Investigating Security Failures and their Causes: An Analytic Approach to Computer Security

John Alexander Halderman

A DISSERTATION

PRESENTED TO THE FACULTY OF

PRINCETON UNIVERSITY

IN CANDIDACY FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE BY THE

DEPARTMENT OF COMPUTER SCIENCE

Advisor: Edward W. Felten

June 2009

Abstract

This dissertation examines security failures in three classes of systems: compact disc (CD) audio recordings containing digital rights management (DRM), touch-screen electronic voting machines, and on-the-fly disk encryption software. In each case, we study a variety of implementations developed by different parties; we analyze their security and discover a range of security flaws, including several entirely new categories of attacks; and we propose new mitigations and defenses for protecting related systems. Each of these studies has already had significant real-world impact, and we extend them with a new methodology for studying the underlying causes of security failures and drawing broader lessons for users, developers, researchers, and policymakers.

We begin with CD-DRM systems—security mechanisms for audio CDs that are designed to limit copying and other uses of the music. In the course of tracing the evolution of these technologies over three generations, we discover a range of new attacks, including numerous ways that attackers could bypass the anticopying measures and ways that disc producers could free-ride on other vendors' copy-protection systems to receive the benefits without paying. We demonstrate a new class of threats, collateral damage to the security of CD-owners' PCs, and argue that they are an inherent risk of DRM. We discuss additional factors that led to these failures, including differences between the incentives of CD-DRM vendors and their record-label customers.

Next, we turn our attention to electronic voting systems, specifically touch-screen direct recording electronic (DRE) voting machines. We perform a detailed security evaluation of two similar implementations, the Diebold AccuVote-TS and AccuVote-TSX, applying both reverse engineering and source code review to reveal security flaws. We show how attackers could exploit these flaws to tamper with election results or disrupt the voting process, and we demonstrate a dangerous new attack vector, voting machine viruses. We compare security problems uncovered in other DRE voting machines to suggest common causes and threats, including failures in voting machine certification procedures and incentives that rewarded features and time-to-market over robustness and security.

Finally, we demonstrate new threats to the security of on-the-fly disk encryption software, which is designed to protect confidential data against an attacker who gains physical access to the computer. We conduct a series of experiments to investigate memory remanence in dynamic RAMs, a phenomenon largely unknown to security research that causes data in RAM to remain intact for a short time after the memory chips lose power. Attackers can exploit this effect to bypass operating system security and recover sensitive memory contents, such as encryption keys. We demonstrate how this would allow an attacker to defeat most popular disk-encryption products. We discuss how the widespread ignorance of this basic hardware behavior relates to abstraction, a fundamental computer engineering principle, and suggest other abstractions that might similarly conceal security threats.

In all three studies we apply new methodology that combines causal analysis with security engineering. We adopt the concept of informative causes of failure to organize and direct our investigations. In the pursuit of causes, we compare security flaws across different implementations to find supporting evidence in suggestive patterns of failures. Like the search for flaws, the search for causes seems resistant to thorough systematization, but it has been a useful tool for guiding us to the broader lessons of these security failures.

Contents

Chapter 1

Introduction

Modern computer systems are among the most complex objects ever devised by humankind, yet lives and fortunes depend on their functioning correctly. As difficult as it is to build reliable computer systems, it is even harder to build ones that are *secure*. A secure system must work correctly even when facing an adversary, an intelligence that actively attempts to make the system misbehave. Adversaries are not bound by the designers' assumptions; they may poke and probe the system using all methods at their disposal, whether this means working around, "tunneling under," or directly attacking the system's defenses. Often we attempt to improve reliability by testing the inputs and conditions that are likely to occur "in nature," during the normal operation of the system, but an adversary will seek to construct inputs or conditions that bring about failure no matter how unusual or unnatural they may be. This is a central challenge faced by the computer security field—how to predict and avoid failures in computer systems in the presence of an adversary.

Most computer security research can be divided into two categories: *defenses* and *attacks*. The first group devises new countermeasures, such as building blocks (e.g. cryptographic functions and protocols), system-level defenses (e.g. firewalls and CAPTCHAs), and implementational techniques (e.g. safer programming languages, formal verification). The

second group exposes and investigates flaws in existing designs; it is the subject of this dissertation. Though these approaches may seem to work at cross purposes, they are in fact symbiotic. New attacks motivate the development of improved defenses, which in turn provide fodder for innovative attacks. The mutual goal of both research areas is strengthened security.

1.1 Why Study Failures?

Inquiries into failure and its causes are central to technical progress: they delineate the practical boundaries of what we can achieve with current technology and highlight areas that are ripe for further research [129, 130]. Studying failures has become a standard practice in many engineering fields [46, 60, 127, 181]. When a bridge collapses [120] or a building falls down [58], or a Space Shuttle burns up [37, 145], scientists and engineers are called upon to understand what happened and seek lessons that will prevent such disasters from reoccurring. Learning from failures is also common in areas of computer science. Krsul et al. [96] cite numerous examples, including case studies of software faults, software testing methodologies for detecting various kinds of flaws, and advances in compilers and development tools in response to past problems.

Attack research examines computer security "through the lens" of failure. This strengthens security in several ways. Most directly, it can reveal flaws in real-world systems, allowing developers and users to correct problems before attackers exploit them. It also puts pressure on developers to be more diligent about security by increasing the odds that problems will be publicly exposed. In offensive scenarios, it may be desirable to bring about the failure of hostile systems—attack research can also strengthen our arsenal.

Beyond these, a large part of what we learn by studying security failures comes less directly, from generalizing specific incidents to draw *broader lessons*. These lessons fall into a diverse variety of categories, for instance:

- Lessons about where to look for problems. Failures in some systems can point to other systems or scenarios in which similar failures are likely, helping direct the search for vulnerabilities.
- Lessons about what we need to defend against. Attack researchers sometimes invent new classes of attacks that can apply in many kinds of systems. Recognizing new kinds of attacks can stimulate the development of novel countermeasures. Many studies that introduce attacks also propose new defenses to combat them.
- Lessons about the practicability of defenses. Security usually comes at a cost. Sometimes when systems exhibit security failures, we can point to defensive techniques proposed in the research literature that seemingly would have prevented the failures if they had been adopted. Careful investigation of why the defenses were not applied sometimes reveals unforeseen costs or contours of the problem that limit the practicality of the proposed solutions. This helps us make sure research approaches closely track the real needs of defenders and directs us to areas where we need to further reduce the costs of security.
- Lessons about risks. Studying past failures can sharpen our intuitions about the relative riskiness of different designs and implementational methods, helping us avoid dangerous approaches and direct testing resources at likely trouble spots. This help designers improve their threat models and allows users to more accurately assess risks and make better cost-benefit tradeoffs.
- Lessons about our assumptions. Attacks are often made possible because system creators, in reasoning about security, made critical assumptions about the behavior of their systems that turned out to be false. Even formal proofs that software is security depend

on models of how hardware will execute it and of how external systems will behave. By uncovering mistakes in these assumptions and models, attack research helps improve the correctness of future designs.

- Lessons about the human elements of security. Systems are built, used, and attacked by people; their motivations, capabilities, and limitations have enormous security consequences. We need to use models of human behavior alongside our models of computer behavior in order to understand the full behavior of a system. For instance, researchers need to understand how much we can expect from designers in terms of engineering prowess and from users in terms of sound judgment about security concerns. Another important question is how the actions of implementers and attacks are shaped by their individual incentives, as this determines how much effort the parties can be expected to invest in defenses and attacks. The answers to these questions can help shape both technical and nontechnical approaches to security problems.

Done right, the study of security failures holds promise far beyond the mitigation of immediate dangers. It can lead to better understanding of threats, safer approaches to design, and new opportunities for discovery.

1.2 How We Study Failures

Computer security failures can be studied *post mortem*, by analyzing malicious attacks, or *ante delictum*, by discovering new vulnerabilities before they are exploited. Researchers have been doing both since the early days of the field. Designers have long studied attacks to strengthen their systems; for example, in a 1979 paper about UNIX password security, Morris and Thompson describe how its design "was the result of countering observed attempts to penetrate the system." [116] A decade later, some of the first studies to focus exclusively

on a particular assault came in response to the Internet worm unleashed by Morris's son; Eugene Spafford's analysis reflected on the value of such endeavors: "What we learn from this about securing our systems will help determine if this is the only such incident we ever need to analyze." [163] (Alas, it was not.) By the mid-1990s, so-called "attack papers" were an established genre. Sometimes they combined the discovery of new attacks with efforts to understand their causes. Dean et al. for instance, discuss how vulnerabilities they discovered in the Java programming language "arise for several reasons, including implementation errors, ... [and on] a deeper level, ... weaknesses in the design methodology ..." [42] These are just a few illustrations of a seasoned practice.

One of the leading exponents of learning from security failures has been Ross Anderson. The practice has long been a focus of his work [5] and is a centerpiece of his book *Security Engineering* [6]. He explains its importance in the preface to the first edition: "New systems are often rapidly broken, and the same elementary mistakes are repeated in one application after another. ... Many of these failures could have been foreseen if designers had just a little bit more knowledge of what had been tried, and had failed, elsewhere." Anderson has developed ways to investigate the causes of failures using ideas from economics [7] and social choice theory [153], such as by modeling different actors' incentives. He and his Cambridge colleagues have combined a lesson-drawing approach with the discovery of new attacks involving emission security [98], smart cards [8], and electronic payment systems [49], among other topics.

Though many security researchers acknowledge the importance of learning from failures, few studies apply a systematic approach to understanding them. Such approaches have been proposed for certain kinds of failure analysis, and we mention a few of them briefly. Vulnerability classification methods (e.g., [3, 12, 97, 99, 19]) create taxonomies of flaws. Grouping failures using one set of characteristics can reveal hidden similarities

among other characteristics, which may illuminate shared causes or implications. It can also be helpful to fit new flaws into preexisting taxonomies, since this can quickly point to applicable lessons from previous failures. Another family of systematic approaches (e.g., [125, 142]) apply formal reasoning techniques such as model checking to determine how a set of vulnerabilities might be used to violate a system's security policies. Researchers could use these techniques to discover new attacks or to rapidly determine the impact of a new vulnerability. The last family we will consider encompasses root cause analysis (RCA) techniques [20] and related problem solving methods. These are systematic but informal methods for seeking out the causes of particular problems with the goal of preventing recurrence. RCA seeks to answer three questions: What happened, why did it happen, and what can be done to prevent it from happening again? [170] It is regularly applied in psychology [141] and in investigations of industrial accidents and defects [20], and increasingly in the medical community [182]. RCA is sometimes used for investigating security breaches in businesses [165], but it is rarely [89] applied in security research.

Unfortunately, there seems to be little consensus in contemporary security research over how broader lessons should be drawn from observed failures. In particular, there is widespread disagreement about five basic issues:

1. What questions should we ask?
2. How should we answer them?
3. How should we report the answers?
4. How should we evaluate whether purported answers are correct?
5. How should we value answers to these questions relative to other contributions?

In order to quantify the inconsistency of contemporary approaches to failure analysis, we reviewed the proceedings of four major security conferences from the past two years,

NDSS [134, 135], Oakland [138, 139], CCS [132, 133], and USENIX Security [136, 137]. While 37 out of 252 papers (15%) focus on new attacks, 11 of these (30%) neglect to draw any nonobvious lessons. Of the 26 that do draw some sort of broader lessons, only half position such analysis as an important contributions by mentioning the lessons in the abstract or featuring them prominently in the introduction. The papers that do draw lessons do so in many different ways: some seek to understand why failures happened in the past, others are only concerned with how we can prevent them from recurring; some explicitly state that they are drawing such conclusions, others do so only implicitly; some limit interpretation to its own section, others blend it with descriptions of the attacks. There appears to be little agreement within the security field about how we should learn from failures.

In contrast to the diverse and usually unstructured approaches to deriving broader lessons, there is considerable uniformity in the way we study the direct lessons from failures. Conferences often treat failure studies as a distinct genre, grouping them into dedicated "attack sessions" even if their subjects are otherwise dissimilar. This uniformity exists because the research community has developed a powerful organizing principle to structure such work: the notion of an attack as a recipe for an assault.

This notion is so central to the way failures are investigated that most researchers do not realize they are applying it, but we find evidence of its impact in the specialized meaning of the word "attack" that has developed within the security community. Researchers sometimes use the noun "attack" in much the same way as nontechnical speakers do, to refer to an actual malicious assault[1], or as military strategists do, to refer abstractly to a class of assaults involving similar means[2]. Yet over the years, use of the word has changed, and

1 Vixie in 1995: "We learned of these weaknesses by studying some successful [malicious] **attacks**, not just by a careful examination of the protocol and the BIND source code." [173]

2 Spafford in 1988: "That is, some dictionary-based or brute-force **attack** was used to crack a selection of a few hundred passwords taken from a small set of machines." [163]

it has developed a third, specialized sense that refers to assault recipes. This sense may have originated in contexts that explicitly framed the search for attacks as a role-playing exercise[3]—but authors have long employed it without this qualification[4]. Today this sense sees widespread use (though some writers still consider it nonstandard), and it has been added as a second definition in some respected security lexicons[5]

As a result, when researchers say they are describing a new attack, this usually means something specific: that they have discovered a previously undocumented security flaw (or a new way to utilize a known one) and documented it, that they have devised a way that an attacker could leverage the vulnerability to cause harm, and often that they have confirmed that the attack works in real systems by constructing a proof-of-concept exploit.

The notion of attacks as assault recipes finds widespread use because it provides valuable structure for research activities. It frees us to reason abstractly about potential dangers, yet it restrains our speculation by forcing us to maintain concrete ties to the harms we wish to prevent. It provides a conceptual framework for the study of security failures, defining the objective (to discover such attacks), the level of abstraction (not particular assaults or general brittleness, but exploitable weaknesses), and the standard of evidence by which results will be validated (demonstrable exploitability).

Returning to the five questions we mentioned earlier, we find that, as a result of this concept, there is general agreement about how to answer four of them with respect to the direct study of attacks. What question should we ask?—How can systems actually be assaulted. How should we report the answers?—By describing specific recipes for assault. How should we evaluate whether purported answers are correct?—By whether they can

3 Morris and Thompson in 1979: "To help develop a secure system, we have had a continuing competition to devise new ways to attack the security of the system (the bad guy) and, at the same time, to devise new techniques to resist the new **attacks** (the good guy)." [116]

4 Bellovin in 1989: "We describe a variety of **attacks** based on these flaws [in TCP/IP], including sequence number spoofing, routing attacks, source address spoofing, and authentication attacks." [17]

5 It it absent in the first edition of the Internet Security Glossary [155] but does appear in the second edition [156].

be demonstrated on real systems. How should we value answers to this question? — We recognize novel attacks as a substantial contribution. One question remain unanswered — How should we discover attacks? — and we will return to it later.

Security research today lacks any parallel organizing concept to answer these questions with respect to drawing broader lessons from failures. This is a lost opportunity. It results in the huge variation in the terminology, approaches, and quality of lesson drawing that we have observed in contemporary studies. Thus, we will now propose such an organizing concept, and in subsequent chapters we will demonstrates its application.

1.3 The Analytic Approach

The concept we propose is the *informative cause*. In investigating failures, researchers should seek to establish the informative causes that contributed to them. Mirroring the notion of the attack, this new framework encompasses a goal, a level of abstraction, and a standard of evidence. We now define it and explain how it provides an organizing framework that structures the process of deriving broader lessons from security failures.

First, the goal. Just as we usually attempt to understand security failures in terms of attacks, our preferred starting point for understanding the broader implications of failures should be understanding the causes — factors that made the failures happen or that made them more severe. The justification of focusing on causes is pragmatic: the concept is flexible enough to restate many of the kinds of lessons described earlier, but, we will argue, restrictive enough to focus our investigations and provide a standard of evidence. There are many types of causes that underlie security failures, but we can classify them roughly into two groups: human factors and technological factors. Human factors include *counter-secure incentives* that motivate developers or testers to invest less heavily in security, or that motivate attackers to invest more heavily in breaking it; *misleading intuitions* on the part

of developers or users about the nature of threats or the value of defenses; and *dangerous practices* or engineering patterns that carry with them a higher probability of security failure. Technological factors include those relating to the *nature of the problem* that the system attempts to solve—some problems seem inherently more difficult to solve securely—; those relating to the *brittleness of the technique*—some techniques are more prone to weakness—; and those relating to the behavior of the supporting technologies on which the security of the system depends.

Second, the level of abstraction. Just as the notion of attacks as assault recipes focuses investigations on the technical causes of failure at a particular useful level of abstraction, the notion of informative causes serves to focus investigations concerning the implications of failures on the level of abstraction most useful for lesson drawing. By requiring that the causes sought be *informative*, we mean that they should be generalizable but also actionable.[6] Causes are noninformative when they are too specific. The immediate cause of a failure might be a particular bug or design flaw, but deriving useful lessons requires following the chain of causality to find reasons why the mistake was made, why it went undiscovered, or why it resulted in harm. On the other hand, causes that are too general are not informative either. If we follow the chain of causality too far, we arrive at uninformative conclusions. All humans make mistakes, all large programs contain bugs, entropy of systems increases over time; while true, we already know these things, and restating them again is of little use for guiding action and improving security. Informative causes, the causes that teach us the most, lie at an intermediate level of abstraction.

Finally, the standard of evidence. The notion of attacks as assault recipes implies a simple test by which we can tell whether a purported attack is correct: does the recipe work against the systems that are claimed to be insecure? Likewise, to demonstrate an

6 Causation is a complex concept, and other areas such as law [161] have developed rich and subtle theories to deal with it. We might compare our notion of the informative cause to the legal concept of the *proximate cause*.

informative cause, investigators must establish a *causal* relationship between it and the failures at issues. This has the effect of constraining our speculation about causes and maintaining a concrete connection between the lessons drawn and the facts of the failures under investigation. One way to establish causality is to argue that if a factor or set of factors were not the case, the problem would not have occurred (or would have been less severe). Doing so at an informative level of abstraction can be difficult, since there may be many plausible causes, some of which are merely coincidental. One approach is to look for similarities and differences in related failures. Such investigations may be latitudinal in scope, closely examining one system and searching for patterns of failures, or longitudinal, searching for similar problems in related systems.

Returning to our five questions, we see that the notion of informative causes, like the notion of attacks as assaults, answers four of them. What questions should we ask? — We should seek factors that made security failures possible or made them worse and that are general enough to guide future action. How should we report the answers? — By detailing specific causes and supporting them with evidence, such as patterns of failures. How should we evaluate whether purported causes are correct? — By judging the strength of the causal connection. How should we value answers to these questions? — Focusing on causes emphasizes that they are a valuable contribution.

Neither the concept of the attack nor our notion of the informative cause answers the last question — How should we discover attacks/causes? The process of finding attacks is usually a creative rather than mechanical exercise. In general, it involves understanding the system's operation across all layers, identifying areas that may be prone to weakness, identifying critical assumptions in these areas, and attempting to violate these assumption. There are many parallels between finding attacks and finding causes. Both involve a directed search through a broad space of possibilities, guided by past experience, and both

involve the creative formulation of hypotheses—original attacks and causes—which are then tested against the facts. It seems likely that the search for causes, like the search for attacks, is inherently resistant to systematization beyond a certain point. The value of both concepts stems from the structure they provide, not from their ability to provide a precise investigatory roadmap.

Combining the notion of attacks as assault recipes with the notion of informative causes gives us a new template for studies of security failures. The most critical change is that it places understanding the broader implications of failures on an equal footing with discovering attacks, highlighting its importance for improving security. Studies in this framework might involve the following components:

1. Investigating the security of a system or a group of related systems with the goal of discovering new attacks.
2. Performing analyses looking for patterns of failures (longitudinally and/or latitudinally) and other factors that reveal informative causes.
3. Discussing the implications of the attacks and causes, drawing further lessons, making predictions, and proposing mitigations.

Shifting the emphasis of failure investigations to co-emphasize attacks and their causes calls for a different term to denote such work, rather than simply "attack research." We propose *analytic* security research. *Analytic* derives from the Ancient Greek ἀνάλυσις (*analusis*), and in turn from ἀναλύω (*analuō*), meaning "I unravel, investigate" [1]. This seems appropriate, since analytic security research begins with complete systems and takes them apart in search of attacks and their causes. The term naturally meshes with the existing lexicon. Some security researchers draw a distinction between "synthesis" (system design) and "analysis" (attack investigation), and code-breaking has long been called "cryptanalysis."

Analytic security research is an approach to investigating security failures that couples the organizing concept of the attack with the organizing concept of the cause. It thus provides an intellectual template for studying security failures in terms of both their direct and broader lessons.

1.4 In this Dissertation

This dissertation contributes by studying security failures and their causes in three classes of systems: digital rights management systems for audio compact discs (CD-DRM systems), direct-recording electronic (DRE) voting machines, and on-the-fly hard disk encryption software. For each class, we apply the analytic security research methodology. We analyze several deployed implementations and identify previously unknown vulnerabilities that pose significant threats to the systems' security. We compare these security failures longitudinally within each implementation and latitudinally across implementations from the same group to inquire into the causes of the failures and to derive broader lessons and implications. We then recommend mitigations both for the specific vulnerabilities and for the deeper problems that germinated them.

Our first case study concerns security failures in CD-DRM systems (Chapter 2), copy protection systems intended to control the use of music distributed on compact discs. We trace the development of this technology through three generations and compare implementations from several vendors. Examining these systems, we discover three classes of attacks: content copying attacks, which defeat the copy protection; protection cloning attacks, which allow music distributors to take advantage of the protections without paying; and collateral damage, side-effects caused by the DRM that weaken the security of end-users' PCs. By analyzing these failures, we find three groups of causes that contributed to them or amplified their severity: the intractable nature of the CD copy protection problem,

mismatched incentives between CD-DRM vendors and their customers, and fundamental tension between DRM and PC security. These findings carry broader lessons for security beyond the context of DRM.

The attacks on first- and second-generation CD-DRM schemes originally appeared in "Evaluating New Copy-Prevention Techniques for Audio CDs," which was presented at DRM 2002 [72], and in "Analysis of the MediaMax CD3 Copy-Prevention System," a technical report [73]. The attacks on third-generation schemes and portions of their analysis originally appeared in "Lessons from the Sony CD-DRM Episode," joint work with Ed Felten, which was presented at USENIX Security 2006 [74]. Additional analysis in Section 2.5 derives from "Digital Rights Management, Spyware, and Security," also joint work with Ed Felten, which first appeared in *IEEE Security and Privacy* [63].

Our second case study concerns security failures in direct-recording electronic (DRE) voting machines (Chapter 3). We analyze the security of two generations of systems from the same manufacturer, the Diebold AccuVote-TS and AccuVote-TSX. We uncover weaknesses that enable three classes of attacks: attacks on the integrity of the election results, attacks on the secrecy of voters' ballots, and attacks on the availability of the voting system. All three could be carried out on a wide scale by a malicious party with minimal access to the voting system — perhaps only temporary access to a single machine or memory card. We find several mechanisms by which malicious software could spread from a point of infection to compromise other voting machines and central election administration systems, and we demonstrate this attack by constructing a vote-stealing virus. Comparing our results with the findings of other studies that examined voting machines from different manufacturers, we find evidence that vulnerabilities like these are widespread in electronic voting system designs. This trend suggest several informative causes for e-voting security failures, including market incentives that push manufacturers to prioritize features and time-to-market

over security and testing. We discuss the implications of these problems and conclude that new approaches to involving technology in the election process are necessary.

Chapter 3 incorporates attacks and analysis of the Diebold AccuVote-TS that originally appeared in "Security Analysis of the Diebold AccuVote-TS Voting Machine," joint work with Ari Feldman and Ed Felten, which was presented at EVT 2007 [61]. It also incorporates attacks and analysis of the AccuVote-TSX that originally appeared in "Source Code Review of the Diebold Voting System," joint work with Joe Calandrino, Ari Feldman, David Wagner, Harlan Yu, and Bill Zeller, which was part of the voting systems review commissioned by California Secretary of State Debra Bowen [28].

Our final case study is about security failures in on-the-fly disk encryption software (Chapter 4), which is commonly used to protect sensitive data against physical attacks such as laptop theft. We investigate memory remanence, a physical phenomenon, little known in the security community, that causes data stored in a DRAM to decay gradually when power is lost rather than being immediately erased. Our findings indicate that residual memory contents can often be recovered after seconds without power, and after minutes if the memory device is first cooled. An attacker could exploit this phenomenon to steal encryption keys, such as those used in disk encryption applications, by quickly cutting power to the machine, then rebooting and running memory imaging tools like ones we have developed. Some amount of decay may be unavoidable in certain attack scenarios, but we have found algorithms for locating keys in partially decayed memory and for correcting the errors. We demonstrate that these attacks compromise the security of five widely used disk encryption utilities, and we suspect that most similar software is also vulnerable. We identify several causes that underlie these failures, including deficiencies in common PC architectures, which provide no safe place for software to store secrets that are in active use, and the role of abstraction in computer engineering, which had the effect of concealing

security-critical details about computer operation from disk encryption designers. These causes provide motivation for changes to memory architecture and other aspects of PC design. They also imply that security problems may arise in other settings where abstractions mask complex behavior.

With the exception of Section 4.7, the results in Chapter 4 first appeared in "Lest We Remember: Cold Boot Attacks on Encryption Keys," which was presented at USENIX Security 2008 [75], and in an extended abstract by the same name that appeared in the May 2009 issue of *Communications of the ACM* [76]. Both are joint work with Seth Schoen, Nadia Heninger, Will Clarkson, Bill Paul, Joe Calandrino, Ari Feldman, Jake Appelbaum, and Ed Felten.

The findings in this dissertation have already had significant real-world impacts. Our work on CD-DRM led to the recall of millions of CDs containing dangerous copy protection software, lawsuits and government investigations into the problems, and it may have contributed to a shift in the music industry's business strategy away from the use of the DRM for music purchases. Our initial voting machine study prompted a number of states to conduct security evaluations of election technology, revise their election procedures, or abandon the use of DRE voting machines altogether. Our work on disk encryption changed the security community thinks about the threat model for physical attacks on memory, prompted users who rely on this technology to reassess the risks they face, and even entered popular culture in dramatizations on two prime-time television programs.

In revisiting these works, we have adapted them to apply our new analytic methodology. Our revisions highlight the value of placing emphasis on both attacks and causes. Lessons scattered throughout the papers now come together in one place, a section describing causes, making them more readily accessible to the reader. Lessons that were implied but not expressly stated are made more explicit by the added structure. The practice of

basing lessons on causes forced us to draw crisper, concise statements of each cause, and it exposed ambiguities in some of the lessons that we drew before, prompting us to refine them. Perhaps most interestingly, the consistent approach now applied across the three case studies highlights causal factors that were common across different classes of systems, such as the role of misleading intuitions about system behavior, the impact of vendor incentives on investment in security and testing, and the disruptive impact of technological change on users' understanding of risks. In future investigations we hope to more fully explore these common causes and their implications for security.

Of course, this exercise is only an initial step in the development of our proposed methodology. Validating and refining it will require significantly more practice, and, in particular, applying it from the start in new investigations rather than retrofitting it to past work. Significant questions remain. Does it provide too much structure, confining our lesson drawing, or does it provide too little structure to make a meaningful impact on the quality of the results? Is there more than can be done to systematize the search for causes? Can we more rigorously define causes and provide a stricter standard by which to judge them? We hope to shed light on these and other questions as we apply the analytic approach to future studies.

1.5 Funding Acknowledgments

This material is based in part upon work supported under National Science Foundation Graduate Research Fellowships. Any opinions, findings, conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reflect the views of the National Science Foundation.

Portions of this research were performed under an appointment to the Department of Homeland Security (DHS) Scholarship and Fellowship Program, administered by the

Oak Ridge Institute for Science and Education (ORISE) through an interagency agreement between the U.S. Department of Energy (DOE) and DHS. ORISE is managed by Oak Ridge Associated Universities (ORAU) under DOE contract number DE-AC05-06OR23100. All opinions expressed in this dissertation are the author's and do not necessarily reflect the policies and view of DHS, DOE, or ORAU/ORISE.

This work was supported in part by the Charlotte Elizabeth Procter honorific fellowship.

Chapter 2

Security Failures in CD-DRM Systems

Compact disc digital rights management (CD-DRM) is a form of copy protection designed to restrict the use and dissemination of music stored on audio CDs. This chapter presents an analytic security study of CD-DRM systems. We analyze the security of six implementations from four different vendors, examining how each of them was intended to work and how they all ultimately failed, and we investigate the causes of these failures. Many of the lessons that emerge are valuable not only for CD-DRM but for DRM generally and for other security applications.

We begin by introducing a taxonomy for CD-DRM systems (Section 2.1). We classify them into three generations: *passive protection*, *active protection*, and *aggressive protection*. This framework allows us to trace the technology's evolution from its inception in 2001 through its demise in 2005. Our analysis reveals how each generation was developed in response to problems with the previous generation's security mechanisms but ended up introducing problems of its own.

Next we evaluate the systems' security, describe attacks against them, and explore possible defenses. We divide the attacks into three categories, each involving a different attacker and victim. The first are *content copying attacks* (Section 2.2), which allow CD owners

Gen.	Year	Method	Implementations	Common Vulnerabilities		
				Copying	Cloning	Collateral
1st	2001	Passive	Midbar CDS-200 Sony Key2Audio SunnComm MediaCloQ	X		
2nd	2003	Active	SunnComm MediaMax 3	X	X	
3rd	2005	Aggressive	First4Internet XCP SunnComm MediaMax 5	X	X	X

Table 2.1—Taxonomy of CD-DRM Systems We classify CD-DRM implementations into three generations based on the techniques they use to prevent copying. This table lists the systems we studied from each generation and the kinds of attacks we found.

to bypass the usage restrictions established by content distributors, negating CD-DRM's *raison d'être*. The systems from all three generations were subject to such attacks. The second group, *protection cloning attacks* (Section 2.3), allow disc producers to benefit from a CD-DRM platform without paying the developers. Both the second- and third-generation systems were susceptible. The final group of threats involve third-party attackers, who, rather than targeting CD-DRM systems directly, exploit weaknesses in them to attack CD owners' computers; we call these *collateral damage* (Section 2.4). Only the third-generation systems we studied exhibited this kind of failure.

Armed with an understanding of these attacks, we conduct an analysis in search of underlying causes that induced or aggravated them (Section 2.5). Our examination traces security failures both latitudinally across contemporary systems and longitudinally across generations. Causes we identify include the nature of CD-DRM's copy protection goal, which faces an uphill battle against Moore's law; the incentives of CD-DRM developers, who underinvest in protecting users due to the structure of outsourcing arrangements; and

inherent characteristics of DRM software, such as tension between content security and user security. The causes we identify predict that other kinds of DRM software will cause security problems. We conclude by summarizing the implications of our findings and pondering the ultimate fate of CD-DRM technology (Section 2.6).

2.1 CD-DRM Technologies

CD-DRM systems must meet difficult requirements. Copy-protected discs must be reasonably compliant with the CD digital audio (CDDA) standard so that they can play in ordinary CD players. They must be unreadable by almost all computer programs in order to prevent copying, yet the DRM vendor's own software must be able to read them in order to give the user some access to the music. Different CD-DRM vendors addressed these challenges in different ways that evolved hand-in-hand with changes in the marketplace and the development of new attacks. In order to make sense of the resulting array of systems, we group them into three technological generations based on the kinds of protection mechanisms they employed (see Table 2.1).

2.1.1 First Generation: Passive Protection

The first generation CD-DRM systems were released around 2001. They employed intentional deviations from the CDDA specification that were designed to cause errors when discs were read with PCs [72]. We refer to these methods as *passive protection* techniques.

Understanding how passive protection works requires some background about CD data formats and their history [107]. The CDDA format was invented by Sony and Philips in the late 1970s. It was extended in the early 1980s with the compact disc read-only memory (CD-ROM) standard, which provided for discs that could store data as well as audio. Recordable (CD-R) and rewritable (CD-RW) disc formats were introduced in the late 1980s and early

1990s by replacing the pitted aluminum in regular CDs with specialized dyes that could be marked by low-power lasers.

The information stored on a CD is organized into functional units called *tracks*. An audio CD usually contains one audio track for each song. The CD-ROM format allows discs to mix audio tracks with data tracks storing arbitrary files, and introduces a track type flag to distinguish between the two. A header before the first track holds a *table of contents* (TOC), which specifies the number of tracks, their starting positions, and whether each contains audio or data [53, 85].

Older standalone CD players are unaware of the data CD standards and interpret all discs according to the CDDA format. If a CD-ROM is inserted into one of these players, the player emits static, since it does not recognize that the tracks have been marked as data and treats the raw bytes as audio samples. Many passive protection schemes exploit such behavioral differences between computer CD drives and CD players. For example, the MediaCloQ system deliberately mismarks every audio tracks as data. Standalone CD players often ignore this designation and play the tracks normally, but most CD ripping applications recognize it and refuse to process them as music [72].

The CD-R and CD-RW writable disc formats have more complicated structures that provide other opportunities for distinguishing between standalone players and drives. CD-R media cannot be erased, so the standards allow data to be written incrementally until the disc is filled. One way to do this is to write several *sessions*, each with its own header and tracks; discs encoded in this way are called *multisession* CDs. Every session has its own TOC that describes the tracks it contains. A new TOC field points to the beginning of the TOC from the previous session. Most CD drives treat the TOCs as a linked list, starting with the *last* session TOC and iteratively following the links to the previous ones [121]. However,

audio CD players and older CD-ROM devices that do not recognize the multisession format read just the TOC from the *first* session and only see tracks listed there.

Many passive protection schemes exploit these differences to make audio tracks inaccessible from CD drives. One widespread technique (used, e.g., by CDS-200 and Key2Audio) involves creating a disc with two sessions but leaving out the TOC entry that would normally link them. CD drives read only the second session TOC, which on these discs omits the audio tracks or contains corrupted entries for them, while CD players read only the first, accurate TOC and play the disc correctly [72].

In addition to TOC errors, copy prevention schemes may place errors in the track data area, either in the audio samples, in error-correcting bits, or in other metadata. For instance, CD-DRM vendor Midbar holds a patent describing how to inject corrupt audio samples while concealing them from audio CD players using CDDA metadata [157]. Other proposed techniques involve writing corrupt audio samples along with incorrect error correcting codes to simulate scratches on the disc. These errors are unrecoverable, so audio CD players interpolate over them, but older CD drives designed primarily for data access do not support interpolation and return faulty samples instead [72].

As passive protection became more broadly adopted, users increasingly reported problems playing protected discs in devices such as DVD players, video game systems, and newer car CD players [72]. Often these devices are powered by general purpose CPUs and are architecturally more similar to PCs than to older ASIC-based CD player hardware; they sometimes contain the same CD drive hardware found in PCs. As one might expect, there were fewer behavioral differences between these devices and PCs, making it harder for passive protection systems to accurately target computers. The growing compatibility problems that resulted forced CD-DRM designers to turn to a radically different technique [73].

2.1.2 Second Generation: Active Protection

Around 2003, a second generation of CD-DRM systems emerged. These designs are marked by a totally different protection mechanism: rather than relying on PCs to behave differently from CD players, they install and execute software on the computer to block read requests from other programs. We call this mechanism *active protection*.

Active protection software must be installed on the computer somehow. This is usually accomplished with the Windows AutoRun feature, which (when enabled) automatically executes software from a disc when it is inserted into the computer's drive. AutoRun lets the DRM vendor's software run or install immediately after the user loads the CD.

Once installed, every time a new CD is inserted the CD-DRM software runs a recognition algorithm to determine whether the disc is associated with the same DRM scheme. If it is, the active protection software interferes with all read requests to audio portion of the disc, except for those originating from the vendor's own music player application. This proprietary player application, which is shipped on the disc, gives the user limited access to the music while enforcing an access policy specified by the record label [73].

The first widely adopted second generation system was SunnComm's MediaMax 3. MediaMax 3 leaves the standard CD audio portion of the disc unprotected, relying entirely on its active protection software to block PC-based copying. If a disc containing MediaMax 3 is placed in a PC with AutoRun enabled, Windows runs a file on the disc called `LaunchCD.exe`, which installs a new Windows service called `SbcpHid`. The service interposes between the operating system kernel and drivers for CD drives. It examines each CD placed in the machine, and when it recognizes a title protected with MediaMax, interferes with attempts to read the audio tracks by introducing random skips and pauses [73].

When AutoRun first starts the MediaMax 3 software, it presents an end user license agreement (EULA). Before displaying the EULA, MediaMax installs the `Sbcphid` service in a

temporary fashion, with its startup-type parameter set to "Manual." This protects the disc during the current session, but the service will not reactivate after the computer shuts down. If the user accepts the EULA, MediaMax installs the software permanently by changing its startup type to "Auto" so that it reloads on each subsequent boot [73]. This design appears to be a compromise between DRM security requirements and legal concerns about installing software without consent (and without the additional protections granted to the developers by the license agreement).

Second generation techniques like MediaMax 3 offer only weak protection, because, as we will discuss in Section 2.2, users can easily disable their active protection components or block them from installing in the first place [73]. Pressure to close these holes eventually spawned third generation CD-DRM techniques.

2.1.3 Third Generation: Aggressive Protection

The third generation of CD-DRM systems were introduced in 2005. Like second generation schemes they employ software-based protection, but they are distinguished by aggressive countermeasures against attempts to block or disable this active protection. The most notable examples of these *aggressive protection* schemes are SunnComm's MediaMax 5 and First4Internet's Extended Copy Protection (XCP).

Sony deployed XCP on 52 titles (representing more than 4.7 million CDs) [36]. We examined three of them in detail: Acceptance, *Phantoms* (2005); Susie Suh, *Susie Suh* (2005); and Switchfoot, *Nothing is Sound* (2005). MediaMax was deployed on 37 Sony titles (over 20 million CDs) as well as dozens of titles from other labels [36]. We studied three albums that used MediaMax version 5—Peter Cetera, *You Just Gotta Love Christmas* (Viastar, 2004); Babyface, *Grown and Sexy* (Arista/Sony, 2005); and My Morning Jacket, *Z* (ATO/Sony, 2005).

MediaMax 5

MediaMax 5 employs active protection software similar to that used by MediaMax 3, but it takes additional steps to head off attempts to block the software from installing.

CD-DRM systems have few defenses against users who permanently disable AutoRun before using any protected discs. However, AutoRun is enabled by default (in Windows versions prior to Vista), and many users will not be aware that they own a CD-DRM album until after they try unsuccessfully to rip or copy it. As a result, the software on the CD will probably be invoked at least once before the user can discover it and begin blocking AutoRun. MediaMax 5 uses this window of opportunity to permanently establish itself on the system.

MediaMax 5's installation behavior reflects a different balance between security requirements and legal concerns. It differs from MediaMax 3 in two ways. First, it permanently copies almost 12 MB of programs and data onto the user's system, including the active protection service, *before even displaying the EULA*; these files remain installed even if the user rejects the agreement. Second, under some common circumstances MediaMax 5 installs its active protection service permanently even if the user always rejects the EULA. This behavior is triggered when:

- A user who previously inserted a MediaMax 3 album inserts a MediaMax 5 album,
- A user who previously inserted a MediaMax 5 album inserts a MediaMax 3 album, or
- A user who previously inserted a MediaMax 5 album and eventually rebooted inserts the same album or another MediaMax 5 album.

These steps do not have to take place in a single session (though the first two can). Weeks or months might elapse between using the first and second disc.

This behavior might be an unintentional bug in code that was intended to upgrade the active protection software if an older version was consensually installed, but it could also be explained by a deliberate attempt to exploit the window of opportunity while AutoRun is enabled. Even if poor testing is the explanation for *activating* the software without consent, it is clear that SunnComm deliberately chose to *install* the MediaMax 5 software on the user's system even if the user did not consent. These decisions are difficult to reconcile with the ethical and legal requirements on software companies, but they are easy to reconcile with the vendor's strategic interests—strengthening security pleases their rights-holder customers, while placing their software on as many computers as possible builds a potentially lucrative platform. We discuss this platform building strategy in detail in Section 2.3.

XCP

XCP combines active and passive protection methods for a multilayered defense. It contains the most notorious example of aggressive protection behavior, a rootkit-like mechanism intended to block attempts to remove its active protection software.

XCP's active protection software consists of a pair of filter drivers called `crater.sys` and `cor.sys` that attach to the CD-ROM and IDE devices [147]. It examines each disc that is inserted into the computer to determine whether access should be restricted. If the disc is recognized as copy protected with XCP, the drivers monitor for attempts to read the audio tracks, as would occur during a playback, rip, or disc copy operation, and replace the audio returned by the drive with random noise.

When XCP installs its active protection software, it also installs a second program—essentially a rootkit—that conceals any files, processes, or registry keys with names beginning with a particular prefix, `$sys$` [147]. The result is that XCP's installation directory,

registry keys, files, and processes become invisible and inaccessible from normal programs and administration tools.

The rootkit is a kernel-level service named `$sys$aries` that is configured to automatically load early in the boot process. When the rootkit starts, it hooks several Windows system calls by modifying the system service dispatch table (the kernel's `KeServiceDescriptorTable` structure), which is an array of pointers to the kernel functions that implement basic system calls. The rootkit changes five of these addresses to point to functions within the rootkit, so that calls to the patched system calls are handled by the rootkit rather than the original kernel functions. These functions are `NtQueryDirectoryFile`, `NtCreateFile`, `NtQuerySystemInformation`, `NtEnumerateKey`, and `NtOpenKey`. In each case, code in the rootkit calls the real kernel function with the same parameters and then filters the results to remove entries marked with the trigger prefix before returning them to the original caller. Before the filtering step, the rootkit checks whether the name of the calling process begins with the magic `$sys$` string; if so, the rootkit returns the unadulterated results from the real kernel function. This allows XCP's own processes to have an accurate view of the system.

Beyond protecting its active protection software, XCP revives the idea of passive protection, incorporating a mild form of it for an added layer of content security. Like most of the first-generation passive protection schemes, XCP's technique exploits a quirk in the way different systems handle multisession discs. Some commercial discs use a variant of the multisession format to combine CD audio and computer accessible files on a single CD. These discs adhere to the Blue Book or "stamped multisession" format [13]. According to the Blue Book specification, stamped multisession discs must contain two sessions: a first session with 1–99 CD audio tracks, and a second session with one data track. The Windows CD audio API contains special support for Blue Book discs, which presents the CD to player

and ripper applications as if it were a normal audio CD. Windows treats other multisession discs as data-only CDs.

XCP discs deviate from the Blue Book format by adding a second data track in the second session. This causes Windows to treat the disc as a regular multisession CD, so the primary data track is mounted as a file system, but the audio tracks are invisible to player and ripper applications that use the Windows CD audio API. This includes Windows Media Player, iTunes, and most other widely used CD applications. This passive protection provides some degree of defense even if the user declines to install the active protection service, deactivates it, or permanently disables AutoRun.

Authorized Players

Increasingly, PCs and portable playback devices that attach to them are users' primary means of organizing, transporting, and enjoying their music collections. The second- and third-generation CD-DRM vendors recognized this trend. Rather than inhibit all use with PCs, as some earlier schemes do [72], XCP and MediaMax provide their own proprietary media players, shipped on each protected CD, that allow certain limited uses of the music subject to restrictions imposed by the music label.

The XCP and MediaMax players launch automatically using AutoRun when a protected disc is inserted. Both players have similar feature sets. They provide a rudimentary playback interface that allows users to listen to protected albums, and they allow access to "bonus content" such as album art, liner notes, song lyrics, and links to artist web sites. The players bypass the copy protection to access music on the disc by using a special backdoor interface in the active protection software.

XCP and MediaMax 5 both permit users to burn copies of the entire album a limited number of times (typically three). These copies are created using a proprietary burning

application integrated into the player. The copies include the player applications and the same active (and passive, for XCP) protection as the original album, but they do not allow any subsequent generations of copying. [74]

Another feature of the player applications allows users to rip the tracks from the CD to their hard disks, but only in DRM-enabled audio formats. Both schemes support the Windows Media Audio format by using a Microsoft product, the Windows Media Data Session Toolkit [113], to deliver DRM licenses that are bound to the PC where the files were ripped. The licenses allow the music to be transferred to portable devices that support Windows Media DRM or burned onto CDs, but the files will not be playable if they are copied to another PC.

2.2 Attacks: Content Copying

All CD-DRM systems share the same principal goal: restricting particular uses of audio content. We find that all three generations are vulnerable to attacks that allow disc owners to bypass these restrictions with ease.

Any evaluation of such attacks must consider the record labels' objectives in deploying CD-DRM. They would ideally like to prevent the music from being made available on peer-to-peer (P2P) file sharing networks, but this goal is not feasible [18]. If even one user can rip and upload an unprotected copy, it will be available to the whole world; in practice, every commercially valuable song becomes available in this way immediately upon its release, if not sooner. No DRM system can achieve this level of protection.

The record labels surely recognize this (and sometimes acknowledge it in public), so we assume their actual goals are more realistic: erecting barriers to disc-to-disc copying and other local uses of the music.[1] Henceforth we will call this the *modest protection goal*. They

1 Another content industry goal might be to invoke legal protection under 1201(b) of the DMCA, which essentially prohibits trafficking in products that circumvent copy protection measures. Using even a weak

most likely believe that some people who are deterred from making their own copies will buy copies instead. By controlling other local uses, such as transferring songs to an iPod, the labels may hope to be able to charge for these capabilities à la carte. How well do CD-DRM systems satisfy these more modest goals? The answer depends on the cost and difficulty of attacking them. We now evaluate several content copying attacks in these terms.

Every CD-DRM system we have studied suffers from vulnerabilities that facilitate attacks by users who want to copy the music illegally or who want to make uses allowed by copyright law but blocked by the DRM. The user can defeat passive protection by any of several means, block active protection software from installing, disable installed active protection software, capture music from the DRM vendor's authorized player application, or uninstall the protection software, among other methods. Many of these circumventions can be conducted easily by a user of average technical skill, defeating even the industry's modest protection goal.

We consider attacks against passive protection measures in Section 2.2.1 and against active protection measures in Section 2.2.2. Since our class of aggressive CD-DRM schemes employ active and/or passive security measures, we discuss them in both subsections. Finally, in Section 2.2.3, we focus on attacks that circumvent protection measures in the authorized player software included in some CD-DRM systems.

2.2.1 Attacks Against Passive Protection

Passive protection schemes exploit differences between the way computers interpret the CD format and the way CD players do. These differences most often stem from deficiencies in computer software and hardware, such as programming errors, design limitations, or insufficiently robust error handling.

---

form of DRM could allow the industry to control the market for products that interoperate with their content. Products made without the industry's cooperation would have to defeat the DRM, and selling them would potentially violate 1201(b).

Initially, some passive protection schemes were quite effective. Soon after the first generation schemes were released, our testing found that they consistently prevented copying with the most popular consumer ripping and copying software [72]. Yet over time, this effectiveness declined as drive and software vendors improved their products. Today, some users experience no effects from the schemes, while many others are able to bypass them at no expense by downloading new or updated software. There are at least three ways to bypass passive protection: replacing CD drives with hardware that does not suffer from the deficiencies employed by the CD-DRM, using updated or substitute software programs that work around the protections, and modifying the discs to obscure protection-related data areas.

Using Robust Hardware

A disc owner who wants to bypass passive protection first needs compatible hardware. Some CD drives are particularly fragile, with buggy firmware that locks up when presented with CD-DRM discs that utilize malformed TOC entries [72]. Better drives handle corrupted data gracefully whether in headers like the TOC or in the audio stream itself. Either they repair the problems on-the-fly or they provide the calling application with a detailed error report to allow it to take corrective action. These behaviors are not specific to copy-prevention systems—rather, they improve performance with all damaged or poorly recorded discs [72].

Though uncommon when the first passive protection schemes were introduced, increased drive robustness has since become the norm, and many drives are now specifically optimized for audio extraction. Nonetheless, this attack does not entirely frustrate the music industry's modest protection goal, since users with older drives may find upgrading their hardware too inconvenient or expensive.

Using Different Software

While the only way to fix an incompatible drive might be to replace it, repairing a deficient application can be as easy as downloading a patch; this makes most software-based attacks on passive protection easy and cheap, defeating the industry's modest protection goal. Software makers can be expected to quickly patch any problems exploited by passive protection schemes if they impact a significant number of users. Changes targetted at defeating particular CD-DRM schemes might run afoul of the DMCA, but as with hardware, the most important improvements for software are general-purpose, such as increased robustness and better modes of failure. For maximum compatibility, CD reading and copying programs can be modified to detect and correct data errors and to recover gracefully even when certain areas of a disc are unreadable.

One effective software countermeasure against passive protection schemes that use erroneous TOC information is to ignore the TOC entirely, and instead to derive a table of contents directly from the audio. Analyzing regions of the disc shows whether they contain audio samples or are transitions between songs, so a simple binary search can reveal where each track begins and ends. This approach combined with improved error correction would defeat nearly all first generation schemes [72].

As early as 2002, many ripper programs had already adapted to passive protection. Feurio 1.64 added special routines for handling defective CDs [65], and EAC 0.9x could detect CD structure using a variation of the binary search method described above [57]; both already supported extended error correction mechanisms. Version 4.0 of the CloneCD copying software added a special mode for audio CDs, which greatly improved its success rate in our tests compared to earlier releases. Coupled with hardware that is compatible with the discs, these programs were able to bypass the content protection features of the CDS-200, Key2Audio, and MediaCloQ schemes [72].

XCP, a third-generation CD-DRM scheme, uses mild passive protection as a secondary defense. It is vulnerable to a variety of attacks involving the use of substitute software. Since this protection mechanism is specific to the Windows CD API, one trivial work around is to use a non-Windows platform, such as Mac OS X or Linux, that interprets the CD format more robustly. Another approach is to use advanced ripping and copying applications that avoid the Windows CD API altogether and issue commands directly to the drive. Widely available programs such as Nero [122] and Exact Audio Copy [179] use this approach and are able to read XCP audio tracks (assuming the active protection is also defeated) [74].

Modifying Protected Discs

Most of the protection schemes that we studied make use of the multisession CD format described in Section 2.1.1. Typically, these schemes store the audio tracks in the disc's first session, which is formatted like a normal audio CD in order to ensure compatibility with standalone CD players. They also include a second session with a TOC header that contains misleading or corrupted descriptions for the audio tracks or other deviations from CD standards. CD players typically do not understand the multisession format and fall back to reading only the first session. Multisession-aware CD drives encounter the erroneous data in the second session, which by various means renders the audio tracks unreadable [72].

This mechanism suffers from a very powerful attack that is easy to carry out, costs practically nothing, and defeats the modest protection goal for nearly all CD-DRM schemes, active and passive. A user can physically modify the CD so that the area where the second session is stored is unreadable. This can be accomplished in a reversible manner using adhesive tape [72], or more permanently using a felt-tipped marker [176]. Most CD drives, when unable to read the second session, fall back to reading the first session alone, just as CD players do. They then have full, unprotected access to the audio.

The only nontrivial part of this attack is determining where to place the tape or markings. CDs record data in a continuous spiral beginning near the central hole, so the second session is stored in a ring towards the outside of the disc [107]. An attacker could proceed by trial and error, starting at the outside edge and obscuring progressively greater portions of the surface until the audio becomes readable. A shortcut may also be available: many discs exhibit a visible band between the sessions. This region, the lead-out and lean-in area, stores a repeated pattern of bits that causes different reflectance than the high-entropy audio samples stored elsewhere, and can show the attacker exactly which area to obscure [72].

The CDS-200 and Key2Audio schemes are vulnerable to this attack, as are MediaCloQ discs when used with certain CD drives [72]. The passive protection mechanism used by XCP can also be defeated in this way [74], as can all active protection schemes, which depend on installing software stored in the second session.

2.2.2 Attacks Against Active Protection

Active protection methods install and run software components that interfere with accesses to the audio portion of the CD. They are vulnerable to attacks that block the software from installing and ones that later deactivate it. Either kind of attack can be executed easily and at low cost, effectively defeating the modest protection goal.

Preventing Installation

The installation phase is the Achilles' heel of every active protection scheme. Users have practically no incentive to install the active protection software, since its primary purpose is to reduce their access to the content. Therefore, the schemes rely on the operating system to launch the software automatically when a protected disc is inserted. This is accomplished

using the Windows AutoRun feature, which allows a CD to specify a file to execute when it is inserted.

There are four main ways that users can bypass active protection installation: [73, 74]

1. Most non-Windows operating systems, including Mac OS X and Linux, do not have an equivalent to the AutoRun feature. CD-DRM systems cannot automatically install on these systems, so their users will not be affected by the protections.
2. Windows Vista, which was released after the CD-DRM systems considered in this study, replaces AutoRun with a new feature called AutoPlay. AutoPlay asks for permission before launching the software on the CD [112], so users who understand what the active protection software does can decline to install it.
3. Users of earlier Windows versions can disable AutoRun using a variety of methods [111]. Many experts recommend doing so as a security precaution apart from CD-DRM concerns.
4. Windows users who do not wish to disable AutoRun can suspend it temporarily each time they use a disc containing active protection by holding the shift key for a few seconds while inserting the CD [111].

Each of these methods will allow the disc to be copied normally, as if it did not contain active protection, provided that the method is used consistently and the software is never allowed to install. If it is installed, the user must take additional steps to disable it.

Deactivating Protection Software

Users can remove or deactivate active protection software by using standard system administration tools that are designed to find, characterize, and control the programs installed on

a machine. This is difficult to prevent if the user has system administrator privileges, but some schemes such as XCP attempt to do so by taking aggressive countermeasures.

The MediaMax 3 and MediaMax 5 active protection software is simple to deactivate since it is a single service named `sbcphid`. Services can be manipulated using the Windows command line service control utility `sc.exe`. To check the status of the service, a user can open a command prompt and issue the command `sc query sbcphid`; if the reported state is RUNNING then the MediaMax service is active. It can be deactivated using the command `sc stop sbcphid`. To permanently remove it, a user can issue the command `sc delete sbcphid`, then delete `%windir%\system32\drivers\sbcphid.sys`, the service's program file. Once the service is deactivated, MediaMax-protected albums can be accessed as if they were unprotected [73, 74].

XCP's active protection is more complicated to deactivate than MediaMax's, because it comprises several processes that are more deeply entangled in the system configuration and that are hidden by the XCP rootkit. However, the necessary steps could be automated to create a "point-and-click" removal tool that would be easy even for novices. For further details, see [74].

Ultimately, there is little a CD-DRM vendor can do to stop users from deactivating active protection software. Vendors' attempts to frustrate users' control of their machines are harmful and will trigger a strong backlash from users. In practice, vendors will probably have to provide some kind of uninstaller—users will insist on it, and some users will need it to deal with the bugs and incompatibilities that crop up inevitably in complex software. Once an uninstaller is released, the vendor can no longer aspire to prevent its users from removing the DRM software, and, once it is removed, determined users will be able to keep it off of their machines.

2.2.3 Attacks Against Authorized Players

The XCP and MediaMax 5 players were designed to enforce usage restrictions specified by the record label. In practice, they provide minimal security because there are many ways that users can bypass the limitations.

For instance, because the XCP and MediaMax 5 players create Windows Media files, they are vulnerable to any attack that can defeat Windows Media DRM (several have been available over the years). DRM interoperation often allows attacks on one system to defeat other systems as well, by allowing the attacker to transfer protected content into the system of her choice in order to extract it.

Another notable class of attacks targets the limited number of burned copies permitted by the players. Both players are designed to enforce this limit without communicating with any remote trusted server; thus, the player must track how many allowed copies remain in state on the local machine.

It is well known that DRM systems like this are vulnerable to rollback attacks [74]. A rollback attack backs up the state of the machine before performing the limited operation (in this case, burning the copy). When the operation is complete, the old system state is restored, and the DRM software is not able to determine that the operation has occurred. This kind of attack is easy to perform with virtual machine software like VMWare, which allows the entire state of the system to be saved or restored in a few clicks. XCP and MediaMax both fail under this attack, which allows unlimited copies to be burned with their players.

A refined variation of this attack targets only the specific pieces of state that the CD-DRM system uses to remember the number of copies remaining. The XCP player uses a single file, `%windir%\system32\$sys$filesystem\$sys$parking`, to record how many copies remain for every XCP album that has been used on the system. (This file is hidden and protected

by the XCP rootkit, which must be disabled before attacking it.) Rolling back this file after a disc copy operation would restore the original number of copies remaining [74].

A more advanced attack modifies the `$sys$parking` file to set the counter to an arbitrary value [74]. The file consists of a 16 byte header followed by a series of 177 byte structures. For each XCP disc used on the machine, the file contains a whole-disc structure and an individual structure for each track. Each disc structure stores the number of permitted copies remaining for the disc as a 32-bit integer beginning 100 bytes from the start of the structure.

The file is protected by primitive encryption. Each structure is XORed with a repeating 256-bit pad. The pad—a single pad is used for all structures—is randomly chosen when XCP is first installed and stored in the system registry in the key `HKLM\SOFTWARE\$sys$reference\ClassID`. Note that this key, which is hidden by the rootkit, is intentionally misnamed "ClassID" to confuse investigators. Instead of a ClassID, it contains the 32 bytes of pad data.

Hiding the pad does not increase the security of the design. An attacker who knows only the format of the `$sys$parking` file and the current number of copies remaining can change the counter to an arbitrary value without needing to know the pad. Suppose the counter indicates that there are x copies remaining and the attacker wants to set it to y copies remaining. Without decrypting the structure, she can XOR the padded bytes where the counter is stored with the value $x \oplus y$. If the original value was padded with p, the new value will be $(x \oplus p) \oplus (x \oplus y) = y \oplus p$, which is y padded with p.

Mitigating Player Vulnerabilities

Given that the DRM vendor must include some kind of limited back door in the mechanisms so that its authorized player can reliably access the music, we ask how this might be accom-

plished securely. The active protection software will have an interface that can be called to get access to the raw data from the disc, or to temporarily deactivate the active protection. This back door interface must be protected so that an ordinary program cannot use it to access the music.

There are at least three ways to protect the back door: [74]

1. *Secret interface*. The back door interface can be kept secret so that ordinary programmers do not know how to call it. This method is disfavored because its security by obscurity method violates Kerckhoffs's Principle [91].
2. *Secret key passed as argument*. There can be a secret key that must be passed to the back door interface, and the active protection software can be programmed to ignore requests that do not contain the correct key. This is superior to the secret interface method because it relies on the secrecy of the key rather than secrecy of algorithm information. However, an adversary who can interpose himself into the back door interface can observe the key, and can act as a man in the middle to modify the calls being made to the active protection software.
3. *Cryptographic protocol*. The vendor's application software can use a cryptographic protocol to communicate with the active protection software. The sort of protocol used for secure remote procedure call on a network would be suitable, with the network messages replaced by calls across the back door interface, so that each back-and-forth pair of messages in the protocol was replaced by a single call and return. This approach makes interface snooping and interposition attacks useless (assuming that the protocol is properly secure). However, it cannot stop an adversary from reverse engineering the vendor's application-level software to learn the protocol and extract any keys.

It seems none of these methods is likely to withstand a sophisticated attack. Stronger security might be possible by adding support to CD drive hardware or by utilizing Trusted

Computing functionality [159], but these methods would require users to shoulder additional costs and limit the use of the player software to a smaller number of compatible PCs.

2.3 Attacks: Protection Cloning

Once installed from a single disc, the active protection systems employed by XCP and MediaMax restrict access not just to the original disc, but to any disc that is protected by the same scheme. This requires some mechanism for identifying these discs.

Disc identification is potentially vulnerable to a variety of attacks. For example, by preventing active protection software from identifying a disc as protected, an attacker can gain unrestricted access to the content. However, since there are many easier ways to bypass active protection (see Section 2.2.2), we focus now on a different kind of attack wherein the attacker attempts to manufacture a new disc, containing different content, that activates a particular active protection system.

To see why this attack is important, consider the CD-DRM vendor's business strategy. The vendor seeks over time to build a protection platform—an installed base of as many computers as possible running the vendor's active protection software. This is valuable because of the difficulty of getting the software installed in the first place and the vulnerability that exists when it is not present. New albums can be more secure if they leverage a large preexisting installed base, allowing the protection vendor to charge a premium for its product.

However, if the mark that the vendor's scheme uses to recognize protected discs can be forged, then a record label or rival vendor could mark its discs without the vendor's permission, thereby taking advantage of the platform without paying. We call this kind

of free-riding a *protection cloning* attack [74]. All second- and third-generation CD-DRM systems are potentially susceptible to it.

Forging only a mark is probably not copyright infringement. Unlike the musical work in which it is embedded, the mark itself is functional and contains little or no expression, and therefore seems unlikely to qualify for copyright protection. In principle, the mark recognition process could be covered by a patent, but we are unaware of any such patent relating to XCP or MediaMax. Even if the vendor does have a legal remedy, it seems worthwhile to design the mark to prevent forgery if the cost of doing so is low.

There are advantages and disadvantages for an entity placing unauthorized marks. Copyright would prohibit rogue publishers from distributing an installer for the active protection software, though they might depend on the existing installed base if the software was included on many widely sold titles. They would also be prevented from employing the components of the protection software that allow users to access restricted copies of the music; however, they could create their own software to provide this capability if they desired. On the other hand, free riding publishers would not be restricted to marking their disc for only one scheme. By identifying their discs as copy protected with a number of schemes (say, both XCP and MediaMax), they could invoke multiple layers of security and enjoy stronger protection than is available with any single technique, all without paying. Preventing protection cloning requires some kind of disc authentication mechanism to control access to the active protection platform—a meta-copy protection technique, if you will.

The remainder of this section discusses the security requirements for CD-DRM disc recognition systems, describes the design of the actual recognition mechanism employed by MediaMax, and shows how this mechanism is vulnerable to protection cloning and other attacks. We conclude by presenting an improved design that better satisfies the requirements.

2.3.1 Disc-Recognition Requirements

Any disc recognition system detects some mark or other distinctive feature of discs protected by a particular copy protection scheme. Ideally, such a feature would satisfy these requirements:

1. *Correctness.* The feature should identify protected discs without accidentally triggering the copy protection on unprotected titles.
2. *Detectability.* It should be possible for the active protection drivers running on client systems to reliably and quickly detect the feature in protected discs. In practice, this limits the amount of audio or data that can be read before deciding whether to apply protection.
3. *Indelibility.* The feature should be hard to remove without substantially degrading the quality of the audio; that is, it should be difficult to physically modify the disc to obscure the mark, or to make a copy of the protected disc that does not itself trigger the protection.
4. *Unforgeability.* It should be difficult to apply the feature to an unprotected album without the cooperation of the protection vendor, even if the adversary has access to protected discs.

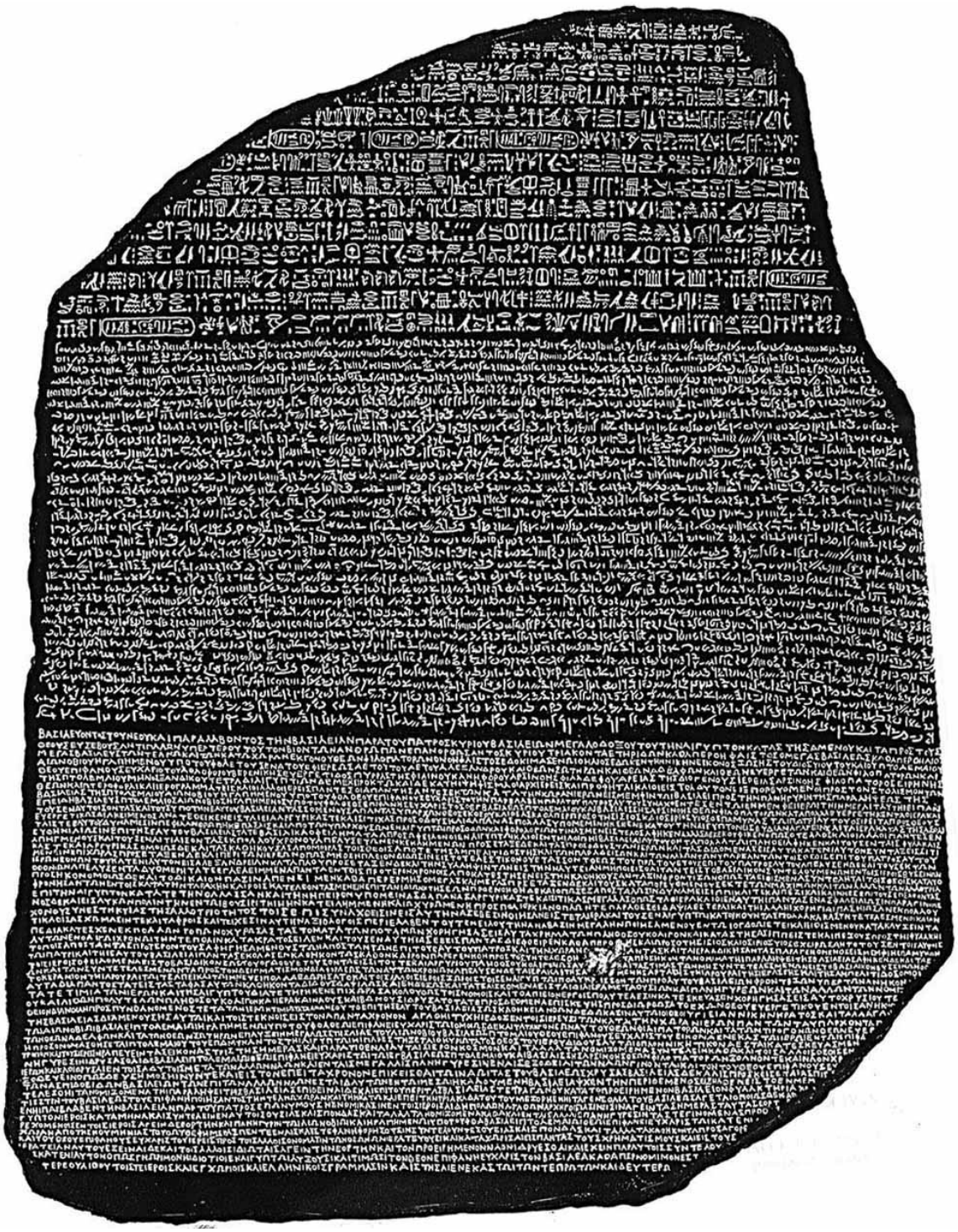
The correctness and detectability requirements are necessary for basic functionality, and indelibility is required to secure the content. Unforgeability is a defense against protection cloning attacks and may also be important for safeguarding access to "bonus" content that is made available only when a marked disc is present.

2.3.2 Reverse-Engineering the MediaMax Watermark

To find out how well the disc recognition mechanisms employed by CD-DRM systems meet the ideal requirements, we examined the recognition system built into MediaMax 3 and MediaMax 5 [74]. This system drew our attention because MediaMax's creators have touted their advanced disc identification capabilities, including the ability to identify individual tracks within a compilation as protected [108]. XCP appears to use a less sophisticated disc recognition system based on a marker stored in the data track of protected discs; these markers can be trivially duplicated, so we did not study them further.

We determined how MediaMax identifies protected albums by tracing the commands sent to the CD drive with and without the active protection software running. These experiments took place on a Windows XP VMWare virtual machine running on top of a Fedora Linux host system, which we modified by patching the kernel IDE-SCSI driver to log all CD device activity.

With this setup we observed that the MediaMax software executes a disc recognition procedure immediately upon the insertion of a CD. It reads two sectors of audio at a specific offset from the beginning of audio tracks—approximately 365 and 366 frames in (a CD frame stores 1/75 second of sound). On unprotected discs, the software scans through every track in this way, but on MediaMax-protected albums, it stops after the first three tracks, apparently having detected an identifying feature. The software decides whether or not to block read access to the audio solely on the basis of information in this region, so we inferred that the identifying mechanism takes the form of an inaudible watermark embedded in this part of the audio stream. Locating the watermark nearly five seconds after the start of the track rather than at the very beginning is a logical choice, since it reduces the likelihood that it will occur in a very quiet passage (where it might be more audible) and makes cropping it out more destructive.



The Rosetta Stone

Locating the watermark amid megabytes of audio might have been difficult, but we had the advantage of a virtual Rosetta Stone. The actual Rosetta Stone—a 1500 lb. granite slab, unearthed in Rosetta, Egypt, in 1799—is inscribed with the same text written in three languages: ancient hieroglyphics, demotic (simplified) hieroglyphics, and Greek. Comparing these inscriptions provided the key to deciphering Egyptian hieroglyphic texts. Our Rosetta Stone was a single album, Velvet Revolver's *Contraband*, released in three different versions: a U.S. release protected with MediaMax, a European release protected with a passive scheme developed by Macrovision, and a Japanese release with no copy protection. We decoded the MediaMax watermark by examining the differences between the audio on these three discs. Binary comparison revealed no differences between the releases from Europe and Japan; however, the MediaMax-protected U.S. release differed slightly from the other two in certain parts of the recording. By carefully analyzing these differences—and repeatedly attempting to create new watermarked discs using the active protection software as an oracle—we were able to deduce the structure of the watermark.

The MediaMax watermark is embedded in the audio of each track in 30 *clusters* of modified audio samples. Each cluster is made up of 288 marked 16-bit audio samples followed by 104 unaltered samples. Three clusters exactly fit into one 2352-byte CD audio frame. The watermark is centered at approximately frame 365 of the track; though the detection routine in the software only reads two frames, the mark extends several frames on either side of the designated read target to cope with inexpensive CD drives, which often suffer from imprecise seeking in the audio portion of a disc. The MediaMax software detects the watermark if at least one mark cluster is present in the region read by the detector.

A sequence of 288 bits that we call the *raw watermark* is embedded into the 288 marked audio samples of each cluster. One bit of the raw watermark is embedded into each unmarked audio sample by setting one of the three least-significant bits to the new bit value

Original bits	Marked bits					
	0␣␣	␣**0**␣	␣␣**0**	**1**␣␣	␣**1**␣	␣␣**1**
.....111	**0**11	1**0**1	11**0**	**1**11	1**1**1	11**1**
.....110	**0**1$\underline{1}$	1**0**$\underline{1}$	11**0**	**1**10	1**1**0	11**1**
.....101	**0**$\underline{1}$1	1**0**1	10**0**	**1**01	1**1**$\underline{0}$	10**1**
.....100	**0**$\underline{11}$	1**0**0	10**0**	**1**00	1**1**0	10**1**
.....011	**0**11	0**0**1	01**0**	**1**$\underline{00}$	0**1**1	01**1**
.....010	**0**10	0**0**$\underline{1}$	01**0**	**1**$\underline{0}$0	0**1**0	01**1**
.....001	**0**01	0**0**1	00**0**	**1**0$\underline{0}$	0**1**$\underline{0}$	00**1**
.....000	**0**00	0**0**0	00**0**	**1**00	0**1**0	00**1**

Table 2.2—MediaMax Watermark Bit Storage To store a bit, the MediaMax watermark encoder overwrites one of the three least-significant bits of an audio sample. It then alters less-significant bits (underlined) when this will reduce the change from the original value.

and then setting the two other bits according to Table 2.2. This design lessens the audible distortion caused by setting the watermark value. The change in the other two bits reduces the magnitude of the difference from the original audio sample, but it also introduces a highly uneven distribution in the three least significant bits that makes the watermark easier for a naïve adversary to detect or remove.

The position of the embedded bit in each sample follows a fixed, apparently pseudorandom sequence for every mark cluster. Each of the 288 bits is embedded in the first-, second-, or third-least-significant bit position of the sample according to the sequence shown in Table 2.3. [74]

The active protection software reads the raw watermark by reading the first, second, or third bit from each sample according to the sequence discussed above. It determines whether the resulting 288-bit sequence is a valid watermark by checking certain properties of the sequence, as shown in Table 2.4. It requires 64 positions in the sequence to have a fixed value, either 0 or 1. Another 192 positions are divided into 32 groups of linked values, each corresponding to a bit a_i. Within each group, three positions are equal to a_i and three share the complement value, $\bar{a}_i$. This allows the scheme to encode a 32-bit value (vector A), though in the discs we studied A appears to take a different random value in

each mark cluster of each protected title. The final 32 bits of the raw watermark may have arbitrary values and encode a second 32-bit value (vector B). MediaMax 5 uses this value to distinguish between original discs and backup copies burned using its proprietary player application. [74]

2.3.3 Attacks on the MediaMax Watermark

The MediaMax watermark fails to satisfy the indelibility and unforgeability requirements of an ideal disc recognition system. Far from being indelible, the mark is surprisingly brittle. Most advanced designs for robust audio watermarks [40, 39] manipulate audio in the frequency domain and try to resist removal attempts that use lossy compression, multiple conversions between digital and analog formats, and other common transformations. In contrast, the MediaMax watermark is applied in the time domain and is rendered undetectable by even minor changes to the file. An adversary without any knowledge of the watermark's design could remove it by converting the tracks to a lossy format like MP3 and then burning them back to a CD. While this would result in some minor loss of fidelity, a more sophisticated adversary could prevent the mark from being detected with almost no degradation by flipping the least-significant bit of just one carefully chosen sample from each of the 30 watermark clusters, thereby preventing the mark from exhibiting the pattern required by the detector.

The watermark also fails to satisfy the unforgeability requirement. Its only defense against forgery is its complicated, unpublished design, but as is often the case, this security by obscurity has proved tedious rather than impossible to defeat. As it turns out, an adversary needs only limited knowledge of the watermark—its location within a protected track and its confinement to the three least-significant bits of each sample—to forge it with minimal loss of fidelity. Such an attacker could transplant the three least-significant bits

2,	3,	1,	1,	2,	2,	3,	3,	2,	3,	3,	3,	1,	3,	2,	3,	2,	1,	3,	2,	2,	3,	2,	2,	2,	1,	3,	3,	2,	1,	2,	3,
3,	1,	2,	2,	3,	1,	2,	3,	3,	1,	1,	2,	2,	1,	1,	3,	3,	1,	2,	3,	1,	2,	3,	3,	1,	3,	3,	2,	1,	1,	2,	3,
2,	2,	3,	3,	3,	1,	1,	3,	1,	2,	1,	2,	3,	3,	2,	2,	3,	2,	1,	2,	2,	1,	3,	1,	3,	2,	1,	1,	2,	1,	1,	1,
2,	3,	2,	1,	1,	2,	3,	2,	1,	3,	2,	2,	2,	3,	1,	2,	1,	3,	3,	3,	3,	1,	1,	1,	2,	1,	1,	2,	2,	2,	2,	3,
1,	2,	3,	2,	1,	3,	1,	2,	2,	3,	1,	1,	3,	1,	1,	1,	1,	2,	2,	3,	2,	3,	2,	3,	2,	1,	2,	3,	1,	3,	1,	3,
3,	3,	1,	1,	2,	1,	1,	2,	1,	3,	3,	2,	3,	3,	2,	2,	1,	1,	1,	2,	2,	1,	3,	3,	3,	3,	3,	1,	3,	1,	1,	3,
2,	2,	3,	1,	2,	1,	2,	3,	3,	2,	1,	1,	3,	2,	1,	1,	2,	2,	1,	3,	3,	2,	2,	3,	1,	3,	2,	2,	2,	3,	1,	1,
1,	1,	3,	2,	1,	3,	1,	1,	2,	2,	3,	2,	3,	1,	1,	2,	1,	3,	2,	3,	3,	1,	1,	3,	2,	1,	3,	1,	2,	2,	3,	1,
1,	3,	2,	1,	2,	2,	2,	1,	3,	3,	1,	2,	3,	3,	3,	1,	2,	2,	3,	1,	2,	3,	1,	1,	3,	2,	2,	1,	3,	2,	1,	3

Table 2.3—MediaMax Watermark Placement The 288 bits of the MediaMax watermark are embedded in successive audio samples by overwriting the first-, second-, or third-least-significant bit according to the sequence shown here.

0,	a_0	a_1,	a_2,	a_3,	a_4,	0,	0,	a_5,	0,	a_6,	0,	a_7,	0,	a_8,	a_3,
a_9,	$\bar{a}_9$,	a_{10},	0,	a_{11},	a_{12},	0,	a_{13},	a_{14},	a_{15},	$\bar{a}_4$,	a_{16},	$\bar{a}_4$,	a_{17},	0,	$\bar{a}_{15}$,
a_{18},	a_3,	$\bar{a}_{12}$,	a_{19},	a_{20},	a_{21},	a_{22},	a_{19},	$\bar{a}_{11}$,	a_0,	a_{23},	a_2,	a_{20},	0,	$\bar{a}_{17}$,	a_{11},
a_5,	$\bar{a}_3$,	a_{21},	0,	a_{12},	0,	$\bar{a}_{16}$,	0,	a_{24},	a_2,	a_{25},	0,	a_9,	$\bar{a}_8$,	$\bar{a}_6$,	a_{26},
$\bar{a}_{18}$,	$\bar{a}_{22}$,	$\bar{a}_7$,	a_{21},	a_{24},	a_{13},	0,	0,	$\bar{a}_7$,	$\bar{a}_9$,	$\bar{a}_{20}$,	a_0,	a_{27},	0,	$\bar{a}_{21}$,	a_6,
a_9,	0,	0,	$\bar{a}_{27}$,	$\bar{a}_8$,	a_4,	$\bar{a}_{25}$,	0,	a_{17},	a_{28},	$\bar{a}_0$,	a_{29},	$\bar{a}_3$,	$\bar{a}_{25}$,	0,	$\bar{a}_{21}$,
a_{30},	0,	a_{23},	a_{18},	$\bar{a}_6$,	$\bar{a}_{17}$,	0,	$\bar{a}_1$,	a_{14},	a_1,	a_{17},	0,	a_{24},	$\bar{a}_{27}$,	$\bar{a}_{12}$,	a_7,
0,	$\bar{a}_0$,	a_{13},	$\bar{a}_5$,	$\bar{a}_{19}$,	0,	$\bar{a}_{14}$,	0,	$\bar{a}_{28}$,	$\bar{a}_{30}$,	$\bar{a}_4$,	0,	0,	$\bar{a}_{10}$,	$\bar{a}_2$,	$\bar{a}_{23}$,
0,	$\bar{a}_5$,	a_{15},	a_{25},	$\bar{a}_{23}$,	a_8,	0,	0,	a_{26},	$\bar{a}_6$,	0,	1,	a_{22},	$\bar{a}_{19}$,	$\bar{a}_{13}$,	$\bar{a}_{22}$,
a_8,	0,	0,	$\bar{a}_9$,	a_{12},	a_{23},	a_{27},	$\bar{a}_{24}$,	$\bar{a}_{15}$,	$\bar{a}_{16}$,	0,	0,	0,	a_4,	$\bar{a}_{27}$,	0,
0,	1,	a_6,	0,	a_{15},	a_{11},	0,	$\bar{a}_{26}$,	a_{19},	a_7,	$\bar{a}_3$,	$\bar{a}_{30}$,	$\bar{a}_{22}$,	a_{28},	$\bar{a}_{29}$,	0,
$\bar{a}_{15}$,	a_{16},	$\bar{a}_5$,	0,	1,	a_{31},	0,	$\bar{a}_2$,	a_{31},	$\bar{a}_{26}$,	$\bar{a}_{18}$,	$\bar{a}_1$,	$\bar{a}_{28}$,	a_{27},	0,	a_{14},
0,	a_{16},	$\bar{a}_8$,	0,	0,	$\bar{a}_{26}$,	a_{18},	a_{30},	$\bar{a}_{30}$,	$\bar{a}_7$,	0,	$\bar{a}_{10}$,	$\bar{a}_{13}$,	$\bar{a}_{31}$,	a_{26},	$\bar{a}_{18}$,
$\bar{a}_{25}$,	$\bar{a}_{13}$,	$\bar{a}_2$,	$\bar{a}_{14}$,	$\bar{a}_1$,	0,	$\bar{a}_{19}$,	0,	$\bar{a}_{24}$,	$\bar{a}_{21}$,	0,	a_{31},	$\bar{a}_{14}$,	0,	$\bar{a}_{31}$,	0,
a_{20},	a_{28},	0,	$\bar{a}_{24}$,	a_{10},	$\bar{a}_{20}$,	a_{25},	$\bar{a}_{29}$,	$\bar{a}_{16}$,	a_{10},	$\bar{a}_{17}$,	$\bar{a}_{20}$,	$\bar{a}_{31}$,	$\bar{a}_{28}$,	a_{11},	a_{11},
a_{22},	$\bar{a}_{10}$,	$\bar{a}_0$,	0,	$\bar{a}_{29}$,	0,	a_{30},	$\bar{a}_{12}$,	a_1,	a_5,	0,	0,	$\bar{a}_{23}$,	a_{29},	a_{29},	0,
b_0,	b_1,	b_2,	b_3,	b_4,	b_5,	b_6,	b_7,	b_8,	b_9,	b_{10},	b_{11},	b_{12},	b_{13},	b_{14},	b_{15},
b_{16},	b_{17},	b_{18},	b_{19},	b_{20},	b_{21},	b_{22},	b_{23},	b_{24},	b_{25},	b_{26},	b_{27},	b_{28},	b_{29},	b_{30},	b_{31}

Table 2.4—MediaMax Watermark Coding and Constraints The MediaMax watermark encodes two 32-bit vectors A and B by expanding them into a 288-bit sequence. Some bit positions have fixed values, while others are always equal to, or always opposite from, bits in the vectors. These constraints allow the decoder to identify marked audio with a low false-positive rate, while the redundancy potentially provides robustness against read errors.

of each sample within the watermarked region of a protected track to the corresponding sample from an unprotected one. Transplanting these bits would cause distortion more audible than that caused by embedding the watermark, since the copied bits are likely to differ by a greater amount from the original sample values; however, the damage to the audio quality would be limited since the marked region is only 0.4 seconds in duration. A more sophisticated adversary could apply a watermark to an unprotected track by deducing the full details of the structure of the watermark, as we did; she could then embed the mark in an arbitrary audio file just as well a licensed disc producer.

As a proof-of-concept, we created a utility called `scmark` that can detect, embed, or remove the MediaMax watermark [74]. The program is invoked on one or more WAVE audio files as follows:

`scmark --detect [-p` ⟨*position*⟩`] [-c` ⟨*count*⟩`]` ⟨*file.wav*⟩ ...
`--embed [[`⟨*A*⟩`]` ⟨*B*⟩`] [-p` ⟨*position*⟩`] [-c` ⟨*count*⟩`]` ⟨*file.wav*⟩ ...
`--remove [-p` ⟨*position*⟩`] [-c` ⟨*count*⟩`]` ⟨*file.wav*⟩ ...

When `scmark` is executed with only the `--detect` switch on a track ripped from a MediaMax-protected album, it detects the watermark almost instantaneously. The ⟨*position*⟩ parameter gives a hint as to the position of the watermark within the file; and the ⟨*count*⟩ parameter indicates how many mark clusters to read or write. For embedding, ⟨*A*⟩ and ⟨*B*⟩ are the two 32-bit vectors encoded in the watermark, as described above. The `--remove` switch searches for an existing watermark and removes it by overwriting it with a random raw watermark that lacks the properties required by the MediaMax detector.

2.3.4 Mitigation: Cloning-Resistant Watermarks

Having shown that the MediaMax watermark fails to provide either indelibility or unforgeability, we ask whether it is possible to securely accomplish either or both of these goals.

As far as indelibility is concerned, prospects are bleak, since watermarking schemes have a poor history of resisting removal [40, 93, 128]. This is especially true against an adversary who has oracle access to the watermark detector, as is the case with a previous application of watermarks to audio copy protection, SDMI [40], as well as with CD-DRM systems. Making marks that are both indelible and unforgeable is likely much more difficult. There seems to be tension between marks that are difficult to remove and ones that are hard to forge. Enforcing both requirements creates two ways to fool the detector: by rendering the mark invisible and by making it appear forged. If, as in CD-DRM systems, either situation leads to the same result (no protection), the attacker's power is magnified.

In contrast, a mark strongly robust to forgery is simple to create based on digital signatures if we are not concerned with its being easy to remove. A simple scheme works as follows:

1. To sign an audio track, the licensed publisher reads a fixed portion L of the audio data (say, the first ten seconds), then computes a cryptographic hash of L and signs it using a public key signature algorithm to derive the signature $S_L := Sign_{K_S}(Hash(L))$. S_L is then stored at a second location in the track by setting the LSB of each sample in the region to the corresponding bit in the signature. A 320-bit DSA signature could be embedded in this way using approximately the same space as one mark cluster of the MediaMax watermark.

2. The publisher keeps the signing key K_S secret and builds the corresponding verification key K_V into the active protection software. When presented with a CD, the software checks for a valid signature. First it reads the audio from the signed area of the track, and it locates and extracts the signature stored in the LSBs in the second location. Next, it hashes the audio and verifies the signature on the hash using K_V. If the signature is

correct, the watermark is valid and genuine; otherwise, forgery or data corruption is indicated.

The scheme could be strengthened against natural errors by applying the mark to several regions of the disc, as in the MediaMax watermark or by the user of error correcting codes.

Forging such a mark would require defeating the digital signature scheme or splicing both L and S_L from a legitimately marked album. Requiring L to be several seconds of audio makes such splicing less appealing.

Clearly this watermark is highly vulnerable to removal. If even a single bit of the hashed region is changed, the mark will not be recognized as valid. Yet the watermark MediaMax actually uses is also vulnerable to corruption of a single bit (in each mark cluster) while being far less resistant to forgery. Though robustness to removal could be improved by various methods, we believe that robustness, while desirable in principle, is of limited value in real CD-DRM applications, and should not be traded off against forgeability. Removal of the watermark is unlikely to be the weakest link protecting the audio, and while the gains from creating a more indelible watermark are slight, the loss to protection-cloning attacks from an easily forgeable mark is potentially substantial.

This basic scheme can be elaborated and improved in various ways. For instance, the scheme as presented is more vulnerable to false negatives due to data corruption than the existing MediaMax mark, since, in our example, the signature is derived from audio data that includes about 250 times the number of samples occupied by the MediaMax mark. We could rectify this by applying an error correcting code to L and storing the syndrome in low-order bits of subsequent samples [45].

2.4 Attacks: Collateral Damage

While content-copying and protection-cloning attacks target the interests of parties that deployed and designed the CD-DRM systems, unsuspecting end-users are the victims of a third category of threats. Several CD-DRM systems contain designs flaws and implementational errors that can be exploited by third parties to attack the security of users' computers. In some cases, the computer becomes vulnerable as soon as a protected CD is inserted, even if the user never consents to the installation of protection software. Though most of these attacks stem from garden-variety security bugs, they are more alarming from users' perspectives than typical failures, because CD-DRM software does not serve the interests of the user, so it cannot be said from the users' standpoint that the benefits justified the risks. Instead, these attacks are *collateral damage* from aggressive protection techniques adopted for the sole benefit of copyright holders and CD-DRM vendors.

In this section we describe several ways that the XCP and MediaMax 5 systems can be exploited by third parties to harm the interests of end users, and then we consider a number of strategies that future CD-DRM implementers might use to reduce the risk of such threats. Only third-generation, aggressive protection schemes have been shown to cause such collateral damage.

2.4.1 Exploiting the XCP Rootkit

XCP puts users' computers at risk by installing a rootkit that allows any software—not just the XCP active protection system—to hide from system administration tools [74]. Malware authors can use the XCP rootkit to hide arbitrary files, registry keys, or processes by naming them with the rootkit's magic prefix, `$sys$`. This is especially harmful in circumstances where OS privilege separation would prevent malware from installing its own rootkit.

Only kernel-level processes can patch the Windows system service dispatch table, and only privileged users—normally, members of the Administrators or Power Users groups—can install such processes. (XCP itself requires these privileges to install.) Thus, malicious code running as an unprivileged user cannot normally install a rootkit that intercepts system calls. However, if the XCP rootkit is installed, it will hide all programs that adopt the `$sys$` prefix so that even privileged users will be unable to see them.

This vulnerability has already been exploited by at least two malicious programs documented in the wild. A backdoor named Ryknos.B hides behind the XCP rootkit while allowing remote parties to download and execute files [105]. A Trojan horse called Welomoch uses the rootkit to conceal a worm that spreads across file-sharing networks [87].

This rootkit facilitates another privilege escalation attack whereby an unprivileged application can crash the system. Russinovich demonstrated this problem using an automated testing (fuzzing) program he created called NTCRASH2 [148]. This utility makes repeated system calls with randomly generated invalid parameters. The original Windows kernel functions handle invalid inputs correctly and the system remains stable, but with the XCP rootkit installed, certain invalid inputs result in a system crash.

We investigated the specific circumstances when these crashes occur. The rootkit's implementation of `NtCreateFile` can cause a crash if it is passed an invalid pointer as its `ObjectAttributes` argument, or if it is passed a valid `ObjectAttributes` structure that points to a `ObjectName` structure with an invalid `Buffer` pointer. It seem that through sheer luck these flaws are not exploitable to execute code; however, they do allow an unprivileged user to bring the system to an unsafe halt.

Building a Better Rootkit?

Is it possible to build a rootkit like XCP's without causing similar security problems? One approach might be to use hard-coded names for the files and other objects to be hidden, rather than a magic trigger prefix. (Alternatively, these names could be included in a signed manifest file.) Though better than the original rootkit, this design is still risky: bugs or permission errors might allow attackers to add files to hidden directories or to inject code into hidden processes after they execute. Furthermore, such a rootkit would hinder Windows system administration by complicating a privilege model already shown to be too complex to manage without difficulty.

2.4.2 Exploiting the MediaMax Player

MediaMax 5 also exposes users' computers to potential attacks. When a MediaMax CD is inserted into a computer, Windows AutoRun launches an installer from the disc. Even before displaying a license agreement, MediaMax copies almost twelve megabytes of files and data related to the MediaMax player to the hard disk and stores them in a folder named `%programfiles%\Common Files\SunnComm Shared`. Burns and Stamos [27] discovered that the MediaMax installer assigns insecure access permissions to these files and directories, allowing any user to modify them.

The lax permissions allow a nonprivileged user to replace the MediaMax executable files with malicious code. If a privileged user later plays a MediaMax CD, the attack code will execute with that user's security privileges. The MediaMax player requires Power User or Administrator privileges to run, so it's likely that the attacker will be able to gain complete control of the system.

We discovered a variation of the attack suggested by Burns and Stamos that allows the attack code to be installed even if the user has never consented to the installation of

MediaMax, and to be triggered immediately whenever the user inserts a MediaMax CD, even without running the player. In the original attack, the user needs to accept the MediaMax license agreement before attack code can be inserted or executed, because the code is placed in a file called `MMX.EXE` that is not copied to the system until after the agreement is accepted. In our attack, the attacker places hostile code in the `DllMain` procedure of a library called `MediaMax.dll`, which MediaMax installs prior to displaying the EULA. The next time a MediaMax CD is inserted, AutoRun executes the installer, which immediately attempts to check the version of the installed `MediaMax.dll` file. In the process, it calls the Windows `LoadLibrary` function, which is not safe to use on untrusted code because it invokes the library's `DllMain` procedure.

Normally, problems like these can be fixed by manually correcting the errant permissions. However, MediaMax aggressively updates the installed player code each time the software on a protected disc is started by AutoRun. As part of this update, the permissions on the installation directory are reset to the insecure state, making repairs much more difficult.

To fix these problems permanently without losing the use of protected discs, users need to install a software patch from SunnComm. Unfortunately, as we discovered, the first version of this patch was capable of triggering precisely the kind of attack it was supposed to prevent. In the process of updating MediaMax, the patch checked the version of `MediaMax.dll` with the same insecure technique as the MediaMax installer. If this file was already modified by an attacker, the process of applying the security patch would set off the attack code. A MediaMax uninstaller that had been issued by SunnComm earlier also suffered from this vulnerability, and the company corrected both it and the patch as a result of these findings.

2.4.3 Privacy Concerns

While not an attack in the same sense as the other collateral damage caused by CD-DRM, XCP and MediaMax 5 both harm user privacy through spyware-like behavior. While the precise definition of "spyware" is has been debated, it was clear even when XCP was developed that the term applies to software that is installed without the user's informed consent, is difficult to uninstall, and transmits information about the user's activities without notice or consent. Both XCP and MediaMax met this definition.

The systems exceed user consent in at least three ways. First, they load and at least temporarily run active protection software before the user has agreed to anything. Second, the license agreements ask the user to consent only to "a small proprietary software program ... intended to protect the audio files embodied on the CD," a description that can hardly be said to cover the full active protection systems, including behavior tracking (described below) and, in XCP's case, rootkit-style interference with system administration. Third, although the EULA disclosures refer to the audio files on the current CD, the installed software perpetually interferes with the use of all titles protected with the same CD-DRM scheme without asking for further consent [63].

Moving on to the second criterion, we find that both XCP and MediaMax were clearly designed to resist detection and removal. Both ship without any kind of uninstaller, and the vendors at first made it exceedingly difficult for users to obtain one. XCP users, for example, had to (1) fill out a Web form containing personal information, then wait days for a reply email; (2) fill out another Web form and accept the installation of more software in the form of a proprietary ActiveX control; (3) after waiting a few more days, at last receive a link to a web page that would perform the removal. Even then, this link only worked for a limited time and could only be used on the machine from which the original request was made.

Beyond limiting access to the official uninstallers, both systems take steps to undermine the user's administrative control of the system: XCP uses the rootkit method described earlier, and MediaMax makes its active protection service invisible in the normal Control Panel GUI.

Finally, the third spyware criterion: both XCP and MediaMax surreptitiously "phone home" to their creators with information about users' listening habits. Both systems were designed to contact a vendor or record label web site whenever the user inserted a protected disc. XCP discs contact a server operated by Sony, `connected.sonymusic.com` [146], while MediaMax discs contact a server operated by SunnComm, `license.sunncomm2.com`. Ostensibly the purpose of these connections was to download images or advertisements to display while the music played, but it also created entries in the Web server logs, noting the users' IP addresses, which discs they had played, and the times, dates, and durations of these uses. Despite this breach in privacy, the vendors' web sites claimed that they did not gather information about users' activities.

This combination of undisclosed data collection, installation without informed consent, and barriers to removal made XCP and MediaMax fit the consensus definition of spyware.

2.4.4 Exploiting the XCP and MediaMax Uninstallers

Users were outraged by the revelation of the rootkit and privacy problems. To appease them, the makers of XCP and MediaMax reluctantly made their uninstallers easier to obtain. Rather than distribute standalone removal utilities, both vendors chose to make these uninstallers available as online applications built with ActiveX controls. Though the MediaMax and XCP uninstallers were apparently developed independently of each other, we discovered that both of them cause serious, persistent vulnerabilities on any computer where they are used.

MediaMax Uninstaller Vulnerability

The MediaMax web-based uninstaller uses a proprietary ActiveX control, `AxWebRemove.ocx`, created and signed by SunnComm. When users visit the MediaMax uninstaller site, the browser prompts them to install the control. If they assent, code on the web page uses the control to uninstall MediaMax. It calls a method called `Remove`, which takes two parameters: `key`, and `validate_url`. When `Remove` is called, it issues an HTTP GET request to `validate_url` to validate `key`. If `key` is valid, the server responds with the message `true,`*<uninstall_url>*, where `uninstall_url` is the web address of a DLL file containing MediaMax removal code. The control retrieves this DLL from the Internet, saves it to a temporary location, and calls a function in it named `ECF7` to carry out the uninstallation. If the function indicates success, the `Remove` routine issues a second HTTP GET request to `validate_url` to report that uninstallation was completed and that the single-use key should be retired.

This design is obviously insecure. The control accepts an arbitrary `validate_url` parameter and blindly trusts the DLL supplied by the server. It neither verifies the authenticity of the server nor the code.

This flaw is amplified into a critical vulnerability by another careless implementational choice. The ActiveX control is not itself removed during the uninstallation process, so its methods can be invoked later by any web page without causing further browser security prompts. An attacker can create a web page that invokes the `Remove` method and provides a `validate_url` pointing to a site she controls. This server can accept whatever key is presented and return an `uninstall_url` pointing to a DLL crafted by the attacker. If a user visits the page on a PC that once ran the MediaMax uninstaller, the latent ActiveX control will download the DLL and execute whatever code the attacker has written.

XCP Uninstaller Vulnerability

We also found that the XCP web-based uninstaller contains the same design flaw and is only slightly more difficult to exploit. It utilizes a proprietary ActiveX control named `CodeSupport.ocx` that is installed in the step two of the three-step XCP removal process documented above. In this step, the control sends a random nonce to the XCP server and stores the same value in the local system registry. Eventually, the user receives a personalized link to a web page that uses the ActiveX control to remove XCP. The page uses the control to check that the system's registry contains the same nonce as in the original request, tethering the uninstaller to the machine from which the request was made. As a consequence of this design, the vulnerable control may be present on a user's system even if she never completed the final step to remove XCP.

Matti Nikki [123] was the first to notice that the XCP ActiveX control contains methods such as `InstallUpdate(url)`, `Uninstall(url)`, and `RebootMachine()` and suspected that these might be vulnerable. He also demonstrated that the control remained installed after the XCP removal process completed, and that its methods (including one that rebooted the computer) were scriptable from any web page without further browser security warnings.

We reverse engineered the `InstallUpdate` and `Uninstall` methods and discovered serious flaws. Each takes as an argument a URL pointing to a specially formatted archive that contains updater or uninstaller code and data files. When these methods are invoked, the archive is retrieved from the provided URL and stored in a temporary location. The `InstallUpdate` method then extracts a file named `InstallLite.dll` from the archive and calls a function in it named `InstallXCP`.

Like the MediaMax ActiveX control, the XCP control does not validate the URL or the downloaded archive. The only obstacle to attack is the proprietary format of the archive file. We reverse engineered the format by disassembling the control. An XCP archive consists of

several blocks of gzip-compressed data containing separate files, each preceded by a short header. At the end of the archive, a catalog structure lists metadata for each block, including a 32-bit CRC that the control checks before extracting the files.

With knowledge of this file format, we constructed an archive containing benign proof-of-concept exploit code. The most difficult detail was the CRC, which seems to use a proprietary algorithm. To avoid the tedious task of reverse engineering the CRC algorithm, we inserted a break point in the verification code and ran the control on an archive containing files we prepared. The verification procedure compares the CRC value stored in the archive to a value it computes on the data; we read out this latter value and applied it to our archive. Thus modified, the archive passed the CRC check and the ActiveX control executed our code. This illustrated why digital signatures, not secret one-way functions, should be used to validate code from untrusted sources.

Following the disclosure of these problems, SunnComm and First4Internet eventually released standalone uninstaller programs that did not leave users' systems vulnerable. These new uninstallers also removed the exploitable ActiveX controls if they were present.

2.4.5 Mitigating Collateral Damage

Compared to other media on which software is distributed, audio CDs have a very long expected life. Many albums will still be inserted into computers and other players decades after they are first purchased. If a particular version of DRM software is shipped on a new CD, that software version may well try to install and run twenty or thirty years after it was developed.

Most software life cycles are so short that we have little experience from which to gauge the risks of such ancient code, but, inevitably, it will carry the cumulative burden of security flaws discovered over many years. It seems implausible that vendors will continue providing

security over this long period, and the risks are further compounded by the likelihood that some vendors will go out of business, possibly resulting in the loss of their source code.

CD-DRM developers must think carefully about their designs if they are to avoid compatibility problems that might damage future systems and to ensure that the vulnerabilities that will inevitably be discovered over this long period can be corrected. In planning for this extended term, conscientious CD-DRM designers will prioritize *safety* so as not to cause collateral damage, and will also strive for lasting *efficacy* so that the anticopying goals continue to be met.

Deactivating Old Software

Safety is arguably much more important than effectiveness, and it is easy to achieve if we are willing to let the DRM protections expire in the future. One approach is to design the DRM software to be inert and harmless on future systems. Both XCP and MediaMax effectively have this property, because they depend on Windows AutoRun, which is being phased out with the adoption of Vista and future Windows versions. On these newer systems, XCP and MediaMax will not start unless the user take affirmitive steps to run them, rendering them relatively inert. Since it is difficult to anticipate what features will be available on future systems, a surer approach would be to take active measures to ensure inertness. One technique is to build in a sunset date after which the software will make itself inert. Another is to prepare a kill switch that the vendor can use to initiate deactivation.

Whether by design or by obsolescence, a sunset would greatly reduce long term risks. It would have relatively little effect on record label revenue for most discs. From the buyer's perspective, most of the value of the disc is short term, after which the music loses its novelty. Over the long term, the DRM cannot boost revenue because the customer no longer care about making whatever uses the DRM attempts to stop. Thus, we expect nearly all

revenue from a particular disc to have been extracted from the customer in a short time after purchase. One way the music industry *can* extract additional revenue is to develop new markets by introducing the music to other listeners. If substantial new markets appear in the future, the producers could press more copies of the album, and these could have updated DRM software with a later sunset.

Keeping DRM Up-to-Date

Both safety and efficacy can be supported by issuing software updates that correct flaws and add countermeasures against new attacks. Updates can be subsequent pressings of the CDs, but existing CDs, whether in the supply chain or a purchaser's music collection, cannot be modified retroactively. Updates for older discs can be delivered by download, like for most PC software, or by replacing the CDs., as Sony-BMG did in a massive recall to rectify the security problems in its XCP and MediaMax albums.

Users generally do cooperate with updates that help them by improving safety or making the software more useful, but updates to retain the efficacy of the software's usage controls are a different matter. Usage controls provide no value to individual users but only reduce what the user can do with music from a disc. Often some of the uses they restrict are allowed under copyright law, so even law-abiding users have cause to reject updates that prolong DRM efficacy.

Users have many ways to stop updates from downloading or installing, such as write-protecting the installed software so that it cannot be updated, or blocking connections to the vendor's download servers using a personal firewall. System security tools of many kinds might be useful for this purpose, since they are often designed to stop unwanted network connections, downloads, and code execution. If customers widely regard CD-DRM software as malware, makers of system security tools will have an incentive to provide tools capable

of restraining it. These factors limit software updates' ability to mitigate content-copying attacks.

A DRM vendor who wants to deliver updates over the objection of users has no good options. They could simply offer updates and hope some users will not bother to block them. For the vendor and record label, this might be better than nothing. Alternatively, the vendor could take more aggressive measures to force users to accept updates, such as blocking all access to the audio content until the latest updates are installed. This is essentially what BluRay's AACS and similar "renewable" DRM systems do. They periodically change the encryption keys used by software players so that outdated versions cannot access any content released later.

This is a dangerous strategy. Locking down the content will not avoid collateral damage due to vulnerabilities in the installed software, and with users refusing to install patches, these dangers can only increase as new attacks come to light. Thus, attempts to boost long term CD-DRM efficacy through software, besides being likely to fail, may actually harm safety by leading users to block all updates to the products.

2.5 Causes of CD-DRM Security Failures

This chapter has been a case study of the design, implementation, and deployment of CD-DRM technologies. We presented detailed technical analysis of the security failures of three generations of systems developed by a number of vendors. This kind of investigation, with its focus on discovering vulnerabilities, has significant intrinsic value. Our work has benefited music labels and investors by exposing defective content security mechanisms and encouraging vendors to be honest about their capabilities and limitations. It has reduced the collateral harm from CD-DRM by revealing security holes that were subsequently corrected through software patches and product recalls. We have established that DRM systems can

pose a threat to user security, but users will be safer in the future because systems like the ones we studied are certain to be thoroughly scrutinized.

Yet there is something lacking. The chapter has so far said little about how to prevent systems from failing in the first place. Many different kinds of stakeholders are eager to know what they can do to head off future problems—vendors and record labels, users and system administrators, lawyers and policymakers, to name a few—and if we were to stop our analysis here we would have little advice for any of them.

Thus, in this section we turn our attention away from our initial question—What is wrong with the system?—and ask instead, Why did these failures occur? Our investigation is guided by the analytic security research methodology that we introduced in Chapter 1. We compare vulnerabilities in systems from different time periods and different vendors, searching for patterns of failure that suggest an informative cause—technological trends, behavioral patterns, organizational structures, theoretical limitations, or other factors that made the observed security failures more likely to occur or increased the risks they posed. Our approach to the development of causes from observed patterns of failure remains largely free-form, like the development of new attacks, but we require that they are supported by evidence from our attack investigation. We are most interested in those that have broad implications and those that suggest specific, practicable remedies.

Several themes emerge. The content security vulnerabilities we observed appear to be related to the inherent difficulty of retrofitting copy protection to an existing open format such as the audio CD. Problems resulting in collateral damage to users' security seem to result from risks that the vendors took because of a market structure that insulated them from liability. Finally, many of these security side effects were exacerbated because of tensions between DRM's goal of protecting content and consumers' desire to secure their PCs.

From these themes we draw lessons for future CD-DRM systems and for broader content security problems. Awareness of these causes, and their consequences for systems like those we have studied, should help future designers of DRM and related security products avoid making the same mistakes.

2.5.1 The CD-DRM Problem

An unmistakable pattern emerges across all three generations of systems: despite increasing sophistication, every CD-DRM product we examined was vulnerable to content copying attacks that undermined its central security goal. A pattern where every implementation in a class of systems fails to achieve its primary security goal implicates the nature of the problem that the systems attempt to solve, indicating that it may be fundamentally difficult, or even impossible. This suggests that CD-DRM's goal of selective incompatibility with PC software is a factor in its failure.

Analysis CD-DRM is premised on the need for protected discs to be usable in standalone CD players—otherwise, producers would encrypt the music and rely on more traditional DRM techniques. Thus, all CD-DRM exploits differences between PCs and CD players in order to block access to the plaintext audio only on the former.

However, as we discussed in Section 2.1, the line between PCs and CD players has increasingly blurred, with CD players becoming more PC-like and PC hardware and software outgrowing their initially brittle implementations. The past decade has seen broad adoption of entire new categories of consumer electronics with support for CD playback—DVD and BluRay players, advanced video game systems, personal video recorders (PVRs)—and this has shifted what users think of as "CD players." These devices often are built from commodity CPUs and general purpose software, which bear little resemblance to the discrete logic and ASICs inside 1980s-vintage CD players, and they frequently share CD drive components

with commodity PCs. Commoditization has driven down costs to the point that even "CD players" increasingly utilize such an architecture. At least on the hardware level, CD-DRM designers are running out of reliable distinctions [73].

Software is a different matter. PCs exhibit software monoculture [67], with a small number of operating systems and CD copying applications dominating the market; CD-DRM can target behavioral quirks in these programs. Yet these behavioral differences are not set in stone. Computers, a far more flexible technology than traditional CD players, are adapting to make differences in their behavior harder to exploit. While CD players are increasingly acting like PCs, PCs are learning to act like more drives, through software updates that correct some of the telltale quirks.

The second- and third-generation CD-DRM schemes further distinguish PCs by their ability to execute certain kinds of software, such as Windows applications. Though this is fundamental feature of computers, it is a poor trigger for a content protection feature; this is because of another capability that is not only fundamental to PCs but also essential to the security, the ability to control *which* software is allowed to run. Thus, users can make their computers appear more like CD players simply by declining to run CD-DRM software. There is evidence that this, too, is becoming easier, thanks to improvements in desktop security.

The sum of these trends effects CD-DRM systems in two ways. First, they gradually erode the effectiveness of each particular CD-DRM mechanism. We have discussed how today's CD drives are more likely to be compatible with first generation discs, while today's car CD players are more likely to be incompatible with them; the active protection systems we studied provide little protection on systems running Windows Vista, since Microsoft removed the AutoRun feature due to security concerns. Second, they leave the designers of new CD-DRM systems with fewer distinguishing behaviors on which to base their schemes.

This effect already forced designers to switch from passive to active protection methods. It is possible that someday they will have no options left.

> **Cause 2.1: PCs and CD Players are Converging**
> As a result, CD-DRM's principal goal—to produce audio CDs that are unusable with most PC ripping and copying applications while remaining compatible with almost all standalone CD players—is difficult to achieve with lasting effectiveness.

Is effective CD-DRM ultimately impossible? By assumption it must be possible to read the plaintext audio from the disc, so, of course, one could extract it with a microscope, or tap the circuits of an old CD player with a logic analyzer, but these are well beyond the capabilities of a typical user. However, we can conceive a more practical attack—one feasible with commodity hardware—that nevertheless would allow PCs to simulate a CD player from 1985 with almost arbitrary fidelity.

Our approach is based on the PC's ability, as the more general purpose device, to emulate the operation of simpler devices such as CD players. This task is made more difficult by the fact that it involves hardware devices, but on a basic level PCs have hardware with equal or greater capabilities than CD players, and hence differences in behavior that are driven by software can be eliminated.

One way to do this involves patching the firmware of popular CD drives to add support for a new read method. Patched drives would be able to download special "read control microprograms" from the host PC—software executed on a simple cross-drive virtual machine with the ability to control the drive's behavior at the lowest level. When instructed by the host, the drive invokes the microprogram, which would typically cause the drive's motor to begin scanning the spiral grove of the disc from beginning to end, at the same speed as a traditional CD player. The software could control the position of the drive's optical pickup

and would receive its raw output. The software would be responsible for decoding this data stream—making out the pattern pits and lands, decoding them into bytes and samples, and breaking them up into frames and tracks. In doing all this, the software could emulate a CD player with arbitrary precision. A drive with a 150 MHz processor will have about 100 clock cycles per bit of audio, enough time to perform moderately sophisticated decoding even with the overhead of the VM.

If the emulation failed to accurately simulate the behavior of a real CD player in some way, CD-DRM system might be able to utilize this defect. However, the deficiency could be corrected by installing an updated microprogram. These updates could be developed and distributed inexpensively thanks to the virtual machine.

With no discernable differences between the drive's emulation and various standalone CD players, any CD-DRM disc would need to be readable by the drive or else unreadable on a vast number of CD players. Though this attack consists entirely of software, it would give a large fraction of users the perpetual ability to defeat any CD-DRM copy protection that retained wide compatibility with CD players. This shows that CD-DRM's security goal cannot, in the limit, be achieved.

Of course, this analysis presumes that users have the ability to patch their drive firmware and to run other necessary software. DRM proponents might try to head off this attack and counteract the technological trends that are weakening CD-DRM. They might try to forge agreements with hardware and software markers, persuade government to adopt technology mandates, or both. The likely aim of these efforts would be to embed a form of CD-DRM into hardware or software—having them recognize protected CDs and deny access—while blocking efforts to read discs by lower-level channels. Hardware markers seem unlikely to comply voluntarily, since such behavior would be a major competitive disadvantage and several drive makers have already shipped products that tout their immunity to certain

CD-DRM. Software makers would have similar disincentives, and such blocks could in any case be bypassed by using open source software. The most significant of these possibilities—though it seems quite remove—is government intervention resembling a "broadcast flag" for audio CDs [44]. Without reciting the arguments against such a harmful policy, we can conclude that in the long run, CD-DRM's goals will be practically impossible to realize as long as computer users retain the freedom to control the operation of their computers.

> **Cause 2.2: CD-DRM Attempts the Impossible**
> CD-DRM's principal goal—to produce audio CDs that are unusable with most PC ripping and copying applications while remaining compatible with almost all standalone CD players will remain defeatable so long as users retain control over their hardware and software.

Implications

- Future CD-DRM schemes will also fail. The protections provided by a given CD-DRM system are at best temporary. Passive systems depend on incomplete or buggy implementations to differentiate between PCs and CD players, and active systems must rely for their installation on insecure features like AutoRun or on exploitable security flaws; thus in the long term, all schemes will diminish in effectiveness as hardware and software mature. In the mean time, future CD-DRM schemes may take increasingly aggressive and risky steps to prolong their effectiveness.
- CD players too are evolving, and the technical distinctions between standalone CD players and PCs are becoming less pronounced; thus, future CD-DRM systems are unlikely to fare better than past ones at protecting content unless they accept the costs of an even higher rate of incompatibility with non-PC players.

- Since CD-DRM systems do not benefit the user, their operation creates demand for hardware and software changes that render them less effective. If labels continue to use CD-DRM over an extended period, countermeasures like our CD player emulation strategy will develop; thus even the more limited goal of a temporary speed bump may someday be unattainable.

2.5.2 Incentives for Underinvestment in Security

Our analysis of XCP and of the two versions of MediaMax reveals a pattern in the flaws that resulted in collateral damage to users: time and again, they resulted from blatant or elementary mistakes—basic access permission errors, executing untrusted code from the Internet, the obviously dangerous rootkit—what might be called "rookie" security errors. A profusion of simple mistakes consistent across products or versions implicates the developers' incentives, indicating that they had reasons to underinvest in security and testing.

Analysis The key to understanding CD-DRM producer incentives is the relationship between the companies that developed the DRM systems (the vendors) and the company that deployed them (the record label). Both XCP and MediaMax were designed and engineered by independent companies—First4Internet and SunnComm, respectively—whose primary business activity was selling the copy protection. The vendors were small companies: SunnComm was a "penny stock," with a market value in the range of $10 million at the time of the rootkit incident, and First4Internet, while privately traded, apparently had only a handful of employees at its peak. The vendors marketed their systems to large music conglomerates including Sony, which had a market capitalization exceeding $35 billion in December 2005.

The CD DRM vendors' primary goal was to create value for the record labels in order to maximize the price the labels would pay for the DRM technologies. In this respect, the

vendors' and labels' incentives were aligned. However, the vendors' incentives diverged from the labels' in at least two ways.

First, the vendors had a higher risk tolerance than most labels, because record labels tend to be large, established businesses with valuable brand names, while the vendors were start-up companies with few assets and not much brand equity. Start-ups face many risks already and are therefore less averse to taking on one more risk. They were also relatively isolated from liability if their products cause damage. With few assets, in the worst case they could simply cease operations and declare bankruptcy–and, in fact, both vendors have. Record labels, on the other hand, have much more capital and brand equity to lose if something goes horribly wrong. Therefore, the vendors saw large upside from making their products more attractive to labels by taking aggressive but dangerous copy protection measures, but they faced limited downside from damage to users if the risks if they took turned out unfavorably. Accordingly, we would expect the vendors to be much more willing to accept security risks than most labels.

The second incentive difference is that the vendors could monetize the installed platform in ways the record label could not. For example, once the vendors' DRM software was installed on a user's system, the software could control use of other labels' CDs, so a larger installed base made the vendor's technology more attractive to other labels. This extra incentive to build the installed base made the vendors more aggressive about pushing the software onto users' computers than most labels would be.

We found evidence of these incentive differences in both XCP and MediaMax. MediaMax's aggressive installation served their platform building strategy and promoted the content security objectives that were the primary selling point of their product. For similar reasons, both XCP and MediaMax employed uninstallers with atypically complex designs in order to inhibit removal of the copy protection software. In each of these cases, the

programmers made elementary security mistakes that caused vulnerabilities on computers where the discs were used.

In short, incentive differences made the vendors more likely than the record labels to (a) underinvest in security design and testing and accept security risks, and (b) push DRM software onto more users' computers. If a label had perfect knowledge about the vendors' technology, this incentive gap would not be an issue—the label would have simply insisted that the vendor protect the label's interests. But if, as seems to have been the case, the labels had imperfect knowledge of the technology, then the vendors would be expected to sometimes act against the labels' interests. (For a discussion of differing incentives in another content protection context, see [66].)

Cause 2.3: Vendors were Insulated from Insecurity Risks
The CD-DRM vendors had limited exposure to the costs of collateral damage caused by their software; this gave them incentives to underinvest in security and testing compared to typical application developers, instead devoting resources to directly profitable aim, such as strengthening content security and building their protection platforms.

The labels were, in practice, poorly equipped to evaluate software security, since their primary expertise was in the unrelated business of music production and distribution. We find evidence for this the now-infamous public comments about the impact of the rootkit by Thomas Hesse, president of Sony's global digital business division: "Most people, I think, don't even know what a rootkit is, so why should they care about it?" [167] Assuming that this reflects his genuine views about the seriousness of the situation, it illustrates that Sony's management were both ignorant of both the security risks and of their own inability to competently assess them.

Implications

- Incentives have a powerful effect on security. The CD-DRM vendors were the party best equipped to reduce security risks by choosing safer designs, employing more careful engineering practices, and conducting security reviews and testing. As Varian and others have argued [171], if the party who is in the best position to prevent problems does not have adequate incentives to do so, security suffers.
- Customers of DRM products (i.e., labels and end users) should be particularly cautious about security. They should seek external review of the products' security and the vendors' security practices before making purchasing decisions. Where vendors' and labels' incentives differ, DRM systems may not serve the labels interests; such costs may not have been considered in the labels' strategic calculus.
- Security consumers should be wary of other scenarios where producer incentives are similarly misaligned or where the producer bears similarly little risk in case of catastrophe, such as other cases where software is built by small startups. This suggests negative security implications from software outsourcing more generally.[2]
- One way to reduce risks of collateral damage in other DRM systems would be to force vendors to share in the risks. Merely holding the vendors financially responsible after the fact would not be sufficient, since, as we have noted, small start-ups tend to have few assets. A better approach might be for the vendors' customers to require them to carry insurance in case of a security catastrophe. This would have two salutary effects. First, it would help ensure that the victims of such damage would be made whole. Second, it would encourage better security practices on the part of the vendors, since insurance underwriters would presumably require an efficient investment in reducing security risk.

2 Other studies have discussed related security risks due to software outsourcing, such as the threat of sabotage [109].

- These problems could have been lessened or avoided through greater transparency about the design and operation of the CD-DRM systems. The labels lacked the technical expertise needed to evaluate the systems' security risks. They could have cheaply harnessed the widespread expertise of the security community by publishing details about how the systems worked before deploying them. Feedback from researchers and other experts would have highlighted many of the potential risks. However, as we discuss below, there may be tradeoffs between the transparency required for PC security and the secrecy needed to accomplish DRM objectives.

2.5.3 Tension between DRM and PC Security

Some collateral damage from the third-generation CD-DRM schemes reflects deliberate choices by the vendors to adopt obviously risky or harmful designs. There can be little question that XCP's rootkit functionality, the installation without consent of MediaMax software, and the lack of uninstallers were put in place deliberately by the vendors. These problems are symptomatic of a tension between CD-DRM and PC security.

This tension takes two forms. First, there is an inherent conflict between the methods that DRM producers need to employ in order to protect content using software and the administrative practices that PC owners need to use in order to maintain security. Second, in the name of protecting content, DRM producers employ techniques and mechanisms that cause greater damage when they contain exploitable bugs than do those employed in most typical application software.

Analysis The use of rootkits and other spyware-like tactics harms users by undermining their ability to manage their computers. If users lose effective control over which programs run on their computers, they can no longer correct vulnerabilities by patching programs,

reduce their attack surface by removing unneeded software, or make informed decisions about the security risks they face.

Concealed programs may contain security flaws, or they may contain faults that hamper the operation of the computer. The XCP rootkit itself did both. As Russinovich and others have pointed out, these problems illustrate the danger of installing software in secret. Users experiencing system instability due to bugs in the rootkit would have great difficulty diagnosing the problem, since they likely would be unaware of the rootkit's presence. Managing a system is difficult enough without spyware tactics making it even harder.

Though it is no surprise that spyware tactics would be attractive to DRM designers, it is a bit surprising that mass-market DRM vendors chose to use those tactics despite their impact on users. If only one vendor had chosen to use such tactics, we could write it off as an aberration, but we found evidence of it in both XCP and MediaMax 5. Why would they do this? One possible explanation is that effectively implementing many kinds of DRM in software *requires* employing techniques that limit the user's ability to monitor and control the operation of the PC.

The reason is simple: most DRM requires keeping secrets from the user. These might include decryption keys for protected content, the plaintext content itself, or the mechanisms for storing state associated with usage limitations. If users were able to monitor and fully understand how the DRM software operated, they would be able to bypass its restrictions.

This is not the case for every DRM design. Systems that involve a secure hardware component such as a TPM [166] could remain fully transparent except for some secret stored in the hardware. Other designs might perform decryption in a secure output device, allowing the software running on the main PC to be entirely open. However, DRM running entirely in PC software has nowhere else to store secrets, so it needs to obscure parts of the PC's operation from the user.

Though most examples of this are less blatant than the rootkit and surreptitious installation we observed in the context of CD-DRM, many other software DRM systems limit users' ability to understand the programs they install. SecuROM, a DRM system for PC video games, detects administrative tools like Process Monitor as well as debuggers and virtual machine monitors and refuses to allow access to protected content while they are running [152]. Another PC game DRM system called SafeDisc was quietly bundled with most versions of Windows XP systems, even those destined for business customers [114]. Some versions of iTunes for Mac OS X disable the OS's system call tracing mechanism [101]. Many DRM systems use software obfuscation to conceal elements of their operation. These examples harm users' security by making it more difficult to search for security problems in the DRM applications and harder to identify application behaviors that create potential security risks.

Cause 2.4: Content Security Opposes User Security
Users need the ability to monitor, understand, and control the operation of their computers in order to maintain effective security; DRM software needs some component that the user cannot monitor, understand, and/or control in order to effectively secure content. Thus, creating secure DRM software requires tradeoffs that inevitably weaken user security.

A second way in which DRM tends to harm the security of users' PCs is by behaviors that, while not necessarily harmful in themselves, produce unusually severe security harm when they contain vulnerabilities. Often these behaviors are employed to strengthen the content protection mechanism, but they force the user to bear additional risks. Typical examples include aggressive installation, running code in kernel mode, installing automatic updates without user confirmation, and executing code from a remote system.

Because of these risky techniques, small errors in the third generation CD-DRM schemes caused disproportionately harmful collateral damage. For example, the MediaMax file permission problem was made more severe because parts of the MediaMax software are installed automatically and without consent. Users who have declined the EULA likely assume that MediaMax has not been installed, and so most will be unaware that they are vulnerable. The same installer code performs a dangerous version check as soon as the CD is inserted, leading to a privilege escalation vulnerability. A CD that prompted the user to accept a license before installing code would give the user a chance to head off the attack. Similarly, the vendors' apparent desire to limit use of their uninstallers led to designs that relied on downloading code using ActiveX controls—leaving users just one design flaw away from being attacked. Indeed, these were such flaws: Both the XCP and MediaMax uninstallers failed to verify the authenticity of the downloaded code, leading to remote code execution vulnerabilities.

Obviously, these vulnerabilities could have been prevented by careful design and programming. But they were only possible at all because the vendors chose to deliver the uninstallers via this ActiveX method rather than using an ordinary download. We conjecture that the vendors made this choice because they wanted to retain the ability to rewrite, modify, or cancel the uninstaller later, in order to further their platform building strategies.

These aggressive installer and uninstaller designs were more difficult, compared to traditional installers and uninstallers, for both vendor and user, so the vendors must have benefited from them somehow. One benefit is to the vendors' content protection goals, since CD-DRM content is most strongly protected with the active protection software running. Another benefit is to the vendors' platform building strategy, which takes a step backward every time a user uninstalls or declines to install the software. Customizing the

uninstaller, for instance, allows the vendor to control who receives the uninstaller and to change the terms under which it is delivered.

We have observed that successive generations of CD-DRM systems turned to increasingly aggressive and risky techniques in order to maintain their effectiveness, and the pressure to do so is sure to increase in response to the evolving technical environment. We would expect future CD-DRM schemes to use even more aggressive installation strategies, such as paying to bundle themselves with other software, as Macrovision paid Microsoft to include its SafeDisc game DRM system with Windows, or perhaps even exploit actual security flaws to install or to run privileged code. Rising costs of installation will increase the value of the DRM platform, causing programs to install even more aggressively and resist uninstallation even more tenaciously.

Cause 2.5: DRM Software Causes Elevated Security Risks
DRM software tends to engages in risky behaviors that, while not inherently harmful, expose users to unusually dangerous attacks if they are implemented incorrectly. These behaviors include executing with root privileges, installing updates without confirmation, and running code from remote sources. Market pressures may drive vendors to take increasing risks in the future.

Implications

- These tendencies to adopt risky techniques explain part of the outrage that users felt in the wake of the Sony Rootkit debacle. Any piece of software carries security risks; DRM is unique because it does not deliver a justificatory benefit to the user in return. Many users did not expect audio CDs to contain software. Users did not want the software, and they recognized that Sony-BMG chose to include it anyway. Unlike (say) a web browser, which necessarily includes complex software components that might

have bugs, CDs need not include software, so users are less willing to accept the risk of security problems in order to get CDs.

- In light of DRM's inherent tension with PC security and the elevated risks posed by techniques that DRM vendors choose to adopt, vendors arguably owe their customers a higher standard of care than do typical application developers. Unfortunately, as we have noted, misaligned incentives between vendors and content distributors can lead vendors to pay less attention to security than normal software creators would.
- The collateral damage caused by XCP and MediaMax surprised many observers. Though millions of dangerous CDs were sold, the problems were not reported by security researchers or security product vendors for seven months. Now that we are aware of these tendencies, other DRM software deserves higher security scrutiny from users and researchers, and we can expect that further severe vulnerabilities will come to light in other DRM software. Unfortunately, the vendors' intentional secrecy and the chilling effects of the DMCA inhibit investigations in this area. There is a need for greater transparency about DRM operation.

2.6 Conclusion

Our analysis of security failures in CD-DRM systems carries wider lessons for content companies, DRM vendors, policymakers, end users, and the security community. We draw four main conclusions.

First, the design of DRM systems is driven strongly by the incentives of the content distributor and the DRM vendor, but these incentives are not always aligned. Where they differ, the DRM design will not necessarily serve the interests of copyright owners, not to mention artists. Policy changes may help realign these incentives.

Second, DRM, even if backed by a major content distributor, can expose users to significant security and privacy risks. Incentives for aggressive platform building drive vendors toward aggressive tactics that exacerbate these risks.

Third, there can be an inverse relation between the efficacy of DRM and the user's ability to defend her computer from unrelated security and privacy risks. The user's best defense is rooted in understanding and controlling which software is installed, but many DRM systems rely on undermining this understanding and control.

Fourth, CD-DRM systems are mostly ineffective at controlling uses of content. Major increases in complexity have not increased the effectiveness of second- and third-generation schemes, and may in fact have made things worse by creating more avenues for attack. We think it unlikely that future CD-DRM systems could do better.

We probably will not have a chance to find out, since it seems doubtful that any future CD-DRM systems will be deployed. The disclosure in 2005 of the XCP rootkit and other collateral damage caused by XCP and MediaMax sparked widespread public outrage, class action suits against Sony, and investigations by the Federal Trade Commission and several state attorneys general. Sony eventually recalled the CDs containing the dangerous DRM and offered compensation to the affected customers. SunnComm and First4Internet left the content protection business and apparently ceased operation, but not before Sony sued SunnComm for negligence and breach of contract. Their main competitor, Macrovision, seems to have quietly withdrawn from the CD-DRM market, and there have been no reports of copy protected audio CDs being marketed in the United States since that time. By 2008, the major record labels had completely shifted their strategy and began selling unencumbered MP3s of most their catalogs. DRM for purchased music appears to be a thing of the past.

Though other DRM makers and copyright owners will presumably tread more carefully in the future, there remains a fundamental tension between DRM vendors' desire to control and limit how computers are used, and the need of users to manage their own systems. Users and DRM distributors will continue to struggle for control of users' computers, and other DRM systems will inevitably cause collateral damage. Security researchers—and policymakers—need to remain alert.

Chapter 3

Security Failures in Electronic Voting Machines

This chapter is a security analysis of the Diebold voting system, which consists primarily of the AccuVote-TS or AccuVote-TSX direct recording electronic (DRE) voting machines and the GEMS election management system. It is based an examination of an AV-TS that we obtained from a private party in 2006 [61], and on a study of the system's source code that we conducted at the request of the California Secretary of State as part of a "top-to-bottom" review of California voting systems [28].

Our analysis shows that the Diebold system does not provide sufficient security to guarantee a trustworthy election. It contains vulnerabilities that could allow an attacker to install malicious software on voting machines or on the election management system. Malicious software could cause votes to be recorded incorrectly or to be miscounted, possibly altering election results. It could also prevent voting machines from accepting votes, potentially causing long lines or disenfranchising voters.

Figure 3.1—Machines We studied the Diebold AccuVote-TS *(top)* and AccuVote-TSX *(bottom)* direct-recording electronic (DRE) voting machines.

Furthermore, the Diebold system is susceptible to computer viruses that propagate from voting machine to voting machine and between voting machines and the election management system. A virus could allow an attacker who only had access to a few machines or memory cards, or possibly to only one, to spread malicious software to most, if not all, of a county's voting machines. Thus, large-scale election fraud in the Diebold system does not necessarily require physical access to a large number of voting machines.

The remainder of this chapter is structured as follows:

Section 3.1 describes the overall architecture and individual components of the Diebold election system and how the system is operated under typical election procedures. In Section 3.2, we report specific vulnerabilities in the AccuVote-TS and AccuVote-TSX, and in Section 3.3 we show how an attacker could combine them to steal votes, disrupt an election, or spread a voting machine viruses. Section 3.4 takes a higher-level view, identifying systemic problems in the design and implementation of the system.

All DREs face fundamental security challenges that are not easily overcome. Section 3.5 reviews other studies of DRE security, focusing on similar weaknesses discovered in systems from other manufacturers. Patterns of failures suggest a number of causes that contributed to the security failures in these systems, which we analyze in Section 3.6.

Section 3.7 discusses several technical and procedural strategies for improving the security of the Diebold voting system, together with their limitations. However, the severity of the design flaws in the Diebold system and many similar electronic voting systems, and our lack of confidence in the ability of changes in election procedures to compensate for them, leads us to conclude in Section 3.8 that the surest way to repair these systems is to redesign them.

3.1 The AccuVote TS and TSX

The Diebold[1] AccuVote-TS (AV-TS) and its newer relative the AccuVote-TSX (AV-TSX) were in recent years the most widely deployed direct-recording electronic (DRE) voting platform in the United States. In the November 2006 general election, these machines were used in 385 counties representing over 10% of registered voters [54]. The majority of these counties — including all of Maryland and Georgia — employed the AccuVote-TS model. More than 33,000 of the machines were in service nationwide at that time.

The AV-TS was launched in 2001 by Global Election Systems, which was purchased by Diebold later that year. By 2006, it was the most widely used touch screen DRE in the U.S. That year, we obtained an AV-TS from a private party and analyzed the machine's hardware and software for security problems. The machine came loaded with version 4.3.15 of BallotStation, the software that runs the election. We were aided by access to the source code to an earlier version, 4.3.1, which leaked to the public two years before. We published a full report [61] in September 2006.

In 2007 we were invited to participate in a new voting machine security study commissioned by the Secretary of State of California and encompassing all the electronic voting systems then in use in that state. As part of this study, we analyzed the security of the Diebold voting system, including the AV-TSX, an updated model introduced in 2003. We were given full source code access but no direct access to the hardware. Among the software we evaluated was GEMS version 1.18.24.0 (Diebold's election management software), BallotStation version 4.6.4, and the AccuVote bootloader BLR 7-1.2.1. The Secretary released our findings [28] in July 2007.

1 Diebold's election division has since rebranded itself "Premier Election Solutions." We continue to use the original name for consistency with our earlier reports.

Before discussing vulnerabilities and attacks, we will first describe the design and operation of the AV-TS and AV-TSX voting machines and the GEMS election management system.

3.1.1 Voting Machine Hardware and Software

The AV-TS and AV-TSX are DRE voting machines. They interact with the voter via a touch-screen LCD display, and they support audio ballots for increased accessibility. They authenticate voters and election officials using a motorized smart card reader, which pulls in cards after they are inserted and ejects them when commanded by software. They include built-in rechargeable batteries to allow voting to continue if power is interrupted.

The machines are configured for each election by inserting a memory card into a slot behind a locked door on the side of the machine. The memory card is a standard PCMCIA flash storage card (a normal Compact Flash card and passive adapter may be substituted). Before the election, the file system on the memory card stores the election definition and other configuration information. After the election, poll workers remove the memory card from the machine and send it to election headquarters so that the electronic vote records can be uploaded for tabulation.

A lockable metal door limits physical access to the memory cards. On the AV-TS, the lock is a standard pin-tumbler cylinder of a lightweight variety commonly used in desk drawers and file cabinets. On the AV-TSX it is a tubular pin-tumbler design often found on bicycle locks.

The AV-TS and AV-TSX both contain smaller thermal roll printers for recording initial and final vote totals. The AV-TSX can also make use of a separate printer attachment for generating a voter-verifiable paper audit trail (VVPAT) record for each cast ballot.

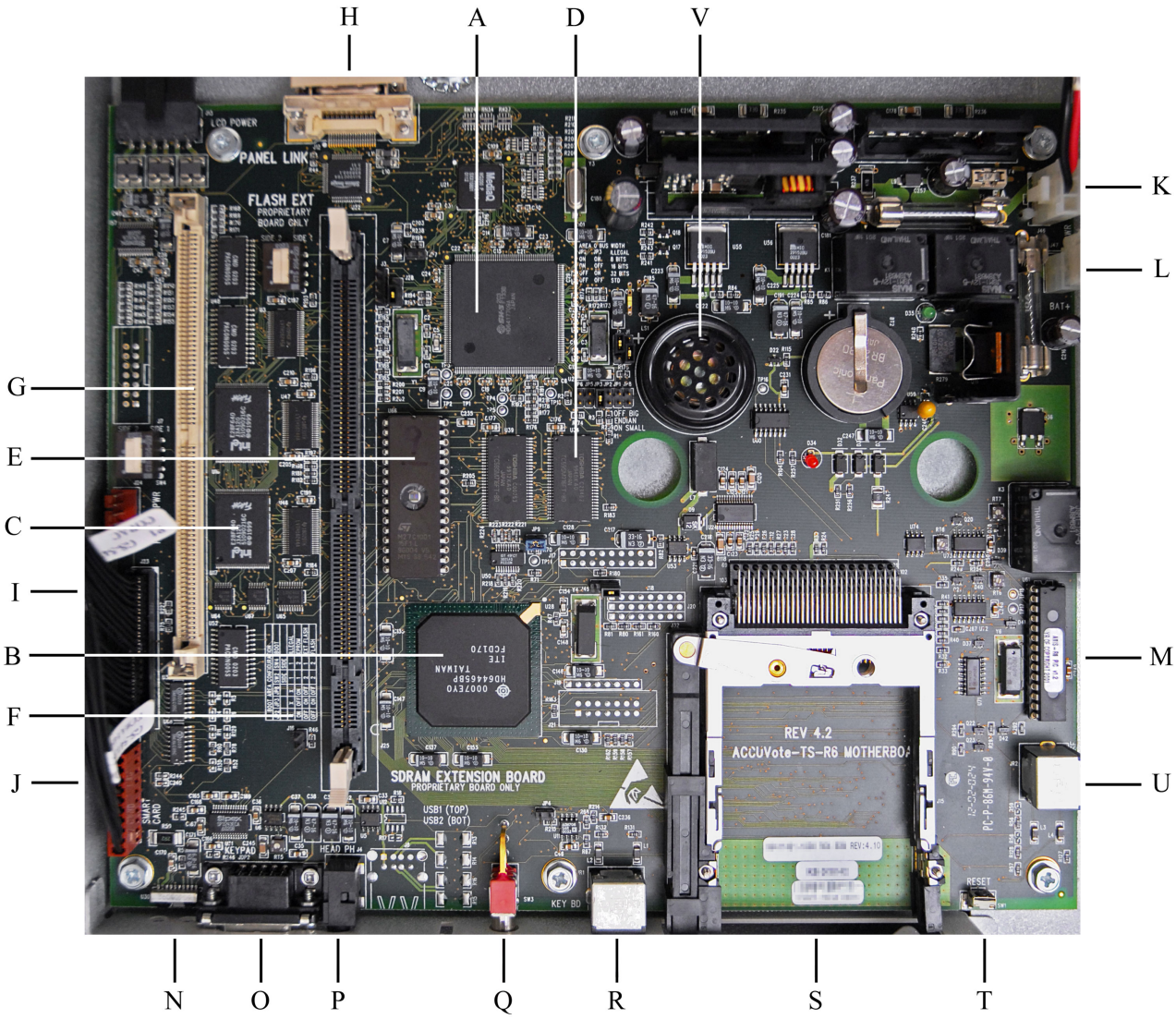


Figure 3.2—AccuVote-TS Motherboard Components: *(A)* Hitachi SuperH SH7709A 133 MHz RISC microprocessor; *(B)* Hitachi HD64465 Windows CE Intelligent Peripheral Controller; *(C)* Intel Strata-Flash 28F640 8 MB flash memory chips (2); *(D)* Toshiba TC59SM716FT 16 MB SDRAM chips; *(E)* M27C1001 128 KB erasable programmable read-only memory (EPROM), in 32-pin socket; *(F)* startup jumper table; *(G)* "Flash Ext" connector; *(H)* touch sensitive LCD panel; *(I)* printer port (connected to thermal roll printer, not shown); *(J)* Smart Card device connector (connected to SecureTech ST-20F reader/writer, not shown); *(K)* D.C. power supply connector; *(L)* battery connector; *(M)* PIC microcontroller; *(N)* IrDA transmitter and receiver; *(O)* serial keypad connector; *(P)* headphone jack; *(Q)* power switch; *(R)* PS/2 keyboard port; *(S)* PC Card slots (2); *(T)* reset switch; *(U)* PS/2 mouse port; *(V)* internal speaker.

Internally, the hardware in both machines strongly resembles that of a laptop PC or a Windows CE hand-held device. The AV-TS motherboard (see Figure 3.2), includes a 133 MHz SH-3 RISC processor, 32 MB of RAM, and 16 MB of nonvolatile flash storage. The AV-TSX motherboard is somewhat more powerful, with a 32-bit Intel xScale processor, 64 MB of RAM, and 32 MB of flash storage. Both machines run Microsoft's Windows CE operating system; the AV-TS runs version 3.0 and the AV-TSX runs version 4.1.

In normal operation, when the machine is switched on, it loads a small, custom boot-loader program from its on-board flash memory. The bootloader loads the operating system, which then executes a proprietary application called BallotStation that provides the user interface for voters and poll workers. BallotStation interacts with the voter, accepts and records votes, counts the votes, and performs all other election-related processing.

3.1.2 Election Management

Election officials manage the voting process using a back-end application called GEMS that runs on a desktop PC at the election headquarters. GEMS is used to conduct many aspects of the election, including designing ballots, downloading election definition files to voting machines, compiling election results, and reporting the election outcome. GEMS is a Windows application, and it typically runs on a system configured by the vendor running Windows 2000 or Windows XP.

At the election headquarters, one or more AccuVote units would typically be connected to the GEMS PC by Ethernet. These machines are identical to units used in the polling place, but they serve a different function: they are used to read and write voting machine memory cards. Election staff insert a PCMCIA Ethernet card into one of the PCMCIA slots on the voting machine and then connect it to an Ethernet hub. The BallotStation application

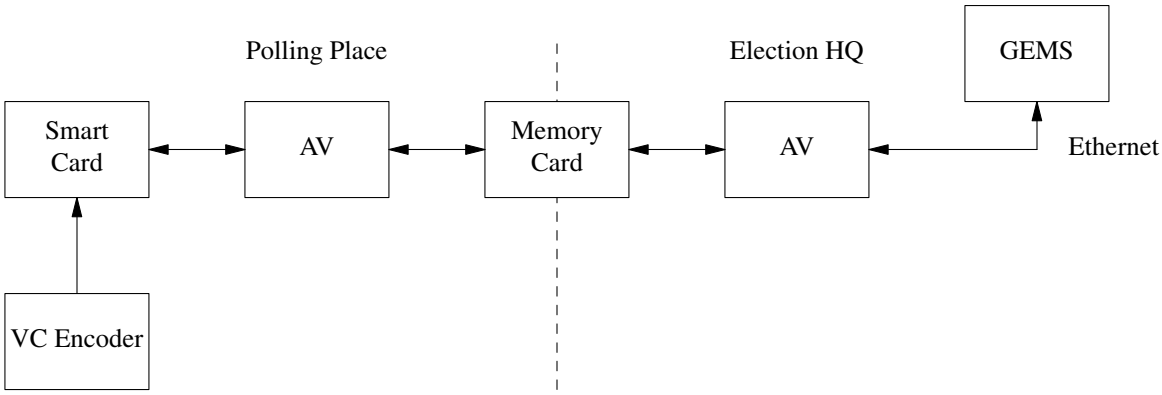


Figure 3.3—Components of the Diebold System In a typical county, there is a GEMS server at the election headquarters that is connected via Ethernet to one or more central-office AccuVote machines. These machines read and write memory cards, which are used to transfer ballots to machines at the polling place and to read back election results. Polling places also contain voter card encoders, which program smart cards that allow voters to access the voting machines.

interfaces with the GEMS server over the network and provides result upload and ballot programming capabilities.

Before the election, staff use GEMS and the networked AccuVote units to write election definition files onto memory cards. They must prepare one memory card per AccuVote unit that will be deployed in the field by inserting a memory card into the networked AccuVote and executing a programming function in GEMS. After the election, as poll workers return memory cards to the election headquarters, workers use the networked AccuVote units to read results files from the memory cards and upload them to GEMS for tabulation.

3.1.3 Voting Machine Operation

All of the machine's voting-related functions are implemented by BallotStation, a user-space Windows CE application. BallotStation operates in one of four modes: Pre-Download, Pre-Election Testing, Election, and Post-Election. Each mode corresponds to a different phase of the election process and is intended to have its own associated election procedures. Here we describe the software's operation under typical election procedures. Our understanding

of election procedures is drawn from a number of sources including [150, 55, 164, 180] and discussions with election workers from several states. Actual procedures may vary somewhat from place to place.

Election Setup Prior to the election, poll workers may configure BallotStation by inserting a memory card containing a ballot description—essentially, a list of races and candidates for the current election—prepared earlier using GEMS. If, instead, a card containing no recognizable election data is inserted into the machine, BallotStation enters Pre-Download mode and awaits instructions from a GEMS server.

After election definitions have been installed, BallotStation enters Pre-Election Testing mode. Among other functions, Pre-Election Testing mode allows poll workers to perform so-called "logic and accuracy" (L&A) testing. During L&A testing, poll workers put the machine into a simulation mode where they can cast several test votes and then tally them, checking that the tally is correct. Because the voting software is in L&A mode, these votes are not counted in the actual election. After any L&A testing is complete, the poll workers put the machine into Election mode. The software prints a "zero tape" which tallies the votes cast so far. Since no votes have been cast, all tallies should be zero. Poll workers check that this is the case, and then sign the zero tape and save it.

Voting When a voter arrives at the polling place, she checks in at a front desk where several poll workers are stationed. She is verified against a list of registered voters. Assuming the voter is registered and has not yet voted, poll workers record that the voter has voted. At this point the poll workers give the voter a "voter card," a special smart card that signifies that the voter is entitled to cast a vote. The voter waits until the voting machine is free and then approaches the machine to cast her vote.

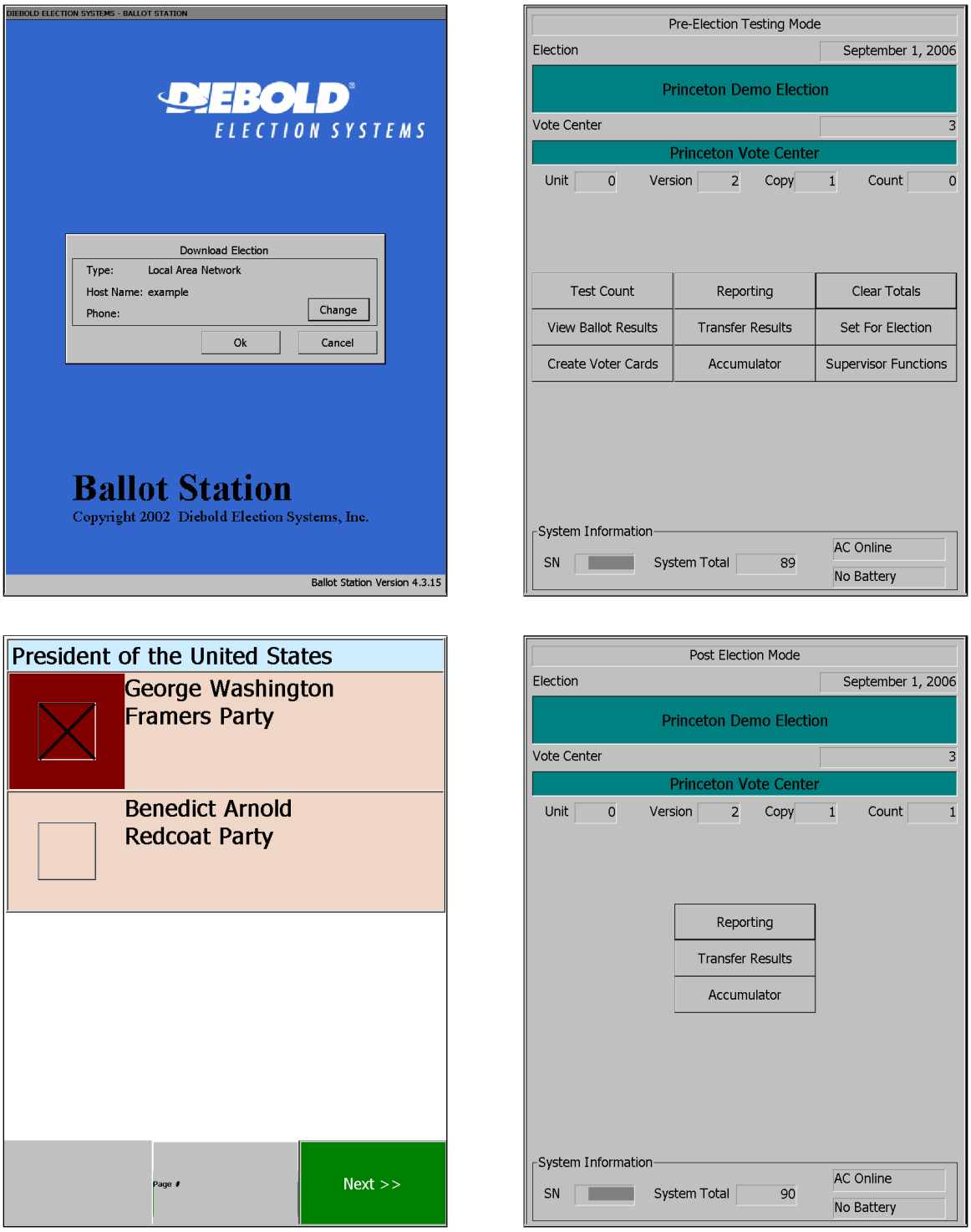


Figure 3.4—Diebold BallotStation Software Screenshots depicting BallotStation's four primary modes: *(a)* Pre-Download, *(b)* Pre-Election Testing, *(c)* Election, and *(d)* Post-Election.

To cast a vote, the voter first inserts her voter card. The machine validates the voter card and presents the voter with a user interface allowing her to express her vote by selecting candidates and answering questions. (Voter using AV-TSX machines, before casting their ballots, may have an opportunity to examine printed VVPAT records and confirm that they accurately represent their intent.) After making and confirming her selections, the voter pushes a button on the user interface to cast her vote. The machine modifies the voter card, marking it as invalid, and then ejects it. After leaving the machine, the voter returns the now-invalid voter card to the poll workers, who may re-enable it for use by another voter.

Post-Election Activities At the end of the election, poll workers insert a special smart card called an "Ender Card" or "Supervisor Card" to tell the voting software to stop the election and enter Post-Election Mode. Poll workers can then use the machine to print a "result tape" showing the final vote tallies. The poll workers check that the total number of votes cast is consistent with the number of voters who checked in at the front desk. Assuming no discrepancy, the poll workers sign the result tape and save it. Members of the public are invited to watch this procedure and to see the contents of the result tape, including the vote tallies.

After the result tape is printed, the election results are transferred to the central GEMS system. Most often this is done by physically transporting the memory cards back to the election headquarters, though machines may be configured to upload results via modem for faster returns. Once results from all machines have reached the central tabulator, the tabulator can add up the votes and report a result for the election.

If a recount is ordered, the result tapes are rechecked for consistency with voter check-in data, the result tapes are checked for consistency with the results stored on the memory cards, and the tabulator is used again to sum up the results on the memory cards. Further investigation may examine the state stored on memory cards and a machine's on-board

file system, such as the machine's logs, to look for problems or inconsistencies. If AV-TSX machines were used, workers may also perform a manual audit of the printed VVPAT records.

3.2 Selected Vulnerabilities

We found numerous vulnerabilities in the AV-TS and AV-TSX machines that threaten the accuracy of the count and the availability of the voting system. Here we describe a few of these problems, focusing on ones that might be exploited as part of a large-scale attack.

3.2.1 Unauthenticated Software Update Mechanisms

Low-level software The AV-TS and AV-TSX include a software update mechanism that allows new bootloader and operating system software to be installed from a memory card inserted into the machine. When the machine boots, it searches the memory card for specially named files. If it finds a file named `fboot.nb0` (for the AV-TS) or `eboot.nb0` (for the AV-TSX), it replaces the bootloader software stored in its internal flash memory with the contents of the file. If it finds a file named `nk.bin` (for either machine), it replaces the Windows CE operating system image [28, 80, 61].

The machines have no effective mechanism for checking the authenticity of these software updates [80]. While they do employ a simple checksum to make sure the files have not been garbled in transmission, they fail to utilize a digital signature or other mechanism that would prevent an attacker from using the software update feature to install malicious code. This means that an attacker can create malicious software updates containing arbitrary code, and the software update mechanism will not be able to distinguish these from legitimate upgrades.

An attacker who has temporary physical access to a memory card — or control of any machine into which a memory card is inserted — can place his own malicious software update files on the card, and this software will be installed on any AV-TS or AV-TSX machine that is booted with that card in place. Alternately, an attacker with unsupervised physical access to the machine for as little as a minute could replace the installed memory card with one containing a malicious software update prepared earlier, boot the machine to install the update, and then reinsert the original memory card. This attacker would need to bypass the lock on the memory card door, but on either machine this is quickly accomplished [2, 61]. Tamper-evident seals might be used to deter attacks, but, as Johnston has demonstrated [88], such seals can usually be defeated with simple techniques.

The machines install these updates without asking the local user for confirmation. While they do display a message indicating that an update is taking place, this message is displayed in a small font and could easily be overlooked by poll workers. The lack of confirmation increases the odds that this issue could be used to spread voting machine viruses.

Application software A second software update mechanism operates after the AV-TS or AV-TSX boots into Windows CE. The Windows kernel launches a program called `taskman.exe`, which eventually starts the BallotStation application. Before running BallotStation, `taskman.exe` searches the memory card for files with the extension `.ins`. These are software update packages in a Diebold proprietary format. Each `.ins` file contains instructions for installing one or more files onto the machine's file system, along with the data to be placed in those files.

Like the bootloader-based update mechanisms described above, the `.ins` update mechanism does not attempt to verify the authenticity of the updates, and it does not maintain a log of software updates that have been performed [80]. Unlike the other mechanisms, it does ask the user to confirm each update by touching a button on the screen. How-

ever, the system trusts update files to accurately describe their contents. Malicious updates could inaccurately describe their purpose, possibly fooling operators into consenting to their installation [28, 61].

Attackers could use the application software update mechanism to install arbitrary code onto the voting machine. For example, they could replace the BallotStation executable file with an altered version. They could also replace other files on the system or destroy data by replacing it with garbage. Conducting such attacks would normally require the ability to write the update file onto a memory card as well as the operator's consent, but even this safeguard can be bypassed because of the vulnerability we describe next.

The code in `taskman.exe` responsible for installing the application updates contains multiple buffer overflows in the way it parses `.ins` files that could allow an attacker to execute arbitrary code on the AV-TS and AV-TSX. For example, a malicious `.ins` file could modify files in the machine's filesystem or launch programs, and it could do these whether or not the user consented to its installation. To exploit these vulnerabilities, an attacker would need the ability to write files onto the memory card. Since the machine does not verify the authenticity of `.ins` files, no secret keys would be required [28, 61].

Service back door The `taskman.exe` program in the AV-TS and older versions of the AV-TSX contains a service feature that causes the system to boot into Windows Explorer instead of the BallotStation application if a file named `explorer.glb` is present on the memory card. Like the special update files that the bootloader recognizes, `explorer.glb` contains no authentication of any kind; its contents are arbitrary. [61] Though some versions of the AV-TSX software contain this feature [80], it was not present in the AV-TSX source code we analyzed.

This feature would allow an attacker with physical access to the machine to install and run malicious software manually from a memory card he has prepared. He could also access the Windows Start menu and control panels, as on an ordinary Windows CE machine.

3.2.2 Unprotected Hardware Debugging Features

AV-TSX The motherboard inside the AV-TSX contains a jumper header marked "Debug" [80]. When a jumper is installed here, the machine's bootloader provides a service menu over its serial port when the machine boots. A feature of the service menu called the "mini monitor" allows arbitrary memory locations to be read or written over the serial port. Since the system maps its internal flash memory into the address range 0xA0000000–0xA4000000, the mini monitor can be used to extract or alter the data stored there. To access this feature, an attacker would need physical access to the inside of the voting machine, which requires removing a small number of screws. The attacker would then interact with the bootloader by attaching a serial cable to the "VIBS Keypad" port in the rear of the machine.

An attacker with access to the inside of the machine's case could also read and write the internal flash memory using the motherboard's JTAG interface. JTAG, the standard IEEE 1149.1 hardware debugging interface, provides faster reading and writing of flash data than the serial port. Unlike the "mini monitor" feature, which is implemented in the bootloader, the JTAG vulnerability cannot be corrected with a software update. We successfully used the JTAG feature to backup and restore the AV-TSX flash memory using a Macraigor Wiggler JTAG reader and a modified version of the open source JTag Tools software.

Using either the mini monitor or JTAG, an attacker could alter any part of the voting machine's software, including the bootloader, operating system, and BallotStation application. These interfaces can also be used to create a perfect copy of the internal flash

memory, including the machine's software, cryptographic keys, and records retained from past elections. These would be useful in constructing future attacks.

AV-TS Though the AV-TS does not contain a JTAG port or mini monitor software, it does have a functionally similar vulnerability. A set of two switches and two jumpers on the motherboard controls the source of the bootloader code that the machine runs when it starts. On reset, the processor begins executing code beginning at address 0xA0000000. The switches and jumpers control which of three storage devices—the on-board flash memory, an EPROM chip in a socket on the board, or a proprietary flash memory module in the "ext flash" slot—is mapped into that address range. A table printed on the motherboard lists the switch and jumper configurations for selecting these devices [61].

An attacker with physical access to a machine could use this feature to install malicious software or create a copy of the machine's internal flash memory. He could create new bootloader-level software that would modify or extract the flash memory. Then he could copy this onto an EPROM chip, install the chip into the motherboard, set the jumpers to boot from the EPROM, and start the machine.

We implemented these capabilities in the form of an EPROM-based bootloader that can backup and restore the complete contents of the machine's flash memory. We began by disassembling the original bootloader (contained on the EPROM that came with the machine) using IDA Pro Advanced [41], which supports the SH-3 instruction set. We created a modified bootloader that searches any memory card in the PC Card slot for files named `backup.cmd` and `flash.img`. If it finds a file named `backup.cmd`, it writes the contents of the on-board flash to the first 16 MB of the memory card, and if it finds a file named `flash.img`, it replaces the contents of the on-board flash with the contents of that file [61].

3.2.3 Exploitable Buffer Overflows in BallotStation

The BallotStation software on the AV-TSX contains multiple buffer overflow errors in code that processes untrusted data. An attacker could exploit these problems to execute arbitrary code on the machine. (The AV-TS may contain identical or similar problems, but we did not investigate this.) We describe these vulnerabilities briefly below, but the nondisclosure agreement connected to the California Top-to-Bottom-Review (TTBR) prevents us from providing further details.

Bug in assure.ini Processing When BallotStation starts, it reads the file `assure.ini` from the memory card. The current election database is defined by whatever database is specified as "current" in this configuration file. BallotStation assumes no line in the file is longer than a short, fixed number of characters and only allocates a buffer of that length on the stack. If a line exceeds this length, excess characters will be written past the end of the buffer. This would allow an attacker to crash the machine and (most likely) execute arbitrary code soon after boot.

This attack is especially serious because `assure.ini` is neither encrypted nor authenticated. An attacker could modify this file without knowledge of any secret keys. An attacker only needs to be able to insert a malicious memory card into a machine, or modify a legitimate memory card that will later be inserted into a machine, to successfully compromise an AV-TSX machine.

Bugs in String-, Script-, and Bitmap-Resource Processing The AV-TSX displays text from the election resource file on the memory card at various stages of the election. This text is stored in language-specific RTF strings. One such string contains the text used to show the current page number at the bottom of every ballot page (excluding the "cast ballot" screen). This text includes two `%d` characters which are replaced with the current page number and

the total number of pages in the current ballot using `sprintf`. For example, the text "Page %d of %d" would be replaced with "Page 5 of 10" on page five of a ten-page ballot. The adversary could instead provide a string like "Page %d of %d%s%s%s%s%s%s%s%s%s%s", which would very likely crash the machine, or other strings to potentially run malicious code during the vote casting process.

Diebold uses a scripting language called AccuBasic to customize the format of reports printed on the AV-TSX, such as the Election Zero report (printed before the election) or the Election Results report (printed after the election). An AccuBasic script is stored on the memory card in the ballot definition file, and the AV-TSX software interprets the script from the card. Wagner, et al. reported buffer overflows and other flaws in the AccuBasic interpreter that allow malicious scripts to execute arbitrary low-level code [174].

The AV-TSX displays bitmap images stored in the ballot definition file at various times during the voting process. A bitmap file begins with header structures describing its data. These headers contain a variety of information, including the size of the bitmap file and the size of the bitmap data. The AV-TSX, assuming these values will be equivalent, creates a buffer large enough to hold the bitmap data and then reads the entire bitmap *file* into that buffer. A malicious bitmap file could be constructed that would lie about its size, claiming to be smaller than it is. The AV-TSX's buffer would be too small to hold the data in the bitmap file, and a buffer overflow would occur during the voting process.

To exploit these vulnerabilities, an attacker would need either control of GEMS; control over the relevant RTF files, AccuBasic scripts, or bitmap files stored on GEMS; or access to the memory card. In some cases, the attacker would need to know a secret key called the Data Key. We discuss how he might learn it later in this section.

Bugs in Election Uploading When an AV-TSX uploads election information to GEMS, BallotStation calls a function that prints out an "election ticket" containing information

drawn from the election database. There are three separate vulnerabilities in this ticket-printing function that allow an attacker who is able to modify the election database to crash the machine and possibly to execute malicious code.

1. The variables containing election attributes are combined using `sprintf` into a buffer `buf` that is 512 bytes long. An attacker able to modify either of these variables could overrun `buf`.
2. The very next statement uses `buf` as an argument to an `sprintf`-style formatting function belonging to BallotStation's printer class. This formatting function contains a format string vulnerability that can be exploited by an attacker who is able to modify either election attribute.
3. Later in the ticket-printing function, a similar formatting function is called every time a file is uploaded. This formatting function contains a format string vulnerability that can be exploited by an attacker who is able to modify a certain other election attribute.

An attacker able to compromise an AV-TSX machine could use this vulnerability to execute malicious code on the central office AV-TSX when election results are uploaded to GEMS.

3.2.4 Insecure Storage of Cryptographic Keys

The AV-TS and AV-TSX use secret keys for various security purposes, including authenticating election definition files, encrypting and authenticating ballot results files, and generating ballot serial numbers. Older Diebold software, like that used in the AV-TS we studied, used hardcoded keys set when the software was compiled. The version of the software in the AV-TSX that we studied retains those hardcoded keys but also allows county election officials to change the 128-bit *Data Key* that the machine uses.

The machine stores the Data Key in a file in its internal flash memory. This file, `bs-security.cf`, resides in the same directory as `BallotStation.exe`. BallotStation encrypts the contents of the file with a third key called the *System Key*. However, the value of the System Key is not a secret—rather, it is the MD5 hash of the machine's serial number. The serial number is stored in the machine's registry (`HKLM\Software\Global Election Systems\AccuVote-TS4\MachineSN`), displayed in the user interface (the parameter "SN" at the bottom of every screen), and printed on the Results Report and other printouts.

As a result, any party with the ability to read data from the machine's internal flash memory can learn the values of Smart Card Key and Data Key. For example, an attacker with temporary physical access to the inside of the machine's case could exploit the hardware debugging features (Section 3.2.2) or insecure update mechanisms (Section 3.2.1) to read the contents of the `bs-security.cf` file and the registry key containing the machine serial number. The attacker could then compute the System Key from the serial number and use it to decrypt the other keys stored in the file.

This attack may be particularly damaging because the design of the Diebold system makes it difficult to use different keys on different machines [28]. Consequentially, all machines within a particular county most likely share the same Smart Card Key and Data Key. An attacker who can extract the keys from a single machine can therefore use them to attack all of the machines and memory cards in the county.

3.2.5 Poor Protection of Critical Election Data

Four types of data files are used in elections: election databases, election resource files, ballot results files, and audit logs. These files are stored on the removable memory cards, with backups stored in the machine's internal flash memory. All files are contained in

standard file systems, with no access control or integrity protection mechanisms. Any process running on the machine can read or write the files arbitrarily.

Election database files (`.edb` files) contain information about races and candidates as well as other ballot text. They are not encrypted, but they are authenticated using a kind of message authentication code (MAC): The MD5 hash of the body of the file is encrypted with the Data Key, and the machine requires this encrypted hash to match a value in the header of the ballot definition.

Election resource files (`.xtr` files) contain RTF strings, AccuBasic scripts, audio, images, and certain other data used during the election. The resources in these files are not encrypted, but each resource is individually authenticated using a MAC like the one used for the election database.

Ballot results files (`.brs` files) contain a record of the votes for each ballot cast in the election. The record of each vote is individually authenticated as in the election resource files. Each vote record is also encrypted using the same Data Key.

Audit logs (`.adt` files) store a partial record of BallotStation's operation. They are authenticated and encrypted using a similar format as ballot results files, except that they use the System Key in place of the Data Key.

An adversary who obtained access to the Data Key as described above could carry out various attacks on these files:

- If the attacker had access to the memory card prior to the election, he could tamper with election files and election resource files in order to exploit other security vulnerabilities in BallotStation, including the vulnerabilities described above that allow the attacker to execute arbitrary code on the voting machine during the election. The attacker could also attempt to alter the ballots in subtle ways that would confuse voters or otherwise disrupt the election.

- If the attacker had access to the memory card *after* the election, he could carry out other attacks by defeating the encryption and authentication of the ballot results files. If the attacker accesses the memory card before the votes on it are loaded into GEMS, then he can tamper with the ballot results file in order to exploit security vulnerabilities in the central office AV-TSX connected to the GEMS server. Using this technique, he might be able to infect a large number of voting machines with a voting machine virus.
- If, during or after the election, the attacker had access to the memory card or the files on the voting machine, he could decrypt or tamper with audit logs. These are secured with the System Key, which is not a closely guarded secret. By tampering with the logs, the attacker could remove evidence of his activities. The audit logs also contain sensitive debugging information, such as stack traces recorded during errors in the software, that could help a malicious party develop further attacks.

Attacks like the ones described above could be carried out automatically by malicious code installed on the voting machine. The code could automatically derive the Data Key, as described above. In that case, the attacker would not need to have physical access to each machine or memory card being attacked.

3.3 Attack Scenarios

Elections that rely on the Diebold DREs are vulnerable to several serious attacks. Many of these vulnerabilities arise because the machine does not even attempt to verify the authenticity of the code it executes. We begin by outlining several methods by which malicious code can be injected. Then we describe two classes of attacks—vote stealing and denial-of-service—that could be carried out by such code.

3.3.1 Direct Attack Installation

An attacker with physical access to a machine would have at least three methods of installing malicious software [81].

The first is to exploit the unprotected hardware debugging features described in Section 3.2.2. On the AV-TSX, the attacker could write his malicious code into the machine's flash memory using a serial connection or the JTAG interface. On the AV-TS, he could create an EPROM chip containing a program that will install the attack code and then open the machine and install the chip.

The second method is to exploit the unauthenticated software update mechanisms described in Section 3.2.1. The attacker could create a modified version of the machine's bootloader, OS kernel, or application software. He could then package it as a software update, store it on a memory card, and boot the machine with the card inserted to install the update. Alternatively, the attacker could exploit the service back door mechanism to boot the machine into Windows Explorer, then install the malicious software by hand.

The third method is to exploit code execution vulnerabilities caused by errors in Diebold's software, such as those described in Section 3.2.3. Doing this would require modifying the data files that are read by the software. Since some of these files are not authenticated, an attacker could tamper with them to launch an attack without knowing any secret keys.

Carrying out the first method requires access to the inside of the machine. The attacker would need to remove several screws and lift off the top of the machine to gain access to the motherboard's EPROM socket, debug jumper, or JTAG port. This can be accomplished with a few minutes of physical access.

The other methods require access to the machine's memory card. If memory cards ship separately from machines, the attacker could intercept a card en route and copy his malicious code onto the card. If, instead, machines ship with memory cards sealed in place, the

attacker would need to gain physical access to the machine. Poll workers, election officials, and technicians often do have such access. For instance, in a widespread practice called "sleepovers," machines are sent home with poll workers the night before the election [162]. In other instances, as Ed Felten has documented [62], voting machines are left unattended in polling places the night before the election.

After gaining access to the machine, the attacker would break the seal and unlock the lock, replace the memory card with one containing his malicious code, reboot the machine to install the code, reinsert the original memory card, and relock the enclosure. On both the AV-TS and AV-TSX, the lock protecting the memory card can be picked in a matter of seconds by a person with moderate practice [2, 61]. In all, this process would require less than one minute of physical access [2] in a manner that would likely raise minimal suspicion from poll workers. The only physical evidence, if any, would be a broken seal.

3.3.2 Voting Machine Viruses

Like desktop PCs, computer voting machines are vulnerable to viruses. Viruses pose a particularly severe threat to voting security because they can spread invisibly in the background, even when procedural safeguards that limit physical access to the machines are followed. Once installed on a single "seed" machine, the virus would spread to other machines by methods described below, allowing an attacker with physical access to a single machine (or card) to infect a potentially large population of machines. The virus could be programmed to install malicious software, such as a vote-stealing program or denial-of-service attack, on every machine it infected.

To prove that this is possible, we constructed a demonstration virus for the AV-TS that spreads automatically, installing our demonstration vote-stealing software on each infected system. Our demonstration virus infects machines by exploiting the unauthenticated low-

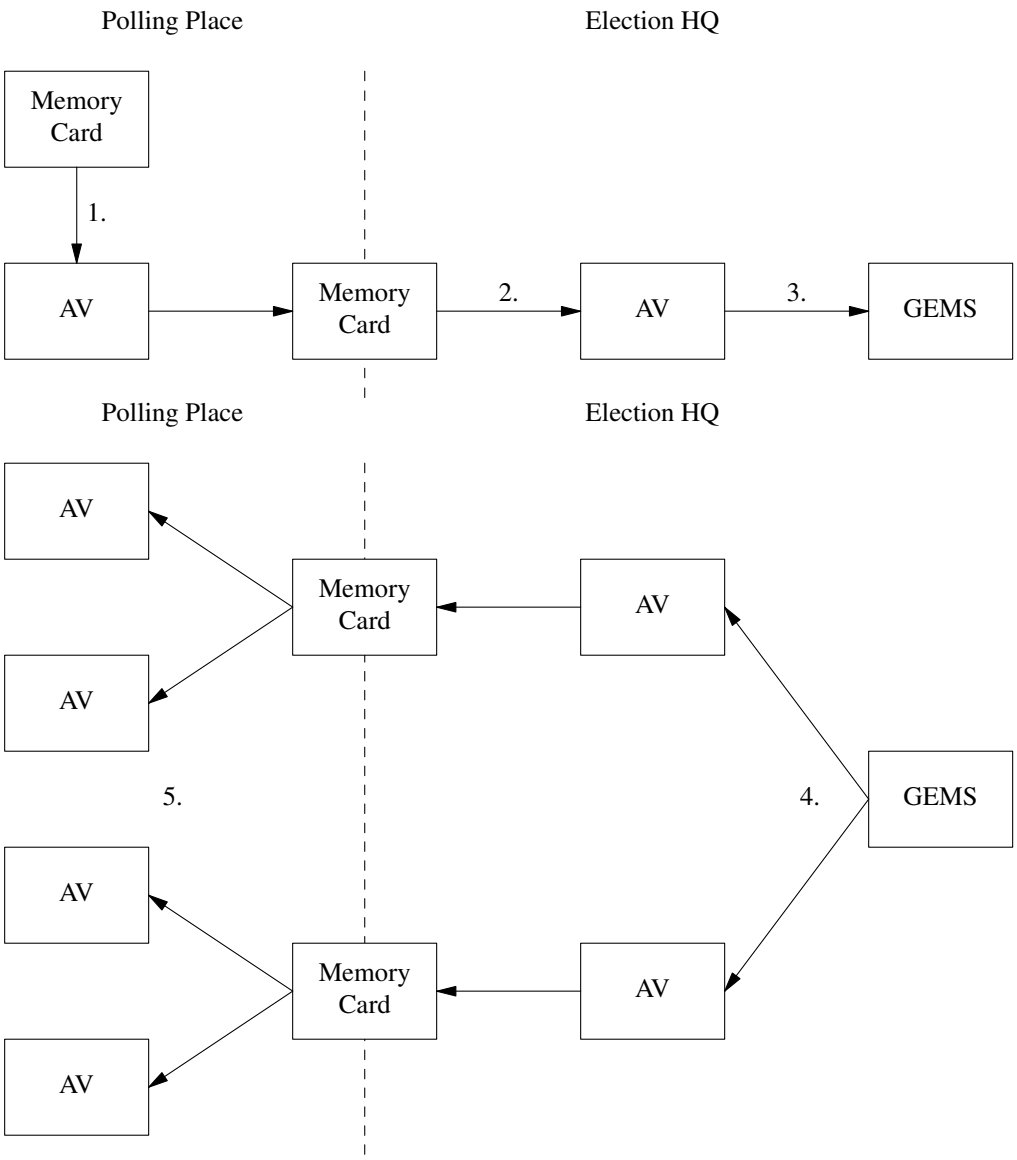


Figure 3.5—Propagation of a Virus over the Course of Two Election Cycles

During the first election: *(1)* An attacker temporarily inserts a memory card containing a voting machine virus into an AccuVote machine, infecting the machine. *(2)* After the election, poll workers remove the memory card containing ballot results from the infected machine and send it to election headquarters for tabulation; the virus has corrupted the files on the card, so inserting it into a central-office AccuVote infects that machine. *(3)* The infected central-office AccuVote attacks the GEMS PC over the Ethernet network by using known vulnerabilities in Windows; when the attack succeeds, the virus infects the GEMS server.

During the next election cycle: *(4)* The virus running on the GEMS server infects memory cards when officials download the new ballots; these cards are placed in voting machines throughout the county. *(5)* On election day, the virus executes its payload, which may involve altering votes or otherwise disrupting the election.

level software update mechanism described in Section 3.2.1. An infected machine will infect any memory card that is inserted into it by placing a firmware update file on the card. An infected memory card will infect any machine that is powered up or rebooted with the memory card inserted, since the machine will automatically install the firmware update. Because cards are transferred between machines during vote counting and administrative activities, such as software updates, the infected population will grow over time. If the virus infects the central office AccuVote machines used to upload and download data from GEMS, it could quickly spread to a large number of polling place voting machines.

Other vulnerabilities that we identified in this chapter create further avenues for spreading viruses that are potentially more dangerous than mechanisms we exploited in our demonstration virus. We now describe one such scenario where a voting machine virus could spread throughout a county's election system (see Figure 3.5). Many variations on this scenario are possible.

The life cycle of such a virus would be a multistep process:

1. **Initial infection of an AccuVote machine**

   The attacker, after developing the virus in advance of the election, would needs only momentary physical access to a voting machine or memory card in order to initiate the infection. He could use any of the direct injection methods described in Section 3.3.1.

2. **Spread to the central-office AccuVote machine**

   On the initially infected voting machine, the virus can manipulate the election database file stored on the memory card by exploiting the issues we describe in Section 3.2.3. The attacker could design the virus to corrupt the file so that it will cause the execution of arbitrary code during the result upload stage.

Later, officials take the memory card with the manipulated election description and place it into a central-office AccuVote machine. When officials initiate the upload function, the attacker's code executes and infects the central office voting machine with the virus.

3. **Attacking the GEMS machine**

   As soon as the virus infects the central-office voting machine, it can begin attacking the PC running the GEMS software. In a typical deployment, as described by Diebold, the GEMS machine and the central-office AccuVote machines attach to a single Ethernet switch and communicate using TCP/IP. This means that the GEMS PC exposes a large attack surface to the machine. Vulnerabilities in the PC's operating system (Windows), network drivers, and network services could all be attacked. The hacker community is already aware of exploitable flaws in some of these components. Even if automatic patches exist for these commodity components, the PC's software may not be up-to-date.

   In the California Top-to-Bottom review, one team was able to use widely available exploit tools to exploit holes in Windows and take control of the GEMS PC from another PC on the same subnet [2]. A virus running on the central AV-TSX could be programmed to perform a similar attack. After gaining control of the GEMS PC, the virus would install itself and proceed to the next phase of its lifecycle. It could hide itself from system administrators and from common security tools using rootkit techniques [79].

4. **Spreading back to the field**

   At the beginning of the next election cycle, the infected GEMS system can spread the virus to the voting machines used in the field. It might spread to AccuVote systems by tampering with the election data files as they are downloaded to memory cards that will be distributed to polling places. By introducing deliberate errors into these files, the virus could exploit vulnerabilities that will allow virus code to execute on the systems during voting. Since typical procedures call for every memory card used in the county to

be created using the GEMS server, this step would allow the virus to infect every machine used by voters.

What harm can a voting machine virus or other malicious code do? Among the most dangerous payloads would be an attempt to shift a race by subtly stealing votes and an attempt to disrupt an election by launching a large scale denial-of-service attack. We discuss these attacks in the next two sections.

3.3.3 Vote-Stealing Attacks

An attacker could use a voting machine virus to reprogram a large number of voting machines to steal votes. When programming the attack, the attacker could decide which votes to steal (e.g., from particular candidates, races, or parties), how to steal them (e.g., by adding, deleting, or switching votes from one candidate to another), and when to execute the attack (e.g., only in closely contested races, or only in precincts with certain voting patterns).

Some AV-TSX machines are capable of producing a voter-verifiable paper audit trail (VVPAT). Though this provides a valuable defense against electronic vote stealing, it will not necessarily be able to detect and correct every kind of attack—particularly in races with a narrow margin of victory.

In a close election, one particularly dangerous scenario would be a widely-spread virus that subtly shifts votes between candidates on both the paper and electronic records. Suppose the candidates are named Alice and Bob. A Bob supporter could reprogram the machines to look for voters who select Alice. One percent of the time, after the voter has selected Alice, they machines could behave as if the voter had picked Bob, displaying a vote for Bob on the confirmation screen and on the printed paper record. A cleverly designed

virus would not interfere with an attempt to correct the problem, so voters who notice the error could cancel the printed VVPAT record and change the selection back to Alice.

An attack like this might shift enough votes to cause the wrong result in a close election, but could it really be done without being detected? Several factors favor the attacker. First, assuming that only a small fraction of voters would carefully review the paper VVPAT record, many voters would overlook the problem and allow incorrect votes to be recorded in both the electronic and paper records. Second, while a few voters might report the problem to poll workers, election officials would have difficulty determining whether the cause was voter error or a problem with the machines. This is similar to problems that Sarasota County, Florida experienced with its DRE voting machines in the November 2006 election [43]. In one race during that election, hundreds of voters reported that the machines displayed the wrong selection on the summary screen, or that they failed to show the race on the summary screen at all. Some observers eventually concluded that the cause was voter error due to a poorly designed ballot layout.

Even if officials suspect an electronic attack, the virus author could take countermeasures to thwart later investigation. The attacker could tamper with the system logs to remove traces of the virus's activity, and remove the virus after the election when the machine powers on again. By the time an investigation is commenced, most of the evidence of the problem could be destroyed.

Finally, even in the best case when officials do detect the virus, they might have difficulty undoing its effects without holding a new election—thus, the vote stealing attack becomes, at best, a massive denial of service attack. It would probably be impossible to tell how many votes had been shifted as a result of the attack, since the electronic and paper records would both reflect the fraudulent result.

Reprogramming the machine to steal votes is relatively straightforward. The machine we studied maintains two records of each vote—one in its internal flash memory and one on a removable memory card. These records are encrypted, but the encryption is not an effective barrier to a vote-stealing attack because the encryption key is stored in the voting machine's memory where malicious software can easily access it (see Section 3.2.4). Malicious software running on the machine would modify both redundant copies of the record for each vote it altered. Although the voting machine also keeps various logs and counters that record a history of the machine's use, a successful vote-stealing attack would modify these records so they were consistent with the fraudulent history that the attacker was constructing.

Such malicious software can be grafted into the BallotStation election software (by modifying and recompiling BallotStation if the attacker has the BallotStation source code, or by modifying the BallotStation binary), it can be delivered as a separate program that runs at the same time as BallotStation, it can be grafted into the operating system or bootloader, or it can occupy a virtualized layer below the bootloader and operating system [92]. The machine contains no security mechanisms that would detect a well designed attack using any of these methods. However it is packaged, the attack software can modify each vote as it is cast, or it can wait and rewrite the machine's records later, as long as the modifications are made before the election is completed.

By these methods, malicious code installed by an adversary could steal votes without being detected by election officials.[2] Vote counts would add up correctly, the total number of votes recorded on the machine would be correct, and the machine's logs and counters would be consistent with the results reported—but the results would be fraudulent.

2 Officials might try to detect such an attack by parallel testing. As we describe in Section 3.7, an attacker has various countermeasures to limit the effectiveness of such testing.

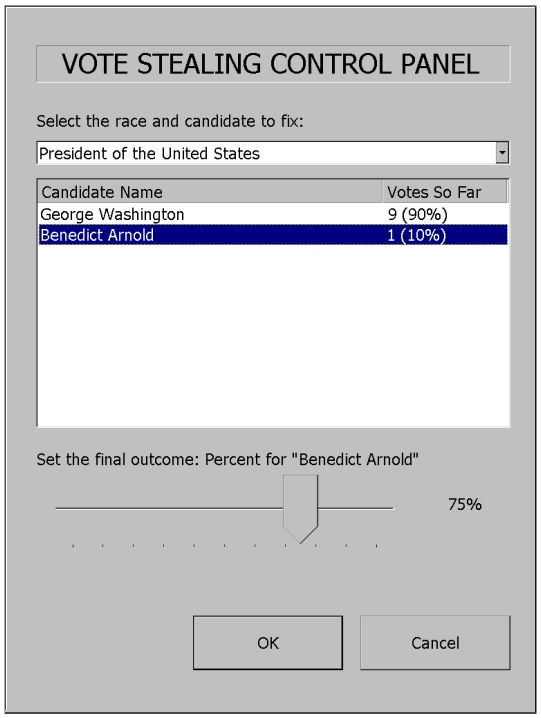


Figure 3.6—Demonstration Attack
Our proof-of-concept vote-stealing software.

We have implemented a demonstration vote stealing attack on the AV-TS to prove that this is possible. Our attack can be launched by exploiting the unprotected service back door described in Section 3.2.1 and running a program from a removable memory card. It displays a user interface, shown in Figure 3.6, that allows the attacker to select the winning candidate and the percentage of votes that candidate will receive. In practice, a real attacker would more likely design a vote-stealing program that functioned invisibly, without a user interface.

3.3.4 Denial-of-Service Attacks

Rather than stealing votes directly, attackers might choose a more passive strategy and attempt to disrupt the election process itself by disabling machines, destroying vote records, or slowing down voting. These attacks could be targeted at precincts that are likely to support an opposing candidate, or even triggered only after the virus detects that the opposing

candidate has won a certain portion of the votes on a machine. Alternatively, the attack could be carried out indiscriminately in hopes of causing such widespread disruption that the election would be postponed.

An attacker would have many choices about when and where to trigger an attack and what kind of damage to do. Some attacks might be very difficult to distinguish from non-malicious hardware and software malfunctions.

One style of such a denial-of-service attack would make voting machines unavailable on election day. For example, malicious code could be programmed to make the machine crash or malfunction at a pre-programmed time, perhaps only in certain polling places.

In an extreme example, an attack could strike on election day, perhaps late in the day, and completely wipe out the state of the machine by erasing its flash memory. This would destroy all records of the election in progress, as well as the bootloader, operating system, and election software. The machine would refuse to boot or otherwise function. We have created a demonstration denial-of-service attack for the AV-TS that operates in this way [61].

If carried out on a wide scale, such an attack could cause massive disruption. Restoring the machine to a working state would require a service technician to open the machine and restore the software using the EPROM slot or JTAG port. The result would be a fresh install of the machine's software, with all records of past and current elections still lost.

A similar style of attack would try to spoil an election by modifying the machine's vote counts or logs in a manner that would be easy to detect but impossible to correct, such as by injecting malicious code that adds or removes so many votes that the results at the end of the day are obviously wrong. A widespread attack of either style could require the election to be redone.

3.4 Systemic Problems

In our analysis of the Diebold system, we found significant systemic weaknesses in its design and implementation as well as in the engineering practices used to develop it.

3.4.1 Systemic Design Weakness

Attack surface Experienced security practitioners often recommend analysis of the "attack surface" of a software system. The attack surface is the interface that is exposed to the attacker. Systems with a large attack surface tend to be more prone to security vulnerabilities.

The Diebold voting system has a large attack surface. Exposed interfaces include: (1) the user interface on the voting machine; (2) the protocol spoken between the machine and the smart card; (3) the content of election database and other files on the voting machine; (4) the memory card, as read by AccuVote units in the field; (5) the content of the ballot results files in the voting machine's memory card, as read by other AccuVote units; and (6) the data transmitted between GEMS and a central-office AccuVote machine, when the two are connected by Ethernet. Some of these interfaces are complex and present many opportunities for attack. All of them could potentially be manipulated by an attacker. Given this, one would expect that the risk of exploitable vulnerabilities is high. That expectation was borne out during our examination of the source code.

Complexity The Diebold system is a complex computing system. Complexity is the enemy of security. All code has bugs; the only way to be sure that software will be secure is to arrange for its design and implementation to be so simple and so small that one can inspect all of it and be confident that all of the bugs and defects in the code are found.

By that criterion, the Diebold software is too complex to be secure. One crude measure of software complexity involves counting lines of source code. The AV-TSX contains 136K

source lines of code, excluding the Windows CE OS. GEMS contains another 116K source lines of code. If the Diebold system were secure, it would be the first computing system of this complexity that is fully secure.

One principle of secure design is to architect the software so that it has a small Trusted Computing Base (TCB), that portion of the software whose correctness suffices to ensure that the system security requirements will be met. The TCB must be protected from attack and must be written to ensure that the rest of the system cannot violate the security policy even if the rest of the system is compromised or malicious.

The Diebold software does not appear to have any clearly defined TCB. It is a monolithic system, with no clear trust boundaries. Due to this architecture, a breach of any part of the software may lead to security violations and breaches of the rest of the software. Because code of any significant complexity or scale inevitably has bugs, defects, and flaws, this architecture makes it all but inevitable that the Diebold voting software will have exploitable security vulnerabilities.

Misplaced trust In our judgment, the Diebold software places too much trust in people and other components of the system. For instance, the software trusts—relies upon—the memory card to contain files from a legitimate, authorized source. In other words, the software is written with the expectation that the contents of the memory card come from a benign source, and the software does not effectively defend itself against malicious files on the memory card. That trust seems misplaced: it is too easy for an attacker to tamper with the contents of a memory card. When that expectation is violated, the integrity of the software can be breached.

This theme appears throughout the voting system. In many places where two components communicate, both components rely on each other to be benign, which renders them vulnerable to attack if the security of the component happens to be breached. For instance,

the GEMS server trusts the central-office AccuVote units and everything else that is connected to its own Ethernet network. This trust is dangerous. While those devices might be protected against physical tampering, they must handle data that comes from the field and thus might be malicious. Those devices are at heightened risk of subversion, and it would be safer if GEMS and other system components were written to defend against subversion by malicious devices on the same network.

This type of pervasive trust makes the Diebold system brittle: a small security breach can have large consequences out of proportion to the initial breach, or a breach in one part of the system can put other parts at risk. That, in turn, places an unnecessary burden on procedural protections, because even a brief violation of procedure or a small, seemingly negligible breach of the chain of custody can have disproportionately harmful effects.

Bidirectional information flow The Diebold voting system includes a bidirectional flow of data. Information flows from GEMS to every unit in the field (via memory cards), and from all units in the field back to GEMS (again, via memory cards). This creates a significant risk of viral infection. In particular, if (1) a memory card or unit in the field can be corrupted and (2) there are any exploitable flaws in the handling of data on the memory card, then a virus may be able to spread from one unit in the field to GEMS and then back to every unit in the field. In practice, we found that both prerequisites are met. Due to the complexity of the data on the memory card, any system of this architecture seems to be at high risk of viral spread.

This is not a necessary property of a voting system. For instance, it would be possible to have one central-office application for programming memory cards for distribution to the field and a second application for reading memory cards from the field and tabulating results, with firewalls to ensure that any penetration of the second application cannot affect the first application. However, the Diebold voting system was not designed with those

kinds of firewalls in place, and it was not constructed in a way that would provide inherent resistance against the spread of virally propagating malicious code.

No way to verify code integrity The Diebold system does not provide any secure way for an election official to verify whether the software resident on the voting machines has been modified. For instance, a cautious election official might wish to occasionally spot-check a random sample of machines to confirm that they have the correct software installed. Unfortunately, GEMS and the AccuVote machines provide no way to do that securely.

Voting machines can be built that make violating code integrity more difficult. For instance, the voting machine could be designed so that its bootloader is stored on write-once storage (PROM or EPROM). The bootloader could make a record of the cryptographic hash of the software that it loads and output that record on the machine's printer. Or, the bootloader could contain a public key and could check that the software is properly signed before loading it. Or, the device could use a Trusted Platform Module or other secure hardware technology, such as that standardized by the Trusted Computing Group [166]. Such a design would allow the operator to verify that the machine's software has not been modified or altered. Neither GEMS nor the AccuVote machines have this capability.

3.4.2 Systemic Implementational Errors

Input validation Input validation is one of the most important practices that developers of security-critical software must follow. Some experts estimate that approximately half of all software vulnerabilities can be attributed to failure to properly validate inputs from untrusted sources. The best practice is to establish a discipline to ensure that all inputs are validated, for instance, by checking all inputs against a template or white list as soon as they are read from any untrusted source and before they are used for any purpose.

We did not find a consistent pattern or discipline of input validation in their source code. Untrusted inputs are occasionally compared against a white list or template describing expected values but are more frequently not checked at all. Integers read from untrusted sources are sometimes bounds-checked immediately after being read but sometimes not. Strings are not usually checked for null-termination and are rarely matched against a white list or regular expression.

Defensive programming Defensive programming is another recommended practice. It involves checking all data provided by other software components just before using the data. Even if one expects that the source of the data has already verified the correctness of the data, each recipient also redundantly checks the data. The philosophy is that the program should be constructed to be robust against unexpected inputs and should fail gracefully even if other components contain unexpected bugs.

The use of defensive programming in the Diebold source code was variable. In a few places, the source code was written defensively, carefully checked all inputs, and appeared to be reasonably robust. In other places, the code made unchecked assumptions about the data it used, was not written defensively, and did not appear to be as robust as it could have been. We noticed that the latter appeared more frequently in places where the programmer might not have been expecting malicious or erroneous inputs (e.g., some of the code that handles data read from the election database or other files on the memory card) and in non-core code (e.g., debugging or logging code, code that is used only to print reports, or code for system administration tasks).

In many places, the failure to program defensively appeared to be of no particular import. However, in some cases, the failure to program defensively led to serious, exploitable security vulnerabilities. The reason that security engineers often recommend applying defensive programming to all code, not just code that is known to be exposed to an attacker,

is that programmers often make unjustified assumptions and fail to anticipate the ways that attackers might be able to provide unexpected inputs. The failure to consistently apply defensive programming techniques probably contributed to the number of exploitable implementation-level vulnerabilities that we found.

Programming languages The choice of programming language can have an influence on the frequency of implementation-level vulnerabilities. The Diebold system uses assembly languages, C, and C++. These programming languages are known to be prone to several common types of security vulnerabilities, including buffer overflows, format string vulnerabilities, and integer overflows. We found instances of all these vulnerabilities in the source code we analyzed.

Many security engineers recommend use of memory-safe, type-safe programming languages, because those languages have inherent resistance to several of the most common types of security vulnerabilities. For instance, until recently, buffer overflows were consistently the number one publicly reported vulnerability [35]. Memory-safe languages, like Java or C#, effectively eliminate buffer overflow vulnerabilities, while programs written in older languages like assembly, C and C++ are known to be at risk for these vulnerabilities.

3.4.3 Deficient Engineering Practices

Our analysis is further informed by an interview that we conducted with Talbot Iredale, Software Development Manager for Diebold Election Systems [86]. We wish to thank Mr. Iredale for his time and his useful insights. Based on what Mr. Iredale told us, Diebold's engineering practices seem to be similar to those of most small- to medium-sized software development firms. These practices may be sufficient for ordinary commercial software, but they are inadequate for meeting the rigorous security requirements of voting software.

No formal threat model or security plan In our interview, Mr. Iredale stated that Diebold has neither a formal written threat model nor a formal security plan for its voting systems. Indeed, we found no evidence in the source code that systematic analysis of threats had been performed. Instead, the security measures that are in place appeared to be *ad hoc*.

No formal security training Diebold has about 25 developers that work on electronic voting systems, including those who focus on documentation, testing, and hardware. When new developers arrive at the company, they do not receive any kind of formal security training. Mr. Iredale states that some developers have security backgrounds but no one is dedicated to handling security issues. They have two small groups of quality assurance testers of approximately four people each, but none of them are dedicated specifically to security or red-team testing.

Weak source code review process Diebold uses standard versioning software (CVS) to manage the development of their source code. Any developer can check code into CVS and the code is not reviewed by other developers before it is committed into the repository. Mr. Iredale states that every CVS check-in causes an e-mail to be sent to developers who are responsible for reviewing the code. Initially, they do "random checks" on most of the code and do a "closer review" of the more critical portions. Although Mr. Iredale claims that 100% of the code is reviewed by another Diebold employee within a few weeks, there seems to be no formal procedure for assigning code to other employees for review. It seems possible that, without formal procedures, some source code could remain unreviewed before release.

No unit testing or red team testing There is no formal requirement to develop a set of unit tests that correspond to each piece of code checked into CVS—the option of doing this is strictly up to individual developers. The testing group will later check for correctness

based on standard test plans. Diebold also lacks any formal procedures for "red team" testing, where the testers play the role of attackers and attempt to break into the system. This type of testing can detect different types of bugs that "white box" unit and system tests might not catch, such as illegal input handling and failure recovery.

3.5 Results from Other Studies

In this section we review past investigations of DRE security failures and place our work in context. Systemic problems like the ones we found are not confined to the Diebold system; numerous studies have reported similar problems in other electronic voting systems.

The first major study of DRE security was conducted in 2002 by Kohno et al. [94], who examined a leaked version of the source code for Diebold's BallotStation application. They found numerous security flaws and concluded that the software's design did not show evidence of any sophisticated security thinking.

Public concern in light of Kohno's study led the state of Maryland to authorize two security studies of the Diebold system. The first study, by SAIC, reported in 2003 that the system was "at high risk of compromise" [150]. RABA, a security consulting firm, was hired to do another independent study of the Diebold machines. RABA had access to a number of machines and some technical documentation. Their study [140] was generally consistent with Kohno's findings, and found some new vulnerabilities.

A further security assessment of was commissioned by the Ohio Secretary of State and carried out by the Compuware Corporation [38], also in 2003. This study examined several DRE systems, including the AV-TS, and concluded that several high risk security problems were present.

In 2006, in response to reports that Harri Hursti had found flaws in Diebold's AccuBasic subsystem, the state of California asked Wagner, Jefferson, and Bishop to perform a study

of AccuBasic security issues. Their report [174] found several vulnerabilities. Later in 2006, Hursti released a report describing several security weaknesses in Diebold's software relating to the automatic installation of unauthenticated software [81].

In 2006, we obtained and examined an AV-TS. In addition to confirming some of the security flaws found in the previous works, we constructed demonstration vote stealing software and a voting machine virus that spreads via the memory cards used to load the ballot definition files and collect election results [61].

In 2007, California Secretary of State Debra Bowen launched a "top-to-bottom" review of the electronic voting systems then in use in California. Voting machine vendors were required to make available to independent reviewers documentation, source code, and several voting machines. We participated and evaluated the Diebold system [28], while other teams studied voting systems produced by Sequoia [21] and Hart InterCivic [83]. In all cases, pervasive problems were reported with the procedures, code, and hardware reviewed.

Also in 2007, Ohio Secretary of State Jennifer Brunner ordered project EVEREST—Evaluation and Validation of Election Related Equipment, Standards and Testing—as a comprehensive review of Ohio's electronic voting machines. As in California's top-to-bottom review, the reviewers had access to voting machines and source code. They examined systems manufactured by Diebold, ES&S, and Hart InterCivic, and, once more, critical security flaws were discovered in all the machines [26].

In 2008, Appel et al. [10] evaluated the security of the Sequoia AVC Advantage machine, which is widely used in New Jersey. Like the previous studies, Professor Appel's found significant problems, including systemic weaknesses in design and implementation and vulnerabilities that could be exploited to spread a vote-stealing virus.

As a result of these studies and others, attacks that can steal votes, disrupt elections, compromise ballot secrecy, and spread malicious code virally have been documented in systems made by virtually every major vendor, including Diebold [61, 28], ES&S [26], Hart InterCivic [83], and Sequoia [10]. We now know that systemic security problems are not confined to the products of any particular manufacturer, but are a pervasive problem in currently deployed DRE voting systems.

3.6 High-Level Causes

In our analysis of the Diebold system, we found significant systemic weaknesses in its design and implementation as well as in the engineering practices used to develop it. We can trace these problems to high-level causes, such as market forces that create incentives to skimp on security, customer demands that add complexity to the voting system, and deficiencies in the testing and verification process. These high-level causes are likely to impact all DRE systems, and they may explain the pervasiveness of the problems noted in the previous section.

3.6.1 Time-to-Market Pressure

We saw in Chapter 2 how a vendor's incentives can lead to underinvestment in security. This effect appears to be at work in the electronic voting context. Diebold's engineering practices, and the systemic implementational errors they produced, may have been driven by intense pressure to bring products to market quickly that gave the company incentives to deprioritize security when developing its products.

Analysis

One source of such pressure was the Help America Vote Act (HAVA) [169], which was enacted by Congress in 2002 following the Florida recount debacle in the 2000 election. Among other reforms, HAVA provided billions of dollars to the states to upgrade their election equipment, but much of the money would only be available until November 2004. This created a window of less than three years during which many states would purchase new equipment. After that, it could be expected, demand would be correspondingly suppressed for a long period of time. The result was a race to ready new products while demand was high. Diebold introduced the AV-TS in 2001 and the AV-TSX only two years later; since then the company has not produced any new voting machine models. Both products reflect this intense time pressure in several ways.

First, the AccuVote machines and GEMS make extensive use of commercial off-the-shelf (COTS) software, including Microsoft Windows. The use of COTS software significantly shortened the product development time for Diebold, but it exposed the system to the ever-growing number of attacks that target Windows.

Second, we found evidence of pervasive errors in the source code, such as the problems we discussed in Section 3.2.3. These indicate that the software was developed rapidly and tested incompletely. