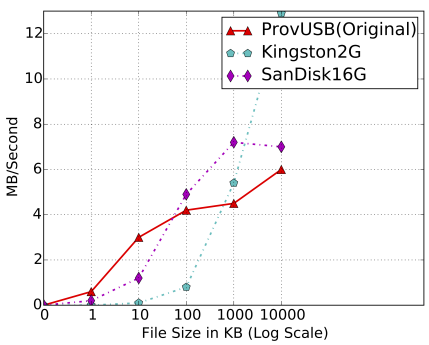


Figure 8: `fileserver` throughput (MB/s) comparison of different devices with different mean file sizes.

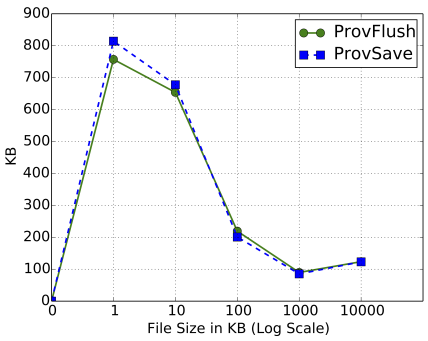


Figure 9: Provenance storage using `fileserver` workload with different mean file sizes.

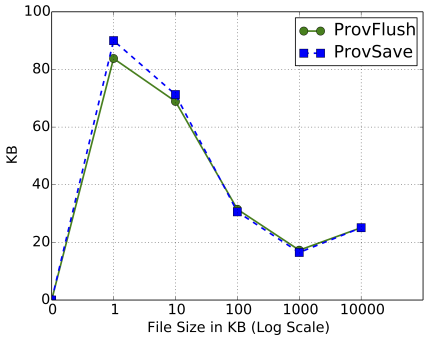


Figure 10: Block control storage using `fileserver` workload with different mean file sizes.

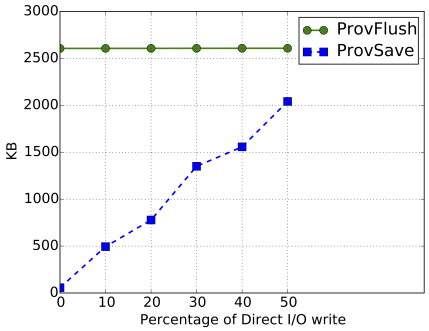


Figure 11: Provenance overhead with ProvUSB optimization using Direct I/O with different read-write ratios.

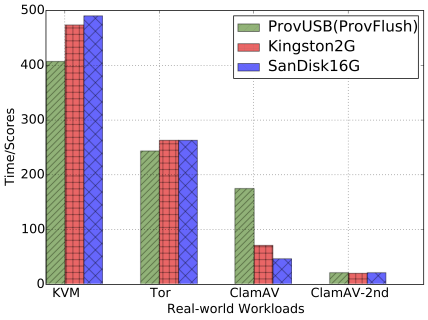


Figure 12: Performance comparison of different USB devices using real-world workloads.

6.3 I/O Operations

To measure the runtime performance of ProvUSB, we used `filebench` [62] to benchmark throughput and latency, then compared the results to the Kingston and SanDisk devices. We chose the `fileserver` workload model within `filebench` as the testing base, because it covers all file related operations, including read, write, create, open, close, delete, and stat. However, as `fileserver` was designed to benchmark file servers, to reflect a plausible usage pattern for a USB storage device, we changed the default configuration as follows: The mean file size in operation ranged from 1 KB to 10 MB; the number of files in total was fixed at 20; the number of operation threads was set to 1; the I/O size was fixed to 1024 bytes, and the run time was fixed to 60 seconds. We selected this configuration to represent a single user making rapid edits to a set of documents, which is a more realistic usage pattern for a USB device.

The throughput of ProvUSB under different configurations is shown in Figure 7. *Original* signifies the original Gumstix device without ProvUSB functionality enabled. *ProvFlush* refers to the ProvUSB device synchronously flushing provenance once created. *ProvSave* is the ProvUSB device using the optimized provenance logging method described in Section 3.5. When the file size is small (1 KB and 10 KB), ProvUSB achieves roughly half of the throughput of the original device. Intuitively, the cost of creating a new provenance record remains the same regardless of the size of the file being accessed. As such, ProvUSB still needs to examine a large number of blocks when files are distributed sparsely in the filesystem, since each file holds the whole block (cluster) even if it is small. As the mean file size grows (from 100 KB to 10 MB), both ProvFlush and ProvSave demonstrate throughput similar to the original device. For example, when the file size is 10 MB, the throughput of Original is 6.0 MB/s, while that of both ProvFlush and ProvSave are 5.4 MB/s (10% overhead).

Comparing ProvUSB to the unmodified Gumstix Overo COM may not be representative of overhead imposed on commercial USB storage devices. To address this, we then compared ProvUSB's original device performance on the same benchmark against commercial storage devices from Kingston and SanDisk. The results are shown in Figure 8. Kingston 2G demonstrates a high throughput jump when the mean file size is bigger than 1 MB, but the lowest throughput when the mean file size is small. SanDisk 16G shows better throughput compared to Kingston 2G when the mean file size is small, and maintains the throughput at around 7 MB/s when the mean file size is greater than 1 MB. The original ProvUSB device has the best throughput when the mean file size is small, and a comparable throughput with SanDisk 16G when the mean file size is 10 MB. In general, as the mean file size increases, the utilization of each block gets better, as well as the utilization of transmission bandwidth of storage devices.

The corresponding latencies for the tests from Figures 7 and 8 are shown in Table 3. Both ProvFlush and ProvSave perform comparably to the original device when the mean

Device/Configuration	1K	10K	100K	1M	10M
Original	1.0	2.1	11.9	105.5	768.6
ProvFlush	3.1	3.6	12.3	104.8	897.6
ProvSave	2.9	3.5	13.5	111.3	856.5
Kingston2G	48.5	42.3	62.3	93.4	386.6
SanDisk16G	3.5	4.3	10.3	70.1	722.4

Table 3: Latency (ms) of ProvUSB under different configurations, Kingston 2G, and SanDisk 16G devices using the `fileserver` workload with different file sizes.

file size is smaller than 10 KB due to the sparse block allocation for small files. With a 10 MB file size, ProvFlush imposes 16.8% overhead while ProvSave introduces 11.4% overhead compared to the unmodified device. Kingston 2G's latency is largest when the mean file size is smaller than 1 MB, but smallest when the mean file size is 10 MB, which also explains its throughput behavior. We suspect that its USB microcontroller is optimized for large bulk data transfer, and the result is reflective of realistic workloads on this device. Aligned with its throughput model, SanDisk 16G shows similar latencies with ProvFlush and ProvSave's when the mean file size is small, and latencies comparable to the unmodified device as the mean file size increases. ProvUSB devices thus perform comparably to commercial devices, providing a realistic assessment of overheads we introduce if our mechanisms are deployed on production devices.

6.4 Provenance Collection

To evaluate the provenance storage costs introduced by ProvUSB, we measured the provenance log size during the `filebench` testing. As shown in Figure 9, both ProvFlush and ProvSave generate around 800 KB of provenance, when the mean file size is 1 KB. Besides the total number of files (20) in operation, the main reason for this relatively large amount of provenance is the large number of I/Os (read and write) performed by `filebench` given the running time, as a result of the small file size. In this case, ProvFlush and ProvSave have 305 and 329 I/Os respectively. This also explains why the throughput and latency of ProvUSB are not optimal when the mean file size is small ($<$ 10 KB). As the mean file size increases, the overhead of provenance logging decreases, since each I/O takes more time, and `filebench` does not repeat the same I/O multiple times. For example, when the mean file size is 100 KB, ProvFlush has 80 I/Os, whereas ProvSave has 73 I/Os in total. If we assume the total amount of data is 200 MB ($10MB \times 20$, based on the `fileserver` workload configuration) when the mean file size is 10 MB, the provenance storage takes around 120 KB, introducing only 0.06% storage overhead.

In addition to provenance logging, ProvUSB also tracks the integrity level of each block. This block control information is loaded automatically every time the ProvUSB device powered on, helping enforce persistent block-level policy control. As shown in Figure 10, regardless of the different scale of the provenance storage overhead, the block control storage pattern looks almost the same as the provenance storage, demonstrating that write operations are dominant in the `filebench` benchmark. In our testing, we find that the number of write I/Os usually doubles the number of read ones, which is the behavior of `filebench` using the `fileserver` workload. To mitigate the impact of repeated I/Os during the workload, we compute the average

Provenance per I/O	1K	10K	100K	1M	10M
ProvFlush	2.76	2.78	3.14	11.91	148.80
ProvSave	2.75	2.78	3.17	11.32	148.20

Table 4: Provenance overhead (KB) of ProvUSB per I/O using the `fileserver` workload with different file sizes.

