LEVERAGING EMERGING STORAGE FUNCTIONALITY FOR NEW SECURITY SERVICES

A Dissertation in
Computer Science and Engineering
by
Kevin Raymond Boyce Butler

© 2010 Kevin Raymond Boyce Butler

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2010
The dissertation of Kevin Raymond Boyce Butler was reviewed and approved* by the following:

Patrick D. McDaniel  
Associate Professor of Computer Science and Engineering  
Dissertation Advisor, Chair of Committee

Trent R. Jaeger  
Associate Professor of Computer Science and Engineering

Bhuvan Urgaonkar  
Assistant Professor of Computer Science and Engineering

Stephen G. Simpson  
Professor of Mathematics

Matthew A. Blaze  
Associate Professor of Computer and Information Science  
University of Pennsylvania

Raj Acharya  
Professor of Computer Science and Engineering  
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

The complexity of modern operating systems makes securing them a challenging problem. However, changes in the computing model, such as the rise of cloud computing and smarter peripherals, have presented opportunities to reconsider system architectures, as we move from traditional “stove-pipe” computing to distributed systems. In particular, we can build trustworthy components that act to provide security in complex systems.

The focus of this dissertation is on how new disk architectures may be exploited to aid the protection of systems by acting as policy decision and enforcement points. We prototype disks that enforce data immutability at the block level on critical system data, preventing malicious code from inserting itself into system configuration and boot files. We then examine how storage may be used to ensure the integrity state of hosts prior to allowing access to data, and how such a design improves the security of portable storage devices. Through the continual measuring of system state, we show through formal reasoning that such a device enforces guarantees that data is read and written while the host is in a good state. Finally, we discuss future directions and how secure disk architectures can be used as the basis for large-scale and distributed system security.
# Table of Contents

List of Figures ix  
List of Tables xiii  
Acknowledgments xiv  

## Chapter 1  
**Introduction**  
1.1 Using Storage to Reduce the TCB .......................... 4  
1.2 Thesis Statement ........................................ 6  
1.3 Contributions of Dissertation ............................ 7  
1.4 Trust Model and Limitations ............................. 8  
1.5 Publications ............................................ 10  

## Chapter 2  
**Foundational Work**  
2.1 Database Machines and Augmented Disks .................. 11  
2.2 Security in Operating Systems ............................ 13  
   2.2.1 Process Isolation .................................... 15  
   2.2.2 System Threats from Rootkits ......................... 16  
   2.2.3 Autonomous Security ................................. 18  
2.3 Secure Autonomous Storage ............................... 19  
   2.3.1 Securing Data with the Filesystem .................. 19  
   2.3.2 Augmented Secure Disks .............................. 21  
2.4 Trust Establishment .................................... 23  
   2.4.1 Trusting the Boot Process ........................... 23  
   2.4.2 Reducing the TCB .................................... 25  
   2.4.3 Devices for Trust Establishment ....................... 26  
   2.4.4 Trusted Storage .................................... 27
# List of Figures

1.1 The size of operating systems and their code complexity has grown tremendously as new features have been added. .................................................. 2
1.2 Full-disk encryption drives are augmented with cryptographic processors. 5
1.3 The trusted computing base of a system can be reduced if storage with autonomous security enforcement is employed. 6

2.1 The original database machine concept was a “processor-per-track” model based on a fixed-head hard disk. ................................................................. 12
2.2 The Security Pipeline Interface (SPI) interposed a security processor between the CPU and I/O interfaces. ................................................................. 18
2.3 The Network-Attached Secure Disk (NASD) architecture. A metadata server provides access control and provides the client with a capability that can be used to send and retrieve information from augmented network disks. 21

3.1 Design of an ASD. .................................................................................... 35
3.2 Performing authenticated encryption using security metadata. When a client makes a read request for blocks 10-20 (step 1), the policy enforcement point, or PEP (usually the processor on the disk) forwards a request to the disk platters, which read the blocks (steps 2,3). The non-volatile memory is consulted to determine the MAC for the integrity set (step 4); in this case, the set covers blocks 10-20. The results are returned to the PEP (step 5), and if the MAC is valid, the data is returned to the OS (step 6). ..................................................................................... 38
3.3 An example of how per-subject capability tokens are represented with an associated capability bitmap. Subject S1 has read access to the first three integrity sets and write access to the next two. Similarly, S2 has read and write access to the first five integrity sets, while SN has no access to any sets. ..................................................................................... 42
3.4 An example of enforcing information flow with disk metadata. A request from a client labeled LOW to read blocks 10-20 is made (step 1). The policy enforcement point, or PEP (usually the processor), consults the non-volatile memory and finds that blocks 10-40 are labeled HIGH (step 2). When this information is returned to the PEP (step 3), it denies the request and no disk access is made.

3.5 A shim layer is used to interface the kernel driver, genbd with the simulated driver layer in DiskSim. Neither the kernel nor DiskSim is aware that there is anything between the gray blocks because workload translates between real time and simulator time.

3.6 Average completion times for various integrity set sizes.

3.7 Average throughput (in IOPS) for various integrity set sizes.

3.8 Relationship between average block request size and integrity set size.

3.9 Completion times for in-house benchmark.

3.10 Read and write throughputs for in-house benchmark.

3.11 Average block request size for in-house benchmark.

3.12 Contour plots for best completion time and throughput. The contour line for 75.5 s shows the trade offs between flash memory and disk size. The plot for 995 IOPS is the same as that for completion time because the same integrity set size is optimal for both.

4.1 The use of tokens for labeling data in the RRD. Part (a): The host system writes a file to unused disk space using the gray token. The write is allowed and the file is labeled accordingly. In part (b), any blocks written while the black token is in the disk are labeled accordingly, and any attempts to write gray-labeled data are denied as long as the gray token is not present.

4.2 The prototype RRD. Block requests are intercepted at the client by the RRD driver and sent to the slug over TCP. netblockd receives the block request and consults BLOCK-LABELS and the optional physical token to determine if the request is allowed. If so, netblockd fulfills the request from the external USB hard drive. netblockd accesses BLOCK-LABELS, the physical token, and the external drive through mount points in the slug’s file system. An asterisk is placed next to each component that we created or configured.

4.3 The RRD prototype, consisting of a Linksys NSLU2 storage link, a Seagate Free Agent Pro external hard drive, and a USB flash memory drive, connected to a hub for clarity. A token is plugged into the leftmost spot on the USB hub.

4.4 Comparison of used blocks versus labeled blocks.

4.5 The number of ranges in BLOCK-LABELS compared with number of labeled blocks.
5.1 SwitchBlade architecture. A physical token contains the set of labels for segments to be exported, in this case, S2 and SBoot, the latter of which is exported as read only. The audit log is read only by default and writable by the disk controller. In this case, the Token Handler constitutes the policy decision point which configures volume management to export the segments allowed by the current physical token according to the policy, which is stored in NVRAM.

5.2 Details of the SwitchBlade prototype. The token handler parses the contents of the USB token, mounted by udev. It then invokes vblade to export the logical volumes allowed by the token. The audit log is also managed as a logical volume that is exported read-only by vblade, but directly writable by the token handler.

5.3 An example of the contents of a SwitchBlade token.

5.4 The process of booting the host system from the Slug disk. The initial phases are the DHCP and TFTP protocols in order to retrieve the kernel, hypervisor and initial ramdisk (initrd) images. Once the host has started booting the system, it discovers the available AoE devices, which will include the root file system for the platform.

5.5 Operation in a virtualized environment. Each ATA-over-Ethernet (AoE) device is a separate block device that either dom0 or the domUs can access. The domUs are unaware that the block device runs AoE.

5.6 The above screen capture depicts Windows Vista and OpenBSD 4.3 running in virtual machines. The terminal window is showing the VMs that Xen is configured to run, which in this case is five different VMs. The menu interface is shown in the lower right. This interface only shows VMs that can be correctly started based on the available devices presented to dom0, which includes the two running VMs and another VM containing OpenSolaris currently in a stopped state.

6.1 Overview of the Firma architecture. The disk is augmented with verification logic for receiving attestations from the host system, and stores secure policy information and cryptographic keys. Secure communication between the host and the disk can occur through enhancements to the disk protocol or out-of-band, such as with a hardware token delivering information between the devices.

6.2 Throughput of Firma prototype when using time-based reattestations

7.1 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.
7.2 Overview of the Kells architecture. The storage device provides public and trusted storage partitions, selectively exposed based on attestations received from the host TPM. 127
7.3 Overview of USB operation with an embedded Linux mass storage device, or gadget. Most portable devices export a FAT32 partition for compatibility. 127
7.4 Sample USB setup between a host and the Kells device. After the typical setup requirements for a USB device, descriptors from the device are retrieved. Only if the device is identified as trusted, and the host is aware of these trusted devices, will the attestation protocol begin. 129
7.5 The operational semantics of the language extensions used to encode Kells operations. The program counter \( \rho \) applies to all actions in the language. 138
7.6 The encoding of the Kells read operation. 139
7.7 The encoding of the Kells write operation. 139
7.8 The formal definition of the two Kells security properties. 140
8.1 Policy storage and labeling inside the ASD, which stores the endorsement key, measurement list and pickle access token for each paired host. Blocks within the secure pickle partition that correspond to pickle files are marked by the PA of the paired host. Note in this example the equivalence between paired hosts Y and Z; they share the same access token \( PA_Y \). 153
List of Tables

3.1 Comparison of capability granularities. 41
4.1 Average completion time in seconds for Postmark 70
4.2 Average Transactions Per Second for Postmark 70
4.3 Average microbenchmark results showing the amount of time spent on disk and network I/O and security operations in the RRD for Postmark. 70
4.4 Average completion times in seconds for FS creation 72
4.5 Average completion times in seconds for the base system copy 72
4.6 Microbenchmark results showing the amount of time spent on disk and network I/O and security operations in the RRD during the base system copy. 72
4.7 The number of packages modified in each upgrade. 74
5.1 Throughput of in KBps for various block sizes as output by measuring the speed of SHA-1 hashing with OpenSSl. 102
6.1 Microbenchmarks of the host operations needed to support Firma. For each measurement, 95% confidence intervals are shown in parentheses. All numbers are shown in milliseconds unless otherwise noted. 118
7.1 The subset of $LS^2$ and extensions used to evaluate the Kells security properties. 147
7.2 Kells performance characteristics – average throughput over bulk read and write operations. 148
Acknowledgments

The saying “It takes a whole village to raise a child” may be an old saw but its meaning still rings true for me. My village has been populated by some extraordinary people whose selflessness and generosity have helped me get to where I am now. I was blessed to have grown up with a wonderful public education system in Canada, where I was afforded every advantage thanks to wonderful and motivated teachers from elementary school onwards. In particular, Ms. Smith, Ms. Phillips, Mrs. Montgomery, Mrs. Schoo-Grigg (who gave me a postage stamp of a firefighting plane that she thought described my personality – the propensity to be able to put out fires was sorely tested in graduate school)... you have my eternal gratitude for helping to mould me as a thinker and challenge me with novel ideas. The teachers in high school were no less influential - in particular, Mrs. DiRocco-Smith’s energy and mental math challenges, Mr. Smith’s introducing us to Chomskyan knowing that a select few of us wanted to learn more than algebra, Mrs. Rochon’s incisive wit, Mr. King’s focus on excellence in physics, Mr. Pleasants’s warm personality and combination of musical technicality and showmanship, even Mr. Levine’s husky but good-natured growl during rugby practice: these are but a few of the impressions of the phenomenal teachers and mentors, who imparted lessons that stay with me even today.

I was fortunate to be able to attend Queen’s University for electrical engineering and to meet some phenomenal friends that I’ve kept to this day. George Pfaff showed me it was possible to be a model student and still have a good time, Chris Edwards was always there with either a joke or a keenly perceptive insight, depending on the situation, and Keith Gerein could always be counted on for his wise remarks (unless they were attributed to me and directed at other people). My housemates in later years, particularly Steve Mayer and his wife Michelle Hapuarachichi, were especially supportive and pleasures to be with. I was equally fortunate to have worked at UUNET for my first job, which opened more doors than I could have imagined. Fabio Guglielmo and John “JP” Pennington took me under their wings and taught me the intricacies of networks from vastly different perspectives of theory and practice, and both were invaluable.

I was perhaps most lucky to have been introduced to the world of research through working at a research lab. Peter Bates at Telcordia Technologies in New Jersey took a chance on hiring me based on my networking knowledge, and showed me a new world of possibilities in terms of how to find interesting ideas and turn them into reality. Peter
schooled me in the lore of the Bell System and was a mentor in many ways to me, personally and professionally. I owe a great deal to him and am proud to consider him a teacher and friend. Simon Tsang also helped me tremendously, always having an ear for my technical problems or asking about adjusting to life in the States. Joe DesMarais was a mentor to me as well, and helped me navigate working while taking classes at Columbia University. When I began going full-time to Columbia, Henning Schulzrinne advised me and showed me how academic research was different. Angelos Keromytis was my first real security teacher, and I learned from his tremendous expertise.

While at Columbia, I took a course on Internet routing taught by John “JI” Ioannidis. JI's knowledge was immense and I was very interested in the material. At the conclusion of the course, and based on my work on a project that had impressed him, JI brought me out to where he worked at AT&T Labs in Florham Park, NJ. It was there that I met Patrick McDaniel, a researcher in the secure systems group. That was truly a momentous day. Patrick had heard about my work and I noticed a marked up copy of the report on his desk, covered in red ink. I expected the worst, but Patrick was enthusiastic about what I’d done and wanted me to continue on this path. Working with JI and more and more with Patrick, with whom I spoke regularly on the phone and provided progress updates, I began to discover in a new way just how rewarding it was to chart my own direction and fully realize a project that challenged me but spoke to my underlying interests. Thanks to his unceasing advocacy, AT&T hired me as an intern in the late fall of 2003. Thanks to the extraordinary members of the labs (one of the finest groups of computer scientists assembled in one place) and their promotion of my case, I was extraordinarily fortunate to be accepted to a number of top-tier doctoral programs. When Patrick told me he was planning to take the job at Penn State, and detailed his plans for building a lab there, it seemed like a long shot compared to the established and highly prestigious programs I was considering. I had tremendous faith in Patrick's abilities to be successful, though, and so did everyone at the labs. Bill Aiello, director of the Secure Systems group and an invaluable mentor to me, told me that Patrick would leave no stone unturned when it came to being successful and would be an uncommonly good advisor. I am happy to say that he was exactly right. Over the more than 7 1/2 years since we first met, Patrick has gone far and beyond the call of what was necessary for him to do in order to ensure that I would be successful. The level of commitment he shows to his students is virtually unprecedented. Patrick gave me everything I needed to get work accomplished, showed me how to do research, how to write papers, how to give a talk, how to truly become an academic, and along the way he became more than an advisor. He is a trusted confidant, a mentor, a teacher, and a friend, and I owe him a tremendous and unpayable debt for allowing me to realize the potential that he saw in me. His wife Megan has been a pillar of strength and support for me and the entire lab, and we would not have been able to do what we do without her. She has made me feel like part of the family and I am tremendously grateful to her for putting up with my being in her house at all hours of the day and night.

While the SIIS lab was created by Patrick, it has been the people that have flocked to
it that have made it the amazing environment it is today. I have had the opportunity to
work with very fine people who have been tremendous colleagues and friends, but want to
highlight three in particular. Patrick Traynor was a dynamic force with an amazing work
ethic balanced by his humour and insight. Will Enck has been my housemate for the
past five years and our lengthy talks, combined with his critical eye and uncanny ability
to ferret out fundamentally interesting problems, have made my own work immeasurably
better. Both Patrick and Will were also founding members of the SIIS lab, and I am
proud to have been with them all since the beginning. Steve McLaughlin did not join at
the beginning, however. As an undergraduate, he heard about some of our research and
wanted to learn more. His reward for his curiosity was being stuck with me working on
a technically gruelling block driver project. I have seen Steve blossom into the highest
calibre of researcher, and his analytical skills combined with his never-say-die attitude
have been instrumental in making my own work successful. It has been an honour to
work with him and an even greater honour for him to consider me as a mentor - if I get
to regularly work with anyone as talented as him later in my career, I will consider myself
fortunate indeed. I would also like to thank Josh Schiffman for his technical assistance
and humorous musings on zombies, as well as Tom Moyer for all of his work with the
storage devices. To the rest of the SIIS lab, it has been a tremendous pleasure and I
hope to work with all of you again.

There have been many people looking out for my well-being both within and outside
Penn State. My thesis committee members took particular care to ensure that I was
heading down the right path, and in their own rights have taught me a great deal. Trent
Jaeger taught me fundamentals of systems and security, and could always be counted on
for honest advice and a dry wit - I am enormously grateful to have had an opportunity
to work with him. Bhuvan Urgaonkar and Stephen Simpson provided me with valuable
ideas about systems and theory. Matt Blaze from the University of Pennsylvania has
been especially notable, having helped guide my career in many ways, including being
my first exposure to destructive research when he allowed me to sit in on his class at
Penn. I am tremendously honoured to have Matt on my committee and as a tremendous
mentor and resource. Others have helped guide my journey including Adam Smith at
Penn State, who has been a guide to the theoretical, and Erez Zadok at Stony Brook,
who graciously agreed to be a letter writer for me and has taught me much about the
art of systems analysis and performance. Simon Byers at AT&T Research allowed me
to live in his home and introduced me to new ways of thinking about security. I hope
to attend another Spicy Food Wednesday with him and his wife Miriam. Notably, I'd
like to thank Jennifer Rexford at Princeton. She has always been quick to offer advice
and new information, helped shape a large survey on BGP and has always been generous
with her time and talent. I am deeply gratified and humbled that she has always made
efforts to help me in every aspect of my professional life.

A few years ago I was introduced to Emily Rine, and we found that we had much
in common. As we saw more and more of each other, we also found differences, but
these served to accentuate our own conversations. Emily has been an intellectual and
conversational foil, a constant bedrock of support, and someone who has known when I need a hand up and when to be left alone to struggle. Through our journeys together we have gotten ever closer and now we prepare for a new life together in a new place, there isn’t anyone I’d rather be sharing the journey with.

Finally, I’d like to acknowledge my family. My great-grandmother drilled the value of education into my grandfather, who passed it on to his children. This solid foundation was crucial to our success, as academic excellence was desired and expected for all of us, and every resource was provided for us to strive for the best education we could get. My grandparents are constant sources of support and wisdom, and have been an incredible inspiration. My uncles and aunts have succeeded spectacularly and have been tremendous role models for me, as well as having given me boundless love and support. My father has been an inspiration in his own way, showing that it’s possible to attain the highest of academic goals. My sister has been a source of unquestioning love and good humour, and is someone I admire a great deal.

My mother raised me single-handedly and has given me every possible opportunity. She has poured her heart and soul into making me who I am now, and there are no words sufficient to thank her for everything she’s done. I dedicate this dissertation to her.
Dedication

To my mother, who gave me everything I could ever want and whose unyielding spirit motivates this work.
Chapter 1

Introduction

In 2007, the US Central Command in Washington, D.C., experienced a security failure of near-catastrophic proportions. Employees found portable storage devices of unknown origin around the premises. These “flash drives” are small and designed to be plugged into a computer’s USB interface, making them readily accessible to a wide variety of systems. Employees at Central Command (CENTCOM), interested in determining the contents of these devices, plugged them into their workstations. Unbeknownst to them, the flash drives contained files infected by malicious code, or malware. When these devices were plugged in, the malicious code propagated itself throughout the CENTCOM network, infecting other workstations and servers. The damage did not end there, however; the malware was capable of traversing network boundaries, and propagated not only into classified networks but also into networks used for active military operations. Since this event, the Department of Defense has imposed a ban on USB flash drives throughout military and civilian machines [1].

The reasons for this critical vulnerability are multifaceted, and stretch beyond issues of user behavior and training. In particular, the incident brings to light the difficulty of maintaining a secure computing infrastructure. Operating systems face greater exposure than ever before. Twenty years ago, the Internet was accessible primary through universities and government environments, and few users knew of its existence. External connectivity for many users, if they had it at all, was often limited to local bulletin board services. By contrast, as of 30 June 2008, it is estimated that 73.6% of the North American population has access to the Internet, and the number of worldwide users is approaching 1.5 billion [2]. Users demand more from their computing experience and as a result, modern operating systems have added increasing amounts of functionality and
compatibility with external peripherals, with new features and advanced graphical interfaces. The net result of this has been an explosion in their size, as shown in Figure 1.1. Windows NT 1.0 comprised approximately 4-5 million lines of code with a 200 member development team; by contrast, Windows XP was roughly 40 million lines of code with an 1,800-member development team [3]. Processes running on these operating systems interact with other processes both locally and remotely—with the Internet, the remote system could be anywhere in the world, and users are increasingly relying on on-demand access to these remote services due to the rise of paradigms such as cloud computing. Users interact with large-scale programs such as web browsers and office productivity suites with codebases in the millions of lines of code. The size and complexity of these programs and of the operating system, combined with the almost limitless interactions they can have with each other and with remote parties, makes a full understanding of them virtually impossible. In short, securing these systems is an almost intractable problem.

Securing operating systems has been a major focus of computer science research for over 40 years. The Multics system [4] was one of the earliest to define security as a goal. Beginning in 1965, the initial project lasted over 10 years. Even with the intense focus on security, demonstrated by hardware features such as segmented address space and process isolation through protection rings [5], and the use of typed programming languages to eliminate buffer overflow attacks [6], as well as a relatively small codebase with a 54,000 line kernel, vulnerabilities were still present in the system. Karger and Schell showed that Multics was vulnerable to a variety of attacks, due to errors in both

\[ \text{Figure 1.1. The size of operating systems and their code complexity has grown tremendously as new features have been added.} \]
design and implementation [7]. As examples, a badly-implemented argument validator caused a privilege violation, while a subtle method of passing instructions in Master Mode showed a design vulnerability that allowed seemingly-independent events occurring in conjunction to transfer privileged information to an arbitrary location on the system.

Unfortunately, current operating systems are significantly more vulnerable to attack than Multics. These systems often emphasize user functionality over security, and services that are open by default in order to prevent users from having to configure them. This decision to make convenience paramount has had inadvertent and unfortunate results. The Code Red worm, launched in July 2001, attacked millions of Windows 2000 machines that had the IIS web server installed. Many of the infected machines appeared to have been office desktop computers, whose users may not have even been aware that their machines were running web servers in the first place [8]. The security problems in the Windows operating system were sufficiently publicized that Bill Gates wrote an all-company memo in 2002 as a “call-to-action” to focus on increasing the security and overall trustworthiness of the entire Windows platform [9].

Other operating systems such as Linux and the Unix variants are similarly vulnerable to intentional or inadvertent misuse. While efforts to secure these commodity systems are in development, such as the SELinux initiative [10], they can suffer from complex policies that are difficult to decipher and relate to high-level security goals. This complexity is a prime factor for the insecure state of modern operating systems: while policies can be written to provide mandatory access control for user protection, and firewall policies can be written to attempt to protect systems from malicious entities on the network [11], the interaction of these policies is difficult to analyze; reconciling them may be intractable [12].

Fundamentally, the key to protecting a computing system must as a necessary first step involve the protection of its trusted computing base, or TCB [13], which defines the minimum set of hardware, firmware, and software that are critical to the overall security of the system. Components outside of the TCB should correspondingly be able to fail without compromising the system; Lampson et al. describe the failure mode of these components as fail-secure [14], where the component may be denied access it should have been granted but not granted access it should have been denied. An example TCB for a desktop computer includes the system hardware, i.e., memory, storage, and CPU, the system BIOS and, due to the monolithic design of commodity operating systems (i.e., the entire operating system kernel runs in privileged mode in a single, unpartitioned kernel space), the entire OS. As mentioned above, the OS alone in a modern Windows operating
system comprises tens of millions of lines of code. It is clear that with a TCB of this enormous size, it is impossible to claim that it is all critically important to the system’s security, and more importantly, that it is secure. The central, and well-remarked upon observation, is thus that the trusted computing base of computing systems is far too large to ensure their adequate protection.

1.1 Using Storage to Reduce the TCB

If a TCB that comprises the operating system is too large to protect the computer, clearly another method of providing protections must be considered. All of the information on a system that does not arrive from external sources must be stored on the system itself. Consider starting a desktop computer. The system BIOS, contained at a predefined address in a computer’s read-only memory, executes and performs a power-on self test, then begins executing a boot loader by reading from the boot sector of the device assigned to be read at startup. While in some cases a user may choose to boot from a peripheral such as a USB device or a CD-ROM drive (e.g., for “live” execution of an OS), and in some enterprises the system may use the preboot execution environment (PXE) [15] to boot from the network, in the vast majority of cases the source of the boot loader, and the operating system subsequently loaded, is the system’s hard disk. In computing systems, the magnetic hard disk has been the primary means of providing persistent information storage ever since its commercial introduction with the IBM 350 disk storage unit [16] in 1956. Since that first 5 MB device, comprising 50 platters each 24 inches in diameter, the information density has increased 300-fold. Modern hard disks can be less than 2 inches in diameter, weighing a fraction of the IBM 350.

Disks have become increasingly intelligent in that period of time as well, and methods of accessing them have become increasingly simplified as more functionality has been abstracted away from the user by the disk. While operating systems—and often, users themselves—needed to be aware of the specific disk geometry details even a few years ago (e.g., the cylinder-head-sector (CHS) disk layout), this form of addressing has been rendered moot with modern disks (e.g., logical block addressing (LBA) exposes disks as a single linear array of bytes). Modern drives perform numerous functions that are unseen to the user, constantly scanning for bad blocks and remapping those parts of the disk to spare sectors, while performing other maintenance functions virtually the entire time that they are powered on.

These fundamental advances in hard disk capabilities have been the basis for proposals
Figure 1.2. Full-disk encryption drives are augmented with cryptographic processors.

to exploit these new features to supplement functionality typically handled by operating systems. For example, Riedel’s active disk proposal [17] aimed to define a new model for disks, based on the belief that these devices would contain their own independent computational abilities. This work in turn is supported by older research into database machines [18], which are specialized hardware elements designed to dedicate processors with individual tracks or heads on a disk, or to a disk itself. These devices would perform searches on databases to the disk, with specialized logic modules that offload processing to the units. The active disk concept mirrors the processor-per-disk architecture of database machines, with specialized logic used to perform operations such as nearest-neighbor search in databases and edge detection in images.

Using the disk to perform processing tasks external to the rest of the system was explored in a security context by Pennington et al. [19]. This work suggested the use of storage systems as intrusion detection monitors, given the independence of storage from the host operating system and using this as an impetus for describing how storage can enforce intrusion detection even if the OS is compromised. This work is predicated on Strunk et al.’s original proposal for embedding intrusion resistance in storage [20], but expanded beyond versioning and post-intrusion detection and recovery to proactively enforce detection rules. This work was important in establishing the potential for disks to be used as a policy enforcement point (PEP), and provides a basis for our investigations into creating a general architecture for storage that revolves around the increased benefits of reduced TCB afforded through storage-based security enforcement. As with the active disk proposal, storage-based intrusion detection, if enforced at the disk\(^1\), would require

\(^1\)The authors describe methods of providing IDS services at file servers, disk array controllers, and
the addition of extra capabilities within the disk.

Commercially, while disks have not implemented this type of specialized processing logic, other advances have been made within the drive enclosure. Recent introductions of hybrid hard disks such as the Seagate Momentus 5400 PSD [21] were designed with performance in mind; by providing non-volatile memory within the drive enclosure, operating systems could use this as an additional cache layer to prevent having to access the slower magnetic media. Specifically, these drives were designed to support ReadyDrive in Microsoft Windows Vista, an initiative to reduce power in mobile laptops [22]. Another advance in hard disk technology has been the emergence of full-disk encryption drives. As shown in Figure 1.2, these are disks that are augmented with processors which provide cryptographic services through customized application-specific integrated circuits (ASICs). These disks are made to protect laptops from compromise if they are stolen, as the disk itself contains a secret key that disallows any access by the operating system without the correct password supplied to it. As a result of these and other advances, disks have become capable of performing actions autonomously with regards to the rest of the system.

1.2 Thesis Statement

Based on the challenges that modern operating systems have enforcing data and system security, and the increasing capabilities of storage devices, we observe that storage is uniquely positioned within a computing system to act as a policy enforcement point. We thus propose a model of security enforcement that is regulated by the disk, according to augmented disks.
a policy specified by the user or administrator of the host machine. These policies will be
enforced by the disk’s firmware and as such, will render decisions made on read or write
accesses, or potentially more complex decisions, independently of the host computer.
Policies can be defined to enforce very different properties, each of which can further
augment the security of stored data and the host computer itself. We will show that
such an approach is viable and that the properties enforced are useful and significantly
add to the overall security of the system without compromising its usability and while
preserving system performance.

As shown in Figure 1.3, such a development would allow reducing the TCB to the disk
itself. In such an architecture, the disk can provide services independently of the operating
system that to date has largely controlled the disk’s functionality. Because of the general
difficulties ensuring the security of a large, complex codebase such as an operating system,
enforcement within the disk provides unique opportunities to enforce novel and to this
point, unexplored security services to storage. For example, a compromised operating
system kernel no longer necessitates that on-disk data is vulnerable, as access may be
independently enforced. As a result, it is possible for these storage devices to aid in the
protection of the operating system against malware and other threats that rely on data
compromise.

The central thesis of this dissertation is therefore:

_Storage that enforces security independently of the operating system can form
  a perimeter around data, reducing the size of the trusted computing base._

Our goals are not to claim that this approach is usable to the exclusion of all other
efforts to secure systems. Rather, we aim to show how autonomously-enforced storage
may benefit system security by offering novel applications that have previously been
difficult or infeasible to provide.

## 1.3 Contributions of Dissertation

We demonstrate our thesis by designing an architecture for disks that can act to enforce
security policies separately from the disk. Our approach demonstrates that security can
be best enforced by the user interacting _directly_ with the disk to ensure that security prop-
erties are enforced even in the presence of a compromised operating system. We show that
our approach is scalable to challenges of increasing complexity, from providing protections
within an operating system, to enforcement of properties between operating systems, and
ultimately describing how long-standing challenges such as enforcement of information flow throughout systems may be feasible through our model for autonomously-enforced storage. In doing so, we aim to fundamentally change the current understandings of what is possible with secure storage and fully explore the myriad benefits of autonomous security enforcement.

Our specific contributions are as follow:

- **Demonstrating the design and simulation of a disk that can autonomously enforce security policy.** We provide an architecture based on componentry found in modern commodity disks that demonstrates how new disks that can provide these new security services may be designed.

- **Minimizing the trusted computing base to remove the operating system and showing how disk-enforced policy can provide system security.** We consider both how an operating system can be protected from persistent attack through the use of storage, and how multiple operating systems residing on a disk can be protected from each other through policy enforced by the disk. Our *segmentation* approach controlled by physical access to the disk is the first example of this policy in use.

- **Demonstrating how storage can be used to bring the operating system back into the TCB.** We show approaches for using commodity trusted hardware on a host computer that allow us to validate the host’s integrity from the storage device. We use these techniques to demonstrate a commodity secure boot mechanism and extend the approach to encompass portable storage. Our mechanism for protecting data on flash drives from untrusted hosts is the first of its kind.

1.4 Trust Model and Limitations

It is important to understand what is actually required to be trustworthy with our approach, i.e., what should appear within the TCB. While our goal is to reduce the size of the TCB compared to current approaches, it is not realistic to claim that we can reduce it to only encompass the disk. Our methodology within this dissertation is to attempt to minimize the components required apart from the disk and to gradually expand our focus based on methods of verifying the trustworthiness of the host system. As we will describe in Chapter 4, we avoid relying on the operating system as a source of trustworthiness, relying instead on forming a *trusted path* between the user and the disk in order to establish a security relationship with the data stored. However, there are components we
must still be reliant on within the system. For example, we trust that the video signal to
the monitor has not been subverted; otherwise, a adversary could force the user to insert
a token at the wrong time or suppress the message to remove the token. We also assume
that the system does not become compromised during the installation or upgrading of
an operating system; we address this issue by ensuring that during the installation pro-
cess, all services not related to installation are disabled. There is also an implicit trust
assumption that the wiring has not been compromised on the host system, that the bus
carrying data to the disk is trustworthy, and that the adversary has not compromised
the system by exploiting direct memory access (DMA) on the system, which is possible
to do if a peripheral with a FireWire or eSATA interface are attached.

There are trusted devices and componentry that we are also reliant on. For example,
we assume that for approaches where a token is used, the token has not been compro-
mised and is resistant to tampering by outside agencies. Similarly, we assume that when
trusted hardware, notably the trusted platform module (TPM) is employed to validate
system security, as we discuss further in section 6, that it has not been compromised.
To date, there have been threats leveraged against the TPM, notably a physical attack
involving micro-probing of the silicon substrate from a chip in the Infineon SLE 66PE
microcontroller family, used to implement the Infineon TPM [23]. The attack, however,
is currently very time and equipment-intensive and requires the use of a Focused Ion
Beam, which is similar to a scanning electron microscope and operates at widths smaller
than 200 nm. It also requires applying a variety of solvents to dissolve various layers
of lithography. The threat with this attack is the potential ability to create counterfeit
chips, though this has not been yet demonstrated.

We do not provide any protections against social engineering, or “phishing” attacks
against the system. If, for example, a user is fooled into installing a malicious operating
system or malicious updates, the system will be compromised and since the software will
be installed to disk, it will be outside of the purview of our system to protect against.
In a similar manner, if an attacker gains physical access to both the computer and the
correct physical token to gain access capabilities to the disk, they will have breached the
trust confines of the system and have the potential to take full control of the system.
Similarly, if malicious information was installed to a system that relies on verification
via trusted hardware, and the malicious code is considered valid for verification against,
then the system will be compromised.

Finally, our primary assumption is that the storage device is trustworthy, as it acts
as a root of trust in our proposal. We believe that this is a valid means of protecting the
system on account of the smaller codebase associated with trusted hardware. However, as noted previously in this chapter, the increased functionality of these devices means that there is additional code complexity and that all of the components within the device need to be similarly trustworthy. If there are physical attacks against the disk, then it may no longer be trustworthy. Issues such as subverting the supply chain so as to install malicious componentry in the disk such as cryptographic chips or compromised magnetic media, non-volatile memory, heads, and armature assemblies, will render the storage device untrustworthy. So will having compiled the firmware using a malicious compiler in a similar manner to Thompson’s proposed attack [24].

1.5 Publications

Parts of this dissertation draw from, or are expansions of, the following publications:


Chapter 2

Foundational Work

The challenges of operating system security, and proposals to meet these challenges, have existed for decades. In this chapter we will consider a historical framework for using storage as an independent security enforcement point.

2.1 Database Machines and Augmented Disks

As discussed in Section 1.1, the use of disks as entities capable of performing processing independently of the operating system was considered in the 1970s with the proposed use of database machines. Storage has been optimized for processing since well before that time; in 1954, Hollander described how the TapeDRUM storage device could act as an intermediate processing element for tallying and subtotaling in enterprise-level inventory systems [25]. It was in 1970, however, when Slotnick devised the concept of a database machine, where processors were associated with the read/write heads of a fixed-head disk, thus associating processing logic with each track on the disk [26]. Each logic cell is connected on a global bus to communicate with a central processor. In this model, an entire database would be stored on the disk and with one disk revolution, the entire database could be read and queries processed without involvement of the CPU. This design is shown in Figure 2.1. The first full database machine was built in 1973, when Copeland et al. designed the CASSM system [27].

Database machines were an active area of research in the late 1970s and early 1980s [28], but the state of the art was negatively assessed by Boral and DeWitt [29], who argued that processor speed was multiplying at a vastly greater rate than storage capacity, rendering the need for on-disk processors moot. However, the rise of “shared-nothing” sys-
Figure 2.1. The original database machine concept was a “processor-per-track” model based on a fixed-head hard disk.

tems [30] reaffirmed the need for processing co-located with storage. In a shared-nothing system, there is no sharing of either memory (i.e., multiple processors sharing a common memory space) or disk (i.e., multiple processors sharing a common storage space). Designing shared-nothing systems introduces complexities because of the requirement for distributed deadlock detection, while load balancing must be accomplished by physically relocating resources across peripherals. In addition, there are additional concerns of cost and efficiency due to the lack of resource sharing necessitated by this approach. Despite these drawbacks, the shared-nothing architecture was extensively used in fault-tolerant systems such as the Tandem (now HP) NonStop system [31]; it and related parallel database systems were extensively surveyed by DeWitt and Gray [32]. Riedel et al. re-visited on-disk database processing by applying the Active Disk architecture to support parallel databases without requiring special-purpose microcode [33].

In a similar vein to the Active Disk research was the Intelligent Disk (iDISK) initiative [34]. This work was similarly motivated by the increasing appearance of embedded processors and memory within commodity disks, in contrast to the specialized hardware previously necessary in database machines. Another observation with the iDISK proposal was the increasing high-speed communication ability of disks, which is necessary for the decision support systems discussed—these disks were designed to work in clusters rather than forming independent perimeters of their own. Active disks themselves were considered extensively by others; Acharya et al. [35] proposed active disks that are stream-processing devices, programmable via disklets, disk-resident code that performs functions in conjunction with code optimized by the host. The Diamond prototype [36] built upon active disks with the addition of searchlets (similar to disklets) that are used as filters for the disk as it finds information on the disk. Diamond presupposes the use of objects,
rather than blocks, to provide additional semantic information. Memik et al. considered active and intelligent disks under the umbrella of smart disks and examined their functionality in terms of query processing and execution in decision support systems [37]. Operations were bundled together to be executed as single operations on the disks, with scheduling performed by a central unit rather than the disk itself. Later work considered a distributed architecture for I/O-intensive applications [38], where smart disks communicate directly with each other as with iDISKs but support a smaller codebase. Son et al. similarly considered clusters of active disks and focused on the improvements in energy consumption that these disk architectures allow [39].

These architectures showed how it was possible to perform processing within the disk, but did not consider the security implications of doing so, nor did they suggest security functionality that could be feasible as a result of having independent processing capability within the system. In addition, many of these architectures are designed to work in conjunction with the host and sometimes with each other, rather than performing any independent functions.

2.2 Security in Operating Systems

OS security is one of the deepest areas of security research within computer science, and its scope is tremendously broad. To concentrate the scope of this work, we focus on material that will be of particular importance later within the dissertation. The best overview of the history of operating system security is by Jaeger [40], who comprehensively describes systems from Multics to modern-day secure OS initiatives. Saltzer covered early work in the field as it was in process, summarizing the ongoing activities as ones that are still being actively researched, including system penetration, usability, correctness, and network security [41]. We cover OS security that uses additional trusted hardware in Section 2.2.3. We briefly described the origin of secure operating systems through the Multics initiative in Chapter 1. Many of the canonical principles of secure OS design were a result of Multics, and were put into axiomatic form by researchers and observers at the time. Of these original principles, perhaps the most fundamental is the concept of the reference monitor as described by Anderson [42]. Conceptually, a reference monitor in an operating system provides the notion that all references by a program to any other data, including program data, or device will be validated against a list of references that has been authorized and that describes available functionality by a user or another program, given as a set of privileges available to the program and often encapsulated in
an access control matrix, as defined by Lampson [43]. The three fundamental properties of a reference monitor are that:

1. The reference monitor is always invoked and performs complete mediation of all security-germane operations;

2. The validation mechanism of the reference monitor is tamper-proof; and

3. The reference monitor’s validation mechanism is verifiable, i.e., it is small enough to be tested and shown to be demonstrably complete.

Virtually every mechanism meant to protect the operating system may be cast in terms of reference monitor principles, and we do the same when describing our work in subsequent chapters. We also make use of the concept of capabilities, data objects that are unforgeable and communicable representations of privilege. Capabilities must be held by a process to access a particular resource with a given privilege, and provide a means of providing authority and preventing privilege escalation, a fundamental problem typified by the confused deputy problem [44]. Capabilities were first deployed as a means to provide protection in operating systems in the Hydra project [45], allowing for separation of policy and mechanism and providing a means for operating system components to be deployed outside of the kernel as user-level processes [46]. Capability-based operating systems were further developed, particularly with KeyKOS [47] and EROS [48], which is notable for providing a capability-based operating system that runs on commodity hardware (the Intel Pentium).

Another fundamental concept that arose from early work in operating system security was that of confinement [49]. The principle of confinement is the assurance that information being processed by a program will not leak, particularly to malicious processes. This principle is the basis for research into isolating processes from each other within operating systems, as well being the basis for the subsequent concerted research into systems and policies that preserve information flow. Of these systems, the most notable early approaches were multi-level security policies first considered by Bell and LaPadula [50] to protect the confidentiality of sensitive information, and similar lattice-based policy devised by Biba [51] for protecting the integrity of information. Myers and Liskov first considered the concept of decentralized information flow control [52] where processes and not just administrators could make decisions relating to allowing access based on information flow primitives. While this was an approach largely rooted in programming languages, leading to the genesis of languages such as Jif [53], recent work has
also considered applying decentralized information flow primitives to operating systems, leading to labeling operating systems such as Asbestos [54] and HiStar [55], as well as Flume [56], a user-level reference monitor for process confinement. We now examine other approaches for separating processes that have been explored in the literature.

2.2.1 Process Isolation

Numerous approaches have been employed to provide isolation between processes in order to maintain the principle of confinement. Robinson et al. [57] considered a methodology for secure operating system design that considers a system as a hierarchical collection of abstract machines, using proofs of correctness of small abstract programs and composition to determine a correctness proof for the system as a whole. These proofs would include the concept of isolating one user’s resources from another, typified by the use of a security kernel such as Multics. The concept of segmentation appeared in this work as well, in the context of virtual memory and protected by capabilities, which at to protect memory access from users without appropriate credentials. Security kernels reached the apex of their development with the Scomp [58] system (described below), which relied on trusted hardware, and Gemini’s GEMSOS system, one of the very few operating systems formally validated and verified to the Trusted Computer System Evaluation Criteria’s level A1 [59].

Verifiable kernels such as UCLA Secure Unix [60] formed the basis for Rushby’s observations about the difficulty in performing this verification [61]. Namely, the problem was seen to be that a single monolithic security kernel attempts to specify policy for the entire system; Rushby’s major insight was that processes should be distributed such that isolation can be afforded. Rushby considered a full-scale distributed system with machines isolated from each other for running processes in isolation before creating the concept of separation kernels, which would afford the same level of inter-process isolation on a single microprocessor. A formal proof of separability further demonstrated how formal verification techniques could enforce the concept of isolation [62]. This investigation into separation of processes led to the concept of microkernels, which attempted to separate all processes not essential to kernel operation into user level processes. These concepts were fundamental to the designs of the Amoeba distributed operating system [63], which looked to connect multiple computers to appear as a single system, and the Mach microkernel [64], designed to augment the functionality of UNIX with privilege separation. The major issue with these designs is the increased reliance on inter-process communication (IPC) due to the movement of many frequently-accessed services out of the kernel,
leading to decreased performance thanks to context switching and accounting. The L4 microkernel [65] was a major development in this area as it led to a significant decrease in IPC. This was due to optimizations such as aggregating messages to reduce the number of system calls, performing direct process switches using registers, preventing any copying from occurring, and lazy scheduling, which attempts to prevent manipulating ready and waiting queues for scheduled processes.

Another important means of providing isolation is through the use of virtualization. Madnick and Donovan [66] outlined how this could be accomplished and why it is important from a security standpoint. They described how a “bare-machine” virtual machine monitor, which acts to only multiplex and allocate physical hardware resources, could improve security, and show that the probability of a security violation with a VMM system is much lower than without. These concepts already existed in mainframe systems at the time such as the IBM VM/370 operating system [67], but subsequent operating systems made use of the VMM specifically from a security approach, notably the VAX security kernel [68]. Disco [69] demonstrated that it was possible to run commodity operating systems on a shared multiprocessor, and allowed for multiple OSes to communicate using standard distributed protocols such as TCP/IP and NFS through a global buffer cache for efficiency. This approach to providing virtualization through commodity operating systems was commercialized, notably with VMWare [70] and Xen [71]. The Denali isolation kernel [72] was designed specifically with inter-process isolation in mind, exposing an abstraction of the x86 architecture in order to gain performance.

### 2.2.2 System Threats from Rootkits

We now turn to discussing the threats against systems that have necessitated the continued development of defensive security mechanisms. While an entire universe of threats exist against a system, we focus our investigation on malicious code. Of the malicious code, or malware, that is being developed and that has been deployed against computer systems, perhaps the most pernicious threat stems from the rootkit. As their name implies, rootkits comprise one or more pieces of software developed for the purpose of achieving administrator access, or root, on a system, and to keep this access in a manner undetectable to system resources and users. In effect, they replace operating system functionality such that true system state is hidden.

Rootkits have been well studied, and those that attack the operating system and reside in the kernel have been demonstrated in both theory and practice, dating back to the author Half-life’s redirection of the `setuid` system call to a compromised version through
loading of a rogue kernel module, allowing for the hijacking of a TTY [73]. Rootkits can be user-mode programs that perform functions such as adding inline hooks into system functions or patching runtime executables dynamically (e.g., system commands such as `ps`, `netstat`, and `top`), or kernel-mode programs that hook into the kernel, layer themselves onto device drivers, or directly manipulate the OS kernel, and sometimes the hardware itself [74]. Rootkits can be persistent, where they survive a system reboot, or non-persistent, where they install themselves into volatile memory and do not survive across reboots [75].

Numerous techniques for hiding rootkits have been implemented, including modification of system files and libraries [76], boot sector modification [77], and altering the ACPI code often stored in the BIOS [78] – this approach may potentially even evade detection by trusted hardware such as a TPM, by causing it to report correct hash values [79]. While many of these attacks can be fended off through integrity protection mechanisms [80,81] and kernel-level rootkit detectors [82,83], increasingly sophisticated rootkits can evade this level of detection. Such attacks can subvert virtual memory [84] or install themselves as a virtual machine monitor (VMM) underneath the operating system itself [85], demonstrating that whoever controls the lowest layer of the system gains the advantage in attacking or defending it.

Rootkits themselves are not used to exploit a system, but are often used in conjunction with exploits to maintain a persistent presence on a system after it has been compromised. In this sense, they often share commonalities with programs such as Trojan horses [24]. Software to exploit systems has been a topic of extensive and ongoing research. Tools that generate exploits are readily available [86], and defending against malicious code, particularly if it is polymorphic, is extremely difficult. Identifying polymorphic viruses bounded in length has been shown to be NP-complete [87], while modeling the polymorphic attacks (such as polymorphic blending attacks [88]) requires exponential space [89]. The transmission vector for these exploits is often a worm [90], which can compromise large numbers of machines in very short time periods [91].

Numerous proposals to defend against rootkits have varied in their complexity and coverage. Signature-based schemes such as `chkrootkit` [92] are limited in that they rely on the operating system to correctly scan for rootkits, which may have subverted the OS to protect against these defenses. Rootkit scanners that are implemented as kernel modules (e.g., `rkscan`) [93] provide better protection, but can only detect a rootkit when it is present, potentially allowing it to have subverted the kernel to protect against these scanners. Kruegel et al. [94] present a scheme to detect rootkits by checking kernel
modules at load time, but this does not protect against a kernel code injection that bypasses the module loader. Once the rootkit is installed, it can modify the boot sequence.

General malware tracking schemes such as Panorama [95] may be useful for preventing rootkit installation but exact a very heavy performance penalty.

### 2.2.3 Autonomous Security

While much of the storage research did not consider how to use the functionality of independent storage elements for security purposes, those in the security community have been examining independent security processors since the 1970s. Honeywell’s Project Guardian, which led to the Scomp system [58], called for a secure front-end processor to control access to the rest of the Multics operating system. Scomp included a security protection module, which sat on a bus between the Scomp processor and mediated access to the I/O and memory. Honeywell also began work on the Logical Coprocessing Kernel (LOCK) [96] program (successor to the Secure Ada Target, or SAT, initiative [97]). One of the keys to this architecture was the inclusion of a hardware coprocessor tasked with making security decisions for the system, called the System Independent Domain Enforcing Assured Reference Monitor (SIDEARM). Processes running on a LOCK system would use the SIDEARM to mediate access to files and objects, thus providing a security enforcement point separated from the rest of the system. While the SIDEARM was separated from the rest of the system, the operating system running over LOCK needed to be modified to redirect all security-sensitive operations to the SIDEARM for mediation.

The Security Pipeline Interface (SPI) [98] was a first attempt to provide a security processor that is interposed directly between a host’s CPU and its I/O interfaces without requiring operating system modifications. As shown in Figure 2.2, SPI was designed to be applicable between any data flow pipeline in a system, and could be a hardware element sitting on a chip or a device that intercepts communications to a network. Placing security in the microprocessor itself was proposed by Best [99], but this model did not provide

**Figure 2.2.** The Security Pipeline Interface (SPI) interposed a security processor between the CPU and I/O interfaces.
separate isolation of security functionality. It was, however, an influential model that led to proposals such as Dyad [100]. Dyad operates on the IBM Citadel prototype, which was a secure coprocessor system. The Dyad secure coprocessor consisted of a processor, ROM, and secure non-volatile memory, and could perform functions such as determining host integrity. The secure coprocessor itself was the basis for further development by IBM [101] and came to commercial fruition as the IBM 4758 [102], which supported functionality such as tampering countermeasures and code-signing techniques to ensure that only authorized code was loaded into the co-processor’s memory.

A successor to Best’s model, called AEGIS [103] proposes the use of a single-chip processor that is the basis for a secure chip platform, using techniques such as physically uncloneable functions (PUFs) as a means of authenticating the chip. By linking physical properties of the chip through characteristics such as logic delays, by which the PUFs are generated, there is an assurance that modifications to the chip will modify these properties and render access to the keys impossible.

Separating security functionality from the rest of the system led to the development of smart devices that could be easily physically separable, such as smart cards. These were designed to contain their own independent memory and processing, and software was designed with this functionality in mind, culminating with the BITS operating system, which derived its security from a smart card [104].

2.3 Secure Autonomous Storage

2.3.1 Securing Data with the Filesystem

Using the filesystem is one of the most commonly used approaches to handling data. Cryptographic file systems such as CFS [105], TCFS [106], and CryptFS [107], provide data protection through encryption at the file system level, allowing users to encrypt at the granularity of files and directories. Other schemes that provide volume-based encryption, e.g., SFS [108, 109] operate transparently to the user but do not provide granularity at a file or directory level.

Schemes such as SNAD [110], which seek to secure network attached disks, or SNARE [111], which provides key distribution across multiple storage devices, require the use of an ancillary metadata or authorization server. SCARED [112] provides data integrity but not at the block layer, so operations cannot be performed by the disk; Oprea provides integrity at the block [113] and file system layer [114], both relying on correct workload characterization to parameterize an entropy threshold, and requiring a source of trusted
storage. Aguilera et al. [115] consider block-based capabilities that rely on access control at a metadata server. All of these solutions provide cryptographic services but do not protect the operating system against exploits.

Plutus [116] allows for the secure storage of data through the filesystem without relying on storage servers to provide guarantees of secrecy or integrity. Individual users are responsible for cryptographically securing the files, and key management and distribution are handled directly by individual clients, rather than metadata servers. Mechanisms are provided to detect and prevent unauthorized modifications to data, to differentiate between read and write file access, and to change user access privileges. Encrypting data directly to disk offers more security to the storage devices, as data leakage through compromises to the server (as well as more prosaic methods of acquiring data, such as stealing server disks) is mitigated. Users can have greater control over their data, as they can set arbitrary policies for key distribution and file sharing. One necessary assumption when discussing these solutions are that the servers will store the data, but cannot be trusted to keep it confidential. The server may attempt to change, misrepresent or destroy data on the system. Plutus does not explicitly deal with servers that attempt to destroy data, but multiple servers are considered a means of assuaging this concern. Blanchet and Chaudhuri analyzed the Plutus protocol using the ProVerif automatic protocol verifier [117] and found a new attack against its integrity. This was the first automated analysis of a storage protocol, and the formal analysis resolved some inconsistencies and ambiguities within the original Plutus paper.

SiRiUS [118] is another proposal that provides encryption in a similar manner to Plutus, but keys are stores and distributed by the server rather than in the decentralized, user-focused manner employed by Plutus. SUNDR [119] is another proposal for security at the file system, and it adds a guarantee called fork consistency for defending against rollback attacks, which attempt to subvert a storage system by allowing information to be retrieved by a user but not more recent information that has been sent to the filesystem by another user. SUNDR accomplishes this without requiring trust in the storage server.

All of these schemes that rely on the filesystem to provide protections are by extension relying on the operating system kernel to be trustworthy. If the kernel is malicious, it may direct the filesystem to arbitrarily subvert the security of any files within the filesystem’s purview.
2.3.2 Augmented Secure Disks

To solve the problem of securing data without relying strictly on the operating system, researchers turned to incorporating hardware-based approaches that used storage devices themselves as a means to protect the security of data. The first on-disk security proposal was Carnegie Mellon’s Network Attached Secure Disks (NASD) project [120,121]. As shown in Figure 2.3, NASD aims to improve throughput and server scalability by providing direct access between clients and storage devices with minimal amounts of server communication. Bypassing the server allows for the elimination of “store and forward” situations where the server must act as an intermediary, decreasing overall performance. The client makes a request to the metadata server, which performs all access control and consistency functions, and provides the client with a capability to access a set of information on specific disks. The client then directly communicates with the disks for its I/O. The disk receives capabilities from the server through a shared secret channel. In the NASD scenario, the server’s role is less about managing individual requests; the resources that are available are instead used for tasks such as policy definition, cache consistency, and namespace management.

The other major change effected through the NASD initiative is the way in which storage is considered. Up to this point, storage solutions worked at the granularity of blocks. The base unit exported by the storage devices are blocks of a fixed length, with no additional metadata regarding the block included. By contrast, the object paradigm, pioneered by NASD, considers variable-sized objects to be the base unit exported. Objects contain an attribute set, metadata exported from the drive, and a sequence of bytes,
thus giving a much richer context association. Objects can correspond to an entire file, or a file fragment, such as a database table, or a data stripe unit.

In NASD devices, data is encrypted on the wire, meaning its confidentiality is protected through transmission, but it remains in cleartext on the disk itself. The encrypted data is integrity-protected through the use of message authentication codes on message checksums. Since data is stored in the clear on NASD storage devices, the system is vulnerable to attacks where the adversary is able to collude with the storage system. Additionally, since all authentication and authorization data is available through the object server, the system is also vulnerable to attacks where the adversary colludes with the object server itself. The cache storage is a point of concern, given the potential access to capabilities that is afforded, particularly if the adversary colludes with the storage server. Furthermore, NASD was the first scheme that relied on individual drives directly participating in security protocols. On the one hand, this potentially increases scalability as computation is distributed; but it also creates the increased potential for malicious activity.

NASD has had a significant impact on the design of secure disk architectures. The Object-based Storage Device (OSD) initiative is the successor to NASD [122]. Much of the OSD protocol, particularly the security model, is strikingly similar to what was originally proposed with NASD. Note that with both OSD and NASD, the disk is not truly autonomous as it relies on a metadata server to provide a significant portion of the security functionality. More recently, Factor et al. provided a new architecture for enforcing access control within storage area networks that leverages the OSD model by extending the SCSI command set [123].

Storage that independently enforced security properties was proposed by Strunk et al., with Self-Securing Storage [20]. These were capable of isolating storage and providing extra data inaccessible to the rest of the system, notably an audit log that could be checked to ensure that violations against specified files were not occurring or could be recovered. Pennington et al. [19] continued working along this vein and, as described in the previous chapter, described disks that were independently secured and capable of performing functions such as intrusion detection [124] (a use model that was described with SPI). More recently, this idea was expanded to include using the disk as a general detector for malware, which Paul expanded upon in his dissertation [125]. This work does not consider a general security architecture for the disk, however, relying more on the disk to cooperate with the host system’s AV scanner (and hence being more akin to a smart disk solution for security).
Sundararaman et al. [126] also considered disk-level policy to provide secure, selective versioning of data, requiring additional capabilities from the disk in the form of type-safe disks [127] capable of distinguishing between inodes and data. These solutions require tighter coupling between the disk and the filesystem; in addition, they rely on the operating system to work in tandem with the disk.

The need to access storage from portable devices and the security problems that consequently arise is a topic that has been well noted. SoulPad [128] demonstrated that the increasing capacity of portable storage devices allows them to carry full computing stacks that required only a platform to execute on. These devices contained an auto-configuring live operating system image, which in turn booted a virtual machine monitor and virtual machines, thus decoupling code and data from the underlying hardware on which it executes. However, platform security was not considered.

Another example of enforcing on-disk policy is through write-once, read-many (WORM) mechanisms enforced through hard disks, commercially available through vendors such as EMC [129] and IBM [130]. These mechanisms have been shown to be vulnerable to insider attack [131]. Sion [132] has explored methods of ensuring policy enforcement at the storage level while considering the limitations of trusted hardware.

2.4 Trust Establishment

Understanding how to establish trust in a computer is one of the more challenging research problems in systems security. The differences between a system that has been cleanly booted, running free of any undesirable software and a system that has been compromised by malware can be non-obvious, particularly on commodity computing systems. We highlight some of the work in this area that is most relevant to our investigation below. An extensive review of the literature may be found in Parno et al.’s survey of trust bootstrapping techniques [133].

2.4.1 Trusting the Boot Process

One common threat to system security is the boot process. The presence of malicious code during the early boot phase (such as in the BIOS or bootloader) can compromise the security of the full system by disabling security mechanisms or loading additional malicious programs. Two categories of approaches, authenticated boot and secure boot, have been used to combat this threat. In both secure and authenticated boot, trust is axiomatically placed in an initial component that starts a code measuring processes.
The first approach, **authenticated boot**, enables a remote verifier to inspect the integrity of the host’s boot process. This is done by performing measurements of every phase of the boot process before executing it and storing those measurements in a secure log. Devices that perform authenticated boot often store these measurements in hardware and use cryptographic keys to link the measurements to the physical machine being measured. The major drawback to authenticated boot is that it offers no security to the host. Even if a malicious program is measured, it is still running on the system. An alternative to authenticated boot is **secure boot**. A secure boot approach also measures code before loading it, but denies its execution if it does not meet some requirements. The end result is a system running only trusted code because only such code can be loaded.

The IBM 4758 secure coprocessor described above allowed for implementation of **trusted boot** functionality as described by Gasser et al. [134]. A trusted boot includes **authenticated boot**, where the system’s boot was verified by an external device, and **secure boot**, where the boot process was staged such that the next stage is only performed if the current stage has been verified\(^1\). The AEGIS system by Arbaugh et al. [135] allowed for such secure booting capabilities by checking each phase of the boot process against a special PROM that contained trusted measurement definitions. AEGIS makes the assumption that firmware in the physical hardware, as well as portions of the BIOS, are trusted. The Copilot integrity monitor [136] is also based on a secure coprocessor, which runs on a PCI board that monitors the system, but significantly differs from AEGIS by not focusing on the boot process, but by verifying that the system’s kernel is in a good state during runtime operation.

Secure coprocessors are expensive, due to the high cost of assuring their physical security. The search for a cost-effective independent security device accessible to the host machine led to the development of the Trusted Computing Group’s Trusted Platform Module (TPM), which provides some functionality such as start-up keys. It is not, however, meant to take the place of a secure co-processor. Attacks against the TPM are possible, as detailed in Chapter 1. Even simpler attacks may potentially circumvent some of the protections provided by the TPM. For example, Kauer demonstrated that a particular vendor’s TPM had a Low Pin Count (LPC) bus was vulnerable to attack, and that by connecting the device’s reset pin to ground, it was possible to reset the TPM without resetting the platform as a whole [79]. In effect, this would allow the unsealing of data protected by the chip. Other attacks may yet surface against the TPM, particularly

\(^1\)There is some debate as to whether “trusted boot” encompasses authenticated and secure boot, or whether trusted boot and authenticated boot are synonymous. For clarity, we differentiate between trusted and authenticated boot.
if Tarnovsky’s attack proves to be possible to perform in a less costly and time-intensive manner. Such physical attacks against hardware have been amply demonstrated in other venues; recently, the Mifare RFID tag used in public transit card systems was successfully reverse-engineered through the use of innocuous chemicals and matching etched silicon groups against known templates using image processing techniques [137].

2.4.2 Reducing the TCB

Measuring a system's trusted computing base is simplified by reducing the size of what should be held inside it. Such work has taken place at the layer of hardware, within the operating system, and through the use of virtual machine monitors. An example of the latter approach is Terra [138], which proposes a full trusted computing architecture predicated on a trusted virtual machine monitor (TVMM), which partitions a hardware platform to allow multiple trusted virtual machines to run. The means for ensuring that the TVMM is trustworthy, however, is not discussed. SecVisor [139], a hypervisor that uses the IO memory management unit in a system (IOMMU) to protect code from DMA writes and also virtualizes both the IOMMU and the CPU’s MMU. Through the small size of the hypervisor and mechanisms built into it, code integrity of the kernel is maintained, but vulnerabilities within the kernel itself are still possible to exploit. Overshadow [140] also uses virtualization, by leveraging extra indirection that occurs as a result of virtualizing memory within an VMM, due to the different views a memory page may represent depending on the guest OS accessing it. Memory pages are encrypted and viewable as cleartext only by the correct guest accessing the page. In this manner, there is a reduction in the amount of trust necessary within the OS, but these approaches require the use of a VMM, which is itself potentially subject to compromise.

The Flicker security infrastructure [141] is a means of reducing the TCB through trusted boot mechanisms. Routines considered critical to security may be run in custom routines called pieces of application logic, or PALs. When these are executed, the system uses the late-launch capability based on a dynamic root of trust to boot into a small protected environment where only the particular logic can run. Because it runs on the system itself, this means of operation is complementary to performing security at the disk itself, but suffers from performance drawbacks on account of the frequent need to invoke the trusted boot routines. TrustVisor [142] combines concepts from Flicker, notably PALs, with the idea of a minimal hypervisor in an attempt to improve performance. Both of these approaches require the rearchitecting of legacy code if it is to be used with the system, as applications need to explicitly have PALs developed for them.
2.4.3 Devices for Trust Establishment

In order to ensure that a platform is trustworthy, it is often helpful to be able to validate this trustworthiness through use of a known trusted device. In the DeviceSniffer proposal [143] a portable USB flash drive allowed a kiosk computer 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 layers. It is also susceptible to being run within a malicious VMM such as SubVirt [85]. The iTurtle [144] 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 in nature and postulated questions 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 [145]. A framework is provided for using the mobile device to establish trust in the platform by measuring its integrity through a trusted boot process. This work makes different assumptions about how portable devices will be used to provide the computing environment; in the proposed model, a mobile phone is used as the authenticating device and relies on a barcode attached to the platform and transmitted wirelessly to the device. Because the verifier is not a storage device, the virtual machine to be run is encrypted in the cloud.

BitLocker [146] is a disk volume encryption feature built into Windows Vista and Windows Sever 2008 that optionally leverages the TPM to protect a volume encryption key (VEK). Bitlocker functions by placing the Windows system volume in an encrypted partition and using the TPM to seal the VEK to a known integrity state. The TPM seal operation encrypts the VEK along with a the set of PCRs. When unsealing the VEK, the TPM’s current PCR set must match the PCRs specified with the key or the TPM refuses to decrypt the key, which effectively denies access to the system volume. The target set of good PCRs are selected by measuring the early boot phase of the Windows system before encrypting the disk. When booting, the TPM measures firmware, the boot loader, and other early boot phase files. Modifications of these files will be detected when the VEK is attempted to be decrypted.

One issue with BitLocker is its inability to protect the disk partition after releasing the VEK to the OS. Since Bitlocker is designed to facilitate just secure boot of the OS,
later compromise of the OS system will go undetected. Another concern with BitLocker is the management of the VEK during firmware updates. In BitLocker, the VEK is placed unencrypted on the disk while an administrator performs changes that would affect the early boot measurements.

2.4.4 Trusted Storage

The Trusted Computing Group’s Storage Work Group has described specifications for interfacing with storage devices, considering the storage device to be a Trusted Peripheral (TPer). The primary specification in support of this effort is the Storage Architecture Core Specification [147], which primarily describes how to negotiate and establish a secure channel between the trusted disk and the host system, largely predicated on the existence of trusted send and receive commands specified in the SCSI and ATA specifications. In particular, the Storage Interface Interactions Specification (SIIS) [148] describes the mapping of secure events, such as sending and receiving secure information to their equivalents in the SCSI command set [149] (i.e., SECURITY PROTOCOL IN/OUT) and ATA command set [150] (i.e., TRUSTED SEND/RECEIVE). The core specification provides a StartSession method for the disk and host to establish communication with each other and provides the basis for a portion of our key establishment with Firma. There is also support for pre-boot authentication, defined in the Opal SSC standard [151].

The specification does not consider mechanisms for trusted boot processes such as secure boot, nor does it describe a methodology for providing attestations of the system state to the disk from the host; this model makes the assumption that the host is implicitly trustworthy and it is the peripherals that could be potentially malicious, and there is no means for a TPer to itself validate whether the host is trustworthy or not. The implications are that once the system is considered to be in a “good” state prior to boot, no further checks of system state are considered. Note that there are no current disks that support the trusted extensions to SCSI or ATA; many of the standards are still in the draft/discussion stage without any reference implementation.

2.5 Usability Concerns

The solutions we present in subsequent chapters rely in part on providing a capability to storage that emanates either from the user or from the host. In Chapter 4, we will see how the use of tokens can provide a trusted path between a user and a disk. A critical factor that will affect the deployability and eventual success in the commercial marketplace of
these token-based solutions, however, is ensuring that they are usable by administrators and users. The use of tokens raises questions about how to deal with the management situation that arises.

Usability and security are often design points at cross purposes with each other. Increasing the usability of a system often provides an increased vector for attack, while increasing security measures can make the system either aggravating to use or completely unusable for the average user. It is of paramount importance for these two concerns to be balanced against each other, as systems deemed to be unusable by their users will be circumvented or abandoned altogether in favor of other systems that do not impose restrictions as onerous, even if they are inferior or less effective from a security standpoint; Hitchens showed that if security mechanisms lacked usability, their effectiveness was curtailed [152].

In their seminal paper on information security, Saltzer and Schroeder described the Principle of Psychological Acceptability, in which they state that human-computer interfaces must be easy to use “so that users routinely and automatically apply the protection mechanisms correctly” [153]. Adams and Sasse further identified the need for user-centered design in security and the need to align users, who often lack security knowledge though conscious of the need for it, with the necessary practices [154].

2.5.1 Threats from Improper Token Usage

By interfacing directly with a disk through the use of a token, there are certain threats to the user that may be exploited:

- The user may forget their token in the disk. As we describe when discussing rootkit-resistant disks (RRDs) in Chapter 4, this leads to the possibility that blocks which should be considered immutable will be able to be overwritten, defeating the protections that should be provided.

- The user may improperly insert their token into the disk and consequently not be able to access the functionality that he or she expects. With RRDs, this could mean not being able to overwrite blocks. While with the SwitchBlade prototype described in Chapter 5, access to desired OSes on the disk may be disallowed.

- The wrong token may be inserted into the disk, providing undesired functionality to the user. With RRDs, an administrator token could be inserted instead of an token, providing a privilege escalation to the user. With SwitchBlade, the wrong
token placed in the disk may provide access to the wrong set of OSes or an undesired security level.

2.5.2 Insertion vs. Proximity-Based Tokens

Corner’s dissertation discussed the issues of using tokens in the context of user authentication [155]. A taxonomy of solutions was provided against an axis of transparency, or the amount of work that a user needs to perform to use the authentication method, versus granularity, defined as the amount of time between an authentication even and the user’s request that depends on it. Corner claimed that inserted tokens, while providing fine granularity, suffer from a lack of transparency because the user must remember to remove the token from the device.

An argument can be made that mitigating the issues of a token left in the drive may be the greatest deterrent to usability of an inserted-token approach, and an argument for a proximity-based wearable token solution as found in ZIA [156] may be more appropriate. Nicholson and Corner investigated system-wide transient authentication through proximity-based tokens and described threats to that model [157]. Namely, this security design is dependent on the wireless token having a range that is both limited and deterministic, and it is possible that an attacker to eavesdrop on the signal and to execute a wormhole attack, whereby repeaters could be used to fool the token and device into thinking they are closer in proximity than they actually are. This technique does not have methods that are provably defeatable. Moreover, in the context of our technique, proximity-based authentication is not a tenable concept because we are interested in more than the user being authenticated to the drive; rather, our model must ensure that the user is explicitly cognizant of her decision to execute a secure operation. Insertion-based tokens are hence an improved solution as they force the user to perform a physical action. It is possible that a wearable token could provide similar abilities if, for example, the user was to push a button on the token that enabled it. This may mitigate the issue of token loss, but such a model nullifies many of the advantages of transient authentication (i.e., the process is seamless to the user) and leaves open the question of how to ensure that the user ends the session with the drive using the enabled token. If the user needs to push the button again to disable the token, then the same issues as removing the token from the physical enclosure exist, e.g., what if the user forgets to perform this operation? If the token automatically times out after a certain amount of time then a window exists for unintended privileged operations to occur. Because of these issues combined with the previously mentioned attacks against wireless tokens, this model does not appear to be
appropriate for our goals.

2.5.3 Tokens vs. Secure Cryptographic Channel

Griffin et al. investigated the feasibility of intrusion detection within workstation disks [124]. They faced with the challenge of how to provide an administration interface to the disk in order to specify and enforce policies. Their solution includes an administration console that directly communicates with the disk using the host as an intermediary. This is achieved through the construction of a cryptographic channel between the console and the disk. The disk must share a symmetric key with the administrative console. The host computer is further modified to include administrative bridging software that demultiplexes TCP packets and repackages requests to the disk as SCSI packets with a modified command descriptor block. The disk is polled by the administrator to determine whether any policy violations have occurred.

Creating a secure cryptographic channel has some appeal for our model. It would eliminate the user from having to be in physical proximity to the disk, as is required with a token insertion operation. It also allows remote policy administration. However, there are significant drawbacks to this approach. The approach relies on the constant maintenance of a secure channel between the administrator and the disk. Not only does this approach not allow for offline operation of the disk, but it also allows an adversary an opportunity to intercede in the path between the administrative console and the disk. The authors acknowledge that the communication path is vulnerable to disabling if an attacker has compromised the system. Not only is this true, but an attacker who has positioned himself between the console and the host on a network interface can effectively block communication between the two entities. The authors argue that a disabling of the channel could be viewed as a policy violation and acted on accordingly; however, service to the disk has now been denied. In addition, by requiring the opening of the network interface during administrative operations, the disk is vulnerable to remote access and modification while in this mode. By contrast, the token-insertion model does not require the network interface to be available when the system is booted into a state where token insertion is required; it would be preferable if the interface was disabled on the host system.

Additionally, by requiring remote administration, the question of when the token should be removed from the disk is raised if an adversary is able to interpose themselves within the communication path. The disk could wait for a command to remove the token, which could be blocked by the host, or it could time out after an interval of not
communicating with the administrative console. In the latter case, an window exists for
an attacker to perform malicious activities with the disk (a time-of-check-to-time-of-use
(TOCTTOU) [158] condition). This is not present when the token is physically removed
from the disk, as at this time, the credential provided by the disk is immediately revoked.
Of course, if the host computer is compromised then it can arbitrarily drop or tamper
with any received request from the administrative console. Finally, key establishment
between the disk and the administrative console is not considered in this model, whereas
our model provides a direct physical interface through use of the inserted token.

2.5.4 Studies of Token-Based Interfaces

Possession of a hardware token with appropriate credentials corresponds to the “something I have” model authentication factor, compared to knowledge of a fact (e.g., with a
password) or something about a user (e.g., biometrics). Little work has involved the use
of tokens that may be plugged into a USB interface. Piazzalunga et al. [159] performed a
usability test of a variety of security devices including USB tokens and how their usability
compared with smart cards. Their tests, which required users to send three protected
emails, showed that employing USB tokens required significantly less time than solutions
where users required smart cards, from under 6 minutes with the two classes of USB
tokens employed to a mean of 10.7 minutes using a smart card. Their tests also showed
that users were often more confused about how to use a smart card versus a USB token,
and this confusion often stemmed from a lack of understanding about how the smart card
connects to its reader. One of the main results from this study dealt with user mobility,
with users moving between workstations to complete their tasks. The tests showed
that users made an average of 2.0 errors when using smart cards compared to 1.3 errors
when using so-called “base type” USB tokens, and 0.3 errors using “advanced type” USB
tokens. The base type USB tokens are simple with no additional functionality, while
the advanced type tokens used by the authors also included a mass storage component,
isolated from the smart card component within the token, allowing the users to use it in
addition as a storage device. This test shows that by increasing the functionality and the
general value of the device to the user.

The only other empirical study to our knowledge that has been published describing
usability of token-based authentication was performed by Weir et al. [160]. This study
differed considerably from Piazzalunga et al. in that it considered 2-factor authentication
as used in electronic banking applications. As such, the devices studied were push-button
tokens that provided an access code to be keyed into the authenticating device, a card-
activated token whereby the user inserts a card into a reader and pushes a button to display an access code to be keyed in, and a chip-and-PIN approach where the user inserts a card into the reader and enters a PIN in order to display an access code that is subsequently entered. Unfortunately, these approaches differ from our usage model for tokens inserted into a disk, but the results are instructive insofar as they give feedback from study subjects as to the perceived convenience and usability of token-based security. Users were required to log into an electronic banking site using graphical directions supplied on a computer screen, and to log off to end the experiment, with three incorrect logins requiring the equivalent of a support call and device reset. As intuitively expected, the simpler the authentication mechanism was, the less time was required for tasks to be performed. In particular, the mean task completion time with the push-button token was 25.20 seconds, versus the 117.8 seconds required with the chip-and-PIN method. In addition, the push-button token scored highest in terms of satisfaction with usability compared to the other methods. More complex authentication methods scored higher in terms of perceived security but the quality of the experience and the convenience was overall much lower, though subsequent use of each device showed that every device was considered more usable with increased exposure and usage. However, the results of the study led the authors to surmise that concerns about security, even with transactions of as great personal import as electronic banking, do not override convenience and usability as factors that are desired in an authentication solution. It is difficult to extrapolate from these results whether similar factors would be at play with token-based security for disks, and the demographics and use cases for these devices may differ considerably (e.g., a system administrator in an enterprise could be more tolerant of a convenience-security trade-off). However, a central result is that making security tokens as easy to use as possible will result in users using the devices properly while minimally hampering convenience and usability.

Lu and Ali recently proposed methods of securing the communication path between a USB hardware token and a host computer [161], which does not pre-supposed that either the communications to, or applications on the host are trustworthy. Their solution describes the means by which keys may be shared between the host and the token and consider biometrics through use of a thumbprint. These methods may be appropriate for dealing with key management, encryption, and integrity issues that may arise between the secure token and the drive; however, the authors do not consider usability issues as a part of their model.
Chapter 3

Designing Disks to Autonomously Enforce Security

To understand how autonomously-secure disks may be used to solve innovative security problems, we first describe how they may be constructed based on the technologies found in current disks, and provide some preliminary use cases to provide the reader with some insight into how these devices may fit into a general security infrastructure. As described in Chapter 1, the existence of hybrid hard drives provide small but fast non-volatile memory to coexist with slower, spinning magnetic disk media. To date, these architectures have been used primarily for improving performance, using the non-volatile memory as cache, and for reducing power by allowing advanced hibernation. We postulate that fast NVRAM may be used as a repository for security metadata and can form the basis for a comprehensive security policy, which may be enforced by the disk on a per-request basis. In effect, we can build a security perimeter around the disk where it enforces security in an autonomous manner.

In this chapter, we present a threat model for an autonomous disk architecture and show how services may be added to allow the disk to protect itself against adversarial operating systems. The disk, for example, may support primitives to allow for information flow control, while requiring the operating system to attest its hardware and software. We present three demonstrative applications of how an autonomously secure disk may be potentially used: to preserve integrity, to enforce access rights through capabilities, and to implement a lattice-based information flow policy. In addition, we demonstrate an initial modeling technique for the autonomously secure disk (ASD) using the DiskSim disk simulator. We have augmented DiskSim with a flash simulator that implements a detailed
model of non-volatile memory. Our extensive evaluations show that security guarantees such as confidentiality and integrity may be implemented with an ASD, exacting only a small performance penalty; our measurements indicated that input/output operations per second (IOPS) are reduced by as little as 2% in a mail server workload.

We begin by illustrating the use of ASDs, then describe their architecture and motivating applications. In Section 3.3 we describe how to emulate ASDs with DiskSim and show how we simulated operation of security features. We then describe how we evaluated performance under a number of differing workloads, then describe issues relating to the integration of ASDs into existing systems.

3.1 ASD Design

3.1.1 Requirements

To date, storage security proposals have often focused on the possibility that the storage is untrusted. In virtually all of these cases, there is an assumption that clients, and often trusted third servers that frequently called upon to supply capabilities and other storage metadata, are completely trustworthy. For example, consider Snapdragon [115], a solution that draws on receiving block-based capabilities from a metadata server. The user trusts that the metadata server has generated an appropriate capability based on its access control model. However, if the metadata server is exploited, the server can generate capabilities allowing an adversary access to any portion of the disk. This model is emblematic of that used by NASD and similar storage solutions. By contrast, in a system such as Plutus [116], storage is cryptographically protected by the client machine, and all cryptographic and key management operations are performed by the client. The user is hence responsible for ensuring that their machine is secure. An exploited client machine would gain the ability to generate arbitrary read and write requests to the storage associated with the key on that machine. More worryingly, any keys that the client was able to collect (e.g., through exploiting other machines) can be used to read and write any portions of the disk where those keys are used.

These scenarios motivate an adversarial model that differs in important ways. We seek to minimize any trust placed into entities that are not associated with the disk itself, and to narrow the attack surface to the disk. Because of the realities of operating system compromise through sophisticated malware (e.g., rootkits), and the inability to state with full assurance that an operating system is in a safe state, it is realistic to assume that the operating system interacting with the disk is malicious. Given these realities, we seek to
Figure 3.1. Design of an ASD.

protect on-disk data to the maximum extent. Correspondingly, we informally define the requirements for ASD design to be the following:

- The amount of data exposure in cases of OS exploitation should be limited to the user alone. Specifically, even if the OS is exploited, it should not be able to modify applications, configuration files, or the data that belongs to any other user on the system.

- Another requirement is thus the need for all policy to be encapsulated within the disk, such that it is the arbiter of access to block data. For this to happen, permissions, or capabilities, must be generated, and the disk must correspondingly be the entity that grants or authorizes these capabilities.

- Finally, a third requirement is for transparency, such that the disk performs all of its operations in a manner independent of the operating system and largely without requiring user interaction. This aids both in the usability of the system, as in regular operation the user should be able to operate the disk as if it is a regular drive, and in security, as decoupling policy from the operating system mitigates exposure and potential surfaces for attack.

3.1.2 ASD Design

Based on the requirements we have outlined, we provide an initial design for an ASD. Figure 3.1 shows an example ASD design. Note that it is similar in many ways to currently existing disks, but the design leverages the fact that there is movement towards placing more computational abilities within disks, as discussed in Chapter 1. Full disk encryption drives such as the Seagate Momentus 5400 FDE.2 [162], and the Hitachi Travelstar 7K200 [163] contain ASICs that perform cryptographic operations, while primitives such
as pseudorandom number generation, encryption algorithms and hash generators are included in firmware. In addition, hybrid hard disks such as the Seagate Momentus 5400 PSD [21] and the Samsung HM16HJI [164], containing both magnetic hard disk components and non-volatile flash memory, are commercially available. The primary additional components to be found in an ASD beyond what exists in currently-available disks are a policy engine to arbitrate requests from the file system, and an interface on disk to accept tokens from users. We briefly describe the role of the non-standard components found on an ASD:

- **Token interface:** We allow users to interact with the ASD by passing credentials to it. These credentials may be physical entities such as a smart card or security token. Credentials carry read and write capabilities that determine the access to be granted to a user to reading and writing blocks on disk. This also acts as a method of limiting the exposure of data on disk, since without a token with the appropriate credentials, it will not be possible to read or write disk blocks (e.g., those that belong to another user).

- **Crypto processor:** Based on the disk setup, blocks may be encrypted and decrypted by the disk on a per-user basis. In addition, the disk may generate keys to be used for encrypting and setting HMACs on tokens, to ensure the confidentiality and integrity of their contents. This has been implemented as an ASIC by disks offering full-disk encryption. Disks have become increasingly able to perform processing within the drive unit; many drives are SMART-enabled [165], allowing them to monitor a variety of disk attributes; these require some degree of processing power from the disk, as do services such as automatic sector remapping. Additionally, storage hardware employing object-based storage (OSD) [166] paradigms, including Panasas Storage Blades [167] and forthcoming OSD drives from Seagate [122], contain significant processing ability to collate blocks into objects and manage attributes. Potential implementations could include burned-in policy through an ASIC or the ability to provide flexible services through EEPROM firmware or FPGAs that may be reconfigurable for different operations. Sufficiently powerful processors could allow the disk to act as a fully autonomous agent capable of acting as its own iSCSI server, for example, if there is an on-board network interface card.

\[1\] We consider a token to be a physical device that provides a capability to the system, such as a pluggable token or a key fob.
• **Policy engine:** The policy of the disk and granting or denial of block requests is handled by a policy engine. Depending on the policies to be implemented, this can be an ASIC as is commonly used for cryptographic operations, or may potentially be a general-purpose processor for more advanced policy architectures. An embedded processor such as the XScale PXA270 [168], along with RAM for caching policy decisions, would be a potential option for more robust policy implementation. Policy evaluation at high speeds is already possible in network processors, which quickly evaluate access control lists for incoming packets at speeds of gigaabits per second, and general processors are more than capable of servicing requests at the lookup speeds necessary to prevent a bottleneck for the disk. Borders et al. show that with their CPOL policy enforcement architecture, given an AMD Athlon XP2200 processor and 512 MB, request processing could be performed in under 6 microseconds, and often in less than half a microsecond if the policy rule was cached [169]. This amount of overhead is sufficiently small that it essentially disappears given the costs of accessing magnetic, or even flash storage. Policy frameworks such as security lattices could be burned into ROM at the time of drive manufacture, allowed to be updated through firmware upgrades, or placed in non-volatile memory accessible only to the policy engine, depending on the degree of flexibility required by the drive customer. Even with the reduced processing capability of an embedded processor, the performance overheads of policy-related operations discussed in Section 3.2 will be minimal relative to disk service times. In addition, many operations can occur in parallel with data retrieval from the drive, thus masking any overheads. With these pieces in place, an ASD could support full-scale policy architectures such as KeyNote [170], STRONGMAN [171], Antigone [172], or OASIS [173].

• **Non-volatile memory:** NVRAM may be used to store additional metadata for the disk, such as a table correlating blocks with their appropriate capability. This would be accessed by the disk prior to making a policy decision. In our proposal, metadata is encapsulated within the drive enclosure, providing a sealed solution. If the drive manufacturer is concerned with physical security issues, either the non-volatile flash memory or the entire drive can be designed in a tamper-proof manner with mechanisms similar to those employed by the IBM 4758 secure co-processor [102]. Additionally, in contrast to using the non-volatile memory in the drive as a disk cache accessible by the operating system, we are able to create scenarios where the operating system is untrusted that will still enforce data security. By leveraging non-volatile memory for security metadata, the cost of accessing this
Figure 3.2. Performing authenticated encryption using security metadata. When a client makes a read request for blocks 10-20 (step 1), the policy enforcement point, or PEP (usually the processor on the disk) forwards a request to the disk platters, which read the blocks (steps 2,3). The non-volatile memory is consulted to determine the MAC for the integrity set (step 4); in this case, the set covers blocks 10-20. The results are returned to the PEP (step 5), and if the MAC is valid, the data is returned to the OS (step 6).

Data will be minimal compared to that of accessing information from the disk itself. For example, the seek time of a high speed server drive such as the Hitachi Ultrastar HUS15K300 73GB hard drive is 3.4 ms with an average latency of 2.0 ms [163]. By contrast, flash memory is very fast to access; Park et al. [174] show that the access time for a 4 KB page of NAND memory is 156 $\mu$s for reads, 652 $\mu$s for writes, and 2 ms to erase the page. By pipelining the operations of the disk such that it seeks and retrieves or writes data while authenticated encryption computations and accesses to nonvolatile memory are occurring, the costs of these operations may be completely masked.

### 3.2 Applications

We now consider methods of using the flash-assisted disk architecture for secure metadata. Our proposed applications are shown in increasing complexity from integrity protection, using flash memory to store message authentication codes for disk blocks, to capability-based access control, which requires more involved management and storage layout considerations. We also consider how to implement information flow using labels, and consider requirements for policy implementation and enforcement.

#### 3.2.1 Support for Authenticated Encryption

Storage is a central conduit for data loss and compromise, and reports of data exposed through lost or stolen disks and laptops occur on an almost daily basis. As just one example, laptops containing the mental health histories of over 300,000 people, and the
full names and social security numbers for almost 2,000 people, were stolen from the Pennsylvania Public Welfare Department [175]. While full-disk encryption is a means of handling the problem of data exposed from stolen disks, it does not facilitate the protection of data integrity. The attacker may modify contents on the disk in arbitrary ways, e.g., by overwriting the encrypted blocks. Depending on where the attacker writes to on the disk, it may be possible to replace certain data with garbled information in such as way that tampering is non-evident. Authenticated encryption deals with this issue.

A promising means of providing data integrity in addition to confidentiality is authenticated encryption (AE). AE schemes, however, are not length-preserving: validating the data integrity requires an authentication tag (i.e., HMAC) to be appended to the calculated value. The question of where disks should store this information has been, to this point, unanswered. The IEEE P1619.1 standard [176] suggests that authentication encryption is best used with devices that support length expansion, such as tape drives. Such length expansion is problematic for disk drives. Storing the information in a set location on the disk could yield potentially expensive seek times, while storing extra data in a sector hidden to the user could require large amounts of additional on-disk metadata and more importantly, potentially expensive changes to platter manufacturing and servowriters to ensure that tracks are not misaligned. In addition, because certain AE schemes are parallelizable, there may be performance benefits to accessing tags in parallel with data. In order to support authenticated encryption in devices which do not have a variable length base unit of storage, such as disk drives, we propose using the NVRAM in the aforementioned architecture as a store for the additional required metadata.

The IEEE P1619.1 standard defines two modes of operation for authenticated encryption. For a given plaintext, these operating modes produce a ciphertext for confidentiality (which is length-preserving due to the use of tweakable ciphers) key pair produce a ciphertext for confidentiality, and a Message Authentication Code (MAC), a keyed hash, for integrity. Each mode also requires an Initialization Vector (IV), which must be unique for every plaintext/ciphertext pair. MACs and IVs that must accompany each ciphertext. We now examine the role of the NVRAM architecture in the process of authenticated encryption for two modes, Counter mode with Cipher Block Chaining (CCM) [177] and Galois Counter Mode (GCM) [178].

CCM mode uses Counter (CTR) mode for confidentiality (the enciphering of plaintext records) and Cipher Block Chaining (CBC) mode for integrity (the calculation of MACs). When a subject with a symmetric key writes a plaintext record to the disk, an IV is generated and used by CBC mode to generate a MAC (up to size $2^{24} - 1$). The IV and
MAC are then stored in NVRAM. CTR mode is used to calculate the ciphertext, which is stored on the disk. Note that the ciphertext itself is the same length as the plaintext. When a subject with the same symmetric key reads the same record, the ciphertext is deciphered, and the resulting plaintext is used along with the IV from NVRAM to calculate a MAC. This MAC is compared against the one stored in NVRAM, and if they are equal, the integrity of the record has been verified.

GCM mode works in a similar manner. It uses the function GCTR for confidentiality and the function GHASH as well as GCTR for integrity. When a subject with a symmetric key writes a plaintext record (which can be any length up to size $2^{39} - 256$) to the disk, an IV is generated and used by GCTR to produce the ciphertext. The ciphertext is then used by GHASH, which performs multiplications over a Galois field, and GCTR to generate a MAC. As in the previous example, both the MAC and IV are stored in NVRAM. When a user with the same key reads the same record, the IV is retrieved from NVRAM and used to produce the plaintext version of the record. The ciphertext is then used to calculate a MAC. If this MAC is equal to the one stored in NVRAM, the integrity of the record has been verified.

The length of the MAC according to IEEE P1619.1 is 128 bits and that of the IV is 96 bits. For this reason, the length of an associated unit of plaintext and therefore ciphertext is an important detail of any implementation of authenticated encryption. If a single disk sector is used as a unit of ciphertext, the space required for storing MACs and IVs is approximately 5.4% of the size of the entire disk assuming 512 byte sectors. For a 1TB system, 54GB of NVRAM would be required. To mitigate this large spatial cost, we consider using integrity sets, fixed size groups of adjacent sectors, for which a single MAC is calculated and stored. They are used as follows. When a subject writes one or more blocks in an integrity set, authenticated encryption is performed on the entire set, and a single MAC and IV are stored for the set. When a subject reads one or more blocks in a set, authenticated decryption is performed on the whole set. The necessary blocks are extracted from the ciphertext, and a MAC is calculated and compared against the one stored in NVRAM. An example of this implementation in our proposed architecture is shown in Figure 3.2.

By controlling the size of integrity sets, one can control the amount of space needed in NVRAM for storing MACs and IVs. Of course, this savings in space is not without a cost in time. The costs of processing GCM and CCM increases linearly in the number of sectors per integrity set; notably, recent work has shown that GCM with underlying AES as the encryption cipher can be computed 3.54 cycles/byte [179]. This implies that
Table 3.1. Comparison of capability granularities.

<table>
<thead>
<tr>
<th>Parameter / Granularity</th>
<th>File</th>
<th>Page</th>
<th>Sector</th>
</tr>
</thead>
<tbody>
<tr>
<td>required NVRAM space</td>
<td>low</td>
<td>medium/high</td>
<td>medium/high</td>
</tr>
<tr>
<td>required processing power</td>
<td>high</td>
<td>low (constant time)</td>
<td>low (constant time)</td>
</tr>
<tr>
<td>coupling with FS</td>
<td>very high</td>
<td>simple with VM</td>
<td>none</td>
</tr>
<tr>
<td>enforcement mechanism</td>
<td>high</td>
<td>simple</td>
<td>simple</td>
</tr>
<tr>
<td>complexity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>representation of capability token</td>
<td>per file/descriptor</td>
<td>per page stored in page table</td>
<td>single, per-subject possibly a crypto key</td>
</tr>
</tbody>
</table>

the cost of computing a MAC for a set of $n$ sectors is equal to the cost of computing $n$ MACs, one for each sector in the set, plus a constant setup time.

Another advantage of using integrity sets arises from how modern operating systems handle block-level requests. When a block-level request arrives at the I/O scheduling layer, requests for adjacent disk blocks are merged together to reduce the number of requests sent to the disk and therefore disk seeks. This in turn also minimizes the number of MAC calculations as the size of an integrity set in sectors approaches that of the mean number of sectors per request.

3.2.2 Capability-Based Access Control

We now examine how to mitigate the loss of confidentiality of an entire hard disk, such as when a disk is stolen, by implementing capability-based mandatory access control within the disk itself. This implies that the disk must maintain a protection state consisting of per-object capabilities for each subject and an enforcement mechanism to mediate all access according to them. In the event that a disk is stolen, an attacker should not be able to acquire capabilities or successfully perform any block requests to the disk. Such a capability based protection state will require two data structures:

1. An unforgeable capability token that can be assigned to a subject based on credentials.
2. An associated descriptor containing per-object capabilities for a subject.

We propose that both of these data structures be maintained in NVRAM to ensure the secure storage and fast availability of the protection state to the enforcement mechanism, which we assume to be part of the disk’s firmware. We do not aim in this section to define semantics such as a subject’s initial capabilities with an object upon creation, as these are a matter of policy. We are presenting a plausible infrastructure for implementing such policies.
3.2.2.1 Capability Representation

Previous approaches to capability-based systems have represented capabilities at various levels of granularity within the OS, such as segments of memory [180] and nodes and pages [48]. The granularity of capabilities in the disk has an impact on the amount of space required in NVRAM, the amount of processing power required in the disk's controller or processor, the complexity of the enforcement mechanism, the degree of coupling with the filesystem, and the representation of the capability token. We examine these factors for three levels of granularity, files, pages, and sectors. The summary of these parameters for the three granularities can be seen in Table 3.1. It is important that regardless of granularity, no block request made without the appropriate capability will be fulfilled.

**Files.** In order for capabilities to be represented at the file level, the disk must receive enough information from the file system to be able to verify that a given capability token is sufficient to access all of the sectors in a file. This implies a tight coupling between the filesystem and the disk. One such architecture for which capability-based access control has been explored is that of type-safe Disks [127]. Maintaining capabilities at the file level requires a tight coupling between the disk and a filesystem that may not necessarily be trusted. It also typically requires more processor overhead than finer grained, more autonomous methods, due to the maintaining of data structures that represent file and directory relationships. It is however advantageous in that it requires less space in NVRAM for capabilities than page or sector level implementations that must maintain capabilities for the entire disk, as they lack knowledge of block allocation. File level capabilities also offer a familiar representation of capability tokens in the form of a file descriptor that can be returned by the `open()` system call.

**Pages.** Most modern microprocessor architectures supply page tables that map each process's virtual address space to pages of physical memory. Pages can be shifted, or...
paged, in and out of main memory into secondary storage, providing processes with address spaces larger than physical memory. Because of this mapping from physical memory pages to those stored on the disk, page table data structures may be used to maintain per-page capabilities. These could be used to control not only how processes access the pages in memory, but how the OS may read and write these pages to disk on behalf of processes. Per-page capabilities are convenient, as most OSes describe block level requests in terms of pages. For example, the Linux bio structure, the basic unit of I/O requests to block devices, is composed of structures that map to pages of physical memory [181]. All an OS need do is include the correct subject credentials in these requests to obtain per page capabilities. These capabilities could then be stored in the page table descriptors for each processes and retrieved when their corresponding pages must be paged in or out.

Page granularity capabilities will require more space on average than file level, as the capabilities for the whole disk are maintained for each subject regardless of the number of files. However, the amount of required space is static, and once allotted, will not change unless more subjects are added. The only coupling between the disk and the FS/VM subsystems is the requirement that VM pages always align with on-disk pages. Only a small processor overhead is needed to initially distribute the per page capabilities. Once the subject has obtained the capability, the OS need only forward it to the disk when reading or writing a page on its behalf.

Sectors. Representing per-sector read and write capabilities as a bitmap or capabilities list for each subject is not practical due to the limited capacity of NVRAM. Instead, we propose maintaining capabilities at the granularity of the integrity sets mentioned in section 3.2.1. This differs from the proposed page granularity in that it does not depend on the VM subsystem to align pages with the disk. Assuming a large enough set size, capabilities could be maintained in static, per subject bitmaps without exhausting the space in NVRAM. Such a bitmap would map two bits, enough to express read, write or read and write capabilities, to each integrity set as seen in Figure 3.3. A subject would obtain a single capability token from the disk which would be used for subsequent disk accesses to map to that subject’s capability bitmap stored in NVRAM.

There are several advantages to this method above the other two. First, because the access control is being done at the same granularity as the authenticated encryption, a subject’s symmetric key could double as their access token. This allows the same infrastructure used for key management to be implicitly used for capability management. This is advantageous as there has been much exploration in the area of key manage-
Second, provided a sophisticated enough on-disk processor, this could also allow access control decisions and authenticated encryption to be done concurrently. Such a pairing of authenticated encryption and access control provides the means to not only protect confidentiality and verify integrity, but to proactively protect integrity by denying unauthorized writes. Third, it requires little processing power. A constant time mapping from the requested sectors in a block request to the corresponding entries in a subject’s bitmap is all that is needed to decide whether the request will be granted.

This method will require more space than the file granularity capabilities, and either more or less than those at page granularity depending on the size of each integrity set compared to the system page size. Due to the static size of the per subject bitmaps, once the necessary amount of space is allotted in NVRAM, it will not increase until the addition of more subjects to the disk.

### 3.2.2.2 OS Constructions

To reduce trust in the OS, the capability token should originate from within the disk. All subjects must provide credentials to the disk to obtain a token, and should not be able to obtain a token by any other means. As subjects exist primarily at the OS level, we examine a typical usage scenario to show the necessary OS level additions to accommodate the distribution of capability tokens by the disk.

A user logs in to a system using a program such as `login`, which executes a shell and makes a call to the virtual filesystem (VFS) to send the user’s credentials and process ID (PID) of that shell to the disk containing the current working directory. Once the disk receives the credentials, it maps them to a capability token or creates a new token that it returns to the driver. The driver adds this token to the Process Control Block (PCB) of the shell who’s PID was specified in the VFS call. Any child processes of the shell will inherit the token into their PCBs as well. Whenever a subject makes a VFS system call such as `open()` or `write()`, the capability token is retrieved from the initiating subject’s PCB at the filesystem level and included in any block-level requests. The token is extracted from the request by the disk’s driver, and included in the request sent to the disk.

Note that in the above scenario, it is possible that the disk creates a new token for each login session. If a process continues execution after the user has logged out, due to the use of `nohup` or a similar method, it could be given its own token that maps to the user’s credentials, as part of the `nohup` command. Also, if no trusted path for the user’s credentials is assumed, they could be stored along with the user’s ID (UID) on an
Figure 3.4. An example of enforcing information flow with disk metadata. A request from a client labeled LOW to read blocks 10-20 is made (step 1). The policy enforcement point, or PEP (usually the processor), consults the non-volatile memory and finds that blocks 10-40 are labeled HIGH (step 2). When this information is returned to the PEP (step 3), it denies the request and no disk access is made.

For the purpose of capability creation, modification and revocation, a single meta-capability will be required per disk. This meta-capability could be used with a privileged utility to write special blocks to the disk containing user credentials and capabilities. The owner of the meta-capability is the disk’s administrator.

3.2.3 Preserving Information Flow

Providing mechanisms that preserve information flow is a complex task. While previous applications have dealt with tokens that may be modified but evaluated with relative ease, the challenges of fully implementing lattices for information flow are commensurately greater. While operating systems such as SELinux, and a small number of applications such as JPMail [53] support information flow measures, little has been done to protect information flows at the storage layer. We first consider policies and models that preserve information flow, then discuss how to represent policy, how to authenticate security levels with the environment outside the disk, and how metadata is handled.

A well-studied and popular means of controlling information flow is through multilevel security (MLS). In the canonical model for MLS that preserves data confidentiality, as proposed by Bell and La Padula [50], subjects and data observe the simple property, which states that no subject can read to a level higher than it is authorized for, and the ⋆-property, which states that no information may be written at a lower level than the subject is authorized for. There are numerous mechanisms that may be enforced for information flow preservation, such as the integrity-preserving model proposed by Biba [51], separation of duties as proposed by Clark and Wilson [185], and the Chinese
Wall model [186], among many others. Depending on what characteristics of the information is considered most important to protect, the policy will be chosen accordingly by system architects.

Policy is represented as a series of labels, which identify characteristics of both users and data. Figure 3.4 shows an example of a disk enforcing an MLS scheme. A user making a request to the disk has been labeled LOW by the operating system and is attempting to read a set of blocks that has previously been labeled HIGH. Because of the simple security property, a LOW user cannot read HIGH data, so the access is denied at the policy enforcement point – in this case, the processor in the disk mediating the operation. The type of policy enforced by the disk may be modified within the firmware. Alternately, in a high-assurance environment, the manufacturer may burn in custom firmware to the drive such that a specific model must be followed. This approach also has the advantage that labels may be defined \textit{a priori}; with knowledge of these and their semantics, reconciliation of semantics between user labels and those understood by the disk may be easier, as systems can ensure label consistency. However, predefining the label space may also limit the flexibility and granularity of expressible policy.

Regardless of whether labels are predefined or if they can be configurable by the firmware, the drive must be able to understand their semantics. In the previous example, policy enforcement mandates understanding a notion of ordering and comparison: semantics must be in place to determine that because HIGH is a more restrictive level than LOW, a user with level LOW may not be allowed to access data labeled HIGH. Existing research by Hicks et al. has considered policy compliance by examining the intersection of labels in security-typed languages and those derived from an operating system [187]. In particular, the SIESTA tool created by Hicks et al. handles the establishment of connections between the Jif security-typed language and the SELinux operating system. With some modification, a similar tool may be used to provide compliance between operating system labels and those deployed by the storage device. Alternatively, consider a high-assurance system where the entire system enforces information flow throughout and all of the hardware and software to be run is known. In this case, many if not all system parameters may be determined in advance, such that for a given system configuration and policy definition, a hash of the system state may be computed and burned into the drive’s firmware prior to its installation in the system. Then, a system possessing a trusted platform module (TPM) may provide a trusted attestation of the state to the hard disk, which will be able to ascertain the policy as being correct and the system as being trusted to preserve information flow.
3.3 Emulating an ASD

Having described the general usage of the ASD, we now describe some preliminary investigations we made into how the drive would work in a real environment, and how to evaluate the performance overheads created by the security operations. To fully understand these characteristics, we emulate an ASD by developing extensions to DiskSim [188], which account for the effects of the security operations on disk and flash memory access.

Because ASDs do not currently exist, we considered a number of avenues for how to model its functionality in a realistic manner. In our initial investigations, we decided on emulation, which was chosen because it offers several advantages over other techniques. Storage device emulation provides more flexibility for experimentation than full implementation, while offering comparably accurate results [189]. It also requires less effort to emulate a single component as part of a commodity system than to create a full system simulation. Because the emulated component functions as a part of the commodity system, it can be tested under workloads more accurately reflecting those created by current and widely used systems. Our reasoning was mirrored by Griffin et al. in their consideration of how to model storage devices [189].

We developed a generic block driver, genbd, to move block requests between the kernel and the user-space emulator, workload, via an asynchronous netlink socket as seen in figure 3.5. workload serves two main purposes: it simulates block requests with Disksim, and satisfies them from a user-space ramdisk. workload interfaces with Disksim using an event-based timing loop similar to that used in the Memulator by Griffin et al. [189]. Each simulated event is scheduled to occur at a certain time. Because Disksim processes requests more quickly than a real disk would, workload delays simulation of subsequent events until the service time has elapsed in real time. We thus provide consistency between simulated and wall clock time and hence, model I/O overhead more accurately.
The following algorithm shows how this operation is performed:

while(1) {
    req = receive_from_genbd();
    if(req == NULL) {
        if(disksim->event_queue != NULL)
            // Simulate all events up to the current real time.
            simulate_disksim_events(disksim->event_queue,
                                        get_real_time());
    } else
        add_request_to_event_queue(req);
}

3.3.1 Modeling ASDs

To simulate security operations and flash memory access, we placed a hook in the DiskSim disk controller code that is called once for each block request. This hook calculates the overhead created by these operations, and adds it to the total time for the disk access. We modeled these operations using the following parameters:

- \( T(f) \): The time to complete the function \( f \) in floating point milliseconds.
- \( ISet(o,n) \): Returns the integrity sets containing the request with offset \( o \) and number of sectors \( n \).
- \( SpannedSets(o,n) \): The number of integrity sets containing all or part of the block request with offset \( o \) and number of sectors \( n \).
- \( R_D(s) \); \( W_D(s) \): Reads or writes the contiguous set of sectors \( s \) to the disk. This operation is simulated by DiskSim.
- \( R_F(o,n) \); \( W_F(o,n) \): Reads or writes the block request with offset \( o \) and number of sectors \( n \). This operation is performed by the flash simulator, as described in section 3.3.2.
- \( AE(n) \): Perform authenticated encryption on \( n \) contiguous disk sectors as described in Section 3.2.1.
- \( R_{ASD}(o,n) \); \( W_{ASD}(o,n) \): Reads or writes the block request with offset \( o \) and number of sectors \( n \) to an ASD. This operation is a composition of the above functions.
We can thus model completion times with the following equations. The time for a read to the ASD is

\[ T(R_{ASD}(o,n)) = T(R_D(ISet(o,n))) + T(AE(n)) \]

\[ + T(R_F(o, SpannedSets(o,n))) \]  

The corresponding time for a write can be modeled as

\[ T(W_{ASD}(o,n)) = T(W_D(o,n)) + T(AE(n)) \]

\[ + T(W_F(o, SpannedSets(o,n))) \]

\[ + T(R_{ASD}(o,n)) \]  

Note that writes include the time for a read to the ASD over the set of specified blocks. When a write occurs, the HMAC of the integrity sets corresponding to the modified blocks must be recalculated and the new result stored to reflect the changes made.

### 3.3.2 Flash Memory Emulator

To support the experiments detailed in the following sections, we made use of the FlashSim simulator developed by Kim et al. [190]. Integrated into DiskSim [188], the resulting driver is comparable in behavior and operation to SanDisk’s SSD Solid-State Drive and BiTMICRO’s E-Disks [191,192]. A flash memory based solid-state disk operates substantially similarly to a conventional block device/hard drive, except that the storage media
is non-volatile memory. Kang et al.’s performance measurements [193] are used to model each NVRAM operation, i.e., 0.027320 $\mu$s per page read, 0.196370 $\mu$s per page write, and 1.5 ms per block erase.

Each page must be erased before it is reused. Thus, a garbage collector is needed to select an appropriate block for erasure when no “fresh” page is available. When the garbage collector is called, a candidate block is selected based on the ratio of the number of invalid pages to valid pages. As highlighted in the next section, the need for erasure on certain writes can induce significant variance in the write delay.

3.3.3 Experimental Setup

All tests described in the following section were performed on a 1.86 GHz Intel Core2 CPU with 1GB of RAM, running Ubuntu Linux with a 2.6.20-16-generic kernel. We used the ext2 file system with the anticipatory I/O scheduler. Note that no actual disk was
used, as all block requests were satisfied from the emulator’s user-space ramdisk (pinned to physical memory using \texttt{mlock()}). We provided a parameter of 512 MB of flash memory to the flash simulator.

The maximum size of each block request was set to 255 sectors, the Linux kernel default, and the maximum number of outstanding requests was set to 32, which is the maximum number of outstanding commands in both the ATA Tagged Command Queuing and Native Command Queueing\cite{194} standards, as well as the upper limit that could be efficiently queued in user-space. To simulate disk requests we used the DiskSim 3.0 simulation environment, and modeled the default Cheetah 4LP drive, a 4.5 GB, 10,000 RPM SCSI disk with a 512 MB cache.

We automated the benchmarking process using the Auto-pilot benchmarking suite\cite{195}, which we configured to run each test a minimum of 20 times, and to compute 95% confidence intervals for the mean elapsed, system and user times using Student’s t-distribution. In between each run the device created by \texttt{genbd} was unmounted to ensure a cold cache.

Two benchmarks were used, Postmark version 1.51\cite{196} and a simple in-house benchmark. Postmark creates workloads similar to those of email servers. It performs transactions on a set of many small files, which are performed in a random fashion representative of how email is accessed. A Postmark transaction consists of either a read or a write and either a creation or a deletion. We configured Postmark to perform 50,000 transactions on 20,000 files ranging in size from 500B to 20KB, using buffered I/O. Our in-house benchmark performs repeated sequential writes and reads of the entire disk, in order to test the effects of large sequential accesses on flash memory. We configured it to perform two writes and reads across the entire disk.

As discussed in Section 3.1, we assume an on-disk ASIC for performing authenticated encryption operations. Both CCM and GCM modes require 128-bit block ciphers. We assume an AES-128 block cipher, which we simulated based on measurements from AES implementations on ASICs\cite{197}. To simulate authenticated encryption on \( n \) 512 byte sectors of data, we calculate the time needed to perform \( n \times 32 \) encryptions using AES-128, 32 being the number of 128-bit blocks in each 512-byte sector.

### 3.4 Evaluation

ASDs must manage more meta-information than traditional disks. Hence, an essential issue is cost; how much does the added function impede the normal functioning of storage. Whether it be due to managing HMACS, capabilities, labels, or any other security
context, there will be trade-offs between the security provided, resource usage, and runtime performance. A prerequisite of an understanding of the feasibility and utility of ASDs is an understanding of this cost. The following study begins to map this space by evaluating the canonical security policy, confidentiality and integrity-guaranteed storage, in our emulated disk system.

The key trade-off parameter of the secure disk is integrity disk set size. More specifically, the set size determines the trade-off between performance and flash memory requirements. In the following experiments, we observe the overheads caused by security operations and flash memory access, as well as disk level statistics which hint at possible methods for optimization. An integrity set size of zero is the baseline case in which no security functions are executed (and no flash memory is accessed).
3.4.1 Small Random Access Workload

The completion times for the Postmark tests are shown in figure 3.6. They increase with an average slope of 0.032 seconds overhead per additional sector in the integrity set size. Similarly, the IOPS for these tests are seen in figure 3.7. They decrease with integrity set size at an average rate of 0.4 IOPS per sector. In both graphs, there is a slope much steeper than the average between the points with set size zero and eight, due to the introduction of flash access to the baseline.

This decrease in performance is due to the steady increase in modified request sizes, seen in Figure 3.8. The rate of decrease in performance is considerably smaller than the rate of increase in read request size for several reasons. First, as integrity sets become larger, fewer reads and writes are done to flash memory. Writes to flash can be especially costly when an erase operation must be performed as a part of garbage collection. Second, integrity set size mainly affects transfer time, which is closely related to request size, while seek time, a more significant contributor to access latency, is more closely related to request contiguity. We also saw some higher than normal disk cache hits rates due to the fact that we could only use 512 MB of the disk because of size limitations of the ramdisk.

In general, for workloads that perform small random transactions, disk performance is inversely proportional to integrity set size by a small constant. This is due to increased read request sizes, which increase transfer time. Some of the effects of the increased sizes are absorbed by a reduction in the number of flash memory accesses.
3.4.2 Large Contiguous Access Workload

The completion times of the in house benchmarks are shown in figure 3.9. Unlike the random workload, in which performance degraded predictably with integrity set size, completion times oscillate with respect to set size. Note that the shortest completion time in this case is not at set size zero, but instead at set size 128. Similar but smaller oscillations are seen in the read and write throughputs as shown in figure 3.10. Note that the set size with the highest read throughput is also 128. This is a result of the layout of tracks on the disk, explored in detail by Schindler et al. in their investigation of track-aligned extents [198]. Demonstrated in Figure 3.11, the same set sizes that have the shorter completion times and higher throughputs also have smaller average request sizes after modification. The reason for the smaller modified requests is better alignment of particular set sizes with request size before request modification. If a block request is slightly larger than an integrity set, the entire neighboring set will be read and its metadata will be updated in flash memory. This is why set sizes that perform well are next to sizes that perform poorly. The good performers have set size slightly less than or equal to the average block request size before modification.

The above results show that unlike the random workload, there is no approximately linear relationship between integrity set size and performance for a contiguous workload with large reads and writes. An important ramification is that performance is not proportional to disk size. This is because disk combined with available flash memory size dictates integrity set size. This is advantageous as the optimal integrity set size may be chosen for a random workload, and I/O system parameters may be tuned to align requests with the chosen set size.

While contiguous workloads with large block request sizes performance is independent of disk size, it was shown above that for random workloads, disk performance is inversely proportional to integrity set size. This raises the question of the feasibility of this scheme for disks much larger than the one used in these simulations. As commodity 1TB disks are now available, we wish to use the results shown here to make predictions about how the trade off between NVRAM size and disk performance.

3.4.3 Large Disk Analysis

We now use the simulation results presented in the preceding section to model the time/space trade offs in large disks. Our model is developed by extracting the salient effect of disk parameters from the emulated environment using linear regression, and projecting
that model on more modern disks. Specifically, this model allows us to understand the performance impact of pairs of Disk size and flash memory size. These two parameters are mapped onto IOPS and completion time by the function

\[\text{Performance}(D, F) = \text{MinSetSize}(D, F) \times m + b\]  

(3.3)

where \(D\) is the size of the disk in MB and \(F\) is the size of the flash memory in megabytes. The function \(\text{MinSetSize}(D, F)\) finds the minimum integrity set size needed to hold all metadata for a disk of size \(D\) MB in \(F\) MB of flash memory. \(m\) is the slope obtained from linear regression, and \(b\) is the intercept. The function \(\text{Performance}\) can compute either IOPS or completion time. Because \(\text{Performance}\) is a function of two variables, we examine contour plot of the level with highest performance.

The contour lines for completion times are shown in figure 3.12(a), and IOPS in figure 3.12(b). The two plots are almost identical because for both the integrity set size of 16 was used, which is the smallest set size that could store all security metadata for the ranges of flash and disk sizes. In order to achieve optimal performance in a 1TB disk, 4GB of flash memory are required. To achieve the same performance on a 4.5GB SCSI drive requires only 156 MB of flash memory. Comparable performance with a 2% decrease in IOPS can be obtained by using the next largest integrity set size of 32.
Chapter 4

Autonomous Disk Enforcement for Rootkit Protection

In the previous chapter, we outlined the general design for a disk that independently enforces security functionality, and described how such a disk could be used to provide a variety of potential applications to the system as a whole. In this chapter, we will show how the design for autonomous security enforcement through the disk provides more than data security. While disks are notably reduced in terms of processing functionality and their ability to specify complex policies compared to what is possible with the main computing system, we show even a limited interface offered to us is sufficient to provide system administrators and users with a solution to a particularly pernicious problem that the operating system cannot successfully defend against—rootkits that aim to make their presence persistent. We begin by discussing why persistent rootkits are as problematic as they are, and how our secure disk design can aid in their eradication from the system.

4.1 The Problem of Persistent Rootkits

Rootkits exploit operating system vulnerabilities to gain control of a victim host. For example, some rootkits replace the system call table with pointers to malicious code. The damage is compounded when such measures are made persistent by modifying the on-disk system image, e.g., system binaries and configuration. Thus, the only feasible way of recovering from a rootkit is to wipe the disk contents and reinstall the operating system [199–202]. Worse still, once installed, it is in almost all cases impossible to securely remove them. The availability of malware and the economic incentives for controlling
hosts has made the generation and distribution of rootkits a widespread and profitable activity [203].

Rootkit-resistant operating systems do not exist today, nor are they likely to be available any time soon; to address rootkits is to largely solve the general problem of malicious software. Current operating system technologies provide better tools than previously available at measuring and governing software [10], but none can make the system impervious to rootkits without placing unreasonable restrictions on their operation. However, while it is currently infeasible to prevent an arbitrary rootkit from exploiting a given system, we observe that preventing them from being becoming persistent is a significant step in limiting both their spread and damage.

We introduce a rootkit-resistant disk (RRD) that prevents rootkit persistence. We build on increasingly available intelligent disk capabilities to tightly govern write access to the system image within the embedded disk processor. Because the security policy is enforced at the disk processor (rather than in the host OS), a successful penetration of the operating system provides no access to modify the system image. The RRD works as follows:

1. An administrative token containing a system write capability in placed in the USB port of the external hard drive enclosure during the installation of the operating system. This ensures that the disk processor has access to the capability, but the host CPU does not.

2. Associated with every block is a label indicating whether it is immutable. Disk blocks associated with immutable system binaries and data are marked during system installation. The token is removed at the completion of the installation.

3. Any modification of an immutable system block during normal operation of the host OS is blocked by the disk processor.

4. System upgrades are performed by safely booting the system with the token placed in the device (and the system write capability read), and the appropriate blocks marked. The token is removed at the completion of the upgrade.

An RRD superficially provides a service similar to that of “live-OS” distributions, i.e., images that boot off read-only devices such as a CD. However, an RRD is a significant improvement over such approaches in that (a) it can intermix and mutable data with immutable data, (b) it avoids the often high overheads of many read-only devices, and (c) it permits (essential) upgrading and patching. In short, it allows the host to gain
the advantages of a tamper-resistant system image without incurring the overheads or constraints of read-only boot media.

In this chapter, we present the design and analysis of the RRD. The system architecture, implementation, and evaluation are detailed and design alternatives that enable performance and security optimizations discussed. We implement the RRD on a Linksys NSLU2 network storage device [204] by extending the I/O processing on the embedded disk controller, and use USB flash memory devices for security tokens. Our implementation integrates label and capability management within the embedded software stack (SlugOS Linux distribution [205]). We further extend the host operating system kernel and installation programs to enable the use of the non-standard RRD interfaces and security tokens: however, in practice, modifications to host operating systems will not be needed.

Our performance evaluation shows that the RRD exhibits small performance and resource overheads. The experiments show an overhead of less than 1% for file system creation and less than 1.5% during I/O intensive Postmark benchmarking. Further investigation shows that the approach imposes a storage overhead of less than 1% of the disk in a worst-case experiment. We experimentally demonstrate the viability of the RRD as a rootkit countermeasure by infecting and recovering from a rootkit collected from the wild. Furthermore, we show through examination of the chkrootkit utility that a large number of rootkits would be rendered non-persistent through use of the RRD.

Mutable configuration and binaries that can compromise the system (such as user cron [206] jobs), can reinfect the system after reboot. However, once patched, the system will be no longer be subject to the whims of that malware. This represents a large step forward in that it introduces a previously unavailable feasible path toward recovery. Note that the RRD does not protect the system’s BIOS (which is burned into system PROM/EPROM/flash). The RRD does, however, protect all portions of the boot process that use immutable code or data, including the master boot record (MBR).

4.2 Rootkit-Resistant Disks

To be successfully persistent, a rootkit must be first able to write itself to a persistent data store. Next, the rootkit must be able to insert itself into the boot process such that it is always executed when the system starts and hence persists across reboots. This latter step can be performed by overwriting binaries or modifying boot configurations. To defend against these attacks, we must consider data protection in a different light
from previous proposals.

To date, storage security has often focused on the possibility that the storage is untrusted. In virtually all previous proposals, there is an assumption that clients, and often trusted third-party servers frequently called upon to supply capabilities and other storage metadata, are trustworthy. Designing a storage system that enforces a security policy independent of the operating system presents several challenges, as the adversarial model differs in important ways. Namely, we must minimize any trust placed into entities that are not associated with the disk itself, and to narrow the attack surface to the disk. Thus, we assume that any operating system interaction with the disk is potentially malicious.

When a rootkit successfully compromises a system, the result is that data is susceptible to exposure. By using RRDs, however, the user can effectively reside at a lower level than the OS by directly interfacing the disk with a physical token to arbitrate access to data. The rootkit will thus be unable to gain access to read and write data on portions of the drive that the user does not have access to, regardless of OS compromise. This provides a level of on-disk protection that has not previously been feasible.

4.2.1 Goals for an RRD

To provide a practical solution for an RRD, we need to ensure that the following four goals are satisfied:

1. **It must protect against real rootkits.** The RRD must demonstrably protect against currently deployed persistent kernel-level rootkits.

2. **It must be usable without user interaction and with minimal administration.** The operation of the RRD should be *transparent* during normal operation. This aids both the usability of the system, as during regular operation the user should be able to operate the disk as if it is a regular drive, and security of the system, as decoupling policy from the operating system mitigates exposure and potential surfaces for attack.

3. **It must be highly performant.** Accessing storage must be feasible with as little performance overhead as possible, given the rigorous demands for I/O throughput.

4. **It must have low storage overhead.** The RRD should consume as little ancillary storage for metadata and use as little additional space on the disk as possible.
4.2.2 RRD Design

Designing a suitable solution that fulfills the above requirements presents the following two challenges:

1. As storage requests travel from a system call to the file system to the storage, context about what is being asked for is lost. For example, knowing whether requests for blocks are related to each other (e.g., are write requests associated with the same file or application) is not possible at the storage layer because this information has been removed. This results in a semantic gap between file and storage systems (as described by many, including Sivathanu et al. [207]). Data security policies are often defined at the file level, but the semantic gap makes the task of extending these policies to the disk interface difficult, if not impossible, to implement within conventional operating systems.

2. Enforcement of security in storage independently of the operating system depends on the availability of a trusted administrative interface. The disk interface has traditionally been limited to that of the system bus, as accessible by CPU and possibly DMA controller. This interface is fully accessible to the OS and thus is effectively compromised if the OS is compromised. As the administrative interface becomes increasingly complex, it becomes more susceptible to compromise. As a result, we seek to keep it as simple as possible.

We fundamentally address the semantic loss by not relying on the file layer to provide context to the disk. Instead, the administrator inserts a token into the disk when data is to be write-protected. The token acts to label the blocks written to disk, such that without the token present, they cannot be overwritten. By doing this, the administrator provides context to the disk: it can differentiate between labeled and unlabeled blocks, and between blocks labeled with different tokens. The token may be physically plugged into the drive (e.g., using a smart card or USB token). We say that any data blocks written under a specific token are bound to that token, such that they are rendered read-only whenever the token is not present. Such data will be immutable to alteration on disk by any processes within the operating system. Only a small subset of the data on a disk will be bound in this manner, notably the binaries and important sectors on a disk (e.g., the MBR) that would otherwise be susceptible to being overwritten by a rootkit.

\[^{1}\text{Note that proximity-based solutions such as ZIA [156] do not convey intention and hence would not be suitable for this design.}\]
The write-time semantics associated with tokens are a natural consequence, given that administrative operations requiring the presence of tokens are performed on system data at well-defined times (e.g., during file system creation, system installation, and package installation).

The physical action of inserting a physical token addresses our second challenge, as the user is a trusted interface to the disk that cannot be subverted by a compromised operating system. In essence, we have reduced the trust problem to that of physical security of the user and her associated tokens. This is an attractive model as well because the notion of performing a physical action to secure data is well-understood. One can even think of the token insertion to be equivalent to using a key in the drive, sliding a write-protect tab on a floppy disk, or adding a jumper to a hard disk. What the token insertion gives us, however, is a much more flexible means of providing a representation of authority than any of those approaches. Specifically, as we will see in the next section, tokens can carry different capabilities on them, and depending on how policy is specified, those capabilities can range in scope and authority. In addition, tokens are individual and unique to a user, and can be revoked by an administrator. The disk can be informed of these revocations in a manner similar to a certificate revocation list, thus ensuring that attempts by a token holder of exercising their authority will be for naught.

As previously noted, our model seeks to protect against persistent rootkits that have compromised the operating system; thus, we consider the user a trusted component in the system rather than an adversary. In addition, physical attacks such as forging of the tokens, or attacking the drive itself by opening it to scan its memory for metadata, are outside our protection model. Implementing tamper-proof interfaces into the drive appear contrary to the marketplace desires for inexpensive, high-performance storage. However, building additional security into the drive enclosure in a similar manner to the IBM 4758 secure co-processor [102] is a design point that is only feasible to achieve if the cost-benefit ratio for a specific application dictates it to be appropriate.

4.2.3 Tokens and Disk Policy

An RRD has two modes of operation. Under normal operation, the RRD is used like a regular disk, without any tokens present. This is the mode of operation that a regular user will always use the disk in, as will the administrator for the majority of the time. Only during an administrative event will the disk be used in administrator mode. We define an administrative event to be one that affects the system as a whole and that only an administrator can execute. Examples of these would be the initial installation
Figure 4.1. The use of tokens for labeling data in the RRD. Part (a): The host system writes a file to unused disk space using the gray token. The write is allowed and the file is labeled accordingly. In part (b), any blocks written while the black token is in the disk are labeled accordingly, and any attempts to write gray-labeled data are denied as long as the gray token is not present.

of an operating system onto the disk and subsequent upgrades to the operating system, e.g., software package upgrades or full distribution upgrades. Administrative mode is activated by inserting a token into the disk. As shown in Figure 4.1, data blocks written then become labeled with the inserted token and become immutable. Blocks labeled as immutable may only be rewritten when the token associated with the label is inserted into the disk. If the block has not been written under a token, or if it is written without the presence of a token, it is mutable and hence not write-protected. By differentiating between mutable and immutable blocks, we can allow certain files such as startup scripts to be only writable in the presence of a token, while not forcing such a stipulation on files that should be allowed to be written, such as log files.

There is a special token in our system that acts in a manner different than described above. Because of the need for processes to be able to write certain file system metadata, such as logs and journals, we introduce the concept of a permanently mutable data block. Blocks labeled permanently mutable (denoted $\ell_{pm}$) during file system creation are writable by all processes (subject to operating system protections, e.g., UNIX permissions), regardless of whether the token is installed or not. Data written under any token other than the permanently mutable token is immutable when that token is not in use. Blocks labeled as immutable may only be rewritten when the token associated with the
for all \(blk\) in Request do

\[\ell_b \leftarrow \text{LABELOF}(blk)\]

\[\ell_t \leftarrow \text{LABELFROMTOKEN()}\]

if \(\ell_b \neq \text{nil}\) and \(\ell_b \neq \ell_{pm}\) and \(\ell_b \neq \ell_t\) then

return ‘Write denied’

end if

if \(\ell_b = \text{nil}\) and \(\ell_t \neq \text{nil}\) then

\[\ell_b \leftarrow \ell_t\]

end if

end for

return ‘Write OK’

\textbf{Algorithm 1:} RRD-write. Only if a block is unlabeled or has the same label \(\ell_t\) as the inserted token will the write be successful; otherwise, if there is a label mismatch or if the block is labeled permanently mutable \(\ell_{pm}\), the write will fail.

In a scenario where the drive is used to separate binaries that may end up as vectors for rootkits and being loaded at boot time, only one token may be necessary. This token would be used only when system binaries are installed, as that would be the only time they would require being written to the disk. Greater isolation may be achieved by using separate tokens while performing different roles, e.g., a token for installing binaries and another for modifying configuration files. By differentiating between mutable and immutable blocks, we can allow certain files such as startup scripts to be only writable in the presence of a token, while not forcing such a stipulation on files that should be changing, such as document files within a user’s home directory.

When the RRD receives a write request for some contiguous region of blocks, \(R = \{b_i, b_{i+1}, \ldots b_j\}\), it obtains the label \(\ell_t\) from the current token. If no token is present in the disk then \(\ell_t = \text{nil}\), in which case the RRD verifies that no mutable blocks are included in \(R\). If \(\ell_t\) is the permanently mutable label \(\ell_{pm}\), any unlabeled blocks in \(R\) are labeled with \(\ell_{pm}\) and the write is allowed. If the token contains any other label, all blocks in the request are checked for equality with that label, and any \(\text{nil}\) blocks are labeled accordingly. The RRD’s write policy is specified in Algorithm 1.

Once a block has been labeled as immutable or permanently mutable, its label cannot be changed. Thus, in a system where immutable data is often written, the possibility of label creep \cite{208} arises. Because of the semantic gap between the file and storage layers, we are unable to perform free space reclamation in storage when files are deleted. This leads to labeled blocks becoming unusable, as while they have been freed by the file system, they remain write-protected by the disk. We find in our examination of label
creep that its effects on an RRD are limited; we investigate this facet of operation through experimentation in greater detail in Section 4.4.3. It is possible to mitigate the effects of label creep through unlabeling of tokens; we explore this in further detail in Section 4.5.

4.2.4 RRD Operation

We now outline the steps by which an RRD would be set up and operated. An RRD may be shipped pre-formatted with the appropriate labeled tokens, or the user may perform the initial set up of the device. It is important at this stage to boot the system where the disk is to be set up into a state that is free of rootkits that may infect the machine; the disk is in a vulnerable state at this point. While a trusted boot prior to the installation of an RRD (and with regards to BIOS integrity) is outside the our scope of feasibility, a method of ensuring a trusted system setup could involve the use of a root of trust installation [209] where the system is booted from a trusted CD and the subsequent boot and installation is measured for integrity at every stage.

Once the system is booted, the disk may be formatted. The simplest and safest manner for setting up the drive is to define a system partition that holds the OS binaries and configuration files, and a user partition for other data. During partitioning, the token must be inserted to protect the MBR. The system partition may then be formatted without an installed token such that the file system is mutable (i.e., data may be written to it). As mentioned above, certain special file system structures (e.g., a file system journal\(^2\)) may need to be always writable. The blocks allocated to these structures should be written with the permanently-mutable token. We describe this process in more detail in Section 4.5.

At this point, the operating system may be installed on the disk. A token should be inserted in the disk to move the RRD into administrative mode. This will render important binaries (e.g., programs installed in the /usr/bin hierarchy in a UNIX-like system) and configuration files immutable. Finer-grained access is possible at this point; for example, if there is a desire for access to writing binaries to be decoupled with access to configuration files, the installer can be partitioned to first write the binaries with an appropriate binaries token before writing configuration files (e.g., the /etc hierarchy) with a configuration token. Once the installation is complete, the user partition may be formatted and the disk is ready to be operated.

\(^2\)Note that we do not currently support full journal functionality in a file system like ext3, because journaled data may be written to labeled blocks at a time when the token is not in the disk. We discuss a small file system modification that supports full ext3 journaling in Section 4.5.2.
Figure 4.2. The prototype RRD. Block requests are intercepted at the client by the RRD driver and sent to the slug over TCP. netblockd receives the block request and consults BLOCK-LABELS and the optional physical token to determine if the request is allowed. If so, netblockd fulfills the request from the external USB hard drive. netblockd accesses BLOCK-LABELS, the physical token, and the external drive through mount points in the slug’s file system. An asterisk is placed next to each component that we created or configured.

Normal operation of the system should preclude the need for any involvement of physical tokens. Only when system updates or changes to the configuration are necessary will tokens be necessary. To prevent the token from affecting other files during an upgrade, it is optimal to perform these operations in their own runlevel, such that processes are stopped and labels are not inadvertently written to incorrect blocks. It is also important that the operating system synchronize its data with the disk by flushing buffers (e.g., through the sync command) when the upgrade is completed and before the token is removed. At this point, the system may return to its previous runlevel and operations may continue as usual. Note that we cannot protect against system upgrades that are already compromised as may be the case, and protections against upgrade tampering are insufficient in current software update mechanisms [210]. An RRD can protect against persistent rootkits that attack the system when the token is not installed, but if a program contains a trojan or other exploit and is installed when an administrative token is present, it will have the ability to attack the boot process and install itself to maintain persistence.

4.3 Prototype Implementation

We implemented an RRD that fulfills block requests over TCP. The choice to use a network interface was made as development of firmware for commodity disks is prohibitively difficult due to a lack of open firmware and development environments. Our prototype RRD provides the same functionality and security guarantees described above, and can be used as the root partition of a running system. Because the prototype exports a non-
standard interface, we developed a driver for disk-to-host communication. This driver would normally not be necessary given a disk with a standard interface, e.g. SCSI, and contributes nothing to the security of our scheme. Finally, we create an installer to simplify the processes of installing a base system on an RRD. We now describe the implementation details of our prototype, as well as the RRD driver and installer.

4.3.1 RRD

The prototype RRD has two components: a Linksys NSLU2 network attached storage link [204], commonly referred to as a “slug”, and an external USB hard disk. The setup of this prototype is shown in Figure 4.2. The external hard disk is a standard model and offers no additional functionality beyond the typical SCSI command interface. The slug is a network storage link, an embedded device that acts as a bridge between an external USB hard disk and a network attached storage (NAS) device. It has one RJ45 jack that provides the network interface, and two USB-2.0 ports used for the exported storage devices. In our case, we use one of the USB ports for the disk and the other for USB thumb drives which constitute the physical tokens. The role of the slug is threefold:

- Receive block requests from the network;
- Store and enforce the RRD’s policy;
- Act as an entry point for physical tokens.

In order to develop software for the slug, we replaced its default firmware with the SlugOS Linux distribution [205]. We then uploaded netblockd, a server program we developed to satisfy block I/O requests over TCP sockets. netblockd consists of 2,665 lines of code and was cross-compiled for the slug using the OpenEmbedded framework [211]. netblockd satisfies all requests from the USB hard disk connected to the slug, taking as a parameter the device name for the specific partition to export. Along with providing the basic block interface, netblockd also implements the security functionality of the RRD.

We detect physical tokens and notify netblockd of their insertion and removal using the udev framework [212]. When a USB thumb drive is inserted to the slug’s open USB port, a udev script mounts it to /mnt/token and signals netblockd. netblockd will then use the label from that token when making access control decisions until it is removed, at which point the token is unmounted and its label cleared from the slug’s memory.

The labels used to write protect disk blocks are stored in the BLOCK-LABELS data structure. This structure is kept in RAM while the slug is active, and saved to flash
memory when powered off. When netblockd receives a write request, it obtains the labels corresponding to the requested blocks from BLOCK-LABELS, and compares them to those on the current physical token, if any. Because it is accessed on every write request, the search and insert operations on BLOCK-LABELS must contribute only negligible overhead to the total disk latency. It also must maintain metadata for potentially all blocks on the disk, within the space constraints of the disk’s main memory, something that becomes more challenging as magnetic storage capacities increase. We explore scalability of the label data structure relative to disk capacity in section 4.4.3.2. We now examine the design of the BLOCK-LABELS data structure used in our prototype.

4.3.1.1 Label Management

File systems will attempt to store logically contiguous blocks, i.e. blocks in the same file, in a physically contiguous manner. We make use of this property by representing the disk as a set of contiguous ranges for which all blocks in a given range have the same label. If data is written to an unlabeled block, the write label from the current token, if there is one, must be added to BLOCK-LABELS. If this block is either adjacent to or between blocks of the same label, the ranges containing these blocks will be merged, reducing the size of the label structure. Searching BLOCK-LABELS requires logarithmic time in the number of ranges, and inserting a new range requires linear time. This is acceptable, as the number of ranges is kept as small as possible.

4.3.2 Host Machine

In a typical RRD scenario, a standard SCSI or ATA driver would suffice for communication between the host and disk. Because our prototype exports a non-standard interface, we needed to implement an RRD device driver for the host machine. Note that while this driver is necessary to communicate with the RRD, neither it, nor the RRD to host protocol, contain any security measures. The protocol contains only operations for reading, writing, obtaining the disk’s geometry and reporting errors. For ease of implementation, we constructed the communication protocol in user space, leaving the block I/O interface in kernel space. The RRD driver consists of 1,314 lines of kernel code and 307 lines of user space code.

In order to use our RRD as a root partition, we needed to mount it at boot time. This required the RRD driver to be run as part of the boot sequence. To achieve this, we created a Linux init ramdisk (initrd) image. This image contains a small ramdisk file system with utilities and drivers to be executed at boot time to prepare the device
containing the root partition. Because the initrd is required to mount the RRD, it cannot be located on the RRD itself. Neither can the kernel or bootloader. We can, however, achieve the same security guarantees with our experimental setup by keeping the bootloader, kernel and initrd on a read-only media such as a CD-R. Note that in case of an IDE / ATA RRD, the BIOS can load these components from the disk at boot time, eliminating the need for the special RRD driver and initrd.

4.3.3 Installer

Performing an installation with an RRD requires knowing when the token should be present in the disk and when it should be removed. This could also require using multiple tokens at different stages of the install, e.g. a binaries token and a configuration token. For this reason, it is desirable for the installer to cooperate with the administrator to simplify the installation process and make it less error-prone. To achieve this, the installer should require as little token changing as possible, while at the same time ensuring the mutual exclusivity of mutable and immutable data. We observe that the majority of data copied to the disk during installation is immutable. Most mutable data, that residing in logs and home directories, is created some time after the base installation.

We wrote a proof of concept installer script to install a base system onto an RRD. The installer’s main function is to differentiate between data that should be mutable and immutable, as well as format portions of the file system as permanently mutable if necessary. It is focused on ensuring the mutually exclusive installation of mutable and immutable data. This is accomplished by installing mutable files and directories, asking the user for a token, and installing any immutable files and directories. We also modified mke2fs to label the appropriate data as permanently mutable. In this case, all structures except inodes are made permanently mutable. Inodes may need to become mutable to attacks in which inodes are pointed at trojan files and directories.

The key design decision in creating the installer is what data should be mutable or immutable. Making this decision is mainly a matter of identifying and write-protecting possible vectors for rootkit persistence. The MBR, boot loader, kernel and any kernel modules must be immutable to prevent overwriting by kernel level rootkits. Similarly, all libraries and binaries should be immutable to prevent user level rootkits from installing trojan versions. Protecting against overwriting is insufficient, as a rootkit may be stored in some free space on the disk, and loaded into memory at boot time. For this reason, any system configurations and startup scripts should be made immutable, along with scripts defining repeatedly executed tasks, e.g. cron. It may also be necessary to make
root’s home directory immutable to prevent a rootkit from being restarted due to some habitual action by the administrator, such as logging in. Header files such as those in /usr/include may be immutable to prevent the insertion of malicious code that could be compiled into applications built on the target system. Finer granularities of labeling may be achieved on a case-to-case basis, at the cost of administrative overhead.

The sequence of operations taken by our installer is as follows. The user is prompted to enter the permanently immutable token, at which point all file system metadata except inode tables are initialized. At this point the user removes the permanently immutable token, and the inode tables are created. Any mutable data including home directories is also written at this time. Finally, the user is prompted to enter the immutable token and the base system is copied to the disk. We evaluate the RRDs effectiveness in protecting the integrity of the base system in section 4.4.4.

4.4 Evaluation

Because storage is a significant bottleneck in most computer systems, we investigate the performance impact of the RRD’s security measures. We accomplish this by running macro and micro benchmarks on our prototype RRD, and measuring the effects of block labeling on completion times and transactions per second. It is also possible for the layout of data on disk to have a dramatic effect on the size of BLOCK-LABELS, causing it to exceed the space constraints of the disk’s NVRAM. Another concern regarding the labeling of data is the amount of disk space lost due to label creep. To better understand
the effects of label creep, we measure the number of blocks in the disk that become unusable after relabeling. Finally, we investigate the ability of our prototype RRD to prevent rootkit persistence in a real system. In our evaluation of RRDs, we attempt to answer the following questions:

1. What are the performance overheads incurred by block labeling in the RRD?
2. How many disk blocks are lost due to label creep under a normal usage scenario?
3. How well does BLOCK-LABELS scale with the amount of labeled data written to disk?
4. Is the RRD effective in preventing rootkits from becoming persistent?

The answers to these questions will guide our analysis, and direct our future efforts at making RRDs more performance and resource efficient.

## 4.4.1 Experimental Setup

All experiments were performed using our prototype RRD consisting of a Linksks NSLU2 (slug) and a Seagate FreeAgent Pro external USB 2.0 hard drive, as shown in Figure 4.3. The slug has a 266MHz ARM IXP420 processor, and 32MB of RAM. The slug ran the SlugOS 4.8 Linux distribution with a 2.6.21 kernel. The base system was stored on a partition on the external disk, and no swap space was used. The host system ran Ubuntu
Linux with a 2.6.22 kernel on a 1.8 GHz Intel Core 2 Duo processor with 1 GB of RAM. In each experiment, the host was connected to the slug over a 100 MBps Ethernet link, or in the case of the scalability experiments, a switch to allow the host to download system upgrades.

4.4.2 Performance

In order to understand the performance implications of the RRD’s policy, we evaluate our experimental prototype under two workloads. The Postmark benchmark is a standard file and storage system benchmark that performs transactions on a large group of many small files. Postmark is intended to simulate a typical load on a mail server, and is a good approximation of a random workload. In order to test the RRD under its expected operating conditions, i.e., administrative operations, we perform a system install to show the affects of block labeling for common administrative tasks.

4.4.2.1 Postmark

We used Postmark version 1.51, configured to create 20,000 files of sizes ranging between 1KB and 20KB, and to perform 100,000 transactions. All other parameters were left to the default values. We ran the test 20 times, using a different random seed for each run and unmounting the RRD between runs to ensure a cold file system cache. The test was performed using two configurations: nosec in which the RRD was connected to the host via a direct Ethernet link and sec which was identical to nosec with the RRD’s security measures enabled.

The completion times for each configuration as reported by Postmark are shown in Table 4.1 and the transactions per second in Figure 4.2, along with the 95% confidence intervals as calculated using a T-distribution with 19 degrees of freedom. Being that Postmark is a random workload, the majority of the time used by security operations is spent searching and modifying the label data structure, BLOCK-LABELS. This task becomes more costly as many small, non-contiguous files are written to disk, increasing the size of the structure. As will be seen in the following experiment, more contiguous workloads can yield better performance.

To better understand the proportion of time spent on security as compared with the other components of I/O, we recorded microbenchmarks from within netblockd. These contain the time spent on disk access, network access and security. The disk access measurement is the total time spent by netblockd executing blocking read() and write() system calls to read and write blocks to the external hard drive. These do not
include the overhead due to synchronization of the file system cache with the disk. The network measurement is the time spent by `netblockd` on blocking `send()` and `recv()` system calls to move data to and from the host. The security measurement is the time spent labeling blocks and performing access checks for write requests.

The results of the microbenchmarks are shown for the `sec` configuration in Table 4.3. Note that these results do not account for the time writing back pages from the slug’s page cache, and thus do not sum to the total execution time for the benchmark. They do, however, confirm that bus and disk access dominate security operations in the RRD. Furthermore, even in an implementation of the RRD within an enclosure that eliminated network overheads, the disk latency dominates the security overhead, such that policy lookups would not be a bottleneck.

### 4.4.2.2 System Install

Because the majority of labeling in an RRD occurs during administrative operations, we perform a simple system install. To achieve a worst-case scenario, we label all data in the system. For each run of the installation, we formatted the RRD with the `ext2` file system and copied Ubuntu desktop Linux version 6.06 to the RRD from the host’s hard disk. While this does not account for all activities of a typical system install, such as extracting archives and creating configurations, it does capture the I/O intensive operation of writing the system to the disk. The base system, which consisted of 949,657 files, was installed on
a 100GB partition on the RRD. We used the same two configurations as in the previous experiment.

The completion times for file system creation on each configuration as recorded by the `time` command are shown in Table 4.4, and base system copy time in Table 4.5. In the case of contiguous raw I/O, as is seen in file system creation, block labeling and policy checking accounts for less than 1% of the completion time. This is due to the small size of `BLOCK-LABELS`, keeping search and insertion times short. The installation portion of the workload shows a larger overhead than file system creation due to the increasing size of `BLOCK-LABELS` as larger portions of the system are written to disk. We further investigate the growth of `BLOCK-LABELS` in section 4.4.3.

The results of the microbenchmarks for the System Install are shown in Table 4.6. Under the more contiguous workload of system installation, the percentage of overhead due to security operations is less than half that of the random workload. Note that in this case, disk I/O has also improved due to the large amount of contiguous writes.

### 4.4.3 Scalability

As explained above, some blocks in an RRD may become unusable due to label creep. We will show that the number of blocks lost in this way represents only a small fraction of the disk in the worst-case scenario. We do this by measuring the difference between the number of disk blocks used by the file system and the number of labeled blocks during common administrative maintenance on the RRD’s host system. Because the RRD maintains labels for potentially every block on the disk, we need to demonstrate that the amount of space overhead used for these labels does not become unreasonable. It is important that the space needed to store labels represent a small percentage of the disk.
4.4.3.1 Measuring Label Creep

In this test, we perform several common administrative tasks to simulate conditions under which labeling would occur on the RRD. We first install a file system and base OS as described in the previous experiment. We then reboot the host system, mounting the RRD as the root partition, and perform two full distribution upgrades: from 6.06 to 6.10 and from 6.10 to 7.04. The numbers of packages modified in each of these upgrades is shown in Table 4.7. At each of these four steps, we record the number of disk blocks used by the file system, and the number of blocks labeled by the RRD. We performed this test with two file systems, ext2 and ext3, which were chosen for their popularity as well as to determine the affects of journaling on label creep.

The results for both file systems are shown in Figure 4.4. ext3 behaves the same as ext2 with the exception of a constant increase of 32,768 blocks to both the used and labeled blocks. This constant increase is due to the journal, which was labeled as permanently mutable at file system creation time. While the overhead due to label creep in both cases is roughly 10% of labeled data, it represents less than 1% of the total space on the partition. Because we tested the worst-case scenario by labeling all data in the base system, we have shown that in the worst case, label creep does not waste significant disk.
4.4.3.2 Label Space Constraints

We now evaluate the space efficiency of the RRD’s label data structure as described in section 4.3.1.1. We are mainly concerned with the size of the BLOCK-LABELS structure relative the number of labeled blocks. We perform the same tests as in the previous experiment, recording both the size of the labeled data and the size of BLOCK-LABELS at each step of FS creation, base copy, upgrade 1 and upgrade 2.

The results are shown in Figure 4.5. From this figure, two things are evident. First, the label data structure is nearly three orders of magnitude smaller than the number of labels it represents. The label data structure also grows with a slower constant rate than the number of labeled blocks for the given workload. The second noteworthy characteristic of these results is that while the number of labeled blocks is larger in \texttt{ext3} than \texttt{ext2} by the constant size of the journal, BLOCK-LABELS remains completely unaffected. This is because the journal is represented by a single range, labeled as permanently immutable at file system creation time. In our implementation of BLOCK-LABELS, every range is 12 bytes in size, making its maximum size less than 40 KB after the second upgrade, while the size of the system on disk was nearly 4 GB.

4.4.4 Security

In order to test the ability of our prototype to correctly protect immutable data, we install a rootkit on a system booted from the prototype RRD, and verify that it fails to become persistent. We chose the Mood-NT rootkit [213], which is a persistent rootkit for Linux. Mood-NT works by acting as a trojan on the system call table. It can hide files, processes and network connections, as well as discover the location of the system call table for different kernel versions. Mood-NT gains persistence by replacing /sbin/init with its own initialization program. Upon reboot, this program installs the appropriate hooks in the system call table, and runs a backed up version of init to initialize the system as usual. This backup is hidden by trojan versions of the \texttt{stat} system call.

We created a base system using our installer script, which was configured to make all system binaries including init immutable, and rebooted the host machine from the RRD. Inspection of the system call table revealed that specific system calls had been replaced with malicious versions. It was apparent however, that the attempt to replace /sbin/init had failed due to a file system error. We rebooted the target machine and inspected the system call table for any signs that the rootkit had reinstalled itself. No
system calls had been replaced, and there was no backed up version of init. We verified that the backup was not in its intended location by rebooting from the host machines internal hard disk, and searching the suspect partition on the RRD. From these results we conclude that the prototype RRD successfully prevented the rootkit from becoming persistent.

Given that the prototype RRD has been shown to successfully protect immutable data from writing in the absence of the appropriate token, we can safely generalize the set of persistent rootkits protected against by the prototype to all those that attempt to overwrite immutable data. This includes all data labeled immutable at installation time by the installer script as described above. Rootkits that normally overwrite files protected by our prototype system include t0rn, which overwrites common utilities such as netstat and ps [214], Suckit, which also overwrites /sbin/init, and Adore, which attempts to create files in /usr/lib and /usr/bin.

To better understand the scope of rootkits that write data to files normally labeled as immutable on an RRD, we examined a popular rootkit-scanning program to see which files and directories it scans for evidence of rootkits. We chose chkrootkit [92], a collection of scripts that scan for rootkits on a variety of UNIX style systems include Linux, BSDs, Solaris and HP-UX. Our examination of chkrootkit version 0.47 revealed over 150 files and directories, labeled as immutable by the RRD, were scanned for modification by 44 known rootkits. chkrootkit performs two main types of checks. It inspects binary and configuration files for the presence of strings normally present in trojaned versions, and it checks for new files created by rootkits in system directories. The magnitude of files and directories examined by chkrootkit shows that RRDs can protect a large set of data normally tampered with by rootkits.

### 4.5 Discussion

#### 4.5.1 System Tokens and atime

It is advantageous for the system partition of the file system to have its files protected through an administrative token. Without the token in place, these files may not be overwritten. A challenge comes with the use of the atime attribute for UNIX-based file systems, however. Consider, for example, an extended Linux file system, e.g., ext2. When binaries are installed to the RRD with an installed token, both the file’s blocks and its associated metadata blocks will be labeled with the token. In a Linux system, whenever a file is accessed, regardless of whether it is modified or otherwise changed, the time it
was accessed, or atime, is affected. Because the administrative token is not meant to be used during regular system operation, metadata blocks associated with any programs written under the token will not be writable. For example, if Firefox is written under an administrative token and it is subsequently opened by a regular user, the inode’s atime attribute will not be refreshed. Generally, atime is seen as a major detriment to performance [215] and in the Linux 2.6 kernel it is possible, and common, to disable atime altogether by mounting the file system with the `noatime` attribute.³ Disabling atime does break POSIX compliance but the number of programs affected by the lack of atime is small; system administrators should verify that their programs run properly in the absence of atime before committing to this change.

### 4.5.2 File system Modification

While an RRD will function with a variety of current, unmodified file systems, there are some small file system modifications that could help to improve the interaction between the file and storage layers. We consider the inode descriptor table in a UNIX-like file system. There are many tens of thousands of descriptor blocks when a file system such as ext2 is installed on a modern hard drive, and subsequently, millions of inodes are capable of being accessed. If a previously mutable inode descriptor block is written while a token is present, the block will become immutable under that token. Subsequently, if there is a write request and free inode descriptors are available in the block, the file system may attempt to write data to the block. This will fail if the token is not present, and the file system will have no knowledge that the write failed because of a lack of access privileges, but would rather be a message such as "BAD BLOCK".

A small change to the file system could be made such that when the error message is received, the request is not retried but rather a different (potentially non-contiguous) block from the inode descriptor table is chosen. In addition, the file system often intersperses write requests across multiple inode descriptor blocks. A very small change that favors contiguous allocation of inodes will minimize the number of descriptor blocks that will be labeled in the event of a write request. A file system tool that instructs journaling file systems such as ext3 to write all changes in the journal to disk, would prevent write denied errors from the disk when attempting to sync journal blocks to labeled disk blocks after the token has been removed.

³The NTFS file system has a similar access time check, which may be stopped by modifying the `NtfsDisableLastAccessUpdate` registry key.
4.5.3 Maintenance Concerns

Performing administrative and maintenance tasks on RRDs is hampered by the necessity of not trusting the operating system. This is a model in stark contrast to what is currently accepted, where disk utilities that run through the operating system provide easy access to administrative functions. Consider, for example, the task of duplicating an RRD token for purposes of redundancy or backup. In a conventional system, this could occur through a program run by the user that presents a graphical user interface, guiding the user step by step through the required functionality. Unfortunately, any opportunity for the operating system to control functions on the disk is an opportunity to incorrectly label data and cause privilege escalation. As a result, maintenance operations must be performed in a manner that allows for direct interfacing with the RRD without the use of the OS as an intermediary. A non-exhaustive list of tasks that the RRD may be directly called upon to perform includes the following:

- Token cloning and disk backup
- Revocation of existing tokens and token escrow
- Large-scale token management and initial RRD configuration

Below, we present some potential solutions to address or mitigate some of these issues. These investigations are preliminary and understanding them in greater detail is an ongoing research initiative. We assume that the RRD has at least two available slots for the insertion of USB tokens, and that it is shipped with two additional tokens: a “backup” token and an “unlabel” token.

4.5.3.1 Token Duplication

To avoid reliance on the OS, one potential solution for token duplication is to ensure that the RRD has two available USB slots for tokens to be inserted. Then, the token to be duplicated is inserted in one slot, while a blank token is inserted in the other slot. Sensing the blank token, the RRD duplicates the contents of the first token onto the second.

4.5.3.2 Backup

With the availability of token duplication, backup without use of the OS is simplified. Backing up data on an RRD is now a matter of duplicating the backup token, retrieving another empty RRD of the same capacity, connecting the two devices together, and
inserting a backup token in each drive. A block copy will then be performed along with a transfer of metadata between drives. Because this is a block copy, the geometry of the disks does not have to be identical. We are investigating the problem of backing up a source drive to a larger destination drive that may incorporate the backup data while simultaneously being able to store other information.

4.5.3.3 Revocation

The unlabel token comes into use if a label is to be revoked from the disk, e.g., multiple users use the disk and label particular files as immutable with their own token, and a revocation occurs. By inserting the token of the revoked user along with the unlabel token, all block labels associated with the token will be erased from the RRD’s metadata. As a result, these blocks will become mutable and all of the data may be reclaimed.

4.5.3.4 Large-Scale Installations and Upgrades

In environments with many homogeneous machines, performing upgrades with a single hardware token is at best cumbersome and at worst infeasible, necessitating an alternative approach. A common method for rapidly deploying software to large numbers of machines is disk imaging or cloning. Our proposed solution calls for a single master machine to broadcast instructions to other RRDs through a channel that does not involve the OS. For example, software may be installed and configured on a single archetypal machine that is trusted. This machine’s hard disk image is then simultaneously copied to many clone machines. Mutable data may be imaged to these clone machines, but when the token to allow modification of immutable data is inserted into the archetypal RRD, it broadcasts as message to the clone RRDs over a shared medium, such as wirelessly or over a USB bus, to allow writing of immutable blocks and labeling them appropriately. When the token is removed from the archetypal RRD, another message is broadcast that prevents further information from being labeled immutable. A similar process is followed when the archetypal system is to be upgraded.

4.5.4 Considerations for Windows Systems

RRDs can maintain their independence of operating systems by ensuring the correct partitioning of mutable versus immutable files during installation and upgrading of the

\(^4\)We assume that the system administrator has created a duplicate copy of this token.
operating system and its applications. While we have focused on a Linux implementation, an RRD solution would also be suitable for Windows installations. The layout of immutable files differs between Windows and UNIX-type OSes, with system-critical files often residing in the `\Windows` and `\Windows\System32` directories, among others. While the installation process on a Windows system would require some small alterations such that immutable files were installed after the token was installed in the disk, the changes in this sense are similar to those required with a UNIX distribution and could be managed in much the same way. The same is true of using applications such as Windows Update to update the operating system; as many system-critical upgrades require a reboot already, the change from a user's standpoint would be fairly small.

Unlike in a UNIX-type system, configuration and other parameters are centralized in the Windows registry. The registry is not a single database, however; it is split into a number of sections, called hives, which localize functionality. For example, the `HKEY_LOCAL_MACHINE` hive stores global settings, while `HKEY_CURRENT_USER` stores settings specific to a particular user. These settings are stored in binary files in specific portions of the operating system. Notably, `HKEY_LOCAL_MACHINE` has its settings stored in a set of files in the `\System32\Config` directory that are associated with subkeys dealing with security and software settings among others. Because these files may be accessed separately from other registry entries, these files may be marked immutable, as they affect the entire system operation in much the same way as files within the `/etc` hierarchy, without requiring reboots for other less-critical registry operations.

Windows Vista supports registry virtualization [216], where applications that require access to read-only settings, such as system registry settings, can be remapped to locations that do not affect the entire system. In a similar manner, some applications made to interoperate with Windows Vista and older versions of Windows support application virtualization, where all of an application’s registry operations are remapped to another location, such as a file in a user’s space. Through these methods, applications may be accessed and upgraded without accessing system-critical portions of the registry and requiring changing of immutable files.

4.6 Summary

We have detailed the design, operation and evaluation of rootkit-resistant disks (RRD), which immutable system binaries and configuration during initial system installation and upgrades—an operation only available when a physical administrator token is plugged into
the disk controller USB interface. Any attempt to modify these immutable blocks when
the administrator token is not present, i.e., during normal operation of the operating
system, is blocked by the disk controller.

We have shown with the RRD concept how disks can be used to provide services
above and beyond simply securing the data on the disk. With RRDs, the system itself
is protected, and this protection model is only possible because of the disk acting as
an autonomous enforcement point, providing security independently of the rest of the
system. The trusted path formed by the user physically inserting a token into the disk is
the key for us to be able to provide a solution that can be used without the involvement
of the operating system.

Several areas must be investigated in making this approach appropriate for general
use. First, tighter integration between the install programs and the RRD is needed. We
need to more systematically identify the parts of the operating system that should be
immutable. Second, integration with intelligent commodity disks over other interfaces
such as SCSI or IDE/ATA is needed. While our performance evaluation indicates that
such integration may not change the performance footprint much, it is essential for large
or high-value systems that the performance/security trade-offs be carefully mapped and
system parameters selected. Finally, we need explore the usability of administrator tokens
as a method for enforcing security. In particular, we need to know how such a tool will
be used by those in enterprise and home environments, and find ways to prevent their
improper use, e.g., disallowing system booting when the token is present. Answers to
these open questions will inform how such a simple mechanism can measurably protect
real systems against rootkits.
Enforcing Inter-OS Isolation with SwitchBlade

To this point, we have discussed how disks autonomously enforcing security policy can be used to provide protections within an operating system, i.e., we have considered primarily *intra-OS security* to this point. We now turn our focus to how these disks may be used to provide security *between* operating systems, i.e., *inter-OS security*. As we will discuss in this chapter, there is an increasing need to not just be able to isolate processes within an operating system from each other, but operating systems themselves from each other, as the facilities and scenarios for running multiple OSes on the same system continues to increase. Our basic design for autonomously-secure storage systems will be shown to provide a novel means for enforcing this isolation.

5.1 Introduction

Operating systems are under constant threat. Any opportunity for OSes to interact with each other raises the possibility of attack, as has been demonstrated repeatedly from the earliest worms onward [90]. Often the only method of ensuring the isolation necessary between multiple operating systems is to devote separate dedicated machines to each OS. This so-called *air gap* security is the strongest confinement mechanism and may be necessary for red-black networks [217], where different operating systems run at different security levels, but it hampers usability and requires extra resources.

The increasingly complex manners in which users operate their systems complicate these goals. Users have the ability to run multiple operating systems, either one at a time
or simultaneously, on their desktop machines. These *multiboot* and *virtualized* systems, respectively, provide great benefits for users who require access to multiple OSes (e.g., a web site designer ensuring consistent page rendering in different environments), without incurring the energy and space requirements of dedicating an entire machine to an OS. In these systems, the ability to *confine* [49] operating systems from each other effect is critical to preventing information leakage, and to mitigate the effects of a compromised OS from attacking other OSes on the system. Numerous approaches, from secure operating system design [57] and security kernels [58,59] to using separation kernels [61] and virtual machines [66,68,72] have been employed to maintain confinement.

Regardless of how strong the confinement mechanisms between applications or operating systems may be, however, an opportunity exists to tamper with other OSes if they share a common storage medium. Mechanisms such as partitioning the disk are constructions enforced by operating systems, and an attacker that compromises the OS will be able to trivially subvert any protections offered by it. Additionally, booting from a live CD such as BackTrack [218] is often sufficient to gain a full view of on-disk contents, free from access control mechanisms enforced by the OS. Encrypted volume solutions such as TrueCrypt [219] and LUKS [220] are reliant on the OS kernel not being compromised; additionally, these programs themselves may be open to vulnerability [221]. Full disk encryption [222] provides no protections when the disk is unlocked for regular operation.

To address these problems, we propose a novel method of providing confinement. In this paper, we present SwitchBlade, a disk protection model that effectively confines operating systems from each other’s effects. An OS is able to run only in its own *disk segment*, a defined set of blocks on the disk that are accessible to the OS as a physically separate disk. A user is able to access segments by plugging a physical token into the drive. These tokens possess capabilities that define read and write access to one or more segments. Even if the adversary ran BackTrack, the only segments visible to them would be those made available through their policy, affording no opportunity to read, write, or interact in any way with other OSes on the disk. The user forms a *trusted path* to the disk by plugging a token directly into it, ensuring that isolation is enforced independently from the rest of the system.

Many modes of operation are possible with SwitchBlade, and we have explored two of these. In *multiboot* mode, the user will access a different view of the disk and thus, available operating systems, depending on which token is plugged into the disk. In *virtualized* mode, a virtual machine monitor (VMM) and guest operating systems are exposed to the user, where the set of available OSes is defined by the token policy.
thus can enforce isolation between sets of virtual machines that may be differentiated by
security class.

The contributions of this chapter are as follows:

• We define the SwitchBlade architecture for achieving storage level isolation, and
  show how it may be used for protecting the integrity of system components and
  the confidentiality of user data.

• We implement a fully functioning prototype for the purposes of validating our con-
  cept and provide a detailed description of how each component in the architecture
  is implemented using commonly available hardware and software.

• We relate our experiences with configuring the disk policy and building systems
  that use SwitchBlade. This includes descriptions of token management, segment
  creation and the installation and use of a hypervisor and guest operating systems.

5.2 System Goals

SwitchBlade is designed to protect operating systems on disk by isolating access to only
segments on the disk specified by capabilities retrieved from a token, physically inserted
into the disk by the user. Below, we consider the goals of our system and the threats
that it is and is not capable of addressing.

5.2.1 Design Criteria

Our approach is motivated by providing a protection mechanism that is simple to ad-
minister and easy to use, and that enforces a model of confinement ensuring that no
operating system will be accessible to another OS specified to be isolated from it. We
explain these goals in further detail:

Management

Users insert a physical token into the disk that specifies policy through capabilities. As
discussed in Section 5.6, a token manager that operates independently of the system
with SwitchBlade installed is where policy is specified. From the standpoint of the OS
or VMM, they have access only to the on-disk segments that are specified on the token,
which provides mandatory enforcement. No further policy decisions need to be made by
SwitchBlade.
Usability

The user does not need to make any changes to the way they use the system beyond plugging the correct token into the disk, providing them access to the desired operating systems. No further modifications are necessary. SwitchBlade also operates transparently to the operating systems, which require little to no modification to be run.

Isolation

SwitchBlade isolates disk segments from each other. Each segment is presented to its OS as a separate disk, each of which contains a master boot record (MBR). While MBR rootkits [223,224] may infect an individual segment, they will be unable to spread beyond their segment to infect other parts of the disk that the token-based policy does not provide access to. A by-product of this isolation is a natural confidentiality and integrity based on the inability to read or write other segments unless authorized by the token. In addition, having the user physically insert a token into the drive creates a trusted path to the disk that is independent of the potentially compromised OS. The problem of confining multiple segments on the disk reduces to one of physical security through token management, reducing the TCB for segment protection to the disk enclosure.

5.2.2 Securing On-Disk Storage

In addition, often Live CD operating system images may be used to bypass any protections on a disk to allow reading and writing to filesystems located at arbitrary locations on the disk. For example, while a partition table may be enforced by a Linux or Windows operating system, an adversary using a live distribution such as Knoppix could obtain full block access to the disk independently of the controlled-access model that would otherwise be available. While solutions such as full-disk encryption (FDE) would provide some mitigations while a drive is in transit, once the drive is in use and unlocked, any protections afforded by FDE become less powerful. In addition, users communicate with disks through the operating system, giving a malicious OS the opportunity to intercept the key meant for unlocking the FDE drive.

5.2.3 Threat Model

We consider an adversary capable of arbitrarily and completely subverting a host operating system. The OS itself is outside of the purview of what can be protected at the storage level in our design, as we do not consider protections inside the OS. Other
solutions are possible for protecting the OS at the storage level, including storage-based intrusion detection [19] and rootkit-resistant disks, described in the previous chapter.

Many security architectures that consider virtualized environments make the assumption that the hypervisor and associated administrative VM is within the trusted computing base (e.g., [225, 226]). We do not make the assumption that these components are automatically trustworthy, given the ability of an adversary to attack storage by installing malicious software such as rootkits affecting a disk’s MBR. In addition, while a virtual machine based rootkit (VMBR) such as SubVirt [85] may currently be unable to operate under a virtual machine monitor due to its lack of handling nested virtualization, self-virtualization in the x86 architecture through the Intel VT [227] and AMD SVM [228] processors makes the threat of VMBRs running under a VMM increasingly possible. Another assumption we make is that there is no trusted path through the operating system without more trust being placed in the VMM, such as with the trusted VMM in Terra [138].

Our enforcement mechanism takes place within the disk. We assume that the attacker does not have the ability to mount physical attacks against the disk, such as opening the drive enclosure to attack the media or X-raying the disk. Physical security measures such as those used in secure coprocessors [101] may be possible if the cost-benefit ratio makes this design point appropriate. In addition, we do not consider physical attacks against the tokens; however, solutions for secure, tamper-resistant tokens such as those found in the IBM Zurich Trusted Information Channel [229] are possible where such defenses are deemed necessary.

5.3 Architecture

SwitchBlade is a protection system consisting of a disk augmented with a processor for providing cryptographic services (such as those available in an FDE drive) and policy evaluation, and non-volatile memory (as found in a hybrid hard disk) for storing policy metadata. The goal of SwitchBlade is to provide storage level isolation for integrity, confidentiality and availability of storage resources. This is accomplished through the use of disk *segments*, which are analogous to the memory segments used in microprocessor architectures to provide hardware level memory protection. A disk segment appears to the host system as a single device on the storage bus, and is addressed by the host as if it is an actual disk. The host should not be able to tell the difference between a set of segments exported by a virtual disk and a chain of devices on a storage bus. The set
of disk segments accessible by the host is determined by the capability on the current token in the disk, allowing the set of devices currently available on the bus by the host to change dynamically through the use of different tokens. The SwitchBlade architecture is shown in Figure 5.1. We now examine in detail how policy is expressed, how SwitchBlade is managed, and how it is used during system boot.

5.3.1 SwitchBlade Policy

The SwitchBlade policy has two main objectives. The first is to define the controlled access to disk segments (hereafter referred to solely as segments). This includes the label state of all segments on the disk as well as the semantics of capabilities stored
Capabilities:
{D1: r, w}
{D2: r, w, d}
{DBoot: r}
{can_create}

Commands:
[delete_disk:D1
 create_disk:D3; 10000; r, w, d]

Figure 5.3. An example of the contents of a SwitchBlade token.

on the physical tokens. The second is to control access to the various administrative functions needed for disk management, including segment creation and deletion. In the SwitchBlade policy, segments represent the objects in the system, and the set of sensitive operations includes the creation, deletion, reading and writing of segments. Subjects are authenticated by the current token in the disk. Each segment has several attributes:

- **name** - The unique identifying name for this disk. In the case of a SCSI or ATA bus, this could be the device identifier of the disk.
- **size** - The amount of disk space allocated for this segment
- **labels** - The set of labels defining read and write access to the segment, as well as the ability to delete the segment.

The attributes for each segment are stored in the disk’s non-volatile memory to allow for concurrent access to the policy metadata and data on the disk. Access control decisions are made for each segment based on the segment’s attributes, and the set of capabilities on the current token in the disk. Along with the specific labels needed for each segment, there is a global capability needed to create new segments. A token with this capability may issue one or more `create_disk` commands, which result in the creation of new segments (covered in Section 5.3.2).

Each physical token contains capabilities, where each capability maps a segment name to a set of labels specifying the operations that may be performed on that segment. Also present in a capability is the optional label to allow for the creation of new segments from the available free disk space. Along with these capabilities is a queue of `create_disk` and `delete_disk` commands to be executed. These details are described in the next section.

An example of token contents is shown in Figure 5.3. When this token is in the disk, segments D1 and D2 can both be read and written, segment D2 can be deleted, and the
boot segment is read-only. Upon insertion into the disk, the command set from the token will be parsed and executed. In this case, $D1$ will be deleted and a new segment, $D3$, will be created with 10GB of reserved space and the set of labels $r, w, d$ (capabilities for segment read, write, and delete, respectively).

Because no assumptions are made about the available disk space or the permissions stored in the capability before `create_disk` or `delete_disk` commands are issued, they may fail. An administrator would like to know about the success or failure of each command, and in the case of a failure, the cause. For this reason, the disk includes a tamper-proof audit log. This log is exported to the host as read-only as long as the disk is powered on. Only the disk controller may append messages to the audit log segment. Common interfaces to the log include the OS file system on the host and the segment menu as described in Section 5.4.2.2.

The enforcement mechanism for the SwitchBlade policy is a part of the disk’s firmware and runs in the disk processor alongside the normal processing code. The enforcement mechanism is completely independent of the rest of the disk controller code, and is for all purposes verifiable. It consists of a single hook that intercepts all I/O requests at the disk interface and inspects their command fields to see if they are requesting a sensitive operation. If so, the current set of labels available for the specified segment on the current token are checked to make sure the request can be fulfilled. If so, the command is passed unmodified to the regular controller code. The independence of the enforcement mechanism and the disk controller code allows for a performance optimization: a pipeline of disk requests may be formed, allowing current requests to be authorized while previously authorized requests are fulfilled.

### 5.3.2 SwitchBlade Management

Up to this point, we discussed disk segments in terms of their attributes and the policies controlling their access and manipulation. We now examine disk segments in terms of their contents and the administrative procedures for managing them. We also describe token management and how SwitchBlade is used by the host system.

The key idiom that we are assuming for general use of SwitchBlade is that each segment contains at most one operating system (although, as we show in Section 5.4.2.2, it is possible to run more than one OS inside a segment). This may be a guest OS in a VM environment, or a standalone OS available as part of a multiboot configuration. Each OS will view its segment as a completely separate disk, and will keep both its file system and swap partition on this segment. A policy may also be defined for segments
that only contain data, which can act as shared storage between multiple running OSes. These segments will most often be readable and writable.

In the case of a multiboot system, the usage of a SwitchBlade is simple - isolation is maintained, since the OS has no knowledge that other storage even exists on the disk. In the case of virtualized operation, there is an opportunity for more powerful isolation mechanisms between running VMs by trusting the VMM to multiplex access to virtual disks over VMs. In this case, the VMM is aware of each segment exported by the disk, and allows each guest OS access only to the segment from which its image was retrieved, using storage virtualization methods discussed further in Section 5.4.2.2.

We believe that the VMM can provide some weak guarantees for several reasons. As we discuss in the next section, SwitchBlade possesses the necessary mechanisms to protect the integrity of the VMM stored in the boot segment, ensuring that the VMM is always started in a good state. More importantly, in the event that the VMM is compromised, the malicious VM is limited to affecting the integrity and confidentiality of the files in segments exposed by the current token in the disk.

Tokens used with SwitchBlade are created and modified by a trusted token manager, discussed further in Section 5.6. The token manager is used to construct capabilities and instruction queues and push them to tokens.

5.3.3 Boot Process

The process of booting from a SwitchBlade is similar to that of booting from a regular disk. The equivalent of a disk’s MBR in SwitchBlade is the boot segment, which has several additional features beyond those found in a typical MBR. First, because segments may be of arbitrary size, the boot segment contains a stage-1 bootloader with enhanced functionality, as it needs to inspect the disk to discover which segments contain OSes. Additionally, the boot segment is almost always kept in a read-only state to protect the integrity of the code involved in the boot process, thus ensuring that the system can be booted into a safe state.

Exporting the boot segment allows SwitchBlade to provide some guarantees of integrity. Namely, the disk can guarantee the integrity of the contents of the boot segment. Once executed on the host, the boot segment can further verify the integrity of higher levels of software. This is achievable by maintaining a measurement of a good boot system on the token, and comparing this value to a measurement of the boot segment. In the case that the measurement of the boot segment does not match the value stored on the token, SwitchBlade will refuse to satisfy requests for blocks in the boot segment, and
will append a message to the above described tamper-proof audit log.

There are two times at which the boot segment should be measured: when the disk is powered on, and when it has previously been exported as writable. In the former case, the boot segment contents may have been tampered with, and in the latter case, they may have been altered by a compromised host OS. In the case of a regular upgrade of the software on the boot segment, which we assume is relatively rare compared to OS upgrades and more akin to a firmware update, the measurement stored on the token will need to be updated to match that of the new boot segment.

Note that this approach provides guarantees about the state of the boot segment but as per our threat model, we do not guarantee against physical tampering of the disk’s hardware and as such, we do not provide attestation of it. This may be possible if a device such as a trusted platform module [230] is installed in the disk for attestation to the host system or an active token; alternately, a secure boot that measures its own integrity in a manner similar to AEGIS [135] is possible. While we defer further discussion of this issue, it is notable that if we consider the physical componentry of the disk to be trusted, the disk acts as a root of trust.

5.3.4 System Management with SwitchBlade

We now describe system management techniques for multiboot and virtualized operation, and examine how they may be implemented using SwitchBlade and the policies, disk management mechanisms and boot process described above. In the case of secure multiboot, a workstation has several operating systems that may not access the data on each other’s segments. The bootloader resides in the boot segment, from where the BIOS retrieves it for execution. The bootloader then probes for available segments and loads the OS from the OS segment. A policy may be specified to allow a shared segment to be accessed by multiple operating systems. In this case, the token will export two segments, one containing the OS that will be run, and one containing the shared data. Once the OS segments are loaded, the shared data segment may be mounted, as well as the segment from which it was loaded as a regular block device by its name.

In the virtualized mode of operation, we wish to have multiple OSes running concurrently, each with access to their own segment and no other, thus preventing any information leakage between OSes through storage\(^1\). In this case, a VMM resides in the boot sector that is trusted to multiplex access to the multiple segments exported by the disk.

\(^1\)We do not claim to provide protection against copy-paste or other forms of information leakage above the storage layer; solutions such as labeled desktops in Trusted Solaris [231] may provide for this level of mediation.
The VMM uses a policy that a guest OS can only access the segment from which it was loaded. In order to protect the integrity of the VMM image, the token should only contain the read capability for the boot segment. In both of the above-described cases, the create_disk and delete_disk capabilities will not be needed as a part of normal system management.

5.4 Prototype Implementation

In order to prove the SwitchBlade concept and have an experimental testbed for security and performance analysis, we implemented a prototype. We first describe the implementation of the disk itself, including the command protocol, the token interface, and segments. We then discuss the deployment of SwitchBlade in virtualized mode, using Xen as a VMM.

5.4.1 SwitchBlade implementation

We implemented a prototype SwitchBlade with the capacity for creating, deleting and exporting segments based on the capabilities present on the physical token. We then used the prototype to build the two types of systems described in Section 5.3.4, secure multiboot and virtualized operation. Because of the prohibitive difficulty of obtaining and modifying the firmware of a commodity disk, we implemented our prototype using a modified network attached storage device. The main difference between the prototype’s interface and that of a commodity disk drive is the use of Ethernet for the physical and MAC layers between the disk and host, as opposed to an ATA or SCSI bus.

In order to minimize differences between our prototype’s command interface and that of a real disk, we use the ATA over Ethernet (AoE) [232] command level protocol between the disk and host. As the name implies, AoE sends ATA commands directly over the Ethernet interface between a client and host. Unlike iSCSI [233], AoE does not use higher level network or transport layer protocols, reducing performance overhead in the network stack normally incurred by protocols such as iSCSI; while this does not allow commands to propagate beyond an Ethernet segment, our assumption is that the disk will be co-located with the firmware in a SwitchBlade system, making this access mode more appropriate. In a typical AoE installation, a host machine, or initiator, issues ATA commands to a target storage device. In the case of the prototype SwitchBlade, the initiator consists of the host machine using one or more aoe block devices, which issue commands to AoE targets on the LAN. The targets consist of the modified NAS devices
running `vblade`\(^2\), a server program that listens for ATA commands via raw sockets. Individual AoE devices are addressed by a major and minor number used by the target to demultiplex commands to each device.

The SwitchBlade prototype as shown in Figure 5.2, consists of two discrete physical components, an external USB hard disk which provides the actual storage and a NAS device which sits between the host and disk that acts as the policy store and enforcement mechanism. For the latter, we use a Linksys NSLU2 storage link [204], commonly known as a “slug.” In order to program the slug to act as a policy enforcement point, we replaced its default firmware with the SlugOS Linux distribution [205], allowing us to leverage a large collection of existing software, as well as write our own. Token handling in the slug is performed using the `udev` [212] device management framework for the Linux 2.6 kernel, which detects when a token is inserted or removed. Upon token insertion, `udev` invokes a token handler program we wrote that extracts the capability and commands from the token. A new instance of `vblade` is started for each segment specified in the capability.

The basic unit of storage over which access control is performed in our prototype is the segment. Segments are implemented in our prototype using the Linux logical volume manager [234]. The segments are specified as logical volumes (LVs) over the single physical volume (PV) that is the external USB storage device. We then use a single instance of `vblade` to export each LV to initiators. If multiple segments are to be exported under a given token, we launch as many instances of `vblade` as there are segments to be exported. Each segment is addressed by the host using its unique major and minor number as exported by AoE, and is visible to processes on the host as a block device containing the major and minor number of the segment, e.g. `/dev/e1.1`, `/dev/e1.2` etc.

5.4.1.1 Audit log

We were able to easily use the implementation of segments to create the tamper-proof audit log. An instance of `vblade` is started to export the audit log read-only at disk power-on. When the token handler executes any commands or detects tampering to the boot segment, it appends detailed messages to the audit log by writing directly to the LV. Any guest OS may obtain the audit log contents by mounting the AoE device with minor number 0 to its local reading the contents of, for example `/mnt/audit_log`. Such functionality can be used as the basis for the implementation of an S4 disk [20] which

\(^2\)The genesis of the name “SwitchBlade” is because of `vblade`, which exports virtual storage blades to the user. We provide the ability to “switch” between these blades while keeping them isolated from each other.
requires the existence of an append-only audit log provided by the disk. We have not investigated other requirements for S4, such as journaled metadata and log-structured writes, as those are orthogonal to our goals with SwitchBlade.

5.4.1.2 Policy management

For each segment in the prototype SwitchBlade, an entry is stored in the disk’s NVRAM, containing the segment name, read and write labels and the minor number used to identify that segment when exported. When a token is inserted into the disk, the capability is extracted, and from it are parsed a set of segment names and the corresponding read and write labels and minor numbers. The pair of labels for each name is compared against that stored in NVRAM and used to determine whether the segment is exported and if it is writable. In the case where only a read label is present, the segment is exported as read-only, and in the case where only a write label or no labels are present, the segment is not exported. An instance of vblade is pointed at the corresponding LV for each segment to be exported, each of which is flagged as read-only. At this point, the initiator will detect the new block device.

5.4.1.3 Boot sector integrity requirements

SwitchBlade verifies the integrity of the boot sector against a measurement stored on the token. As discussed in Section 5.3, the boot segment must be measured after the disk is powered on, and after the boot segment has been exported as writable as the disk is vulnerable to tampering in either circumstance. To ensure that measurements are always done at these times, the prototype uses a UNIX temporary file that is cleared either by tmpfs at disk power down, or by the token handler when the boot segment is exported as writable. Any time the file is not present, the SHA-1 hash of the boot segment is computed with OpenSSL [235] and compared against the hash stored on the slug. If the contents of the boot segment have been intentionally altered by an administrator, e.g. for a bootloader upgrade, it is up to the administrator to ensure that the proper new measurement is stored on the token.

5.4.2 Client System

In this section, we explain how SwitchBlade is implemented to operate in either multiboot or virtualized mode. We begin with the multiboot scenario and then cover the installation procedure for the virtualized operation.
Figure 5.4. The process of booting the host system from the Slug disk. The initial phases are the DHCP and TFTP protocols in order to retrieve the kernel, hypervisor and initial ramdisk (initrd) images. Once the host has started booting the system, it discovers the available AoE devices, which will include the root file system for the platform.

Figure 5.5. Operation in a virtualized environment. Each ATA-over-Ethernet (AoE) device is a separate block device that either dom0 or the domUs can access. The domUs are unaware that the block device runs AoE.

5.4.2.1 Multiboot Operation

In order to provide secure multiboot, we use the Preboot Execution Environment (PXE) [15] network-based boot, supported by many common network cards. The slug acts as a server that provides the kernel image for the host to boot from. In order to boot using the AoE targets made available by the slug, some modifications to the OS are typically needed. We experimented with the Ubuntu and Debian Linux distributions, and modified the initrd image to load the AoE driver and mount the AoE segment for the root file system. For Windows XP, the additional steps involved installing the AoE driver, and then using the open-source gPXE software [236] to boot over AoE. The slug’s PXE boot process is illustrated in Figure 5.4. When the system is first powered on, it sends a broadcast request looking for a DHCP server, which sends a response indicating the location of the TFTP server. In this case, both servers are on the slug. The host then sends a request to the TFTP server for the network bootstrap program (NBP), responsible for retrieving
the files necessary to boot. For multiboot operation, these are the kernel and \texttt{initrd} images, or some next stage bootloader. For virtualized operation, the hypervisor image is also sent to the host. The host begins its boot using these images, and once the boot has progressed far enough to load the AoE drivers, the root file system can be mounted and the boot can proceed as normal.

Note that the need for PXE boot and OS boot modifications is necessary only because we deployed the SwitchBlade logic on the slug. On a production device, the kernel image would be booted directly off the boot segment over an ATA or SCSI interface and the need for AoE would be obviated.

\subsection{Virtualized Operation}

Figure 5.5 provides an overview of how SwitchBlade operates in virtualized mode. We use the Xen [71] VMM to support running virtual machines. The host system contains a processor with AMD virtualization extensions, allowing Xen to run unmodified guest operating systems. The initial setup involves installing the hypervisor and administrative domain (collectively called dom0), which is achieved in two phases. The first phase is installation of a base Linux environment on the root segment (using a Debian installer) and manually loading the AoE driver, while modifying \texttt{initrd} to boot an AoE root file system. The second phase involves copying the kernel and \texttt{initrd} images to the slug in order to allow PXE boot.

Once the system has been rebooted, Xen is installed and the kernel and \texttt{initrd} are again copied to the slug along with the hypervisor. A final reboot allows the slug to serve the VMM image, and creation of guest domains (domUs) can occur at this point. We installed four different guest OSes on SwitchBlade, including Windows XP, Windows Vista, OpenBSD 4.3, and OpenSolaris. To demonstrate that each segment considers itself to be a physical disk, we created a fifth segment and installed a dual-boot configuration of Windows XP and Ubuntu Linux 8.04.1 inside of it.

To provide a usable interface to the user, we have developed a menu interface that shows the user which virtual machines are currently able to boot given the inserted token. The user is also presented with an audit log that shows token activity from the disk. The interface is shown in Figure 5.6. This figure contains four different windows. The top two windows are VNC connections to the VMs that are running, namely Windows Vista and OpenBSD 4.3. The lower-left window is a terminal showing the VMs that the hypervisor is currently configured to run. The list includes five different VMs plus a listing for dom0. The window in the lower right is the user interface which only shows VMs that
5.4.3 Experience

To illustrate the typical use of SwitchBlade in a production environment, we describe our experiences performing several common administrative tasks. We cover regular operation, which consists of segment and token management and segment creation. We also describe the response of different OSes to unexpected token removal during regular operation, when given no warning from the SwitchBlade and when gracefully shutdown by the hypervisor.

5.4.3.1 Segment Creation

Each physical token contains a capability and a list of one or more create and delete disk commands.

create_disk:boot:1000:boot_r:boot_l:1:1
create_disk:vd1:10000:vd_r:vd_l:2:2

Instructing the disk to create a 1GB segment called “boot” with the specified read and write labels, and minor number 1 and a 10GB segment called “vd1” with read and write labels, and minor number 2.
write labels \texttt{vd\_r} and \texttt{vd\_w} and minor number 2. Upon inserting the token, the disk removes the commands from the token, creates the segments, and pushes the names of each segment with the provided labels back to the token. The names of each segment along with labels and minor numbers are then stored in the disk’s NVRAM. Because the boot segment should normally be read-only, we edit the token contents using a trusted workstation by removing the write label from the entry for the boot segment. Upon reinserting the token, the boot sector was exported read-only.

In order to verify that the disk successfully verifies the integrity of the boot sector, we modified replaced one block of the boot segment with random bytes and rebooted the SwitchBlade. At power on, the disk would no longer export the boot segment and recorded a message to the audit log containing the measurement from the disk and the expected value from the token.

Once the token containing a \texttt{create\_disk} command for the boot segment is inserted, installation of the boot segment OS can proceed. The installation requires very few modifications from the standard installer. During the installation, the kernel module for the ATA over Ethernet driver is loaded and the boot segment block device is found. The installation procedure is able to correctly install the base system to the boot segment. Once the base installation finishes, the bootloader installation is skipped since the Slug is responsible for booting the host system, through the use of PXE.

\textbf{5.4.3.2 Regular Operation}

Once the boot segment is installed, virtual machine creation can commence. Installation of the virtual machines requires no special modifications to the installation procedures. In order to interface with the virtual machines, Xen provides a VNC interface. These interfaces present the same view the user expects if installing the OS to a physical platform and not in a virtual machine. Once the virtual machine OS is installed, the OS sees a traditional physical platform, with no indication that it is actually running within a domU on the host system.

Along with the VNC interface for the virtual machines, the menu interface is provided for interaction with the boot segment, or dom0. The menu application interfaces with dom0 to determine what virtual machines are available, and presents the user with only valid virtual machine choices. The interface is show in the lower right corner of Figure 5.6. The interface is simple, only allowing the user to start and stop the listed VMs shown in the upper half of the window. The lower half of the window contains the audit log kept by the disk.
5.4.4 Abnormal Token Removal

If a token is removed from the system while the system is running and without notification to the utility running on the host, a number of different scenarios may emerge. If there is no notification to the host at all then the partition will simply disappear. To an OS using that particular segment, it will be as if the disk suddenly became completely inaccessible for reads and writes. If the token is re-inserted into the drive, then depending on the robustness of the OS to losing connectivity with a disk, recovery mechanisms may be employed. Otherwise, the OS will have crashed and will require reboot and the filesystem may require cleanup through a utility such as `fsck`.

In a multiboot setting, if the VMM receives a message indicating that segments are about to become unavailable, a more graceful shutdown mechanism is possible. The VMM can signal the disk (e.g., through a special SCSI or ATA command) to keep segment access available. The disk can refrain from removing access to the segments even if the token is removed, until it receives another signal from the host that the guest operating systems dependent on the affected segments have shut down. At this point, the segments will become unavailable. If there is concern that the VMM may have been hijacked after sending the signal to refrain from removing segment access, and refuses to send the completion signal even after the token is removed from the disk, the disk may consider a time-based expiration window. In this case, after a policy-dependent period of time, even if the shutdown signal is not received from the host, if the time window expires then access to the affected segments will be removed.

5.5 Evaluation

We now analyze the security of our architecture against several types of attacks on integrity and confidentiality and evaluate the performance of our prototype implementation.

5.5.1 Security Analysis

The purpose of SwitchBlade is to provide storage level isolation to protect certain portions of storage from compromised programs running on the host. We provide a non-exhaustive list of attacks that are indicative of how systems may be attacked and the defenses that we provide. In addition, we discuss how SwitchBlade provides complete mediation of block requests in a manner consistent with reference monitor guarantees.
Live CD attacks

One attack against which traditional storage devices are defenseless, is a takeover of the host through the use of a live CD. By booting a system from a live CD, an attacker can bypass file system-level security policies, accessing all data on the disk through the block interfaces. In the case of SwitchBlade, a live CD would in the best case not be able to access any segments due to the lack of physical token. Assuming a more powerful adversary that may possess some tokens, the accessible disk blocks are limited to those exported by the tokens.

Compromised OS

A second example of an attack in which access to “bit-bucket” style storage devices is unrestricted, is the case of a compromised OS on the host system. Control of the OS allows an adversary to linearly scan the entire logical block address space, accessing all data on disk. Assuming a secure multiboot system, in the case of OS compromise, only the data in the segment of the currently running OS is accessible. Similarly, in a red-black isolation system, if the VMM is compromised, access to data on disk is limited to the set of segments visible to the VMM.

This example illustrates the limitations of using TPM-based integrity measurement alone. In the event that a host OS or VMM is subverted, it can simply stop providing measurements of newly loaded code to the TPM, effectively stopping integrity measurement. Even when recorded correctly, integrity measurements provide minimal forensic value in comparison to the audit log maintained by SwitchBlade. While a measurement list can reveal that some untrusted or unknown program was loaded, it says nothing about what disk accesses it may have attempted or if they were allowed.

Availability attacks

Another attack class not previously addressed by storage devices in multi-user systems is that of attacks on resource availability. In this case, the resource is free space. SwitchBlade may protect against attacks in which disk space is intentionally exhausted, or disk quotas surpassed. Because each segment has an associated size, quotas can easily be enforced by assigning a different segment to each user, where that segment’s size is that user’s quota. Similarly, because the permission to create new segments is limited to physical tokens containing create_disk commands, the global amount of free space on the disk may also be controlled.
Extortionware

An attack on the availability of data on the disk that has emerged is that of extortionware, also called ransomware [237]. Extortionware refers to malicious applications that encrypt disk contents with a public key, to which the corresponding secret key is known only to an attacker who’s demands must be met in order for the encrypted data to be recovered. In the secure multiboot case, only the data on the segment containing the OS running when a user executes extortionware is lost. In the red-black VM case, if the extortionware is installed via social engineering and the VMM is left uncompromised, only the data in the segment corresponding the to guest OS on which it was executed is lost.

Complete Mediation guarantees

The above-described security semantics of SwitchBlade are only useful if we can guarantee that it provides complete and correct mediation of block requests. To do this, we reason about the verifiability and complete mediation properties of the SwitchBlade enforcement mechanism.

We can reason about complete mediation in terms of authorization hook placement. In the case of the SwitchBlade reference monitor, hook placement is simple enough to be verified in two dimensions, vertical and horizontal. We define horizontal hook placement as the number of hooks needed to mediate all resources, i.e., all of the execution points requiring authorization checks, and vertical hook placement as the overall complexity of each authorization check. Horizontal hook placement in SwitchBlade is nearly trivial. A single hook intercepts the serial queue of requests from the host. Because the semantics of each command are independent of other commands, the enforcement mechanism does not need to check for conditions that lead to common TOCTTOU attacks [238], in which dependent file system operations can lead to undetected policy violations. The vertical complexity of hooks is also minimal. Requests to SwitchBlade arrive over an ATA interface from the host. Each hook need only extract the ATA protocol’s command code from the command descriptor block and check the target segment. The hook then checks for the label associated with that command for the target segment in the token plugged into the disk.

5.5.2 Performance Evaluation

In order to better understand the performance bottlenecks of our prototype and how they may be improved upon, we conduct a performance evaluation of two aspects of the
<table>
<thead>
<tr>
<th>Block (Bytes)</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slug</td>
<td>447.30</td>
<td>1375.51</td>
<td>3230.91</td>
<td>4880.61</td>
<td>5731.66</td>
</tr>
<tr>
<td>Xeon</td>
<td>25908.53</td>
<td>70506.03</td>
<td>181938.9</td>
<td>300317.70</td>
<td>370849.11</td>
</tr>
</tbody>
</table>

Table 5.1. Throughput of in KBps for various block sizes as output by measuring the speed of SHA-1 hashing with OpenSSL.

We use OpenSSL to perform cryptographic operations on the slug. To get an initial estimate of the throughput of computing sha1 hashes on the slug, we ran `openssl speed sha1` command on both the slug and a machine with a 2.8 GHz Intel Xeon processor for comparison; the results of this test is shown in Table 5.1. At these speeds, measuring a 580 MB boot segment, the size needed for the Xen hypervisor and Debian dom0 kernel, took approximately 2 minutes and 45 seconds.

Obviously, a more efficient method of measurement is needed. This can be achieved either by reducing the size of the boot segment, or by improving the cryptographic hardware available to the SwitchBlade. The former case may be achieved by using a more minimal boot system, such as VMWare Server ESXi [239], which requires only 32 MB of storage for the base system. We performed the same measurement on a 32 MB segment in approximately 8.15 seconds, which in the context of a system reboot is insignificant. In addition to reducing the size of the boot segment, performance gains may be made by offloading cryptographic operations to a special-purpose co-processor. High-performance ASICs can compute SHA-1 hashes at rates greater than 1.5Gbps [240].

We used the Postmark benchmark utility to evaluate the performance of the disk. We found that the read throughput was 2.75MB/sec and the write throughput was 1.07MB/sec. These low numbers can be attributed to the 100Mbps link between the slug and the host system. With a commodity disk connected to the SATA/SCSI/ATA bus, higher throughput is possible.
5.6 Discussion

5.6.1 Token Management

As discussed in Section 5.3, there is a trusted token manager that is used in conjunction with the SwitchBlade. In a large-scale environment of deployed systems, the token manager may reside on a dedicated administrative workstation, isolated from the network.

The manager has interfaces that allow for programming the token with its capabilities, placing `create_disk` and `delete_disk` commands on the token for the creation of new disk segments, and generating a label for the token, which in turn may map to a globally unique physical identifier on the token. Tokens may then be duplicated by inserting a blank token along with the token to be copied into the token manager, which would transfer the information to the blank token while retrieving its identifier for accounting purposes.

The administrator would keep a copy of a token separately from users to provide an easy method of revocation: the token manager can push a `revoke` command to the token that is revoked, which in turn would push that information to a SwitchBlade when it is plugged into the drive. SwitchBlade would then keep the revoked token’s label in NVRAM such that if the token were plugged in by a user, it would not be accepted.

5.6.2 Disk Administration

Communicating with SwitchBlade

Because the operating systems running in a SwitchBlade environment are not trusted, performing any maintenance and administrative tasks are more difficult. Above, we described a “sneakernet” mode of communicating information from the token manager to SwitchBlade. Griffin et al. [124] considered the issue of how to communicate with a disk in a secure manner, as we previously discussed in Chapter 2. Their proposed method was to install a secret key in the disk’s firmware and create a cryptographic tunnel between an administrative console and the disk, with the host system interposing on these requests and routing them to the disk through extensions to the underlying SCSI or ATA protocol. This form of communication requires the operating system to be modified to accommodate these requests, which may be arbitrarily dropped if the OS is compromised. We considered out-of-band methods of communicating with a disk in Section 4.5, including potentially sharing a common bus or a wireless signal from a master machine that disks would listen for. If the environment dictates the need for rapid
revocation [241], one of these communication methods may be appropriate.

**Backup**

Using the token manager, an administrator can encode the “backup” capability onto a token for a single use. When this token is inserted into SwitchBlade and another device is connected, the segments specified to be exposed by the administrator will be transferred between disks. Metadata such as capability mapping will also be transferred. The administrator may choose to have all segments exposed for backup or to only backup a subset of the available disks. Because an administrative token with access to all segments may expose every segment on the disk, an administrator must be judicious regarding its use. For example, in a red/black storage setup, it may be desirable to perform a backup as two operations, for transferring red and black segments separately, to prevent exposure of different-colored segments to each other.

**RAID Operation**

Operating SwitchBlade as a RAID configuration is possible if enforcement occurs at the level of the hardware RAID controller, which enforces segmentation. If a disk fails, it must be securely erased, particularly if it contains security-sensitive information: guidelines for disposal of sensitive media should be followed. Rebuilding a disk requires an administrative token programmed with the **rebuild** capability. The writing of data from multiple segments to the new disk media happens at this stage and is unavoidable; thus, the RAID controller must be trusted to properly map the interleaved data to its correct disk location and to subsequently enforce segmentation.

**5.6.3 SwitchBlade Applications**

In this section, we provide two examples of how SwitchBlade’s ability to enforce isolation through token-based policy may be used for real applications. The ability to provide a token-based policy specified directly by the user creates the potential for interesting use cases. As discussed throughout, the isolation between operating systems gained through SwitchBlade allows deployment in environments such as red/black networks, assuming that the host met the standards for red networks. The token-based interface lends itself to allowing other applications as well.
Voting systems

SwitchBlade can enforce a separation of duties by requiring multiple capabilities to allow access to a segment. This may be a useful method for enforcing election procedures. For example, officials from two or more parties may need to be present in order to enable certain functionality in a voting machine, such as supervisor tasks or recount mechanisms. This would be enforceable by specifying multiple capabilities necessary for the segment to become available and giving each token one capability, requiring all of them to be inserted into the system for the segment to become available and the corresponding functionality to be unlocked.

Trusted auditing

SwitchBlade may be used to enforce segments that are write-only segments except in the presence of a trusted auditor, who would insert a token in order to read the segment and examine the log that is written. Such enforcement mechanisms may be used to ensure compliance with internal controls as specified in Sarbanes-Oxley legislation. While write-once, read-many (WORM) systems allow append-only access (also possible with SwitchBlade), a write-only policy may be even more beneficial as it does not allow employees the ability to scrutinize the logs in advance of an audit. This “write-once” device is considered a viable method of ensuring a trusted audit [242].

5.7 Towards More Complex System Security with Autonomous Storage

This chapter has presented SwitchBlade, a novel storage device architecture to provide isolation that is both defined and enforced below the storage interface. SwitchBlade enforces an isolation policy based on capabilities stored on physical tokens presented to the disk, providing a trusted path for policy configuration and user authentication. SwitchBlade provides isolation guarantees equal to those of physical separation, removing the need to trust OSes and VMMs to enforce isolation policies. We describe our experiences in building, configuring, using and evaluating a SwitchBlade prototype, and show how it may be used to implement secure multiboot and red-black style VM based systems.

We have shown that autonomously-enforced security through the disk can be used to provide solutions to security issues within the operating system, through application of block immutability as a technique to defend against rootkits. We have shown in this
chapter that inter-OS security properties, namely isolation, are possible through a similar token-based mechanisms. These mechanisms can work in conjunction with each other, and provide a basis for whole-system security.

Combined with the previous chapter on rootkit-resistant disks, we have shown how token-based approaches to disk security can be used to provide an environment where security can be enforced directly through interaction with the disk, in a manner independent of the host. This is the first work to show that such an approach can lead to practical solutions for host security.

There are limitations to this token-based approach, however, as our investigations have uncovered. From the work on RRDs, we saw how our abilities to enforce semantic security at the file level were hampered by our lack of knowledge regarding filesystem metadata. This is a result of the semantic gap between the filesystem and the block layer. Being able to enforce finer-grained mechanisms that allow us to leverage this information requires a level of trust in the host that to this point, we simply cannot make.

Additionally, we described in this chapter that our current approach provides guarantees about the boot segment’s state, but that we do not guarantee against physical tampering of the disk hardware and hence, do not attest it. Having a level of trust in the hardware that originates from a core root of trust within the device, such as a trusted platform module, may provide the ability to make stronger guarantees and therefore the ability to provide enhanced security services.

What these two examples show is that without further involvement of the host and being able to trust both its software and hardware, we will be fundamentally limited in the properties that we can enforce. This issue will form the focus of the next two chapters.
Chapter 6

Storage-Enabled Secure Boot
with Firma

Our work in the previous chapter has shown how autonomous storage can be used to enforce policy independently of the operating system, allowing us to provide protections even in the event of OS compromise. This chapter will show how we can use the disk as a means of attesting the good state of the operating system, allowing us to add the host system back into the TCB with measured guarantees.

6.1 The Need for Commodity Secure Boot

An important security goal is to only allow processes running code that is known to be high integrity to access security-sensitive data. For example, secrets should only be made available to processes trusted not to leak them. Also, high integrity data should only be made available to processes trusted to preserve their integrity and perform updates correctly. As loading a single process depends on the correct functioning of several software layers (e.g., BIOS, bootloaders, VMMs, etc.) in modern systems, a secure system requires a mechanism that verifies each of these layers is satisfactory and that protects access to security-sensitive data based on the results of this verification.

A secure boot mechanism aims to provide the guarantees described above. Secure boot was first introduced in the AEGIS system design [135]. Assuming that the machine’s hardware is trusted, the integrity of a software layer is valid if and only if: (1) the integrity of each of its lower layers is verified to be valid and (2) transitions to a higher layer occur only after verification of that layer is complete. Secure boot ensures
that if a system is running, then all its processes satisfy the verification requirement for integrity (i.e., at least up to the extent of secure boot). Further, certain components, such as operating systems and virtual machine monitors (VMM) are verified to enforce protection of security-sensitive data access. The need for these protections is particularly acute in VMMs, given that they must provide isolation assurances between the multiple simultaneously-executing virtual machines that they manage.

While the notion of secure boot has been around for a long time and there are several commercial systems that provide a form of secure boot (e.g., [102]), these secure boot approaches either require specialized hardware or depend on the careful management of cryptographic data. In the first case, special hardware is required to store and/or protect secrets that define the root of trust. AEGIS required a special PROM board [135], and current products still require hardware extensions [102]. In the second case, controlling the release of encryption keys serves as a form of secure boot, as the secret keys are only provided to systems that can be verified to meet specified integrity criteria [219,220,222], but such systems require significant key management. In addition, once the keys are released, all data may be accessed, so if low integrity software is subsequently run data protection may be circumvented.

Our aim is to enable secure boot on commodity systems with minimal management overhead. In this paper, we propose to leverage the processing capabilities found in modern disks to enforce secure boot. The idea is that the host must prove its integrity to the disk prior to receiving security-sensitive data. This enables the disk to protect the data secrecy by only releasing data to systems trusted not to leak the data. Further, the disk can also protect data integrity by only releasing it to high integrity systems. Unlike previous secure boot mechanisms, using the disk as the mediation device also enables the protection of disk data updates. Only authorized systems can update particular disk volumes. We demonstrate this approach by constructing Firma, a novel storage architecture for securely booting a trusted virtual machine monitor (TVMM).

A major goal in our design of the Firma system is to enable secure boot with minimal administration. A key insight is that a Firma disk can act as a remote platform in an attestation protocol. The host can use its TPM to generate attestations that the Firma disk can verify prior to granting access to the next layer. Thus, Firma administration requires that enable a Firma disk to become an attestation client (e.g., obtain the host’s TPM key and high integrity file information), and we show how the root-of-trust-installation method [209] can be used to supply this information easily. Finally, we note that a Firma disk verifier makes a good client for remote attestation as both the TPM and disk will
be administered by the same party, so concerns about physical attacks on the TPM are not relevant, as any attackers with physical access would also have access to the disk.

In this work, we make the following contributions:

- We present the Firma architecture, which demonstrates how to perform secure boot of a VMM using storage as a root of trust. We show that intelligent disks may be used to measure the state of a host before allowing the hypervisor and privileged VMs to be loaded, providing guarantees of the VMM’s integrity state.

- We detail the management tasks necessary to support the Firma architecture, showing that new systems can be installed and configured for secure boot in a small number of automated steps.

- We develop a prototype of Firma using commodity hardware and software. We evaluate the costs of attesting a hypervisor and show that the time to get a VM loaded from a securely-booted VMM requires an overhead of slightly over one second.

### 6.2 Design of Firma

Firma is designed to protect on-disk storage by determining whether a host system is trusted to run an operating system, a process performed by measuring the state of the host from before boot time until the OS kernel is loaded from the disk. Firma can be used to protect a single system, by loading the OS kernel into the root partition of the disk. We focus on a use case of providing a trusted VMM, given the increased demands for system integrity necessitated by managing multiple simultaneously-executing OSes. The
high-integrity system resulting from Firma facilitates trust that a VMM’s policies (e.g., full isolation vs sharing) are initialized to a good state. Figure 6.1 provides an overview of the components that comprise Firma. These are used to address the following design goals:

1. **Secure boot from storage:** Firma provides a staged secure boot mechanism that allows the disk to assess the state of the host system at set points of the boot process, such that if the BIOS or boot loader is compromised, it will be detected and no further boot processes will be allowed. This allows the host to provide trust guarantees and support services such as a trusted hypervisor, whose policy can be guaranteed by the system. Unlike AEGIS or use of secure coprocessors such as the IBM 4758/4764, we do not require secure hardware beyond a disk augmented with functionality that is currently feasible.

2. **Continuous enforcement of system state:** Beyond load-time guarantees, Firma provides a means for the system to attest its state to the disk in a periodic manner. This architecture supports measurement of binaries loaded from the disk’s root partition, but is general in scope and has the ability to support runtime integrity attestations.

3. **Ease of Management and Usability:** Once Firma is set up, there is nothing that needs to be done by the user in order to boot the system. Management of the disk may be performed by a user communicating directly with the device, so as to bypass the host in the event that it is in a bad state. We cannot necessarily rely on a secure channel that has a path through a potentially malicious host, as even if the host was unable to affect the confidentiality or integrity of data emanating from the disk, it could still arbitrarily drop the data. As a result, we consider that communication with the disk can involve plugging the disk directly into an administrative interface, or through the use of an administrative hardware token that can be physically carried by the administrator.

Firma is a protection system consisting of a disk augmented with a processor for providing cryptographic services (such as those available in an FDE drive [162]) and non-volatile memory (as found in a hybrid hard disk [21]) for security-critical keys and policy metadata, combined with a host with a mechanism for providing attestations (e.g., a TPM) to the disk. The disk and host communicate over a management channel out of band to regular requests to the disk. The disk contains a *root partition* that is static,
and only released to the host once its good integrity state has been established. This partition may contain the kernel and binaries for a single operating system, and Firma is fully usable in this configuration. We consider a primary use case, however, of installing a virtual machine monitor onto the disk’s root partition, allowing it to provide a trusted base for other virtual machines to be executed from the disk.

As with the SwitchBlade design presented in the last chapter, we consider in our threat model an adversary capable of arbitrarily and completely subverting a host operating system. The OSes running on the virtual machines from the disk are outside of the purview of what can be protected at the storage level in our design, as we do not consider protections inside the OS. Other solutions, previously discussed, are possible for protecting the OS at the storage level. Beyond the threats we considered with SwitchBlade, we consider physical attacks against the host system’s TPM to be outside of our protection domain. Delivering policy securely to the drive is possible in numerous ways. With a TCG trusted peripheral disk, policy can be securely communicated using the special security commands available for SCSI and ATA. These have yet to be deployed in any disks, however, and are not currently present in the Linux implementation of either of these protocols. One method of delivering policy that we describe in our implementation of Firma (see Section 6.4) is the use of a physical token. We assume that these tokens are trustworthy, and describe elsewhere how tokens can be protected.

6.3 Management and Operation

The basis for trusting Firma’s VMM is predicated on its installation in a trusted manner and the ability to validate the integrity state of the host system. This section describes how a trusted installation process, pairing with the host, runtime operation may be used to provide these guarantees.

6.3.1 Installation

The installation of software necessary for Firma comprises installation of the VMM and kernel to the root partition of the disk. This process may be performed by the drive manufacturer at the time of fabrication, or by the user installing or updating the base software. In either case, the process requires use of a trusted installer. We use a root of trust installer (ROTI) [209], which establishes a system whose integrity can be traced back to the installation media.

The system performing the installation must contain a trusted platform module
The TPM is a tamper-evident chip capable of performing some cryptographic operations, as well as providing non-volatile storage for root keys that it generates. Every TPM contains an endorsement key (EK), a 2048-bit RSA public/private key pair created when the chip is manufactured. As we describe below, this provides us with a basis for establishing the unique identity of the TPM, which is essential to being able to verify 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 have been subverted at this time.1 As described in more detail below, the system’s core root of trust for measurement (CRTM), which contains the boot block code for the BIOS, provides a self-measurement to attest this good state.

2. An administrator authorizes the disk to accept writes to the root partition, and the VMM (and if necessary, supporting OS for privileged domains, i.e., dom0 for the Xen hypervisor) is installed to the root partition off the custom installer. Every file in the root partition is hashed, and the list of hashes is kept in a file that is sealed (i.e., encrypted) by the installing system’s TPM. This process links the installing TPM with the installed code and the filesystem hashes.

The interface to the disk, and authentication mechanisms for the user, are defined by the device manufacturer; our prototype, described in more detail in Section 6.4, uses USB tokens inserted into the disk to deliver these capabilities (i.e., administrator access). Other solutions, such as the use of keypads or direct interfacing with a trusted console are also possible. Note that if the installing system is different from the host system (as would be the case if installation was performed by the manufacturer, for example), the CRTM measurement and the TPM key used to seal the hash list would be needed to verify the installation. Note that the TPM’s EK is not permitted off the module, so an encryption key is generated by the TPM for this operation. This encryption key, along with the CRTM measurement, can be supplied in an out-of-band manner (e.g., including the key on a slip of paper along with the disk or including it on a token), and specified directly to the disk through its administrative interface.

---

1 This restriction is not necessary after the installation has occurred, as malicious changes to the system state will be measured by the CRTM.
6.3.2 Deployment to the Host

In order to assure that we can attest to the integrity state of the host system, we need to pair it with the disk; in other words, we need to establish a method of establishing a unique binding between the host and the disk. We assume that the host system contains a TPM.

When a host system is first installed, it can be considered to be in a “greenfield” state, and we make the assumption that there is no pre-existing malware or subverted hardware on the system at this stage. In a similar fashion to the installed system, we assume the ability to retrieve the CRTM measurement in an offline manner if the integrity of the hardware is a concern. Our first step is to retrieve information to identify the TPM in the host machine. While the EK is unique to the TPM, there are privacy concerns with exposing it, and it cannot be used to perform signatures [243]. Instead, an attestation identity key (AIK) public/private key pair is generated as an alias for the EK, and strictly used for signatures. However, while the EK is kept in the TPM’s non-volatile memory, the AIK is stored in volatile memory. Therefore, for it to be persistent across host system boots, we must store both the public and private AIK on the disk. We cannot reveal the private AIK; fortunately, the TPM provides another persistent key pair, the storage root key (SRK), used for encrypting keys stored outside the TPM. Thus, the SRK encrypts the private AIK before it is sent to the disk. Formally, the set of operations occurs as follows. Given a host’s TPM $H$ and a disk $D$, the following protocol flow describes the initial pairing of the host to the disk and the initial boot:

**Pairing**

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

**Boot**

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 prior to the boot process, while the subsequent measurements occur after the system has booted. Through the administrative interface to the disk, an administrator sends a command to allow the disk to operate without secure boot constraints for the initial system boot. This is done in order to provide a list of measurements that the disk can use to verify the system’s integrity state. 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, (d) the VMM, and (e) the OS running on the VMM. We use the Linux Integrity Measurement Architecture (IMA) [244] as a framework for supporting attestation. 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 $\text{Sign}(PCR, N)_{AIK^{-}}$, 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 in the attestation. At this point, the host must be restarted; this is a one-time operation.

The above process could also involve the setup of a session key associated with a secure channel between the disk and host. For example, the IKEv2-SCSI protocol [245] provides methods of establishing an IKE security association between the disk and the host. As there are many methods of performing such actions, we do not further elaborate on them here.

6.3.3 Regular Operation

On subsequent boots, the disk will revert to secure-boot mode. It will request an attestation prior to the VMM being loaded and will not allow the load to occur unless the attested integrity state matches the one obtained during the pairing. Similarly, after the VMM is loaded, a system measurement and subsequent quote is required before a new VM may be loaded from the VMM. If the check fails, further access to the disk will not be permitted.

When the system has booted to the kernel, it can continue to receive attestations of the system integrity state. While our current prototype enforces integrity measurement with IMA, it is possible to use any mechanism, including runtime execution monitors such as Patagonix [246]. Our solution defines a framework for any monitoring system to be able to enforce access to the disk based on system state.

We have two choices for placing monitors in the system. The first is having a runtime
monitor within the VMM itself, where it checks for violations of system integrity. The IMA approach that we currently use is an example of this. This approach is simple to implement but it has the disadvantage of being potentially vulnerable to attack if the VMM was ever to be compromised. The other location for a monitor could be within a separate privilege-separated virtual machine that is highly restricted by the VMM policy in terms of its read and write access. A thin OS could be running in this VM, in a manner similar to the management VM in Terra.

6.3.4 Error Handling

Fundamentally, our approach of measuring data before releasing access allowing access, while necessary for a secure boot mechanism, makes recovering from errors problematic. AEGIS addressed this issue by specifying that upon a failure, either a secondary ROM (which mirrors the address space of all internal components) is consulted, or a recovery kernel from ROM kept within the AEGIS board is booted from, which contacts a “trusted” host over a network. Establishing this procedure for the disk presents many challenges. Disks and file systems provide a number of solutions to this issue. Journaling file systems such as ext3 and JFS provide a means for recovering updates which may have resided in the operating system buffer cache at the time of access to storage was revoked. Similarly, in the event of a compromise, versioning file systems [247] and storage devices [126] provide a means for rolling back changes to a pre-compromise state. If disk access must be maintained, copy on write schemes [248,249] may be used instead of simply denying write access altogether. This will preserve the integrity of the data in storage while preventing a host compromise from denying disk access to critical applications.

Random environmental errors are well-protected by robust error correcting codes such as Reed Solomon preventing bit errors on hard disks. Concern about larger-scale individual disk failures can be mitigated by using RAID storage systems. In a RAID configuration verification logic may be performed by the RAID controller, and the root partition may be striped amongst multiple disks but present a single interface to outside the disk.

Administrative access may be appropriate under a failure condition, particularly one that results from a monitoring or attestation error. A log of events that is accessible to the administrator can provide information on the nature of the error that occurred, and can be further investigated to determine whether it was the result of natural phenomena, component failure, or malicious tampering.
6.4 Prototype Implementation

The Firma prototype consists of a host side on which the secure VMM and VMs run and the storage root of trust that enforces secure boot via access control, and provides continuous enforcement. We used a similar architecture for designing Firma as we did with SwitchBlade, deploying a storage root of trust using the Linksys NSLU2 [204] network attached storage device running the OpenSlug Linux distribution. All communication between the host and storage occurs via the ATA over Ethernet (AoE) protocol, which extends the ATA disk command set to storage devices sitting on local area networks. As this setup is substantially similar to that used with SwitchBlade, we direct the reader to Section 5.4 for details about AoE setup and use of the vblade daemon. We now detail how the Firma prototype provides a secure boot capability with continuous integrity enforcement.

6.4.1 Secure Boot

To provide the necessary mechanism for secure boot, the storage root of trust needs two things, a means of authenticating the host and a criteria for trusted host configurations. For management purposes, a secure channel to the disk is also needed to deliver these to the disk. For this purpose, we use a USB token which carries the EKs associated with all TPMs on hosts that might access the drive and the set of approved binary measurements for software that may run on those drives. The EKs serve to authenticate hosts and verify that they are using authentic TPMs. The measurement lists may be used both for verifying the integrity of the trusted VMM at boot time and of applications loaded in VMs during continuous monitoring.

As several stages of the secure boot sequence require binary measurements to be delivered to the disk, we made a simple extension to the AoE protocol to allow for binary strings to be delivered to the disk either as part of another block request or as an independent message.

The secure boot process is carried out by the prototype as follows. When the system begins booting, the BIOS is hashed and this hash is extended into a PCR on the TPM. The BIOS then measures and loads the next stage which in our prototype is the PXELINUX bootloader (part of the SYSLINUX project)\(^2\). The PXE bootloader loads each binary needed by the system, including the hypervisor, kernel, and initial RAM disk image. These are measured and the hashes extended into a PCR on the TPM. When the system begins booting, the BIOS is hashed and this hash is extended into a PCR on the TPM.

---

\(^2\)The use of PXELINUX is needed only by our implementation, in an actual implementation a traditional disk based bootloader like GRUB or LILO would be used.
components are being loaded, AoE is used to communicate with the disk, which sends the signed measurement list in its current form to the disk. If at any stage in the boot process, incorrect binaries are detected, the `vblade` instance is stopped, thus preventing access to the disk by malicious binaries.

### 6.4.2 Continuous Enforcement

The above described disk protocol extensions may be used to extend the secure boot mechanism in storage to a continuous monitor that allows re-attestations of up to date system state by the VMM. It is then the job of the disk to verify these measurements against its own integrity criteria as delivered by the physical USB token. The AoE driver on the host take a parameter specifying the number of seconds between re-attestations. It then sends an up to date set of measurements from the host TPM to the disk at the specified time interval.

All modifications needed for re-attestations were made to the linux AoE driver. TPM measurements are cached in the Xen dom0 kernel to alleviate the overhead of obtaining a TPM quote on the host. If time-based re-attestations are made, we use a kernel timer to schedule the construction and transmission of the attestation command.

In the event that the disk receives an untrusted measurement either during secure boot or continuous monitoring, a mechanism is needed to deny the hosts access to storage and, if desired, provide a secure environment from which the system may be recovered. All attestations arriving at the storage root of trust are checked against the disk’s integrity criteria by our utility, `VerifyQuote`. This utility reads the public key matched to the private key used to sign the TPM quote, along with the TPM quote itself. `VerifyQuote` then validates the signed quote using standard RSA signature validation. If the verification process fails, the instance of `vblade` exporting the disk’s root partition is killed, preventing any further access.

If a trusted environment is needed for recovery of either the system or data, a failed attestation may also lead to the launching of a new `vblade` instance, which exports a disk partition containing a minimal trusted environment that may be used to inspect the contents of storage. We also modified `vblade` to maintain an audit log of failed attestations, which could be potentially useful in determining when the system entered an untrusted state.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Host System</strong></td>
<td></td>
</tr>
<tr>
<td>Measure boot binaries</td>
<td>24.9388 (24.9113, 24.9662)</td>
</tr>
<tr>
<td>TPM Extend</td>
<td>39.9934 (39.9887, 39.998)</td>
</tr>
<tr>
<td>TPM Quote</td>
<td>880.037 (880.03, 880.045)</td>
</tr>
<tr>
<td><strong>Disk Firmware</strong></td>
<td></td>
</tr>
<tr>
<td>Verify quote</td>
<td>70.0msec.</td>
</tr>
</tbody>
</table>

Table 6.1. Microbenchmarks of the host operations needed to support Firma. For each measurement, 95% confidence intervals are shown in parentheses. All numbers are shown in milliseconds unless otherwise noted.

6.5 Evaluation

In this section, we evaluate the performance of Firma as described in preceding sections. We examine performance of the host and of the disk during the various operations each performs under the Firma architecture. All experiments were performed on a Dell Optiplex 745 with a dual-core Intel Core 2 Duo 6300 at 1.86GHz, 4GB of RAM, and an internal 160GB SATA drive. The prototype also includes a Linksys NSLU2 (“Slug”) connected to the host over a 100Mbps Ethernet interface. Attached to the Slug is a 160GB Seagate FreeAgent Go external USB hard disk. Also attached to the Slug was the token, a 2GB Lexar Firefly USB drive. The base operating system was Debian 5.0 (“Lenny”) running a modified 2.6.24 Linux kernel and a stock Xen 3.2.1 hypervisor. Table 6.1 shows the time for each operation in the Firma architecture.

When the host system begins the boot process, it begins by loading a bootloader which in the prototype is the PXE bootloader. In order to gain access to the next stage boot files like the hypervisor, kernel, and initial RAM disk, the host sends a TPM quote to the disk. This TPM quote takes an average of 880ms to generate on the host, which is not atypical for current TPMs. This quote is then transmitted to the firmware, where it is validated. This validation takes an average of 70 milliseconds to complete. After the quote from the host is validated, the boot files are loaded. Before each file is loaded, it is measured by the bootloader, and these measurements extended into the TPM. The TPM extend operation takes an average of 40 milliseconds to complete. This extend operation will happen three times during the boot process, once each for the kernel, initial RAM disk image, and the hypervisor. The overall additional boot delay is $t_{over} = 880ms + 70ms + (40ms \times 3) = 1070ms$, or just over 1 second, most of which is attributed to the TPM quote operation. This additional overhead is not unreasonable even on systems with sub-20 second boot times. This can be further reduced by prefetching the boot files
while the quote generation and validation is occurring, thus eliminating the additional overhead.

During normal system boot, the disk is accessed in order to load the boot time binaries mentioned above. A standard SATA disk in our setup has a measured throughput of 70MB/s. A standard bootloader, in this case GRUB, has a total size of about 130K, while the boot files loaded initially have a total size of about 8MB. The hypervisor is 370KB, the kernel is 1.7MB and the initial RAM disk image is 6.2MB. All of the boot files are compressed to increase transfer speed. In addition to these boot files, the system must read several other binaries and their configuration files from the disk during the boot process. For a stock Debian Xen system, these transfers total 46MB with a boot time of 93 seconds. This is higher than some other hypervisor systems, such as VMware ESXi which has a disk footprint of only 32MB. This footprint includes everything needed to boot the system. In our prototype system, measuring the entire VMware ESXi disk footprint takes an average of 9 seconds.

In order to understand the impact of Firma’s continuous enforcement on I/O intensive workloads, we benchmark our prototype under a range of re-attestation time intervals. Figure 6.2 shows the throughput of our Firma prototype as the frequency of re-attestation is varied from 1 second to 60 seconds. The lower bound for time frequency based attestations is limited by the TPM, which can offer a new quote every 880 ms in our system. The horizontal lines in Figure 6.2 show the mean throughput of a base Firma system (i.e. without re-attestation) as well as the 95% confidence intervals. In all cases, the system that is re-attesting shows little to no throughput degradation. The base Firma system showed an average throughput of 1.839MB/s. The system with re-attestation showed at most a 2% overhead, well within range of experimental error.

![Figure 6.2. Throughput of Firma prototype when using time-based reattestations](image-url)
6.6 Discussion

6.6.1 Dynamic Root of Trust for Measurement

Our design has centered around the use what the TCG refers to as a Static Root of Trust for Measurement (SRTM). The SRTM refers to the chain of trust rooted in the hardware (the motherboard’s CRTM), which measures each phase of the boot process up to the OS. Since the measurement gathering code is rooted in hardware, the correctness of subsequent measurements not based in hardware are subject to the integrity of this SRTM. Due to the variability of BIOS code and underlying hardware on the average compute, no single well-known good measurement value can be known by a Firma disk a priori to pairing. As a result, the disk must gather measurements of this early part of the boot process when pairing.

This has several drawbacks. When the owner of the host wants to make changes to the host’s BIOS or changes hardware components, the disk must be re-paired to obtain the new set of early boot measurements. Another issue is the initial trustworthiness of the BIOS and hardware. While we assume the greenfield state of the host during pairing to be trusted, the potential for undetected errors or exploits still exists.

To obviate these issues, a dynamic root of trust for measurement (DRTM) can be used. A DRTM is instead rooted in software and does not depend on the system’s underlying hardware configuration. The Open Secure Loader (OSLO) [79] is a bootloader that implements such a DRTM. For AMD processors supporting the Secure Virtual Machine (SVM) extensions, and in particular the instruction SKINIT (which disables direct memory access to physical memory within a secure loader block), OSLO can use SKINIT to measure and execute a bootloader after clearing the processor pipeline and memory. This effectively allows the system to start from a clean slate without depending on the code loaded previously. Additionally, the DRTM bootloader measurement is stored in a PCR specifically designed to be used only via the SKINIT command.

What this means for us is that a Firma disk can pair with a host without having to gather SRTM measurements such as the BIOS. As a result, we can pre-compute the correct measurement lists directly on the disk, as we can hash them to determine what their values should be. Additionally, we can avoid making the assumption that the host is not running malicious firmware during pairing.
6.6.2 Lazy Attestation

Firma incurs performance overheads both at the host while generating binary measurements and at the disk while verifying measurements. These overheads may be addressed using lazy attestation and verification. In lazy attestation, a trusted VMM may cache integrity measurements until a TPM extend operation. For example, in our implementation, measurements were cached in a Xen dom0 kernel which delivered the attestations to the disk at intervals. Using lazy evaluation a disk may hide the performance cost of integrity verification behind disk seek times. This is done by the disk processor which first initiates a disk seek and then performs integrity verifications for any attestation commands from the host. As disk response times dominate the disk processor time needed to check a quote, the overhead of integrity verification is nullified.

6.6.3 Beyond Unique Pairing

One advantage the Firma approach has over an on host secure boot mechanism is that a Firma disk can be used to secure boot on multiple hosts. If implemented on a portable disk, a Firma disk could be modified to pair with multiple host machines. Then when a user later plugs the disk into a previously pairing host machine, Firma could use the host specific information obtained from the pairing to expose the associated AIK to the host and to verify its attestations.

As we discovered, there are fundamentally different ways in which users approach portable storage, and while Firma provides an excellent means of providing a secure boot mechanism, the security guarantees we look for from a host system in the case of portable storage are different. This investigation of ensuring the security of portable storage is the focus of the next chapter.
We continue our examination of using trusted hardware on the host as a means of validating integrity and attesting that good state to the storage device. We turn our focus to portable storage, where the need for attesting hosts is arguably even greater than with fixed disks, due to the different usage model such devices entail.

Recent advances in materials and memory systems have irreversibly changed the portable 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 [250]. Such devices change the way we store our data; single keychain devices can simultaneously hold, for example, decades of personal email, millions of documents, dozens of movies, thousands of songs, and many virtual machine images. These devices are incredibly convenient, as we can effortlessly carry the artifacts of our digital lives with us 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 [251] 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 [252] and Agent.bz [253] 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 [1], as discussed in Chapter 1. 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.

Solutions that consider this problem from the standpoint of portable storage have been minimal to date. Solutions such as full-disk encryption [222] and Microsoft’s BitLocker to Go [254] 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 [146] (for fixed disks) uses a trusted platform module (TPM) [243] 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 a viable solution for mobile storage, as data is bound to the single physical host. Additionally, BitLocker does not assess the integrity of the host at any time after the data is mounted. In another effort, the Trusted Computing Group (TCG) has considered methods of authenticating storage to the host through the Opal protocol [151] 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, 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. These exchanges are repeated periodically to ensure the integrity state of the system remains stable. 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.

As with Firma, our design uses the commodity Trusted Platform Module (TPM)

---

1Named for the Book of Kells, which is traceable to having resided at the Abbey of Kells, Ireland, in the 12th century thanks to information on land charters written into it.
found in the majority of modern computers as our source for trusted hardware, 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., genuinity [255]) 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 [19,120,256] to encompass a system that actively vets the devices that make use of it. In so doing, we 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 [257], and prove that the storage can only be accessed by hosts whose integrity state is valid (within a security parameter $\Delta_t$).

- 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.

### 7.1 Overview

Figure 7.1 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 process 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 host validation 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.

### 7.1.1 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:

#### 7.1.1.0.1 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. In this mode, the user does not choose whether to be able to use the public versus trusted storage; all decisions are dependent on the storage policy defined within the device.

#### 7.1.1.0.2 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 programs that the user can run is made available. One is a program that prompts 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, but it is then incumbent upon the user to ensure that information be properly managed on the trusted host. 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. It is also possible for the user to have both partitions opened simultaneously if allowed by the device policy. This allows for situations such as having created a presentation on a secure workstation and desiring to place it in the public space, such that another laptop could be used to access the information. This is an example of explicit declassification being possible by the user and moves the security model from mandatory, as in transparent mode, to discretionary. Because this model entails additional possible risks, the user’s role in data protection is magnified.

Opening the trusted and public partitions can be subject to further policy as well, such as the ability to only open public partitions on specific devices, or by certain users. Such a policy may be appropriate if there is a fear of reading or executing potentially compromised data that could have been placed on the public data partition of the device, and having sensitive hosts reading this data.

### 7.1.2 Threat Model

As before, 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 [258] 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 [259] contains a secure area for encryption keys that is sealed with epoxy glue. 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 the Firma secure boot mechanism described in the previous chapter is a means of ensuring that the system has been booted into a high-integrity state, protecting data from being subverted by malware. Integrity-based solutions exist that protect the host’s internal
storage (hard disks), including storage-based intrusion detection [19] and rootkit-resistant disks (described in Chapter 4). For reasons described in our trust model in Chapter 1, we declare physical attacks against the host’s TPM outside the scope of this work. The use of the TPM is 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 [151] includes provisions for trusted peripherals, addressing the issue by requiring devices to contain TPMs. Software-based attestation mechanisms such as SWATT [260], which does not require additional trusted hardware, may also be used.

7.2 Design and Implementation

We now turn to our design of the Kells architecture, shown in Figure 7.2, 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. We begin with a short background on USB.

7.2.1 USB: Background

To better understand our design, it is instructive to understand how USB operates, particularly in the context of storage devices. We thus describe the basic operation of the protocol, shown in Figure 7.3.

USB is fundamentally a client-server style protocol with the concept of a host, which
initiates all transactions, and a device, which receives requests from the host and services them. Every USB device is comprised of up to 16 endpoints. These endpoints support four types of possible transfers. Control transfers provide device identification, configuration requirements, and other commands; every USB device supports control transfers and contains endpoint 0, which is dedicated for control transfers. Bulk transfers are used to deliver large amounts of information without any bandwidth or latency guarantees. These two transfer types are supported by all USB devices supporting the mass storage specification. Other transfer types include interrupt transfers, which are used for small amounts of information that is polled for, and isochronous transfers, which provide guarantees for real-time media. We do not discuss the latter two transfer types further.

Devices adhering to the mass storage specification contain a specific subset of functionality. As described above, these devices support bulk transfers, and a very limited subset of control transfer commands. Above these operations in the mass storage stack is the SCSI command layer. The number of SCSI commands that are supported within this stack is also severely restricted, with only seven commands officially specified for support (e.g., READ, WRITE, REQUEST LUN).

Because we designed Kells in a Linux environment, we briefly examine how the USB mass storage stack is implemented in Linux. Note that there are two separate stacks, one for hosts (which initiate communications) and one for devices (which receive commands and perform the requested operations). In Linux, USB devices are called gadgets to disambiguate the term; we use the terms interchangeably. Within the host’s storage stack, a control thread is spawned when a gadget is attached, and a probe() command is called to retrieve the gadget information and set configuration parameters. The gadget is registered as a SCSI controller and requests are handled in a way befitting the attached gadget. For example, if the gadget does not support the Read(6) SCSI command (a 6-byte command block), the stack will rewrite it as an extended read command (Read(10)).

Due to the wide variety of devices that are supported by the Linux USB host stack, and the wide range of implementations, there are provisions for gadgets that do not conform to the USB specification. The file unusual_devs.h contains a repository of devices that do not perform as expected (e.g., they define multiple targets, return mismatched tags, etc.). Specific flags to deal with the unusual behavior are set based on the vendor and product IDs received from the gadget.

The Linux gadget USB mass storage stack is far more limited in its functionality. This

---

2 Other operating systems such as FreeBSD and Windows have their own USB stack implementations, but are similar in design.
stack is meant to be deployed on embedded devices containing a USB device controller. PCs only possess host controllers, preventing them from acting as gadgets. A PC can emulate a gadget device using the Linux Dummy/Loopback USB host and device emulator driver, but this is meant for assisting in driver development rather than deployment, as USB traffic is simulated with it.

### 7.2.2 USB Interface

The challenges that make securing portable storage unique are due in part to the tension between maintaining device compatibility and ensuring device functionality. In the previous section, we described that a USB device is inherently a slave to 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 7.4 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. As we describe later, though, the host will be able to differentiate the device as trusted by examining the vendor and product ID string descriptors provided. The host will set flags accordingly in order to send the correct commands to the device.

While all USB devices support control transfers, their use with mass storage devices is limited to supporting only two message types (Bulk Only Mass Storage Reset and Get Max LUN). Almost all USB mass storage devices perform their operations using only bulk transfers; they operate in bulk-only, or BBB mode. The only devices that do not conform to this operation set are certain floppy drives that also support control and interrupt transfers; these operate in Control/Bulk/Interrupt, or CBI mode. However, it is beneficial for us to use control transfers for the host-device channel because these receive guaranteed bandwidth: up to 20% of bus bandwidth is reserved for high-speed (i.e., USB 2.0) devices, while by comparison, bulk transfers are best-effort. Thus, using control transfers ensures that information to and from the host is received as quickly as possible. In addition, this differentiates the operation of trusted hosts while requiring no changes to regular device operation. 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 trust protocols and trusted storage.

If the host recognizes the device as trusted, it will send an Accept Device-Specific Command (ADSC), which will only be issued if the host considers the gadget to be a device using CBI mode rather than BBB. A control transfer is a three-stage transaction, consisting of a setup, data and acknowledgement phase. The setup phase allows the host to send a code 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.

### 7.2.3 Designing the Storage Device

The Kells device requires the ability to perform policy decisions that are independent of the host. As a result, there is logic that needs to be executed on these devices not found on regular portable storage devices. Notably, the device requires a way 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. Currently no support for “hotplug” operations are implemented from the gadget’s point of view; therefore, we implemented some additional logic to be passed to the daemon when a connection is first sensed. USB devices are capable of determining when they are plugged into a host, as they detect voltage on the power supply line from the bus and switch a pull-up resistor. This also notifies the host that the device has been plugged in.

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, to the best of our knowledge, has not been currently allocated by the USB Forum as of February 2010.\footnote{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.}

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 \texttt{insmod()} command to link the correct backing store with the gadget driver. If both trusted and public partitions are to be made available then a separate interface is exposed for it. Each partition is presented to the host as a separate device, so the trusted and public partitions appear as separate “disks”. This means that a new device descriptor will be asked for by the host and the same protocol detailed in Figure 7.4 will be followed. However, the host should not issue another ADSC command to the device for the public partition, as it should realize
that the device is already mounted in a trusted state and the next partition will be public. Since we cannot assume that the host will necessarily do this, the device will explicitly respond with a status code in response to the ADSC that indicates the partition is public, and will present the partition ignoring any further non-standard control transfers.

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 and their credentials, which are supplied directly to the device through methods such as biometrics. Many portable storage devices include built-in fingerprint scanners, such as the Index Security BioStik [261] and the SanDisk Cruzer Profile [262]. Policy can then be configured to allow or disallow the device to be plugged into specific machines.

### 7.2.4 Modifications to Host

A host must be capable of recognizing that the Kells device is trusted and that sending information to it differs from a standard USB mass storage transaction. One of our goals was to require minimal changes to the host for operation, but because we are working directly at the USB layer itself, some changes are necessary to the USB driver. In our discussion of the Linux USB mass storage stack in Section 7.2.1, we described that hosts can set flags for devices that do not operate in strict accordance with the USB specification, as checked by the `unusual_devs.h` device repository. For our system to work, we define a flag `IS_TRUSTED` that changes the control flow of USB operations. In addition, we define the device to be capable of CBI mode operations, even though it has been advertised as performing bulk-only 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. While it may be desirable to reduce the number of additional programs that need to run, a major design consideration in the kernel is separating policy from mechanism. The attestation daemon both retrieves boot-time attestations using the Linux Integrity Measurement Architecture (IMA) [244] and can act as an interface to any runtime monitoring systems on the host (we discuss this further in Section 7.3.1). It can also provide an interface for receiving third-party updates (see Section 7.3.2).
7.3 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. Attestation of host integrity happens in a manner substantially similar to Firma; we direct the interested reader to Section 6.3.

We note that the protocol employed by Firma 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 [79]. 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. Datta et al. [257] showed that if DRTM and SRTM are supported on the same device then the SRTM is vulnerable to a code modification attack staged during the late launch process. However, DRTM has also been shown to 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 [263]. 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, which obviates the problem of identifying which TPM the device is communicating with; hence, we do not require a visual channel as provided with Seeing-is-Believing [264]. The cuckoo attack described by Parno [265] 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.

7.3.1 Managing Runtime Integrity Attestations

To perform authentication of the host, it is incumbent upon the device to be able to compare the attestations received with a known set of good values. As a result, a portion of non-volatile memory within Kells is provided for recording this information, which includes a unique identity for the host (such as the public portion of the AIK keypair), the list of measurements that are associated with the host (for attestation verification), and policy-specific information, such as whether the host being attached to should allow
As previously described, 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. For example, if the host system uses the Patagonix system for detecting covertly-executing rootkits [246], it could provide a response to the disk on being queried as to whether the system has any hidden binaries currently executing. A software-based integrity monitor such as Pioneer [266] and solutions that use the TPM such as BIND [267] may also be appropriate for runtime monitoring. 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 $\lambda$ before the attestation period $\Delta_t$ expires, in case the host neglects to send the new attestation. Every bulk transfer (i.e., reads and writes) involves the issuance of a status return code. By providing this code (in the $\text{bCSWStatus}$ field of the command block returned), the device can notify the host that an attestation is required. At this point, a command transfer is sent from the host to the device to begin the runtime verification process.

Algorithms 2 and 3 describe the write behavior on the device. We have implemented a buffer for writes that we term a quarantined 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. When an attestation arrives, every write that has been buffered to that time is written to storage. However, if an attestation does not arrive and the buffer fills, further writes to the device are denied. This could potentially happen if the host is heavily loaded. 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 4 describes the semantics of the read operation. Reads occur as normal unless an attestation has not been received within time $\Delta_t$. 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).

### 7.3.2 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 [268]. 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.

7.4 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 [257]. 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 7.3.1 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 $\Delta_t$ 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 $\Delta_t$ as read.

7.4.1 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
code real programs. Any property that is provable using the $LS^2$ proof system 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 ($n$), as well as pairs
of these to represent data structures for attestations and block requests. The expressions
used for encoding values in Kells is shown in Table 7.1.

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 ($H$) by the Kells device ($D$).

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. Send($D$, req) @ $t$ holds if the thread $D$ send the results of
the request req to the host at time $t$. See the predicates in Table 7.1. 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 GoodState(H, (t, treq, (l, n)), (tatt, 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.

7.4.2 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 7.3.1 using the above de-
scribed language. This includes a description about assumptions concerning the char-
acteristics 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 7.5. We then formally state the two properties and show that they hold
for the encoded versions of Kells operations.

7.4.2.1 Encoding Kells Operation

The encoding of the read operation is shown in Figure 7.6 and the write operation in
Figure 7.7. 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 make one assumption about the underlying hardware of the Kells device,
namely, that it has a hardware timer that can repeatedly call the program that performs
commits from the write request queue (KCommit in Figure 7.7).
KRead: 1. att = read D.RAM.att-loc 
2. (t, req) = receive 
3. n' = sread req,att 
4. send n'

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 7.6. The encoding of the Kells read operation

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 is needed to prevent a dequeued request from being lost by KCommit in the event that a fresh attestation has not arrived after the request has been dequeued. This is needed as existing statements prevent us from simply enqueuing the request again while waiting for a fresh attestation. 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 7.5. Both of these operations take a block I/O request and an attestation as arguments. A block request \( (t, t_{\text{req}}, (l, n)) \) from the host consists of the program counter at arrival time \( t \), an absolute arrival time \( t_{\text{req}} \) and a sector offset and data pair.

The encoded version of the Kells read program (KRead) is shown in Figure 7.6. We assume the existence of a running thread that is responsible for requesting new attestations from the host at a rate of \( \Delta t \) and placing the most recent attestation at \( D.RAM.att-loc \). 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 7.7. 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 \( \rho \) 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 \( LS^2 \).
(SEC) ⊢ ∀ (t_{req}, (l,n)), (t_{att}, sig), t \text{ s.t.} (t_{req}, (l,n)) = \text{Recv}(D) @ t \\
\land (t_{att}, sig) = \text{Recv}(D) \\
\land e = \text{SRead}(D, (t, t_{req}, (l,n)), (t_{att}, sig)) \\
\therefore \text{GoodState} (H, (t, t_{req}, (l,n)), (t_{att}, sig)) \\

(INT) ⊢ ∀ (t, t_{req}, (l,n)) \text{ s.t.} (t, t_{req}, (l,n)) = \text{Peek}(D) \\
\land (t_{att}, sig) = \text{Recv}(D) \\
\land SWrite(D, (t, t_{req}, (l,n)), (t_{att}, sig)) \\
\therefore \text{GoodState} (H, (t, t_{req}, (l,n)), (t_{att}, sig)) \\

Figure 7.8. The formal definition of the two Kells security properties.

7.4.2.2 Proof of Security Properties

The (SEC) and (INT) properties may be stated formally as shown in figure 7.8. Both properties ultimately make an assertion about the state of a host at the time it is performing I/O using the Kells device. \text{GoodState} 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, \Delta_t represents the length of time during which an attestation is considered fresh past its reception. Thus, \text{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 \text{Fresh}(t, t_{req}, t_{att}) to state that an attestation is fresh w.r.t. a given request. The attestation is received at wall clock time t_{att} and the request at time t_{req}. Attestations are received at the \( t^{th} \) clock tick, as obtained using the program counter \( \rho \). As described above, Kells will check if a previous attestation is still within the freshness parameter \( \Delta_t \) before stalling the read or queueing the write. This is the first case in the definition of \text{Fresh} below. If a request is stalled, the next attestation received is verified before satisfying the request. In this case, a \text{Reset} must not occur between the receipt of the request and the check of the next attestation.

Detecting such resets is a key motivation in choosing a \( \Delta_t \). That is, a system should not be able to enter a bad state, submit a block request, reset (clearing the PCR value) and reinitialize to the previous state in time less than \( \Delta_t \). We show in section 6.5 that \( \Delta_t \) values between one and two seconds have minimal performance impact.

\text{GoodState}(H, (t, t_{req}, (l,n)), (t_{att}, sig)) = \text{Fresh}(t, t_{req}, t_{att}) = \\
\text{Fresh}(t, t_{req}, t_{att}) \\
\land v = \text{Verify}((t_{att}, sig), AIK(H)) \\
\land \text{Match}(v, \text{criteria}) \\
\land (t_{att} < t_{req} \land t_{req} - t_{att} < \Delta_t) \\
\lor (t_{req} < t_{att} \land \neg \text{Reset}(H) \text{ on } [t, \rho]) \\
\lor \text{Match}(v, \text{criteria})
Theorem 1. \texttt{KRead} maintains the (SEC) security property.

\textit{Proof.}

Assume that the following holds for an arbitrary program trace.

\[ \exists (t_{\text{req}}, (l, n)), (t_{\text{att}}, \text{sig}), t, e \text{ s.t.} \]
\[ (t_{\text{req}}, (l, n)) = \text{Recv}(D) \oplus t \]
\[ \land (t_{\text{att}}, \text{sig}) = \text{Recv}(D) \]
\[ \land e = \text{SRead}((t, t_{\text{req}}, (l, n)), (t_{\text{att}}, \text{sig})) \]

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

Theorem 2. \texttt{KCommit} maintains the (INT) security property.

\textit{Proof.}

Assume that the following holds for an arbitrary program trace.

\[ \exists (t, t_{\text{req}}, (l, n)), (t_{\text{att}}, \text{sig}) \text{ s.t.} \]
\[ (t, t_{\text{req}}, (l, n)) = \text{Peek}(D) \]
\[ \land (t_{\text{att}}, \text{sig}) = \text{Recv}(D) \]
\[ \land \text{SWrite}((t, t_{\text{req}}, (l, n)), (t_{\text{att}}, \text{sig})) \]

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

It is worth noting that our model makes some sacrifices of secrecy in order to maintain consistency. Because we measure the host after information is released to be read, in the space of time between the time that the read request is processed by the Kells device and the time when the next attestation occurs (i.e., \( \delta_t - t_{\text{read}} \)), it is possible that the host is no longer in a good state. As described above, this window can be made quite small.
with minimal performance overhead; however, the possibility to exploit it still exists. A potential option to deal with this problem is to use the quarantine buffer to also cache read requests, such that information is only released to the host after an attestation occurs. Under such a setup, the host could be waiting as long as $\delta_t$ after it issues a read request until the request is serviced by the Kells device. Another situation could also arise: say, for example, that the host issued a flurry of read requests to the device starting soon after the previous attestation is given. If these requests are large, not only will the device not begin servicing the requests until after the next attestation arrives, but service time could become a non-trivial factor; thus, the host could potentially go bad after that next attestation and before the queue of read requests is fully serviced. The optimal situation would be to have an attestation period $\delta_t = 0$ but this is infeasible using the TPM and given that measurements must be sent with each request. To fully prevent the specter of releasing information to a host that could potentially be compromised, the only sure-fire way is to ensure that the host itself performs enforcement of its own such that when a runtime monitor perceives a host compromise, it immediately shuts down the USB connection to the Kells device. However, our solution of time-based attestation does provide a reasonable level of protection on its own against releasing information to compromised hosts. We favor protection of the information on the device, so the quarantine buffer provides us with full integrity protections.

7.5 Policy Extensions for MLS

The concept of multiple partitions used with Kells, along with a processor capable of enforcing policy, allows for interesting methods for enforcing more complex policies. For example, consider the public and private partitions available to a user when the Kells device is plugged into a trusted host. Let us rename them LOW and HIGH, respectively. Because a Kells device is cognizant of the requests that it receives, it is possible for it to enforce security policy at the granularity of these partitions, as policy can be embedded on the token and enforced in a manner that is dependent on the host that the token is plugged into.

For example, a particular employee’s workstation in a company may be cleared to level HIGH. As a result, when the Kells drive is plugged into this machine, the policy is activated on the drive as the host is recognized. Under a Bell and LaPadula policy, the simple property will mean that the LOW and HIGH partitions will both be accessible for reading. However, because the workstation is cleared HIGH, a consequence of the
property is that the LOW partition will be set as read-only on the Kells drive, and any attempt to write to this partition will result in failure. Conversely, policy may be configured such that a LOW host is capable of writing to HIGH. The simple property will not allow reading of this partition. In this case, a special partition will be mounted that is write-only, meaning that it will appear empty to the user. Information can be written to this partition, and it will be managed by Kells by placing the blocks in sequential order on the partition, but no indication will be returned to the user as to the outcome of the write operation. To allow return codes would present an opportunity for a covert channel to be available for ferreting information about the drive contents. This is a highly centralized form of information flow control, where a central administrator predetermine what machines are capable of being cleared to what levels.

Note that it is possible to include users in this equation as well; as discussed previously, users can be authenticated to flash drives through mechanisms such as fingerprint scanners. Were these to be employed as part of a Kells device then the on-device policy could be used to control access as a product of the clearance level of both the user and the host that Kells is plugged into.

As described in Section 7.1.1, certain users may possess the ability to have multiple partitions open simultaneously with concurrent read and write accesses. Under an MLS model, this would be allowed exclusively by trusted parties on trusted workstations. These entities would be able to allow endorsement and declassification of information between LOW and HIGH partitions. This is a model that is compatible with the Bell-LaPadula scheme, as users are trusted to make decisions regarding the integrity of data. This must be a manual process performed by a trusted user rather than an automatic mechanism supported by Kells, however, to prevent potential Trojan horses from maliciously performing this declassification.

Note that such a model is a more complex use case than private and public partitions, and indeed, it is possible to set up a Kells device with an arbitrary number of partitions. For the sake of simplicity and clarity, however, our investigation to date has dealt with a strictly bi-level scenario. However, let us consider having potentially multiple categories; that is, multiple public and private partitions, each with their own access privileges. In effect, we have created a lattice. As long as the lattice is encoded to the Kells device with ordering and dominance relationships made explicit, providing read and write access is not difficult. This is a one-time operation as well, so there are no ongoing costs of performing these accesses, apart from ensuring that the read and writes to the given partitions are in compliance with the specified lattice.
Given the fact that a Kells drive is capable of recognizing reads and writes to specific partitions, we are actually capable of enforcing a more dynamic policy. Namely, we can constrain the ability to access particular partitions on the Kells drive based on the first read or write to a particular partition, i.e., we can enforce a Chinese Wall policy [186]. Consider a trusted workstation (and trusted user) who are cleared to HIGH access. As a result, both HIGH and LOW partitions are immediately accessible. Now consider that there are partitions LOW-BIZ, LOW-PERSONAL, and HIGH-BIZ. Policy dictates that the drive can be used to store business information or personal information, but never the twain shall meet. In this case, all partitions are initially accessible to the user. As soon as she writes some data into her LOW-PERSONAL partition, Kells will recognize that this partition is being written, and unmount the other two. Note that because this is dynamic, decisions are made on the basis of writes - all partitions allowed to be exposed by policy will be exposed, but due to the mount process, partitions need to be readable in order for the mount process to succeed. The Kells device does not recognize the difference between reading files and directories, either. A heavyweight solution to deal with this particular case is to allow introspection on requests to determine the number of sectors being read and their location. Based on this information, Kells can make a decision as to whether access beyond mount data is being attempted and policy can be applied accordingly.

7.6 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. The board contains a Texas Instruments 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

---

5Due to extreme supply shortages, we were unable to procure a BeagleBoard or our preferred platform, a small form-factor Gumstix Overo device. Future work will consider how these devices may change our performance characteristics.
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 7.2, 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 $\Delta t$ (the periodicity of the host integrity re-validation) 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 7.2, 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, it is reasonable to expect that the overheads would be negligible in production systems.
7.7 Summary

This chapter 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. We designed Kells to operate over USB, and our implementation shows how such a device can be developed. We use the $LS^2$ logic to prove that storage may only be accessed by hosts in a valid state, through preservation of secrecy and integrity properties. 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.

With the previous chapter, we have shown an approach for using storage as the lynchpin of a system for validating a system’s integrity state. With assistance from commodity hardware, namely, a Trusted Platform Module, and small changes to some host services, we can support a model for allowing the host to be a part of a system’s trusted computing base. The next chapter explores how we can use this new symbiosis to provide finer-grained information flow properties.
Table 7.1. The subset of \(LS^2\) and extensions used to evaluate the Kells security properties.

### Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Use in Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(att = (\tau_{\text{att}}, \sigma))</td>
<td>An attestation consisting of wall clock arrival time (\tau_{\text{att}}), and a signature, (\sigma).</td>
</tr>
<tr>
<td>(req = (t, \tau_{\text{req}}, (l, n)))</td>
<td>A block request consisting of a stored program counter (t), a wall clock time (\tau_{\text{req}}), a disk location (l) and a value (n).</td>
</tr>
</tbody>
</table>

### Language Features (\(^*\) indicates an extension)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Use in Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>send req</td>
<td>Send the result of request (req) from Kells to the host.</td>
</tr>
<tr>
<td>receive</td>
<td>Receive a value from the host.</td>
</tr>
<tr>
<td>proj(_1) (e)</td>
<td>Project the first expression in the pair resulting from (e). proj(_2) (e) projects the second expression.</td>
</tr>
<tr>
<td>(^*)enqueue (req)</td>
<td>Enqueue the request (req) in the Kells request queue.</td>
</tr>
<tr>
<td>(^*)peek</td>
<td>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.</td>
</tr>
<tr>
<td>(^*)dequeue</td>
<td>Dequeue a block request from the Kells request buffer.</td>
</tr>
<tr>
<td>(^*)sread (req, \sigma)</td>
<td>Perform a secure (attested) read.</td>
</tr>
<tr>
<td>(^*)swrite (req, \sigma)</td>
<td>Perform a secure (attested) write.</td>
</tr>
</tbody>
</table>

### Predicates (\(^*\) indicates an extension)

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Use in Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send((D, req) @ t)</td>
<td>The Kells disk controller ((D)) sent the result of request (req) to the host at time (t).</td>
</tr>
<tr>
<td>Recv((D, req) @ t)</td>
<td>The Kells disk controller ((D)) received the request (req) from the host at time (t).</td>
</tr>
<tr>
<td>Reset((H) @ t)</td>
<td>Some thread on the host machine ((H)) restarted the system at time (t).</td>
</tr>
<tr>
<td>(^*)Peek((D) @ t)</td>
<td>The Kells disk controller ((D)) peeked at the tail of the request queue at time (t).</td>
</tr>
<tr>
<td>(^*)SRead((req, \sigma))</td>
<td>sread was executed in the program trace.</td>
</tr>
<tr>
<td>(^*)SWrite((req, \sigma))</td>
<td>swrite was executed in the program trace.</td>
</tr>
<tr>
<td>(^*)Fresh((t, \tau_{\text{req}}, \tau_{\text{att}}))</td>
<td>The attestation received at time (\tau_{\text{att}}) was received recently enough to be considered fresh w.r.t. a request that arrived at (\tau_{\text{req}}).</td>
</tr>
<tr>
<td>(^*)GoodState((H, req, \sigma))</td>
<td>The host ((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.</td>
</tr>
</tbody>
</table>

### Configuration (\(^*\) indicates an extension)

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Use in Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\sigma)</td>
<td>The store map of ([\text{location} \mapsto \text{expression}]). This is used in the semantics of \text{read} and \text{write} as well as the write request queue.</td>
</tr>
<tr>
<td>(^*)((h, t))</td>
<td>The Kells requests queue, implemented as a pair of pointers to the memory store (\sigma).</td>
</tr>
<tr>
<td>(^*)(\rho)</td>
<td>The program counter. This counter is initialized to (t_0) at reboot time and increments once for each executed action in the trace.</td>
</tr>
<tr>
<td>Configuration (Δt)</td>
<td>Read</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td></td>
<td>Run (secs)</td>
</tr>
<tr>
<td>No verification</td>
<td>36.1376</td>
</tr>
<tr>
<td>1 second verification</td>
<td>36.5768</td>
</tr>
<tr>
<td>2 second verification</td>
<td>36.6149</td>
</tr>
<tr>
<td>5 second verification</td>
<td>36.3143</td>
</tr>
<tr>
<td>10 second verification</td>
<td>36.2113</td>
</tr>
</tbody>
</table>

Table 7.2. Kells performance characteristics – average throughput over bulk read and write operations
Towards Cooperative Interaction Between Storage and the Operating System

8.1 Introduction

Having shown how the host computer can be allowed into the trusted computing base rooted in storage through validation of its integrity, we now begin to examine how allowing augmented storage to interact with the host’s operating system can provide even more robust security services. In particular, we look at the issue of how an OS that supports labeling can provide finer-grained access, down to the block level, compared to the partition-based granularity we demonstrated with Kells.

To achieve this goal, we first leverage some of the unique characteristics of Asbestos labels [54]. The decentralized nature of Asbestos labels, and the ability to serialize and store them on disk, makes the file system label-aware and allows for self-contained policies to be stored on stable storage. Combining these serialized, self-contained policy descriptions with the storage-centered host validation mechanisms used in Firma and Kells allows us to protect policy-critical files, in order to provide an environment where decentralized information flow control (DIFC) can be supported even when portable storage is used. In particular, we describe how we can use secure bootstrapping to identify the host, determine its level of integrity and trust, and recover the privileges associated with it—stored on disk during the initial “pairing” process. We consider mechanisms for protecting policy-critical files, as well as mechanisms for full disk block labeling that would
allow us to enforce data protections regardless of the host integrity level. In summary, we provide a demonstration of how secure disks can work in concert with operating system primitives to support policy mechanisms and improve the security of portable storage.

8.2 Asbestos Label Persistence

The Asbestos operating system [54] aims to improve security by containing the effects of application bugs. An Asbestos label is a function mapping tags and levels: every tag in the system (represented by a unique, opaque identifier), is associated with one of five possible Asbestos label level values—each corresponding to a different privilege, integrity and secrecy level (the “⋆” level denoting privilege and levels 0, 1, 2, 3). Apart from all tags that are mapped to levels explicitly, a label also utilizes a default level that is applied to all tags not appearing in the label.

Asbestos uses a “split label” design for processes: the tracking label keeps track of all contamination and privilege acquired by the process, while the clearance label tracks the level of contamination that the process is cleared to receive with respect to each tag. Alternatively, one can think of the clearance label as the lowest acceptable integrity level incoming information must be at. Using this split label model and different default levels for tracking and clearance labels (1 and 2 respectively), Asbestos labels can implement decentralized information flow control (DIFC). By modulating these labels directly, or through a high-level policy description language [269], one can implement a wide variety of application-defined, kernel-enforced security policies.

Asbestos labels can be stored persistently on the file system. Similar to processes, files carry two labels: a file tracking label and a file clearance label, representing the contamination level acquired when reading the file and the privilege necessary to modify it, respectively. All file system labels are immutable and are set at file creation time. A consequence of this is that declassifying data out of a file requires a privileged process to copy the data to a new, “uncontaminated” file. The pickle primitive allows applications to serialize and store privilege on the file system. Any runtime Asbestos tag can be “pickled” and stored on a special pickle file, along with a privilege level and a key, which overcomes the complication that run-time tag names are non-persistent and random. A special “unpickle” operation allows privilege to be recovered for that tag (e.g., after application restart) if various constraints are satisfied. All pickle files can be unpickled at level 3, by anyone—since that is the most restrictive (i.e., least privileged) level. Unpickling at more privileged levels is controlled through labels on pickle files themselves, as well
as access control checks (i.e., keys associated with each pickle file). With the pickle mechanism in place, all file system labels can be described in terms of pickled tags. Therefore, controlling access to pickle files is essential for the security and integrity of the file system, since pickles are the key to controlling file system labels and serialized, persistently stored privilege.

8.3 Assumptions and Security Goals

Using the advanced capabilities of ASDs, we seek to address certain security concerns related to portable disks. Specifically, we are guided by the following goals and assumptions:

- We assume that the portable drive is equipped with a tamper-proof administrative interface, that can be accessed securely so as to perform certain privileged administrative tasks (e.g. firmware/software upgrade). It could be reliant on physical security, such as requiring the use of a hardware key to access the interface, or policy on what machines are considered administrative could be set within the ASD and upon verification of the administrative machine’s identity, it would have the capability to perform privileged operations.
- We assume that data stored on the portable drive are labeled. In particular, we assume that the file system is using the Asbestos label persistence mechanism.
- Portable disks can be connected to many different hosts. We want to provide data isolation between hosts, by ensuring that information generated from one host is by default not accessible to any other host the drive may be connected to, unless it was explicitly stored as “unprotected” (or “unclassified”) data.
- To ensure that the practical value of portable hard drives is not eliminated by our security mechanisms, we support the notion of “trust equivalence”. We do not define static trust levels hosts can belong to, but we support the notion of two hosts being considered of equal trust and integrity levels. Declaring two hosts as equivalent is a privileged operation that requires secure access to the portable drive’s administrative interface.
- We first assume that all hosts the drive is connected to are attempting to access data through the labeled file system. Attempting to protect against malicious hosts who try to access the raw disk blocks poses additional challenges and is discussed separately in Section 8.4.2.2.
8.4 Protection Mechanisms

8.4.1 Pickle File Protection

Information stored on disk by a system using Asbestos labels may or may not carry file labels protecting access to it. In our model, all unlabeled information on the disk is considered “unclassified” and is accessible to any system the portable medium is connected to.

As described in Section 8.2, access to all labeled data depends on the ability to unpickle the necessary privilege, or, in other words, on the ability to access the relevant pickle files. Therefore, by controlling access to pickle files, one could control the amount of privilege that could be recovered from disk and, consequently, the disk data that may become available. Our proposed mechanism uses the ASD to control access to pickle files, based on the identity of the host the portable disk is connected to.

During the process of pairing with a new host $X$, the ASD stores along with the host’s fingerprint (i.e., $EK(X)$ and $ML(X)$) a pickle access token, $PA_X$, unique to that host. Access to all pickle files stored on disk by $X$ is controlled at the disk block level: when a new pickle file is created by $X$, the ASD will mark the relevant disk blocks with $PA_X$. All accesses to pickle file blocks are controlled by $PA$ token checks: when attempting to access disk blocks corresponding to pickle files, the ASD checks the current “active” $PA$ (i.e. the $PA$ of the host the ASD is currently paired with) against the $PA$ associated with the disk blocks in question. If the $PA$s do not match, the pickle file blocks access to the pickle file blocks is denied. Notice that this mechanism is independent from higher-level, file system pickle file access control mechanisms.

By using $PA$s, the ASD prevents access at the disk block level, making the pickle file inaccessible from other hosts—even if the user is able to present the necessary credentials for an unpickle operation. Therefore, by controlling access to pickle files created by each host, we are able to control the amount of privilege that can be gained through unpickle operations and limit each host to its own separate, isolated view of the file system.

The pickle access mechanism assumes that the ASD will be able to identify pickle blocks and mark them accordingly. This can be achieved by storing all pickle files in a secure area on the disk, an operation feasible for disks supporting the Opal trusted storage specification [151]. Alternately, the pickle operation can issue an $ioctl()$ to notify the disk. Both solutions rely on the correct implementation of the pickle operation, which is one of the things that can be measured during the integrity measurements performed using the TPM.
We introduce the idea of *equivalence* between two hosts, which can be specified as a policy parameter through the disk’s administrative interface. By making two hosts equivalent, we instruct the ASD to operate in exactly the same way (i.e., apply the same policy) when either of two hosts’ fingerprint is detected. Host equivalence is clearly marked on the portable disk’s list of known hosts. Note that equivalent hosts share the same PA and, therefore, have the identical access rights to pickle files. This would allow file sharing between equally trusted machines, e.g., two secure workstations in the same environment.

### 8.4.2 A Step Further: Full Labeling

Using the PA mechanism, the ASD can implement complete pickle file isolation, and the notion of host equivalence can enable full file sharing between hosts: *all* pickle files created by X are inaccessible to hosts not equivalent to X. This level of protection is a significant improvement over the current situation, and may be adequate for most cases.

However, the mechanism may be too coarse-grained for certain scenarios: we may want two hosts to share access to some, but not all, disk blocks. Moreover, the sets of shared disk blocks between different pairs of hosts may differ: host X may be sharing one set of its disk blocks with host Y and another (overlapping or not) set with host Z.

Implementing such shared sets of disk blocks could be achieved through the use of multiple PAs per host, each representing a different category of trust between hosts, as shown in Figure 8.1. The notion of implementing fine-grained policies using such dynamic categories, is very analogous to Asbestos labels themselves: In essence, this block sharing behavior would require the equivalent of Asbestos labels at the block level, which would allow for the definition of such sharing, as well as many other, disk block access policies.

Moving to a full block labeling mechanism enforced at the disk level would also result in a significant change of the security model: disk block protection would no longer be transparent and mandatory, but user-visible, user-controlled and discretionary.
Users would be able to instruct the drive to create new categories by generating the disk equivalent of Asbestos tags—which we call dtags—used to label the disk blocks belonging to that category.

Although this mechanism may seem redundant in the presence of a trusted labeled file system, its merits become apparent when one considers how the portable drive would operate in a less trusted or friendly environment: disk block access policies are defined and enforced within the portable drive itself. This self-contained system is able to protect data even in when the host is not co-operating, and could be used to protect from malicious users, or even lost or stolen drives. Essentially, the disk can act as a defense-in-depth mechanism by using the policy received from the host as a type of anomaly detection at the block layer if the request received from the host appears to be inconsistent with what has been laid out by the policy received from it. This could trigger a runtime attestation to ensure that the host is in a good state before fulfilling the request.

### 8.4.2.1 Labeling Pickle Blocks

By replacing PAs with labels, we could implement more complex policies regarding how the ASD would restrict access to pickle disk blocks. Once could envision a mechanism operating in the following manner: When an ASD is paired with a new host $X$, it generates a new dtag $x_0$ for that host. Unless the user requests the creation of a new dtag for $X$, $x_0$ will be used to provide the same security that the PA mechanism would: all pickle blocks are labeled with $x_0$ and accessing them would require holding $x_0$ privilege. If the user requested the creation of a new category (by using the relevant ioctl() call), the ASD would create a new dtag $x_1$, associate it with $EK(X)$, and make $x_1$ the “active” dtag—i.e. the one used to label the disk blocks of all pickle files created by $X$ from that point on.

Along with each host’s $EK$ the ASD would store a list of dtags the host holds privilege over. When pairing with a new host $Y$, a user with secure access to the ASD administrative interface could grant $Y$ privilege with respect to dtags other than $y_0$, and grant $y_0$ privilege to already existing hosts.

### 8.4.2.2 Full Block Labeling

By labeling pickle blocks we still use block labels only to determine access control rights over pickle disk blocks, essentially implementing a capability system. Consequently, we do not need to use the Asbestos split-label design. Using the full features of Asbestos labels (multiple dtag levels and split-label design) would enable block-level information
flow tracking for this mechanism. However, implementing dtag management logic and block labeling capabilities within the ASD brings us a step closer to a fully labeled disk able to support disk-enforced, application-level policies. Apart from labeling pickle file disk blocks for host access control, the disk also needs to support labeling of all disk data blocks, based on check-pointed application-level Asbestos label policies. Either periodically or on-demand, file system labels would be translated to block access restrictions, and pushed to the ASD’s secure storage using trusted SCSI or ATA commands.

While this disk-wide block level policy enforcement entails some complexity within the ASD (requiring label book-keeping logic, label checking functions etc.), proposals such as the rootkit-resistant disk prototype show that the administrative overhead of such operations can be very low, on the order of 1%. Additionally, this enforcement model provides some unique benefits. By downloading application level policies to the ASD, in combination with secure pairing and pickle disk block protection, the ASD would be able to provide a complete, self-contained data security solution. A self-contained, policy-enforcing portable ASD would be able to provide protection against loss or theft: the drive would refuse to grant access to pickle files when paired with unknown hosts. If the pickle mechanism is bypassed and a malicious user attempts to access raw disk blocks, the ASD would still refuse access, due to failed disk block label checks. Additionally, full disk block labeling would be able to protect against compromised or low-integrity systems—as detected by the security measurement capabilities of the ASD—by performing label checks within the disk based on the last high-integrity label policy downloaded to it (and not having to rely on the integrity of the operating system).

8.4.3 Discussion

Labeling individual blocks using application-defined policies would also allow us to implement interesting file sharing and locking semantics. By applying different labels to blocks belonging to the same file, one could create potentially usefully locking mechanisms, as well as implement file access control policies at a fine granularity. Applications that manipulate large files internally to implement their own storage logic (e.g. databases storing row data inside a file) could benefit from such a mechanism, but block labels are expected to be immutable and applications would need take this into consideration. Additionally, making certain disk blocks inaccessible might cause anomalies: making file system metadata blocks accessible to a process would reveal a lot of information about the file, even if some of the actual data blocks are not accessible by that process.

Also, note that disk block labeling would require the ASD to be responsible for both
label book-keeping and policy enforcement, and therefore implementation is dependent on the memory and CPU capabilities of the ASD. As shown with our previous designs, however, both fixed and portable drives increasingly incorporate advanced memory, CPU and network capabilities, making them able to run fully programmable embedded version of operating systems, such as Linux.

8.5 Summary

This chapter has proposed an architecture for block level policy enforcement at the disk layer, to provide multiple levels of security for the potentially many systems that can attach to portable storage. This allows users to access information on their disk without exposing sensitive information to potentially malicious systems or those that may become compromised during use. We use Asbestos labels in conjunction with the storage-based validation procedures to provide guarantees of host integrity, and describe how we can ensure that the appropriate security policies are maintained depending on the host connected to. The collaborative security between the host system and the disk provides an interesting new security model and challenging issues as we further explore this problem space and implement prototypes demonstrating these new functionalities.
Chapter 9

The Future of Autonomous Storage Security

The Department of Defense has recently begun relaxing its stance on portable flash drives, though it is attempting to maintain strict control over how they are used and by whom [270]. Under the new directives, flash drives are meant to be used as a “last-resort” option when there is no other way to transfer information from one location to another, and auditing will occur. Vice Admiral Carl Mauney stated that “Use will be permitted only in DOD computers that are in compliance with requirements for hardware that allows for safe transfer of data” [271]. Through the use of portable flash drives augmented with their own intelligence such as IronKey media, which contains an anti-malware scanner and cryptographic chip, in conjunction with on-host file sanitization utilities, the goal is that malware can be prevented from propagating as it did in 2007.

What these new developments show is that device security is going to be a major concern for a long time to come. The military found that going without portable storage was too restrictive to getting necessary work done, and new ways for protecting these devices has been suggested. Businesses that find themselves at similar risk for both leaking data and being exposed to malicious information will be increasingly at the forefront of demanding such devices and solutions for their continued safe and secure business operations.

As a result, the time is ripe for implementing increased security functionality into storage devices. As we have demonstrated in this dissertation, practical solutions for data and host protection in the face of vulnerable operating systems are possible with currently-available componentry and technical know-how.
Consider the Department of Defense. How would the autonomous storage security designs that we have investigated have helped mitigate or eliminate the dangerous CENTCOM incident? Consider an average workstation that the rogue flash drive could have been plugged into. Had the workstation been using Firma, we would know that the system was booted into a good state and if a malware scanner had been part of the continuous integrity measurements, it may have caught the vulnerability - in this case, the malware was a variant of the Conficker worm, which exploits a vulnerability in the Windows Server service [252]. As the continuous attestation mechanism may have detected this, writes to the disk could have been prevented. Conficker attempts to make itself persistent and starts when the victim host is booted, by saving a copy of its library routines in a random file within the subdirectory. If a rootkit-resistant disk had been employed on the workstation, this directory would have been installed with the immutable token in the drive; as a result, even if the host had been infected, rebooting the system would have eradicated Conficker from it. Finally, Kells storage devices could have been resistant to this attack in the first place. By not exposing the private partition of the drive until the host had been validated to be in a good state, had the flash drive been plugged into a compromised system or one that did not perform integrity validation, then nothing could have been written to the drive. Conficker attempts to write itself to the autorun.inf file in order to propagate malware to the victim - in this case, that would not have been possible. Had this been written to a public Kells partition, then the partition would have been quarantined by the workstation, preventing it from being infected.

As this example shows, the solutions we have proposed are complementary to approaches the DoD is already taking such as maintaining strict control over who can use flash drives and ensuring the ongoing maintenance of security architectures on computers, as well as an emphasis on correct user behavior. The human factor is always going to be an issue at cross-purposes with security - this entire scenario was the result of curiosity getting the better of employees who should have known better than to plug a drive of unknown provenance into their valuable machines. Nonetheless, by attempting to prevent an attack from being successful in the first place and by mitigating the damage done by it, our solutions can prevent tragedy and aid in making the aftermath as painless as possible.
9.1 Future ASDs

9.1.1 Object-Based Storage

One of the most controversial decisions we made in our investigation was to focus as much as possible on strictly the disk alone and considering a block-based paradigm. By doing this, we certainly lost a great deal of metadata that could have been tremendously helpful in helping make decisions related to security. In particular, had we considered object-based storage, we would have had access to significantly more metadata. The OSD security model, based on NASD, allows access control to be performed on a per-object basis. With OSD storage, the role of the filesystem’s role is simplified - it just needs to find objects, but those objects are self-contained in terms of attributes. This could have led to focusing directly on the implementation of ideas such as information flow in storage, because much of the information required, such as labels, could be directly applied to an object. Unfortunately, the promise of object-based storage devices has not yet passed in reality. Despite significant interest in the idea from companies such as Seagate, there are no OSDs commercially available, particularly in the context of desktop or portable storage.

Some of these concepts have been used in huge compute clusters: the Lustre filesystem [272] is similar to NASD in that is uses objects but within the context of a large storage cluster with a dedicated metadata server. Ceph [273] uses this approach, as does Panasas [274]. These do not satisfy the goals that we had set out to satisfy, namely a storage device that could be self-contained and autonomously enforce security for a traditional computing environment. While the Linux kernel has finally included support for object-based filesystems through *exofs*, added to the mainline kernel in April 2009, this is maintained by Panasas, whose drives are not available to the public. It appears as though block-based access is not going away, and there are no immediate plans for the average consumer or administrator to be able to use object-based storage on their computers. The approaches that we have investigated in this work are fundamental and will be relevant regardless of whether storage access methodologies change in the future. Because of our ability to work in conjunction with the operating system, we area able to receive context about data semantics, and bridge the semantic gap by penetrating the conundrum between the block layer and the filesystem.
9.1.2 Changing Storage Paradigms

9.1.2.1 Effects of Solid-State Storage

The last few years have been noteworthy for the rapid change in the way that storage devices have evolved. Our work has been predicated on taking advantage of these changes to demonstrate how new and emerging devices can be leveraged for providing new security functionality. This continues to be the case, and nowhere is that more apparent than in the rise of solid-state disks (SSDs). While there will be a place for magnetic storage for years to come—one cannot ignore the massive amounts of storage that these devices are capable of storing at ever-decreasing costs—it is clear that for performance and access optimization, as well as their rugged construction and resistance to shock and drops, SSDs will continue to grow in popularity, both in low-end mobile computing devices and in high-end computing. This means that a new storage hierarchy is forming, but does this change, or even worse, nullify, the arguments that we have made in this work?

On the contrary, we have seen a variegation in the types of non-volatile memory used to construct SSDs. Notably, the market has differentiated between lower-cost multi-level cell (MLC) NAND memory, where each cell can contain multiple bits, and more-expensive single-level cell (SLC) memory, which is more robust and handles many more write cycles than MLC but is lower in density on account of storing only one memory bit per cell. Access is faster on SLC devices, but the costs of SLC devices are many times greater than with MLC. Considering our ASD design where we use fast NVRAM as a storage area for metadata, it is easily conceivable that SSDs can be constructed in a similar manner where MLC flash takes the place of rotating magnetic storage and a smaller amount of SLC flash holds metadata and policy information that can be quickly accessed and replaced. As technologies develop and mature, we can expect that there will continue to be different hierarchies for storage media in terms of cost versus performance, so the general design elements that we have introduced will be adaptable even as specifics may differ.

9.1.2.2 Future Storage Devices

The way in which we access storage in the future is of interest in its own right. The fact remains that individual computers will continue to have local storage, but how much of it will stay local depends on the adoption of cloud-based services. Google is staking a claim that the majority of data can be stored in the cloud and its Chrome OS is an example. The operating system will resemble a web browser, applications will be run from the web,
and data will be stored there as well. The local storage will primarily support boot-time operation and browser management, along with caching for offline operation. Whether this model will be suitable for all users remains to be seen - it is one that requires the user to always have access to their computing device and to a network. The fact that the DoD is still reliant on the use of flash drives is an indication that the need for portable storage is not going away, while the sheer amount of data that may be stored on a conventional hard disk makes it difficult to envision that for all users, even those with high-bandwidth networks, will wish to outsource all of their data and applications to the cloud. With a 1 Mbps upload connection to the Internet, even performing backup of a 1 TB disk could take many weeks.

What may change, however, is the devices that are used for storage. As we alluded to above, massive-scale storage arrays are capable of leveraging object-based storage paradigms due to the increased computation available to the storage cluster. In fact, to look, for example, at Network Appliance storage devices is to see the active disk philosophy in practice. These devices are really full-scale servers, complete with general-purpose CPUs and stocked with RAM and NVRAM. The NetApp FAS6080 can hold up to 8 dual-core AMD Opteron processors, comes with 64 GB of RAM and 4 GB of NVRAM, and is capable of supporting over 1 PB of storage [275]. These devices already are fully capable of being seen as ASDs, and can easily support arbitrary security policies.

Moving from massive-scale to small-scale storage, we have witnessed the dramatic increase in adoption of smartphones. In 2009 alone, over 170 million smartphones were purchased worldwide [276]. These phones have many gigabytes of available storage, and are always on-hand to users, who carry them everywhere. With advanced mobile operating systems such as Android powering these phones, an opportunity exists for storage to be carefully managed on these devices, and given the powerful processors and memory increasingly available on them, they can easily fulfill the role of acting as an ASD as well. Already these small, ubiquitous devices have taken critical roles as portable storage devices: portions of the movie *The Lord of the Rings* that needed CGI rendering were transported on Apple iPods between filming and post-production locations [277], thus showing that such portable devices with physical interfaces for transferring data will be valuable for the foreseeable future.¹ For us, this means that even if the medium on which data is stored is to fundamentally change away from the model of hard disks and flash drives, currently the primary conduits of information transfer, the methodologies

---

¹This reliance on physical versus networked data transfer also lends further credence to Tanenbaum’s oft-repeated quote, “Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway.” [278]
that we have described in this work will be relevant.

9.2 Future Work

9.2.1 System-Wide Information-Flow Assurances

This work is the first to consider information flow enforcement in the context of storage devices. This is a rich area to be mined, particularly determining what services storage can easily supply to the operating system such that enforcement happens in tandem. The previous chapter began this consideration, but there is considerable room for implementation and formalizing the understanding of how these interactions may occur. This can include investigating other policy enforcement models such as the Biba integrity model, high and low water mark for confidentiality and integrity, respectively, Chinese Wall integrity policy, and Clark-Wilson integrity. In particular, we look to the CW-Lite model [279], which provides information-flow guarantees between processes with less filtering interfaces required than with the Clark-Wilson model. Further to this model, an interesting area of research will be considering how CW-Lite may be extended to the storage layer, and in a similar manner to how tools such as Gokyo [280] determine information flows from policy by examining permissions, and how automated tools can be used to detect information flow violations at the storage level.

Further to this model is the integration of storage-level enforcement of information flows with policy decision points elsewhere in the system and determining compliance. With the ability to maintain multiple roots of trust and secure interfacing between storage and the host system, we will consider how to use these facilities to communicate information-flow policies between these two layers and through the application layer, such that the system is able to enforce information flow assurances from the application layer through to storage. Knowledge through multiple layers of policy and the interaction between these layers will be beneficial to solving the semantic gap of policy information between layers, and may allow for detecting issues such as covert flows. A tremendously valuable system architecture will be one that supports and adds to the functionality of current initiatives such as the FlowWolf integrity-flow system [281] and the Shamon distributed reference monitor [282], as well as further investigate the complexity of policy compliance issues between levels, by comparison of our efforts to work that has previously been performed in other intelligent disk proposals such as semantically-smart [207] and type-safe disks [127]. This work serves leaves us positioned to consider the security implications of these intelligent disk systems, as previous proposals have not considered
security beyond rudimentary capability-based applications.

9.2.2 Firmware Verification

Our work to date has been prototyped with devices using commodity Linux kernels and small, customized operating system distributions. These are generally smaller than the OSes used in general-purpose operating systems and have been invaluable for rapid development and evaluation. They are also considerably smaller codebases than the commodity desktop systems that users currently run. To fully realize the ASD as a reference monitor as classically defined by Anderson, however, we would require showing that a formally verifiable codebase is possible to achieve. Recent work has shown that verifiable kernels are possible: the seL4 kernel is only 8700 lines of code and has been formally verified \[283\]. Unfortunately, the firmware for storage devices is notoriously difficult to gain access to. With projects such as OpenCores\(^2\) that provide chip designs in HDL, however, it may be possible to build a ground-up design that can be fully verified for correctness. With advances in static analysis techniques and the ability to analyze large codebases \[284\], combined with the smaller codebases of these embedded devices, such verification may be within the realm of possibility.

9.2.3 Provenance-Enabled Storage

Data provenance \[285, 286\] traces the genesis and subsequent modification of data as it is processed within and across systems. Such information indicates the pedigree of data \[287–289\] and enhances, among other functions, experimental replay \[290\], auditing \[291\], and malicious behavior detection \[292\]. This is an area that has been of tremendous interest to the scientific community, but mechanisms for providing secure provenance are largely experimental. We have been examining the idea of a provenance monitor that records provenance information as it enters a system and records every manipulation and interaction with the data. The provenance monitor enforces similar guarantees to a reference monitor \[293\]: it must provide complete mediation of provenance, be tamper-proof, and be simple to verify.

Autonomously-enforced storage is well-suited to act as a provenance monitor for the same reasons that make it attractive in an operating system: placing the provenance monitor within an ASD reduces the system’s TCB. Such a system may require more cooperation with the host than can be provided if the system is assumed to be automatically

\(^2\)http://www.opencores.org
untrusted; our work with Firma and Kells has described a method of ensuring that the system is in a good state. In particular, portable storage capable of recording provenance can allow for interesting new applications, such as allowing forensics so that infected files can be traced to the source. If, for example, a file was transferred to a Kells device that was later found to be tainted with malicious code, by examining the provenance of read and write transactions of data blocks associated with these tainted files, it would be possible to determine the scope of potentially-compromised machines. Managing the potential explosion of state as the provenance of files is recorded will be a challenge that may be met through some of the large-scale cryptographic systems that we have previously designed; notably, the use of cryptographic proof systems such as Merkle hash trees have been used to efficiently represent routing updates [294] and to scalably attest content in the presence of a slow TPM [268] provides a potential solution to dealing with large-scale provenance information.

9.3 Concluding Remarks

Decades of research into the security of operating systems have led to similar conclusions: security is improved by reducing the size of the trusted computing base. What we have shown in this work is that because of a fundamental rethinking in the capabilities that can be offered from storage, devices, they now present an excellent TCB within the computer. As a result, new security services are available that can protect not only data, but the security of the system as a whole.

Starting from a TCB primarily composed of just storage, and making the assumption that the operating system can be arbitrarily adversarial, we showed that it was still possible to provide solutions to mitigate the effects of malicious code on an operating system. Moreover, we showed that operating systems could themselves be protected from other misbehaving systems sharing common storage on a disk. With a small amount of trusted hardware support from a host, we showed it was possible to bring the host, including the operating system, into a TCB whose core root of trust remained storage. Looking towards new ways in which storage is used, we provided similar guarantees for portable storage, and described how a landscape where the operating system could be considered trustworthy allows for new methods of cooperative enforcement between the host and its storage.

While this dissertation has been wide in scope, it is clear that we have just scratched the surface of understanding how secure storage architectures can improve security in
myriad new ways. We have outlined fundamental design principles for device design and shown how even with significant constraints on what we know about stored data, significant and practical solutions are possible. By expanding our scope to encompass interaction between the storage and the rest of the computing system, we lay the groundwork for significant new advances to be possible. It is our fervent hope that this dissertation, while drawing on decades of past work, represents a bold new way of looking at storage and provides future researchers with a framework and vision towards finding ever more novel ways of keeping users and their data safe and secure.


URL http://portal.acm.org/ft_gateway.cfm?id=291026&type=pdf&coll=Portal&dl=ACM&CFID=14870899&CFTOKEN=42950280


URL http://eprints.ecs.soton.ac.uk/16580/


Vita
Kevin Raymond Boyce Butler

Education
The Pennsylvania State University, University Park, PA, USA. Ph.D., Computer Science and Engineering, August 2010.
Queen’s University, Kingston, ON, Canada. B.Sc., Electrical Engineering, May 1999.

Awards and Honors
- Pennsylvania State University Alumni Association Dissertation Award (2010)
- Symantec Graduate Fellowship (2009)
- Penn State CSE Graduate Research Assistant Award (2009)
- IUSSTF Travel Award (2007)
- Special Award of Meritous Service, ICISS (2007)
- Pennsylvania State University Graduate Fellowship (2006)
- C. Norwood Wherry Memorial Graduate Fellowship (2004)

Professional Experience
- Research Intern, Symantec Research Labs, Mountain View, CA, Summer 2009
  Designed and implemented an architecture for protecting portable storage.
- Research Intern, Seagate Resarch, Pittsburgh, PA, Summer 2008
  Developed security infrastructures for distributed storage in a home user environment.
  Investigated security threats in BGP and deployed cryptographic constructions for efficient security solutions.
- Intern, Flarion Technologies, Bedminster, NJ, Summer 2002
  Implemented packet diversion and IP header modification in BSD for mobile IP testing.
  Investigated routing alternatives and service assurance in multiple service provider hybrid fibre-coax environments.
  Monitored and diagnosed backbone connections across the Canadian network.