Kells: A Protection Framework for Portable Data

Kevin R.B. Butler
Department of Computer & Information Science
University of Oregon, Eugene, OR
butler@cs.uoregon.edu

Stephen E. McLaughlin and
Patrick D. McDaniel
Systems and Internet Infrastructure Security
Laboratory (SIIS)
Penn State University, University Park, PA
{smclaugh,mcdaniel}@cse.psu.edu

ABSTRACT

Portable storage devices, such as key-chain USB devices, are ubiquitous. These devices are often used with impunity, with users repeatedly using the same storage device in open computer laboratories, Internet cafes, and on office and home computers. Consequently, they are the target of malware that exploit the data present or use them as a means to propagate malicious software.This paper presents the Kells mobile storage system. Kells limits untrusted or unknown systems from accessing sensitive data by continuously validating the accessing host's integrity state. We explore the design and operation of Kells, and implement a proof-of-concept USB 2.0 storage device on experimental hardware. Our analysis of Kells is twofold. We first prove the security of device operation (within a freshness security parameter Δ_t) using the LS^2 logic of secure systems. Second, we empirically evaluate the performance of Kells. These experiments indicate nominal overheads associated with host validation , showing a worst case throughput overhead of 1.22% for read operations and 2.78% for writes.

1. INTRODUCTION

Recent advances in materials and memory systems have irreversibly changed the storage landscape. Small form factor portable storage devices housing previously unimaginable capacities are now commonplace today–supporting sizes up to a quarter of a terabyte [15]. Such devices change how we store our data; single keychain devices can simultaneously hold decades of personal email, millions of documents, thousands of songs, and many virtual machine images. These devices are convenient, as we can carry the artifacts of our digital lives wherever we go.

The casual use of mobile storage has a darker implication. Users plugging their storage devices into untrusted hosts are subject to data loss [16] or corruption. Compromised hosts have unfettered access to the storage plugged into their interfaces, and therefore have free rein to extract or modify its contents. Users face this risk when accessing a friend's computer, using a hotel's business office, in university computer laboratories, or in Internet cafes. The risks here are real. Much like the floppy disk-borne viruses in the 1980's and 90's, malware like Conficker [22] and Agent.bz [32] exploit mobile storage to propagate malicious code. The compromise of hosts throughout military networks, due to malware propagated by rogue portable storage devices, has already led to a ban of their use by the US Department of Defense [28]. The underlying security problem is age-old: users cannot ascertain how secure the computer they are using to access their data is. As a result, all of their information is potentially at risk if the system is compromised. This paper attempts to address this conundrum by responding to the following challenge: *How can we verify that the computer we are attaching our portable storage to is safe to use?*

Storage security has recently become an active area of investigation. Solutions such as full-disk encryption [25] and Microsoft's BitLocker to Go [18] require that the user supply a secret to access stored data. This addresses the problem of device loss or theft, but does not aid the user when the host to which it is to be attached is itself untrustworthy. Conversely, BitLocker (for fixed disks) uses a trusted platform module (TPM) [36] to seal a disk partition to the integrity state of the host, thereby ensuring that the data is safeguarded from compromised hosts. This is not viable for mobile storage, as data is bound to the single physical host. In another effort, the Trusted Computing Group (TCG) has considered methods of authenticating storage to the host through the Opal protocol [37] such as pre-boot authentication and range encryption and locking for access control. These services may act in a complementary manner to our solution for protecting mobile storage from potentially compromised hosts.

In this paper, we introduce *Kells*[1], an intelligent USB storage device that validates host integrity prior to allowing read/write access to its contents, and thereafter only if the host can provide ongoing evidence of its integrity state. When initially plugged into an untrusted device, Kells performs a series of *attestations* with trusted hardware on the host, repeated periodically to ensure that the host's integrity state remains good. Kells uses integrity measurement to ascertain the state of the system and the software running on it at boot time in order to determine whether it presents a safe platform for exposing data. If the platform is deemed to be trustworthy then a trusted storage partition will be exposed to the user; otherwise, depending on a configurable policy, the device will either mount only a "public" partition with untrusted files exposed or will not mount at all. If at any time the device cannot determine the host's integrity state or the state becomes undesirable, the protected partition becomes inaccessible. Kells can thus ensure the integrity of data on a trusted storage partition by ensuring that data can only be written to it from high-integrity, uncompromised systems. Our design uses the commodity Trusted Platform Module (TPM) found in the majority of modern computers as our source for trusted hard-

[1]The Book of Kells is traceable to the 12^{th} century Abbey of Kells, Ireland due to information on land charters written into it.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

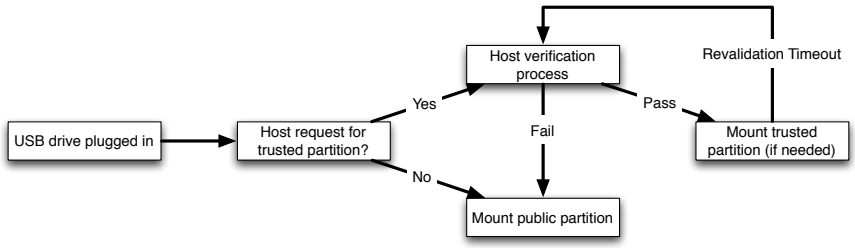

Figure 1: Overview of the Kells system operation. Attestations of system state are required to be received successfully by the device in order for the integrity of the host to be proved, a necessary precondition for allowing data to be available to the host.

ware, and our implementation and analysis use it as a component. *We note, however, that it is not integral to the design: any host integrity measurement solution (e.g., Pioneer [26]) can be used.*

Kells diverges substantially from past attempts at securing fixed and mobile storage. In using the mobile storage device as an autonomous trusted computing base (TCB), we extend the notion of self-protecting storage [3, 11, 21] to encompass a system that actively vets the devices that make use of it. A Kells device is active in order to be able to make these policy decisions. While this is a change from the passive USB devices often currently employed, we note that an increasing class of storage devices include processing elements such as cryptographic ASICs. We thus provide a path to enjoying the convenience of now-ubiquitous portable storage in a safe manner. Our contributions are as follows:

- We identify system designs and protocols that support portable storage device validation of an untrusted host's initial and ongoing integrity state. To the best of our knowledge, this is the first use of such a system by a dedicated portable storage device.
- We reason about the security properties of Kells using the LS^2 logic [6], and prove that the storage can only be accessed by hosts whose integrity state is valid (within a security parameter Δ_t).
- We describe and benchmark our proof of concept Kells system built on a DevKit 8000 board running embedded Linux and connected to a modified Linux host. We empirically evaluate the performance of the Kells device. These experiments indicate that the overheads associated with host validation are minimal, showing a worst case throughput overhead of 1.22% for read operations and 2.78% for writes.

We begin the description of Kells by providing a broad overview of its goals, security model, and operation.

2. OVERVIEW

Figure 1 illustrates the operation of Kells. Once a device is inserted, the host may request a public or trusted partition. If a trusted partition is requested, the host and Kells device perform an attestation-based exchange that validates host integrity. If this fails, the host will be permitted to mount the public partition, if any exists. If the validation process is successful, the host is allowed access to the trusted partition. The process is executed periodically to ensure the system remains in a valid state. The frequency of the re-validation process is determined by the Kells policy.

2.1 Operational Modes

There are two modes of operation for Kells, depending on how much control over device administration should be available to the user and how much interaction he should have with the device. We review these below:

Transparent Mode.

In this mode of operation, the device requires no input from the user. The host verification process executes immediately after the device is inserted into the USB interface. If the process succeeds, the device may be used in a trusted manner as described above, i.e., the device will mount with the trusted partition available to the user. If the attestation process is unsuccessful, then depending on the reason for the failure (e.g., because the host does not contain a TPM or contains one that is unknown to the device), the public partition on the device can be made available. Alternately, the device can be rendered unmountable altogether. A visual indicator on the device such as an LED can allow the user to know whether the host is trusted or not: a green light may indicate a good state while a flashing red light indicates an unknown or untrusted host.

User-Defined Mode.

The second mode of operation provides the user with a more active role in making storage available. When the Kells device is inserted into the system, prior to the attestation taking place, a partition containing user-executable programs is made available. One is a program prompting the user to choose whether to run the device in trusted or public mode. If the user chooses to operate in trusted mode, then the attestation protocol is performed, while if public mode is chosen, no attestations occur. In this manner, the user can make the decision to access either partition, with further policy that may be applied on trusted hosts opening untrusted partitions, to prevent potential malware infection. These hosts may quarantine the public partition, requiring a partition scan prior to allowing access. Such a scan can also be performed by the device. Such a scenario could be useful if there is a need or desire to access specific media (e.g., photographs, songs) from the public partition of the disk while using a trusted host, without having to mark the information as trusted. Trusted partitions on a Kells device are unlikely to be infected to begin with, on account of any host using this partition having to attest its integrity state. This is essential, since a user would not be hesitant to load or execute content from a partition that is considered trusted.

Note that the policies described above are but two examples of the methods of operation available with this infrastructure. For simplicity, we have described the coarse-grained granularity of trusted and public partitions. Within the trusted parition, however, further fine-grained policy can be enforced depending on the identified host; for example, blocks within the partition may be labeled depending on the host writing to them, with a data structure keep-

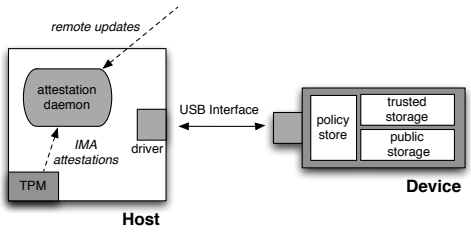

Figure 2: Overview of the Kells architecture.

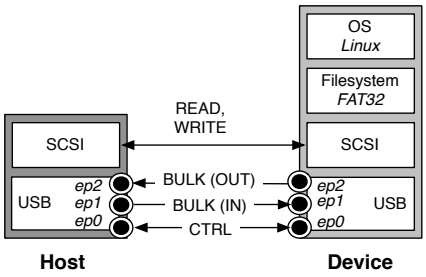

Figure 3: Overview of USB operation with an embedded Linux mass storage device, or *gadget*.

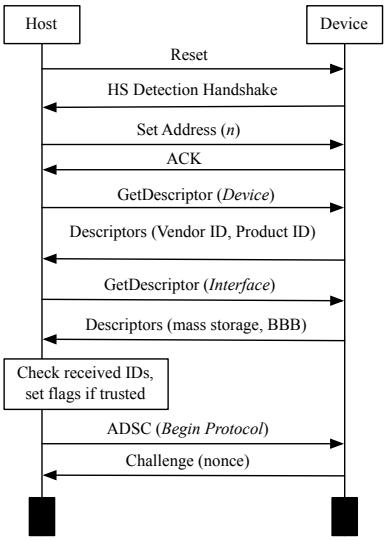

Figure 4: Sample USB setup between a host and the Kells device.

ing track of the labels and access controls to data (e.g., encrypting labeled data and only decrypting based on the host having access to this label, as specified by device policy).

2.2 Threat Model

We assume the adversary is capable of subverting a host operating system at any time. While we do not specifically address physical attacks against the Kells device, such as opening the drive enclosure to manipulate the physical storage media or modifying the device tick-counter clock, we note that defenses against these attacks have been implemented by device manufacturers. Notably, portable storage devices from IronKey [1] contain significant physical tamper resistance with epoxy encasing the chips on the device, electromagnetic shielding of the cryptographic processor, and a waterproof enclosure. SanDisk's Cruzer Enterprise [24] contains a secure area for encryption keys that is sealed with epoxy glue. Tamper-resistance has also been considered for solutions such as the IBM Zurich Trusted Information Channel [38]. Such solutions would be an appropriate method of defense for Kells. In addition, we assume that any reset of the hardware is detectable by the device (for example, by detecting voltages changes on the USB bus and receiving cleared PCR values from the TPM).

Kells does not in itself provide protection for the host's internal storage, though an adaptation of our design can be used to provide a similar protection mechanism, as with the Firma storage-rooted secure boot system [2]). Integrity-based solutions exist that protect the host's internal storage (hard disks), including storage-based intrusion detection [21] and rootkit-resistant disks [3]. As is common in these systems, we declare physical attacks against the host's TPM outside the scope of this work. As previously discussed, the TPM is used as an implementation point within our architecture and other solutions for providing host-based integrity measurement may be used. As a result, we do not make any attempt to solve the many limitations of TPM usage in our solution. Additionally, we do not consider the issue of devices attesting their state to the host. The TCG's Opal protocol [37] includes provisions for trusted peripherals, addressing the issue by requiring devices to contain TPMs. Software-based attestation mechanisms such as SWATT [27], which does not require additional trusted hardware, may also be used. Finally, we rely on system administrators to provide accurate measurements of their systems, which must be updated if there are changes (e.g., due to configuration or updates). Without updates, Kells will not be able to provide access to the trusted partitions of these systems.

3. DESIGN AND IMPLEMENTATION

We now turn to our design of the Kells architecture, shown in Figure 2, and describe details of its implementation. There are three major components of the system where modifications are necessary: the interface between the host and the device, the storage device itself, and the host's operating system.

3.1 USB Interface

We begin by describing the basics of USB operation in order to aid in understanding the design of Kells. This is a brief overview; more details may be found in the appendix.

The basic USB mass storage device stack is shown in Figure 3. At the USB layer are *endpoints*, the means by which USB commands are send and received. USB mass storage devices primarily use *bulk* transfers, a best-effort service, to transmit information, but every device also supports the flow of *control* transfers over endpoint 0. Above the USB layer is a SCSI emulation layer supporting a limited subset of SCSI commands, such as reads and writes.

Within operating systems that support USB, such as Linux, the number and functionality of supported devices is large and diverse. To support devices (or *gadgets*) that do not conform to the USB specificaton, Linux contains a repository for these devices and sets flags when they to modify the host behavior, in order to correctly operate with these devices.

USB is a master-slave protocol, meaning that all commands must be initiated by the host. This model is conceptually at odds with a device such as Kells, which independently enforces security policy. Therefore, we reconsider how the device interacts with the host.

Figure 4 gives an abridged overview of the device setup process at the USB layer. As with any USB device, high-speed detection and bus addressing is performed before information is requested by the host. The host requests the device descriptor, which includes information such as the device's vendor and product ID, as well as

a unique serial number. When the host requests the interface descriptor, the Kells device identifies itself as a mass storage device that operates in the typical fashion of only performing bulk transfers. The host will set flags accordingly in order to send the correct commands to the device.

Almost every USB mass storage device performs its operations using only bulk transfers. However, we use control transfers for sending trusted commands to Kells. Control transfers reserve a portion of USB bus bandwidth, ensuring that information is tranferred as quickly as possible. If a Kells device is plugged into a host that does not support attestation operations, the host will access the public partition as a standard mass storage device, oblivious to the trusted protocols and storage. If the host recognizes the device as trusted, it will send an Accept Device-Specific Command (ADSC). The setup phase of the command allows the host to initiate the attestation protocol, while the attestation information is sent through the data stage, and the gadget sets a response code that includes a challenge. Further stages of the attestation protocol continue as control transfers between the host and device, and all other read and write operations are suspended until the protocol completes. The attestation protocol is described in detail in Section 4.1.

3.2 Designing the Storage Device

Kells requires the ability to perform policy decisions independent of the host. As a result, logic must execute on these devices, which require a means of receiving command transfers from the host and to use these for making the correct access decisions.

The basic architecture for the storage device is an extension to the Linux USB gadget stack, along with a user-space daemon that is in charge of policy decisions and accessing other important information. Within the kernel, we added new functionality that allows the device to receive special control transfers from the host. These are exported to user space through the *sysfs* interface, where they are read as strings by the daemon tasked with marshaling this data.

When plugged in, the daemon on the device sets a timer (as USB devices contain a crystal oscillator for driving clock signals), and waits to determine whether the host presents the proper credentials. The device presents itself to the host as a typical mass storage device operating in bulk-only mode, differentiating itself with the vendor ID. We use the vendor ID `b000` which has not been currently allocated by the USB Forum as of June 2010.[2]

If an ADSC command containing authenticating information from the host is not received within this time period, operation on the device defaults to public operation. If the device is configured such that the policy does not allow any partitions to be mounted, the device will not present any further information to the host. If the protocol fails, the failure is logged in the storage device's audit log, which is unexposed to the host. Depending on the defined policy, either the public partition will be exposed or no partitions on the device will be mounted at all. If the protocol is successful and the host attests its state to the device, the daemon presents the trusted partition to be mounted, by performing an `insmod()` command to link the correct backing store with the gadget driver.

Within the Kells device is a policy store, which contains information on every known host, its *measurement database* to compare attestations against, and policy details, such as whether the host is authenticated as an administrative console and whether the host should expose a public partition if the attestation check fails. Optionally, the device can also store information on users credentials supplied directly to the device through methods such as biometrics. Configured policy can allow or disallow the device to be plugged into specific machines.

[2]Because this is a proof of concept design and implementation, we have not registered a vendor ID with the USB Forum yet; however, based on our results, we may consider doing so.

3.3 Modifications to Host

A host must be capable of recognizing that the Kells device is trusted and sending information to it differs from a standard USB mass storage transaction. Our goal was to require minimal changes to the host for operation, but because we are working at the USB layer, some changes are necessary to the USB driver. We define a flag `IS_TRUSTED` in the Linux *unusual_devs.h* device repository, letting the host know that the device accepts control transfers.

Because the host must interact with its trusted hardware and perform some logic, we designed an *attestation daemon* that runs in the host's user space. The attestation daemon both retrieves boot-time attestations using the Linux Integrity Measurement Architecture (IMA) [23] and can act as an interface to any runtime monitoring systems on the host (see Section 4.2). It can also provide an interface for receiving third-party updates (see Section 4.3).

4. ATTESTATIONS AND ADMINISTRATION

A key consideration with Kells is managing metadata and credential information in a manner that maintains usability and simplicity of the device. We describe in this section details of how this management occurs.

4.1 Attesting Host Integrity

In order for a host connecting to the Kells device to be trustworthy, it must be installed and maintained in a manner that protects its integrity. A way of ensuring this is through the provisioning of a secure kernel and supporting operating system, from which measurements of system integrity can be made and transferred to the Kells device. The maintainer of the host system is thus required to re-measure the system when it is installed or when measurable components are updated. Solutions for ensuring a trusted base installation include the use of a root of trust installer (ROTI) [33], which establishes a system whose integrity can be traced back to the installation media.

The system performing the installation must contain trusted hardware such as a TPM. Every TPM contains an *endorsement key* (EK), a 2048-bit RSA public/private key pair created when the chip is manufactured. This provides us with a basis for establishing the TPM's unique identity, essential to verifying the installation. The stages of this initial installation are as follows:

1. The installation media is loaded into the installer system, which contains a TPM. This system needs to be trusted, i.e., the hardware and system BIOS cannot be subverted at this time.[3] As described below, the system's *core root of trust for measurement* (CRTM), containing the boot block code for the BIOS, provides a self-measurement attesting this state.

2. A measurement of each stage of the boot process is taken. Files critical to the boot process are hashed, and the list of hashes kept in a file that is *sealed* (i.e., encrypted) by the TPM of the installing system. This process links the installing TPM with the installed code and the filesystem. A Kells device in *measurement mode* can record the measurements from the system, or this can be performed in another manner and transferred to the device at a later time, through placement of the list of hashes in a secure repository.

[3]This restriction is not necessary after installation, as malicious changes to the system state will be measured by the CRTM.

We first identify the host's TPM. While the EK is unique to the TPM, there are privacy concerns with exposing it. Instead, an *attestation identity key* (AIK) public/private key pair is generated as an alias for the EK, and strictly used for signatures. However, the AIK is stored in volatile memory. Therefore, both the public and private AIKs must be stored. The TPM provides the *storage root key* (SRK) pair for encrypting keys stored outside the TPM. Thus, the SRK encrypts the private AIK before it is sent to the device. Formally, the set of operations occurs as follows. Given a host's TPM H and a device D, the following protocol flow describes the initial pairing of the host to the device and the initial boot:

Pairing

(1) H : generate AIK = (AIK^+, AIK^-)

(2) $H \rightarrow D : AIK^+, \{AIK^-\}_{SRK^-}$

Measurement

(3) $D \rightarrow H : \{AIK^-\}_{SRK^-}$

(4) $D : n =$ Generate nonce

(5) $D \rightarrow H : Challenge(n)$

(6) $H \rightarrow D : Attestation = Quote + ML$

(7) $D : Validate(Quote, ML)_{AIK+}$

Steps 1 and 2 occur when the host has been initially configured or directly after an upgrade operation, to either the hardware or to files that are measured by the IMA process. Subsequent attestations use this list of measurements, which may also be disseminated back to the administrator and stored with the AIK information so as to allow for remote updates, discussed further in Section 4.3.

The following states are measured in order: (a) the core root of trust for measurement (CRTM), (b) the system BIOS, (c) the bootloader (e.g., GRUB) and its configuration, and (d) the OS. Measurements are made by with the TPM's *extend* operation, which hashes code and/or data, concatenates the result with the previous operation, and stores the result in the TPM's *Platform Configuration Registers* (PCRs). The *quote* operation takes the challenger's nonce n and returns a signature of the form $Sign(PCR, N)_{AIK^-}$, when the PCRs and n are signed by the private AIK. The measurement list (ML), which contains a log of all measurements sent to the TPM, is also included.

The above protocol describes a *static* root of trust for measurement, or SRTM. There are some disadvantages to this approach, since the BIOS must be measured and any changes in hardware require a new measurement; additionally, it may be susceptible to the TPM reset attack proposed by Kauer [13]. Another approach is to use a *dynamic* root of trust for measurement (DRTM), which allows for a *late launch*, or initialization from a secure loader after the BIOS has loaded, so that it does not become part of the measurement. SRTM may be vulnerable to code modification if DRTM is supported on the same device [6]. DRTM may also be potentially vulnerable to attack; the Intel TXT extensions supporting DRTM may be susceptible to System Management Mode on the processor being compromised before late launch is executed, such that it becomes part of the trusted boot and is not again measured [39]. For this reason, it is an administrative decision as to which measurement mode the system administrator should use for their system, but we can support either approach with Kells.

Note that we are directly connecting with the host through the physical USB interface. The cuckoo attack described by Parno [20] may be mitigated by turning off network connectivity during the boot-time attestation process, such that no remote TPMs can answer in place of the host. However, if the host can access an oracle that presents TPM-like answers, a means to uniquely identify the host is necessary. We are actively investigating these methods.

4.2 Managing Runtime Integrity Attestations

```
1: (att, t) ← read.RAM.att
2: if |req.time − t| < Δ_t ∧ GoodAtt(att) then
3:     Perform the write req as usual.
4: else
5:     if WriteBuffer.notFull() then
6:         Buffer the request for later write back once a fresh attestation
           is received.
7:     else
8:         Stall until there is space in the write buffer.
9:     end if
10: end if
```

Figure 5: Write(req) algorithm.

```
1: (att, t) ← read.RAM.att
2: if GoodAtt(att) then
3:     for Requests buffered before t do
4:         Perform the write req as usual.
5:     end for
6: end if
```

Figure 6: Commit() algorithm.

```
1: (att, t) ← read.RAM.att
2: if |req.time − t| < Δ_t ∧ GoodAtt(att) then
3:     Perform the read req as usual.
4: else
5:     Stall until a fresh attestation is received.
6: end if
```

Figure 7: Read(req) algorithm.

To perform authentication of the host, the Kells device must compare received attestations with a known set of good values. A portion of non-volatile memory is used for recording this information, which includes a unique identity for the host (e.g., the public AIK) the host's measurement list, and policy-specific information, (e.g., should the host allow administrative access).

We provide a framework for supporting runtime integrity monitoring, but we do not impose constraints on what system is to be used. The runtime monitor can provide information to the storage device as to the state of the system, with responses that represent good and bad system states listed as part of the host policy. Our design considers attestations from a runtime monitor to be delivered in a consistent, periodic manner; one may think of them as representing a *security heartbeat*. The period of the heartbeat is fixed by the device and transmitted to the host as part of the device enumeration process, when other parameters are configured.

Because the device cannot initiate queries to the host, it is incumbent on the host to issue a new attestation before the validity period expires for the existing one. The Kells device can issue a warning to the host a short time period λ before the attestation period Δ_t expires, in case the host neglects to send the new attestation.

Algorithms 5 and 6 describe the write behavior on the device. We have implemented a buffer for writes that we term a *quarantine buffer*, to preserve the integrity of data on the Kells device. Writes are not directly written to the device's storage but are stored in the buffer until an attestation arrives from the host to demonstrate that the host is in a good state. Once a successful attestation arrives, the buffer is cleared, but if a failed attestation arrives and access to

the trusted partition is revoked, any information in the write buffer at that time will be discarded. In a similar manner, Algorithm 7 describes the semantics of the read operation. Reads occur as normal unless an attestation has not been received within time Δ_t. If this occurs, then further read requests will be prevented until a new successful attestation has been received.

To prevent replay, the host must first explicitly notify Kells that the attestation process is beginning in order to receive a nonce, which is used to attest to the freshness of the resulting runtime attestation (i.e., as a MAC tied to the received message).

4.3 Remote Administration

An additional program running on the host (and measured by the Kells device) allows for the device to remotely update its list of measured hosts. This program starts an SSL session between the running host and a remote server in order to receive new policy information, such as updated measurements and potential host revocations. The content is encrypted by the device's public key, the keypair of which is generated when the device is initialized by the administrator, and signed by the remote server's private key.

Recent solutions have shown that in addition to securing the transport, the integrity state of the remote server delivering the content can be attested [19]. It is thus possible for the device to request the attestation proof from the remote administrator prior to applying the received policy updates.

In order for the device to receive these updates, the device exposes a special administrative partition if an update is available, signaled to do so by the attestation daemon. The user can then move the downloaded update file into the partition, and the device will read and parse the file, appending or replacing records within the policy store as appropriate. Such operations include the addition of new hosts or revocation of existing ones, and updates of metadata such as measurement lists that have changed on account of host upgrades. This partition contains only one other file: the audit failure log is encrypted with the remote server's public key and signed by the device, and the user can then use the updater program to send this file to the remote server. The server processes these results, which can be used to determine whether deployed hosts have been compromised.

5. REASONING ABOUT ATTESTATIONS

We now prove that the Kells design achieves its goal of protecting data from untrusted hosts. This is done using the logic of secure systems (LS^2) as described by Datta et al. in [6]. Using LS^2, we describe two properties, (**SEC**) and (**INT**), and prove that they are maintained by Kells. These two properties assert that the confidentiality and integrity of data on the Kells device are protected in the face of an untrusted host. To prove that Kells enforces the two properties, we first encode the Kells read and write operations from section 4.2 into the special programming language used by LS^2. These encodings are then mapped into LS^2 and shown to maintain both properties. Both properties are stated informally as follows.

1. (**SEC**) Any read request completed by Kells was made while the host was in a known good state. This means that an attestation was received within a time window of Δ_t from the request or after the request without a host reboot.

2. (**INT**) Any write request completed by Kells was made while the host was in a known good state with the same respect to Δ_t as read.

5.1 Logic of Secure Systems

The logic of secure systems (LS^2) provides a means for reasoning about the security properties of programs. This reasoning allows the current state of a system to be used to assert properties regarding how it got to that state. In the original work, this was used to show that given an integrity measurement from a remote host, the history of programs loaded and executed can be verified. In the case of Kells, we use such a measurement to make assertions about the reads and writes between the host system and Kells storage device, namely, that (**SEC**) and (**INT**) hold for all reads and writes. LS^2 consists of two parts: a programming language used to model real systems, and the logic used to prove properties about the behavior of programs written in the language. This section begins with a description of the language used by LS^2, followed by a description of the logic and proof system.

LS^2 uses a simple programming language, hereafter referred to as "the language," to encode real programs. Any property provable using LS^2 holds for all execution traces of all programs written in the language. Our aim is to encode Kells operation in the language and formally state and prove its security properties using LS^2. The main limitation of the language (and what makes it feasible to use for the verification of security properties) is the lack of support for control logic such as if-then-else statements and loops. Expressions in the language resolve to one of a number of data types including numbers, variables, and cryptographic keys and signatures. For Kells operation, we use numeric values as timestamps (t) and data (n), and pairs of these to represent data structures for attestations and block requests. The expressions used for encoding values in Kells is shown in Table 1.

The language encapsulates operations into *actions*, single instructions for modeling system-call level behavior. Program traces are sequences of actions. There are actions for communication between threads using both shared memory and message passing. In the case of shared memory, `read` l and `write` l, e signify the reading and writing of an expression e to a memory location l. As Kells adds security checks into these two operations, we introduce language extensions `sread` req, att and `swrite` req, att, which are covered in the following section. Finally, the actions `send` req and `receive` are used to model communication with the host ($\mathcal{H}$) by the Kells device ($\mathcal{D}$).

Moving from the language to the logic proper, LS^2 uses a set of logical predicates as a basis for reasoning about programs in the language. There are two kinds of predicates in LS^2, *action predicates* and *general predicates*. Action predicates are true if the specified action is found in a program trace. Furthermore, they may be defined at a specific time in a program's execution, e.g. $\mathsf{Send}(\mathcal{D}, req)$ @ t holds if the thread $\mathcal{D}$ send the results of the request req to the host at time t. See the predicates in Table 1. General predicates are defined for different system states either at an instant of time or over a period. One example of such a predicate is $\mathsf{GoodState}(\mathcal{H}, (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig))$, which we defined to show that the host system is in a good state with respect to a particular block request. The exact definition of GoodState is given in the following section.

5.2 Verification of Kells Security Properties

We verify that Kells operations maintain the (**SEC**) and (**INT**) properties in several steps. First, we rewrite the algorithms described in section 4.2 using the above described language. This includes a description about assumptions concerning the characteristics of the underlying hardware and an extension of the language to support the write queueing mechanism, along with the operational semantics of these expressions as shown in Figure 8. We then formally

Table 1: The subset of LS^2 and extensions used to evaluate the Kells security properties.

Expressions

Expression	*Use in Validation*
$att = (\mathtt{t_{att}}, sig)$	An attestation consisting of wall clock arrival time $\mathtt{t_{att}}$, and a signature, sig.
$req = (t, \mathtt{t_{req}}, (l, n))$	A block request consisting of a stored program counter t, a wall clock time $\mathtt{t_{req}}$, a disk location l and a value n.

Language Features (* indicates an extension)

Feature	*Use in Validation*
`send` req	Send the result of request req from Kells to the host.
`receive`	Receive a value from the host.
$\mathtt{proj}_1\ e$	Project the first expression in the pair resulting from e. $\mathtt{proj}_2\ e$ projects the second expression.
*`enqueue` req	Enqueue the request req in the Kells request queue.
*`peek`	Peek at the item at the head of the Kells device's write request queue. If the queue is empty, halt the current thread immediately.
*`dequeue`	Dequeue a block request from the Kells request buffer.
*`sread` req, att	Perform a secure (attested) read.
*`swrite` req, att	Perform a secure (attested) write.

Predicates (* indicates an extension)

Predicate	*Use in Validation*
$\mathsf{Send}(\mathcal{D}, req)\ @\ t$	The Kells disk controller ($\mathcal{D}$) sent the result of request req to the host at time t.
$\mathsf{Recv}(\mathcal{D}, req)\ @\ t$	The Kells disk controller ($\mathcal{D}$) received the request req from the host at time t.
$\mathsf{Reset}(\mathcal{H})\ @\ t$	Some thread on the host machine ($\mathcal{H}$) restarted the system at time t.
$^*\mathsf{Peek}(\mathcal{D})\ @\ t$	The Kells disk controller ($\mathcal{D}$) peeked at the tail of the request queue at time t.
$^*\mathsf{SRead}(req, att)$	`sread` was executed in the program trace.
$^*\mathsf{SWrite}(req, att)$	`swrite` was executed in the program trace.
$^*\mathsf{Fresh}(t, \mathtt{t_{req}}, \mathtt{t_{att}})$	The attestation received at time $\mathtt{t_{att}}$ was received recently enough to be considered fresh w.r.t. a request that arrived at $\mathtt{t_{req}}$.
$^*\mathsf{GoodState}(\mathcal{H}, req, att)$	The host ($\mathcal{H}$) attested a good state w.r.t. the request req. Meaning that the host was in a good state when the request was received.

Configuration (* indicates an extension)

Configuration	*Use in Validation*
σ	The store map of $[location \mapsto expression]$. This is used in the semantics of `read` and `write` as well as the write request queue.
$^*(h, t)$	The Kells requests queue, implemented as a pair of pointers to the memory store σ.
$^*\rho$	The program counter. This counter is initialized to t_0 at reboot time and increments once for each executed action in the trace.

$$\text{(enqueue)} \quad \rho, (h,t), \sigma[t \mapsto \_], [x{:=}\ \mathtt{enqueue}\ e; P]_I \longrightarrow \rho+1, (h, t+1), \sigma[t \mapsto (e, \rho)], [P(0/x)]_I$$

$$\text{(dequeue)} \quad \rho, (h,t), \sigma[h \mapsto e], [x{:=}\ \mathtt{dequeue}; P]_I \longrightarrow \rho+1, (h+1, t), \sigma[h \mapsto e], [P(0/x)]_I$$

$$\text{(peek)} \quad \rho, (h,t), \sigma[t \mapsto e], [x{:=}\ \mathtt{peek}; P]_I \longrightarrow \rho+1, (h, t), \sigma, [P(e/x)]_I$$

$$\text{(sread)} \quad \rho, \sigma[l \mapsto e], [x{:=}\ \mathtt{sread}\ (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig)]_I \longrightarrow \rho+1, \sigma[l \mapsto e], [P(e/x)]_I \quad \text{if } \mathsf{GoodState}(\mathcal{H}, (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig))$$

$$\text{(swrite)} \quad \rho, \sigma[l \mapsto \_], [x{:=}\ \mathtt{swrite}\ (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig)]_I \longrightarrow \rho+1, \sigma[l \mapsto e], [P(0/x)]_I \quad \text{if } \mathsf{GoodState}(\mathcal{H}, (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig))$$

$$\text{(sreadD)} \quad \rho, \sigma[l \mapsto e], [x{:=}\ \mathtt{sread}\ (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig)]_I \longrightarrow \rho+1, \sigma[l \mapsto e], [P(0/x)]_I \quad otherwise$$

$$\text{(swriteD)} \quad \rho, \sigma[l \mapsto \_], [x{:=}\ \mathtt{swrite}\ (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig)]_I \longrightarrow \rho+1, \sigma[l \mapsto \_], [P(0/x)]_I \quad otherwise$$

Figure 8: The operational semantics of the language extensions used to encode Kells operations. The program counter ρ applies to all actions in the language.

state the two properties and show that they hold for the encoded versions of Kells operations.

5.2.1 Encoding Kells Operation

The encoding of the read operation is shown in Figure 9 and the write operation in Figure 10. The primary challenge in encoding Kells operations using the language was the lack of support for conditional statements and loops. Note that their addition would also require an extension of the logic to handle these structures. To alleviate the need for loops, we assume that the Kells device has a hardware timer that can repeatedly call the program that performs commits from the write request queue (`KCommit` in Figure 10).

We extend the language with three instructions for working with the Kells write request queue: `enqueue`, `dequeue` and `peek`. The first two operations are straightforward and are assumed to be synchronized with any other executing threads. The `peek` operation prevents a dequeued request from being lost by `KCommit` if a

```
KRead: 1. att = read D.RAM.att-loc
       2. (t, req) = receive
       3. n' = sread req,att
       4. send n'
```

Figure 9: The encoding of the Kells read operation.

```
KWrite:  1. (t, req-pair) = receive
         2. enqueue (t, req-pair)

KCommit: 1. att = read D.RAM.att-loc
         2. (t, req) = peek
         3. swrite req,att
         4. dequeue
```

Figure 10: The encoding of the Kells write operation.

fresh attestation has not arrived after the request has been dequeued. If the queue is empty, `peek` halts the current thread.

To capture Kells mediation, we add the checks for attestation freshness and verification into the semantics of the `read` and `write` actions by introducing the `sread` and `swrite` actions. The semantics of these two actions are shown in Figure 8. Both of these operations take a block I/O request and an attestation as arguments. A block request $(t, \mathtt{t_{req}}, (l, n))$ from the host consists of the program counter at arrival time t, an absolute arrival time $\mathtt{t_{req}}$ and a sector offset and data pair.

The encoded version of the Kells read program (`KRead`) is shown in Figure 9. We assume the existence of a running thread that is responsible for requesting new attestations from the host at a rate of Δ_t and placing the most recent attestation at $\mathcal{D}$`.RAM.att-loc`. Lines 1. and 2. receive the attestation and request from the host respectively. Line 3. invokes the secure read operation which runs to completion returning either the desired disk blocks (sread) or an error (sreadD). Line 4. sends the resulting value to the host.

The encoded version of the Kells write program (`KWrite`) is shown in Figure 10. `KWrite` simply receives the request from the host in line 1. and places it in the request queue at line 2. t contains the value of ρ at the time the request was received. The majority of the write operation is encoded in `KCommit`, which retrieves an enqueued request, arrival time and the most recent attestation, and performs an `swrite`. Recall that `KCommit` runs once in a thread invoked by a timer since a timed loop is not possible in LS^2.

5.2.2 Proof of Security Properties

The (**SEC**) and (**INT**) properties may be stated formally as shown in Figures 11 and 12. Both properties ultimately make an assertion about the state of a host at the time it is performing I/O using the Kells device. GoodState, defined in Figure 13, requires that an attestation (1) is fresh with respect to a given block I/O request and (2) represents a trusted state of the host system. In the following two definitions, Δ_t represents the length of time during which an attestation is considered fresh past its reception. Thus, GoodState can be seen as verifying the state of the host w.r.t. a given I/O request, independent of the state at any previous requests.

We use the predicate $\mathsf{Fresh}(t, \mathtt{t_{req}}, \mathtt{t_{att}})$ to state that an attestation is fresh w.r.t. a given request. The attestation is received at wall clock time $\mathtt{t_{att}}$ and the request at time $\mathtt{t_{req}}$. Attestations are received at the t^{th} clock tick, as obtained using the program counter ρ. As described above, Kells will check if a previous attestation is still within the freshness parameter Δ_t before stalling the read or queueing the write. This is the first case in the definition of Fresh in Figure 14. If a request is stalled, the next attestation received is verified before satisfying the request. In this case, a Reset must not

$$
\begin{aligned}
(\mathbf{SEC}) \vdash\ \forall\ & (\mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig), t \text{ s.t.} \\
& (\mathtt{t_{req}}, (l, n)) = \mathsf{Recv}(\mathcal{D}) \mathbin{@} t \\
\wedge\ & (\mathtt{t_{att}}, sig) = \mathsf{Recv}(\mathcal{D}) \\
\wedge\ & e = \mathsf{SRead}(\mathcal{D}, (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig)) \\
\supset\ & \mathsf{GoodState}(\mathcal{H}, (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig))
\end{aligned}
$$

Figure 11: Definition of Kells secrecy property.

$$
\begin{aligned}
(\mathbf{INT}) \vdash\ \forall\ & (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig) \text{ s.t.} \\
& (t, \mathtt{t_{req}}, (l, n)) = \mathsf{Peek}(\mathcal{D}) \\
\wedge\ & (\mathtt{t_{att}}, sig) = \mathsf{Recv}(\mathcal{D}) \\
\wedge\ & \mathsf{SWrite}(\mathcal{D}, (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig)) \\
\supset\ & \mathsf{GoodState}(\mathcal{H}, (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig))
\end{aligned}
$$

Figure 12: Definition of Kells integrity property.

occur between the receipt of the request and the check of the next attestation.

Theorem 1. `KRead` maintains the (**SEC**) security property.

Proof.

Assume that the following holds for an arbitrary program trace.

$$
\begin{aligned}
\exists\ & (\mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig), t, e \text{ s.t.} \\
& (\mathtt{t_{req}}, (l, n)) = \mathsf{Recv}(\mathcal{D}) \mathbin{@} t \\
\wedge\ & (\mathtt{t_{att}}, sig) = \mathsf{Recv}(\mathcal{D}) \\
\wedge\ & e = \mathsf{SRead}((t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig))
\end{aligned}
$$

We know that t is the value of ρ at the time the request was received because we assumed Recv occurred in the trace at time t. By definition of SRead, we have $\mathsf{Fresh}(t, \mathtt{t_{req}}, \mathtt{t_{att}})$, $\mathsf{Verify}((\mathtt{t_{att}}, sig), AIK(\mathcal{H}))$, and $\mathsf{Match}(v, criteria)$ all hold. Thus, GoodState holds, and (**SEC**) is provable using LS^2 with extensions. Because `KRead` is implemented in the language with extensions, (**SEC**) holds over `KRead` by the soundness property of LS^2.

Theorem 2. `KCommit` maintains the (**INT**) security property.

Proof.

Assume that the following holds for an arbitrary program trace.

$$
\begin{aligned}
\exists\ & (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig) \text{ s.t} \\
& (t, \mathtt{t_{req}}, (l, n)) = \mathsf{Peek}(\mathcal{D}) \\
\wedge\ & (\mathtt{t_{att}}, sig) = \mathsf{Recv}(\mathcal{D}) \\
\wedge\ & \mathsf{SWrite}((t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig))
\end{aligned}
$$

We know that t is the value of ρ at the time the request was received, by (enqueue). By definition of SWrite, we have that $\mathsf{Fresh}(t, \mathtt{t_{req}}, \mathtt{t_{att}})$, $\mathsf{Verify}((\mathtt{t_{att}}, sig), AIK(\mathcal{H}))$, and $\mathsf{Match}(v, criteria)$ all hold. Thus, GoodState, holds, giving that (**INT**) is provable using LS^2 with extensions. Because `KCommit` is implemented in the language with extensions, (**INT**) holds over `KCommit` by the soundness property of LS^2.

6. EVALUATION

We performed a series of experiments aimed at characterizing the performance of Kells in realistic environments. All experiments were performed on a Dell Latitude E6400 laptop running Ubuntu 8.04 with the Linux 2.6.28.15 kernel. The laptop TPM performs a single quote in 880 msec. The Kells device was implemented using a DevKit 8000 development board that is largely a clone of the popular BeagleBoard.[4] The board contains a Texas Instruments

[4]Due to extreme supply shortages, we were unable to procure a BeagleBoard or our preferred platform, a small form-factor Gum-

Configuration (Δ_t)	Read			Write		
	Run (secs)	Throughput (MB/sec)	Overhead	Run (secs)	Throughput (MB/sec)	Overhead
No verification	36.1376	14.196	N/A	35.4375	5.6437	N/A
1 second verification	36.5768	14.025	1.22%	36.4218	5.4912	2.78%
2 second verification	36.6149	14.011	1.32%	35.9895	5.5572	1.56%
5 second verification	36.3143	14.127	0.49%	35.7969	5.5871	1.01%
10 second verification	36.2113	14.167	0.20%	35.7353	5.5967	0.84%

Table 2: Kells performance characteristics – average throughput over bulk read and write operations

$$\begin{aligned}&\mathsf{GoodState}(\mathcal{H}, (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig)) = \\ &\quad \mathsf{Fresh}(t, \mathtt{t_{req}}, \mathtt{t_{att}}) \\ &\wedge\ v = \mathsf{Verify}((\mathtt{t_{att}}, sig), AIK(\mathcal{H})) \\ &\wedge\ \mathsf{Match}(v, criteria)\end{aligned}$$

Figure 13: Definition of Goodstate property.

$$\begin{aligned}&\mathsf{Fresh}\ (t, \mathtt{t_{req}}, \mathtt{t_{att}}) = \\ &\quad (\mathtt{t_{att}} < \mathtt{t_{req}}\ \wedge\ \mathtt{t_{req}} - \mathtt{t_{att}} < \Delta_t) \\ &\vee\ (\mathtt{t_{req}} < \mathtt{t_{att}}\ \wedge\ \neg\mathsf{Reset}(\mathcal{H}) \text{ on } [t, \rho])\end{aligned}$$

Figure 14: Definition of Fresh property.

OMAP3530 processor, which contains a 600 MHz ARM Cortex-A8 core, along with 128 MB of RAM and 128 MB of NAND flash memory. An SD card interface provides storage and, most importantly for us, the board supports a USB 2.0 On-the-Go interface attached to a controller allowing device-mode operation. The device runs an embedded Linux Angstrom distribution with a modified 2.6.28 kernel. Note that an optimized board could be capable of receiving its power from the bus alone. The TI OMAP-3 processor's maximum power draw is approximately 750 mW, while a USB 2.0 interface is capable of supplying up to 500 mA at 5 V, or 2.5 W. The recently introduced USB 3.0 protocol will be even more capable, as it is able to supply up to 900 mA of current at 5 V.

Depicted in Table 2, our first set of experiments sought to determine the overhead of read operations. Each test read a single 517 MB file, the size of a large video, from the Kells device. We varied the security parameter Δ_t (the periodicity of the host integrity revalidation) over subsequent experiments, and created a baseline by performing the read test with a unmodified DevKit 8000 USB device and Linux kernel. All statistics are calculated from an average of 5 runs of each test.

As illustrated in the table, the read operation performance is largely unaffected by the validation process. This is because the host preemptively creates validation quotes and delivers them to the device at or about the time a new one is needed (just prior to a previous attestation becoming stale). Thus, the validation process is mostly hidden by normal read operations. Performance, however, does degrade slightly as the validation process occurs more frequently. At about the smallest security parameter supportable by the TPM hardware ($\Delta_t = 1\ second$), throughput is reduced by only 1.2%, and as little as 0.2% at 10 seconds. This overhead is due largely to overheads associated with receiving and validating the integrity proofs (which can be as large as 100KB).

Also depicted in Table 2, the second set of tests sought to characterize write operations. We performed the same tests as in the read experiments, with the exception that we wrote a 200MB file. Write operations are substantially slower on flash devices because of the underlying memory materials and structure. Here again, the write operations were largely unaffected by the presence of host validation, leading to a little less than 3% overhead at $\Delta_t = 1\ second$ and just under 1% at 10 seconds.

Note that the throughputs observed in these experiments are substantially lower than USB 2.0 devices commonly provide. USB 2.0 advertises maximal throughput of 480Mbps, with recent flash drives advertising as much as 30MB/sec. All tests are performed on our proof of concept implementation on the experimental apparatus described above, and are primarily meant to show that delays are acceptable. Where needed, a production version of the device and a further optimized driver may greatly reduce the observed overheads. Given the limited throughput reduction observed in the test environment, we reasonably expect that the overheads would be negligible in production systems.

stix Overo device. Future work will consider how these devices may change our performance characteristics.

7. RELATED WORK

The need to access storage from portable devices and the security problems that consequently arise is a topic that has been well noted. SoulPad [4] demonstrated that the increasing capacity of portable storage devices allows them to carry full computing stacks that required only a platform to execute on. DeviceSniffer [35] further considered a portable USB device that allowed a kiosk to boot, where the software on the drive provides a root of trust for the system. As additional programs are loaded on the host, they are dynamically verified by the device through comparison with an on-board measurement list. This architecture did not make use of trusted hardware and is thus susceptible to attacks at the BIOS and hardware levels. The iTurtle [17] was a proposal to use a portable device to attest the state of a system through a USB interface. The proposal made the case that load-time attestations of the platform was the best approach for verification. This work was exploratory and postulated questions rather than providing concrete solutions.

Garriss et al. further explored these concepts to use a mobile device to ensure the security of the underlying platform, using it as a kiosk on which to run virtual machines [10] and providing a framework for trusted boot. This work makes different assumptions about how portable devices provide a computing environment; in the proposed model, a mobile phone is used as authenticator, relying on a barcode attached to the platform transmitted wirelessly to the device. Because the verifier is not a storage device, the virtual machine to be run is encrypted in the cloud.

Others have considered trusted intermediaries that establish a root of trust external to the system, starting with Honeywell's Project Guardian and the Scomp system, which provided a secure front-end processor for Multics [8]. SIDEARM was a hardware processor that ran on the LOCK kernel, establishing a separate security enforcement point from the rest of the system [31]. The first attempt to directly interpose a security processor within a system was the Security Pipeline Interface [12], while other initiatives such as the Dyad processor [40] and the IBM 4758 coprocessor [7] provided a secure boot. Secure boot was also considered by Arbaugh et al., whose AEGIS system allows for system startup in the face of integrity failure. Numerous proposals have considered how to attest

system state. SWATT [27] attests an embedded device by verifying its memory through pseudorandom traversal and checksum computation. This requiers verifier to fully know the memory contents. Recent work has shown that SWATT may be susceptible to return-oriented rootkits [5] but this work itself is subject to assumptions about SWATT that may not be valid. Similarly, Pioneer [26] enables software-based attestation through verifiable code execution by a verification function, reliant on knowledge of the verified platform's exact hardware configuration. A study of Pioneer showed that because it is based on noticing increases in computation time in the event of code modification, a very long execution time is required in order to find malicious computation as CPU speeds increase [9]. Software genuinity [14] proposed relying on the self-checksumming of code to determine whether it was running on a physical platform or inside a simulator; however, Shankar et al. showed problems with the approach [29].

Augmenting storage systems to provide security has been a topic of sustained interest over the past decade. Initially, this involved network-attached secure disks (NASD) [11], an infrastructure where metadata servers issue capabilities to disks augmented with processors. These capabilities are the basis for access control, requiring trust in servers external to the disk. Further research in this vein included self-securing storage [34], which, along with the NASD work, considered object-based storage rather than the block-based approach that we use. Pennington et al. [21] considered the disk-based intrusion detection, requiring semantically-aware disks [30] for deployment at the disk level.

8. CONCLUSION

In this paper, we presented Kells, a portable storage device that validates host integrity prior to allowing read or write access to its contents. Access to trusted partitions is predicated on the host providing ongoing attestations as to its good integrity state. Our prototype demonstrates that overhead of operation is minimal, with a reduction in throughput of 1.2% for reads and 2.8% for writes given a one-second periodic runtime attestation. Future work will include a detailed treatment of how policy may be enforced in an automated way between trusted and untrusted storage partitions, and further interactions with the OS in order to support and preserve properties such as data provenance and control of information flow.

9. REFERENCES

[1] IronKey. http://www.ironkey.com, 2009.

[2] K. Butler, S. McLaughlin, T. Moyer, J. Schiffman, P. McDaniel, and T. Jaeger. Firma: Disk-Based Foundations for Trusted Operating Systems. Technical Report NAS-TR-0114-2009, Penn State Univ., Apr. 2009.

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

[4] R. Cáceres, C. Carter, C. Narayanaswami, and M. Raghunath. Reincarnating PCs with portable SoulPads. In *ACM MobiSys*, 2005.

[5] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *ACM CCS*, Nov. 2008.

[6] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A Logic of Secure Systems and its Application to Trusted Computing. In *IEEE Symp. Sec. & Priv.*, May 2009.

[7] J. G. Dyer, M. Lindermann, R. Perez, et al. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 39(10):57–66, Oct. 2001.

[8] L. J. Fraim. Scomp: A solution to the multilevel security problem. *IEEE Computer*, 16(7):26–34, July 1983.

[9] R. Gardner, S. Garera, and A. D. Rubin. On the difficulty of validating voting machine software with software. In *USENIX EVT*, Aug. 2007.

[10] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Trustworthy and personalized computing on public kiosks. In *ACM MobiSys*, June 2008.

[11] G. A. Gibson, D. F. Nagle, K. Amiri, et al. A Cost-Effective, High-Bandwidth Storage Architecture. In *ASPLOS*, 1998.

[12] L. J. Hoffman and R. J. Davis. Security Pipeline Interface (SPI). In *ACSAC*, Dec. 1990.

[13] B. Kauer. OSLO: Improving the Security of Trusted Computing. In *Proc. USENIX Security Symp.*, Aug. 2007.

[14] R. Kennell and L. H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *Proc. USENIX Security Symp.*, Aug. 2003.

[15] Kingston Technology. DataTraveler 300: World's first 256 GB Flash drive. http://www.kingston.com/ukroot/flash/dt300.asp, July 2009.

[16] C. Lomax. Security tightened as secretary blamed for patient data loss. *Telegraph & Argus*, 4 June 2009.

[17] J. M. McCune, A. Perrig, A. Seshadri, and L. van Doorn. Turtles all the way down: Research challenges in user-based attestation. In *USENIX HotSec*, Aug. 2007.

[18] Microsoft. BitLocker and BitLocker to Go. http://technet.microsoft.com/en-us/windows/dd408739.aspx, Jan. 2009.

[19] T. Moyer, K. Butler, J. Schiffman, et al. Scalable Web Content Attestation. In *ACSAC*, 2009.

[20] B. Parno. Bootstrapping trust in a "trusted" platform. In *USENIX HotSec*, Aug. 2008.

[21] A. G. Pennington, J. D. Strunk, J. L. Griffin, et al. Storage-based Intrusion Detection: Watching storage activity for suspicious behavior. In *Proc. USENIX Security*, 2003.

[22] P. Porras, H. Saidi, and V. Yegneswaran. An Analysis of Conficker's Logic and Rendezvous Points. Technical report, SRI Computer Science Lab, Mar. 2009.

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

[24] SanDisk. SanDisk Cruzer Enterprise. http://www.sandisk.com/business-solutions/enterprise, 2009.

[25] Seagate. Self-Encrypting Hard Disk Drives in the Data Center. Technology Paper TP583.1-0711US, Nov. 2007.

[26] A. Seshadri, M. Luk, E. Shi, et al. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM SOSP*, 2005.

[27] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *IEEE Symp. Sec.& Priv.*, May 2004.

[28] N. Shachtman. Under Worm Assault, Military Bans Disks, USB Drives. *Wired*, Nov. 2008.

[29] U. Shankar, M. Chew, and J. D. Tygar. Side Effects are Not Sufficient to Authenticate Software. In *Proc. USENIX Security*, 2004.

[30] M. Sivathanu, V. Prabhakarn, F. I. Popovici, et al. Semantically-Smart Disk Systems. In *USENIX FAST*, 2003.

[31] R. E. Smith. Cost profile of a highly assured, secure operating system. *ACM Trans. Inf. Syst. Secur.*, 4(1):72–101, 2001.

[32] SRN Microsystems. Trojan.adware.win32.agent.bz. http://www.srnmicro.com/virusinfo/trj10368.htm, 2009.

[33] L. St. Clair, J. Schiffman, T. Jaeger, and P. McDaniel. Establishing and Sustaining System Integrity via Root of Trust Installation. In *ACSAC*, 2007.

[34] J. Strunk, G. Goodson, M. Scheinholtz, et al. Self-Securing Storage: Protecting Data in Compromised Systems. In *USENIX OSDI*, 2000.

[35] A. Surie, A. Perrig, M. Satyanarayanan, and D. J. Farber. Rapid trust establishment for pervasive personal computing. *IEEE Pervasive Computing*, 6(4):24–30, Oct.-Dec. 2007.

[36] TCG. *TPM Main: Part 1 - Design Principles*. Specification Version 1.2, Level 2 Revision 103. TCG, July 2007.

[37] TCG. *TCG Storage Security Subsystem Class: Opal*. Specification Version 1.0, Revision 1.0. Trusted Computing Group, Jan. 2009.

[38] T. Weigold, T. Kramp, R. Hermann, et al. The Zurich Trusted Information Channel – An Efficient Defence against Man-in-the-Middle and Malicious Software Attacks. In *Proc. TRUST*, Villach, Austria, Mar. 2008.

[39] R. Wojtczuk and J. Rutkowska. Attacking Intel Trusted Execution Technology. In *Proc. BlackHat Technical Security Conf.*, Feb. 2009.

[40] B. Yee and J. D. Tygar. Secure Coprocessors in Electronic Commerce Applications. In *Proc. USENIX Wrkshp. Electronic Commerce*, 1995.