Storage Size	KVM	Tor	ClamAV
Workload Files	581 MB	152 MB	581 MB
Provenance	186 KB	81 KB	471 KB
Overhead	0.03%	0.05%	0.07%

Table 5: Provenance storage overhead using ProvUSB (ProvFlush) is less than 0.1% for different workloads.

provenance overhead (provenance + block control storage) per I/O, as shown in Table 4. Even when file size is small ($<$ 10KB), both ProvFlush and ProvSave show a decrease in overhead due to reduced interaction with the filesystem, such as searching the allocation table and updating file metadata. As the file size increases, the overhead of provenance is amortized accordingly. For instance, when the file size is 10 MB, the provenance overhead is no more than 1.5% of the original storage.

The difference between ProvFlush and ProvSave is obscured in `filebench`, as shown in Figure 9 and Table 4, due to the effect of the system cache (page cache) in the Linux kernel, which may buffer a significant portion of the USB storage partition in memory. To verify the advantage of ProvSave, we used direct I/O (bypassing the system cache) to read or write the Linux kernel 4.4 zipped source file (83 MB) 100 times for 6 runs, each of which randomly generated the read-write ratio based on a uniform distribution from 0 to 1. We then measured the provenance log size for ProvFlush and ProvSave with the proportion of writes ranging from 0% to 50%. The results are shown in Figure 11. As the proportion of writes increases, the provenance storage overhead increases almost linearly in ProvSave, from 0 KB to 2 MB, whereas the storage overhead in ProvFlush stays almost constantly at 2.6 MB, since ProvFlush does not distinguish read operations from write ones to filter loggings. When the probability of write operations is less than 40%, ProvSave reduces provenance storage cost by over 50%. While system cache is desirable, large volume USB storage (e.g., large external hard drives) would benefit from ProvSave since only a small portion of the storage can be cached. Applications bypassing the system cache (e.g., databases) would also benefit from ProvSave. The size of the block integrity storage stays the same (5696 bytes) during the testing regardless of the number of write operations.

6.5 Real-world Workloads

To arrive at a more accurate estimate of ProvUSB's performance in practice, we chose three real-world workloads using USB storage devices, then compared the performance of ProvUSB (ProvFlush) against the Kingston and SanDisk devices. In the KVM workload, an Ubuntu 14.04 image was loaded from the storage device to create and install a KVM virtual machine automatically on the host machine. The Tor workload benchmarked the portable Tor browser in storage devices by accessing the browser performance benchmarking web site [4]. The final workload unzipped the Ubuntu 14.04 image file in device storage and then ran ClamAV [39] to

scan the whole image directory for virus. All measurements are in seconds, as shown in Figure 12, except the Tor measurement, which are scores given by the benchmarking web site, and are divided by 10 to fit into the figure.

Compared to commercial USB devices, ProvUSB takes less time to finish the new virtual machine creation and installation. In the Tor browser benchmarking, all the three devices share similar scores. Within the ClamAV virus scanning benchmark, ProvUSB doubles scanning time compared to the other two devices. This is not surprising considering that unlike with the KVM and Tor workloads, where not all the files within the storage are needed to finish the task, a normal virus scanning has to touch every file in the storage to fulfill the job. To confirm the effect of system caching on system performance, we relaunched ClamAV immediately after completion of the first test. As expected, scanning completes much faster in the second iteration (approx. 20 seconds) regardless of which storage device is used. The corresponding provenance logging by ProvUSB was also collected for all workloads, shown in Table 5. Compared to workload file size, the overhead of provenance storage is consistently less than 0.1%. The ClamAV workload generates the most provenance, providing more insight as to why ProvUSB imposes higher performance overhead during virus scanning. In the majority of cases, enabling provenance collection mechanisms has little effect on real-world workloads.

6.6 Case Study

We now present a scenario in which ProvUSB can help to track (and possibly prevent) the spread of malware. The Stuxnet virus leveraged multiple USB-based attack vectors in an attempt to spread to its intended targets. One of these methods of propagation was to embed a malicious payload in an `autorun.inf` file on the storage device [18]. Until recently, this file denoted a special system script that was automatically executed by Windows operating systems upon connecting a device to the host.

In this scenario, we have an administrator Alice (Host A) and a normal user Bob (Host B), and Bob's host has been infected with Stuxnet. Alice has used Host A to configure and distribute ProvUSB devices throughout the organization, including installing a policy that marks Host A as high integrity (H) and Host B as low integrity (L). In an attempt to deliver a critical system patch, Alice now plugs her ProvUSB device into the infected Host B.

Case 1: Detect Malware Propagation. Once the storage is plugged into Host B, Stuxnet writes a malicious `autorun.inf` to the device, yielding the provenance record *<201509161750,B,w,2505,1282560,512>*, consisting of a timestamp, host machine identifier, operation, block number, file offset, and amount. Let us assume that block *2505* is the only block needed to save the file. While the storage partition of the device has been infected, ProvUSB is able to collect provenance as normal. Eventually, Alice may discover the infection on one of the hosts in her network. She can then use the `f2b` tool to identify the blocks associated with `autorun.inf`, query the provenance to reconstruct the chain of infections, and prepare a recovery plan.

Case 2: Integrity Assurance. Using Host A, Alice has configured her ProvUSB device to automatically run diagnostic utilities on the host via an `autorun.inf` script. As a result, prior to the attack ProvUSB will possess a block control record *<2505,H>* marking block 2505 as high integrity. When Alice plugs her device into Host B, Stuxnet will again attempt to infect the device. ProvUSB will compare the access request *<w,2505,integrity(B)>* against the integrity label *<2505,H>*, and transition the block's integrity label to *<2505,L>*. When Alice plugs the device back into Host A, block 2505 will not be permitted to flow to the high integrity host. This example demonstrates that integrity protection can be enabled on *any* critical system blocks on a ProvUSB device, preventing low integrity objects from flowing to high integrity hosts, thereby quarantining sensitive machines in the network from USB-borne malware.

7. DISCUSSION

Smart Storage Devices: While low cost USB thumb drives cannot run their own operating systems, a variety of enterprise devices contain CPUs or cryptographic coprocessors, including products from IronKey [32] and Kingston [38]. We believe these devices already contain the necessary hardware to support adapted versions of ProvUSB. There are many embedded devices in the market that support full operating systems. Gumstix Thumbo [26] runs an ARM processor and embedded Linux. Intel Edison [29] and Google Project Vault [24] provide OS distributions in an SD-card form factor. USB Armory [30] embeds an ARM processor and Linux in the USB stick. Given this trend of increasingly sophisticated storage devices, we are confident that device provenance is achievable for USB storage.

Provenance Relay: With ProvUSB, provenance logging requires less than 0.1% storage overhead compared to the workload file size of a given benchmarking run. Nevertheless, the device would run out of space given enough time, depending on the storage size and the frequency of workloads. Ideally, provenance should be relayed to a trusted server using a secure channel directly by the device itself. Enterprise versions of some secure USB devices (e.g., IronKey) are already capable of communicating with such a server [31]. When networking is not viable, we rely on the owner or sysadmin of the device with root permission to export the provenance manually, making sure enough space is left for normal functionality. Unlike provenance logging, block integrity information is always kept within the device to consistently enforce policy control within the device. This space could be reserved based on the storage size, For example, a 16 GB storage with 33,554,432 blocks (assuming 512 bytes per block) needs 268 MB in total, since we require 8 bytes per block to store integrity metadata.

System Caching: As a side effect of caching in the host operating system, the timestamps for some ProvUSB provenance records do not accurately reflect the actual access time. Modern operating systems apply multi-layer caches to improve the performance of USB storage. For instance, when mounting a smaller storage partition (1 GB), we observed that the Linux kernel reads most storage into the page cache to speed up future operations. Once data is in the page cache, I/O operations for the USB storage partition occur in memory, outside of the view of the USB storage driver. When write operations occur, a dedicated kernel thread periodically flushes data to the block device (write-back). A by-product of this behavior is a reduction in benefit of ProvUSB's provenance storage optimization, shown in Figure 9, though the optimization is still useful for large storage and certain applications. Fortunately, host caching does not impact the forensic validity of block

provenance collected by ProvUSB. Provenance is eventually generated for all writes to USB storage, and for the first read to a given block after enumeration. Although repeated reads to the same block may be masked by host-level caching, it is known that the host accessed the block.

Filesystem Integrity: In order for ProvUSB devices to be mounted by high-integrity hosts, blocks containing the filesystem's metadata need to be made world read-writable. This leaves a small window of opportunity for malware propagation. If attackers discover an exploitable bug in the host's filesystem subsystem, this may allow them to infect the host and the device. Otherwise, this offers limited advantage to attackers, as writing data to the filesystem blocks would merely result in corrupting the storage partition. We are not aware of any malware that propagates through the filesystem metadata of a USB storage device.[2]

8. RELATED WORK

need for smart USB storage devices armed with advanced features has been well noted both by industry and in the literature. By running the Knoppix OS on a USB storage device, Cáceres et al. take snapshots of a running operating system that can later be resumed in another virtual machine [14]. Surie et al. use a smart USB device to measure the software stack of host machines, but not a hardware-based root of trust, and are therefore vulnerable to software attacks targeting the BIOS, bootloader or kernel [63]. McCune theorizes that host system state can be attested via USB, but does not design or implement a system [44]. Bates et al. measure timing characteristics during USB enumeration to infer characteristics of the host machine, but are unable to reliably identify specific instances of similar machines [6, 42]. By leveraging the TPM in the host machine, Butler et al. design and implement Kells [12], exposing the private partition of USB storage based on the measurement of host machines. Tian et al. propose GoodUSB [67] and USBFILTER [68] to defend against malicious USB firmware in the devices for host machines, while Cinch [3] leverages virtualization to achieve the same goal.

For traditional storage systems, Gibson et al.'s NASD adds secure capabilities to SCSI disks [23]. Strunk et al.'s self-securing storage (S4) mitigates attempts to tamper with data by internally auditing all requests from the OS [61]. Both NASD and S4 operate at the high-level objects rather than the low-level blocks directly. Pennington et al. present a storage-based intrusion detection system [53]. Unlike ProvUSB which is designed to be filesystem agnostic, Sivathanu et al.'s semantically-smart disks (SDS) leverage filesystem information to improve performance of the standard SCSI interface [60] and database management systems [59]. Butler at al. design rootkit-resistant storage by labeling certain blocks as immutable to prevent the corruption of the operating system [13], but do not record data provenance or ensure integrity of other data blocks.

Data provenance provides a means for defending against security threats. Provenance has been employed to detect compromised nodes in data centers [5, 21, 65, 74], detect data exfiltration [35, 41], enrich access controls [7, 50], and enforce regulatory compliance [7, 8]. Mechanisms that facilitate the capture of provenance have been proposed for a variety of system layers including filesystems [46, 47], operating systems [8, 22], system libraries [27], workflow engines [2, 15], and network middle boxes [5, 74], among others. Whereas past approaches such as PASS focused on a particular filesystem type (`EXT2` [47]), ProvUSB is compatible with any SCSI storage devices regardless of filesystems.

[2]However, some filesystems, e.g., NTFS, allow storing small files into metadata. In this case, the provenance of ProvUSB devices still works, though the block-level policy control may not be easily extended to cover metadata.

9. CONCLUSION

Modern USB storage devices do not support provenance, which means that there is no way to answer questions such as when and where a piece of malware infected and impacted the system. In this paper, we presented ProvUSB, which adds provenance for each I/O operation at the block level and enforces a provenance-based integrity policy for each block. ProvUSB introduces a small overhead ($< 1s$) during device enumeration, reaches 90% throughput as mean file size increases, and imposes only a small provenance storage overhead ($< 0.1\%$) in real-world workloads. To our knowledge, ProvUSB is the first provenance-aware USB storage solution that enforces data integrity within storage, working at the ubiquitous, non-circumventable block layer.

Acknowledgements

This work is supported in part by the US National Science Foundation under grant numbers CNS-1563883, CNS-1540217, and CNS-1540218, and by the Florida Center for Cybersecurity (FC^2) seed grant program.

10. REFERENCES

[1] M. Al-Zarouni. The Reality of Risks from Consented Use of USB Devices. *Edith Cowan University, Perth, W. Aus.*, 2006.

[2] I. Altintas, O. Barney, and E. Jaeger-Frank. Provenance Collection Support in the Kepler Scientific Workflow System. In *Int'l Conf. on Provenance and Annotation of Data*. 2006.

[3] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, et al. Defending against Malicious Peripherals with Cinch. In *USENIX Security Symposium*, Aug. 2016.

[4] Basemark, Inc. Basemark browsermark. http://web.basemark.com/.

[5] A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou. Let SDN Be Your Eyes: Secure Forensics in Data Center Networks. SENT'14, Feb. 2014.

[6] A. Bates, R. Leonard, H. Pruse, K. R. Butler, and D. Lowd. Leveraging USB to Establish Host Identity Using Commodity Devices. NDSS '14, February 2014.

[7] A. Bates, B. Mood, M. Valafar, and K. Butler. Towards Secure Provenance-based Access Control in Cloud Environments. CODASPY '13, 2013.

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

[9] Belkasoft, Inc. SSD Forensics 2014. https://belkasoft.com/en/ssd-2014, 2014.

[10] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, MITRE Corporation, Bedford, MA, 1973.

[11] K. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-2574, MITRE Corporation, 1975.

[12] K. Butler, S. McLaughlin, and P. McDaniel. Kells: A Protection Framework for Portable Data. In *ACSAC*, 2010.

[13] K. R. Butler, S. McLaughlin, and P. D. McDaniel. Rootkit-Resistant Disks. In *ACM CCS*, 2008.

[14] R. Cáceres, C. Carter, C. Narayanaswami, and M. Raghunath. Reincarnating PCS with Portable Soulpads. In *MobiSys*, 2005.

[15] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. VisTrails: Visualization Meets Data Management. In *SIGMOD*, 2006.

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

[17] W. W. W. Consortium et al. Prov-overview: an overview of the prov family of documents. 2013.

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

[19] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the IEEE Symposium on Security & Privacy*, 2000.

[20] T. Fraser. LOMAC: MAC You Can Live With. In *USENIX ATC*, 2001.

[21] A. Gehani, B. Baig, S. Mahmood, D. Tariq, and F. Zaffar. Fine-grained Tracking of Grid Infections. In *IEEE/ACM GRID*, Oct 2010.

[22] A. Gehani and D. Tariq. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proceedings of the 13th International Middleware Conference*, 2012.

[23] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, and et al. A Cost-Effective, High-Bandwidth Storage Architecture. In *ACM ASPLOS*, 1998.

[24] Google, Inc. ProjectVault. https://github.com/ProjectVault.

[25] Gumstix, Inc. COMS. https://store.gumstix.com/coms.html.

[26] Gumstix, Inc. Thumbo. https://store.gumstix.com/thumbo.html.

[27] R. Hasan, R. Sion, and M. Winslett. The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance. In *USENIX FAST*, 2009.

[28] Infineon, Inc. Embedded TPM. http://www.infineon.com/cms/en/product/channel.html?channel=db3a30434422e00e01442555a5f713f5.

[29] Intel, Inc. Intel Edison. http://www.intel.com/content/www/us/en/do-it-yourself/edison.html.

[30] Inverse Path, Inc. USB Armory. https://inversepath.com/usbarmory.

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

[32] IronKey, Inc. IronKey Enterprise S1000 Encrypted Flash Drive. http://www.ironkey.com/en-US/encrypted-storage-drives/s1000-enterprise.html.

[33] T. Jaeger, R. Sailer, and U. Shankar. Prima: policy-reduced integrity measurement architecture. In *ACM SACMAT*, 2006.

[34] Jaikumar Vijayan. Infected USB drive blamed for '08 military cyber breach. http://www.computerworld.com/article/2514879/security0/infected-usb-drive-blamed-for--08-military-cyber-breach.html, 2008.

[35] S. N. Jones, C. R. Strong, D. D. E. Long, and E. L. Miller. Tracking Emigrant Data via Transient Provenance. In *3rd Workshop on the Theory and Practice of Provenance*, TAPP'11, June 2011.

[36] B. Kauer. OSLO: Improving the Security of Trusted Computing. In *USENIX Security Symposium*, 2007.

[37] Kevin Poulsen and Kim Zetter. U.S. Intelligence Analyst Arrested in Wikileaks Video Probe. http://www.wired.com/2010/06/leak/.

[38] Kingston, Inc. DataTraveler USB Drives. http://www.kingston.com/us/usb.

[39] T. Kojm. Clamav. http://www.clamav.net/.

[40] K. H. Lee, X. Zhang, and D. Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. NDSS, 2013.

[41] K. H. Lee, X. Zhang, and D. Xu. LogGC: Garbage Collecting Audit Log. CCS, Nov. 2013.

[42] L. Letaw, J. Pletcher, and K. Butler. Host Identification via USB Fingerprinting. In *IEEE SADFE*, 2011.

[43] S. Ma, X. Zhang, and D. Xu. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *ISOC NDSS*, 2016.

[44] J. M. McCune. Turtles All the Way Down: Research Challenges in User-based Attestation. In *USENIX Workshop on Recent Advances on Intrusion-tolerant Systems*, 2008.

[45] Michael K. Daly. The Advanced Persistent Threat. In *USENIX LISA*, 2009.

[46] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, and et al. Layering in Provenance Systems. In *USENIX ATC*, 2009.

[47] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware Storage Systems. In *USENIX ATC*, 2006.

[48] D. Nguyen, J. Park, and R. Sandhu. Dependency Path Patterns As the Foundation of Access Control in Provenance-aware Systems. In *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance*, TaPP'12, 2012.

[49] OLEA Kiosks, Inc. Malware Scrubbing Cyber Security Kiosk. http://www.olea.com/product/cyber-security-kiosk/.

[50] J. Park, D. Nguyen, and R. Sandhu. A Provenance-Based Access Control Model. In *Proc. Int'l Conf. on Privacy, Security & Trust (PST)*, 2012.

[51] B. Parno. Bootstrapping Trust in a "Trusted" Platform. In *Proc. Workshop on Hot Topics in Security*, 2008.

[52] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping Trust in Commodity Computers. In *Proc. IEEE Symposium on Security and Privacy*, 2010.

[53] A. G. Pennington, J. D. Strunk, J. L. Griffin, C. A. N. Soules, G. R. Goodson, and et al. Storage-based Intrusion Detection: Watching Storage Activity for Suspicious Behavior. In *USENIX Security Symposium*, 2003.

[54] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler. Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In *ACSAC*, 2012.

[55] J. Rrushi, H. Farhangi, C. Howey, K. Carmichael, and J. Dabell. A quantitative evaluation of the target selection of havex ics malware plugin. *Industrial Control System Security (ICSS) Workshop*, 2015.

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

[57] S. Shin and G. Gu. Conficker and Beyond: A Large-scale Empirical Study. ACSAC '10, 2010.

[58] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring data integrity in storage: Techniques and applications. In *Proceedings of the 2005 ACM workshop on Storage security and survivability*, 2005.

[59] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Database-Aware Semantically-Smart Storage. In *USENIX FAST*, 2005.

[60] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *USENIX FAST*, 2003.

[61] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. Soules, and G. R. Ganger. Self-securing Storage: Protecting Data in Compromised System. In *USENIX OSDI*, 2000.

[62] Sun Microsystems, Inc. and FSL at Stony Brook University. Filebench. http://filebench.sourceforge.net.

[63] A. Surie, A. Perrig, M. Satyanarayanan, and D. J. Farber. Rapid Trust Establishment for Pervasive Personal Computing. *Pervasive Computing, IEEE*, 6(4):24–30, 2007.

[64] Symantec, Inc. Trojan.Zbot. http://www.symantec.com/security_response/writeup.jsp?docid=2010-011016-3514-99.

[65] D. Tariq, B. Baig, A. Gehani, S. Mahmood, R. Tahir, A. Aqil, and F. Zaffar. Identifying the Provenance of Correlated Anomalies. In *ACM SAC*, 2011.

[66] The USB Device Working Group. USB Class Codes. http://www.usb.org/developers/defined_class, 2015.

[67] D. J. Tian, A. Bates, and K. Butler. Defending Against Malicious USB Firmware with GoodUSB. In *ACSAC*, 2015.

[68] D. J. Tian, N. Scaife, A. Bates, K. Butler, and P. Traynor. Making USB Great Again with USBFILTER. In *USENIX Security Symposium*, 2016.

[69] TrouSerS. The open-source TCG Software Stack. http://trousers.sourceforge.net/.

[70] J. Tucek, P. Stanton, E. Haubert, R. Hasan, et al. Trade-offs in Protecting Storage: A Meta-Data Comparison of Cryptographic, Backup/Versioning, Immutable/Tamper-Proof, and Redundant Storage Solutions. In *IEEE MSST*, 2005.

[71] USB Implementers Forum, Inc. USB On-The-Go and Embedded Host. http://www.usb.org/developers/onthego/.

[72] USB Implementers Forum, Inc. USB Mass Storage Class CBI Transport. http://www.usb.org/developers/docs/devclass_docs/usb_msc_cbi_1.1.pdf, 2003.

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

[74] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure Network Provenance. In *SOSP*, 2011.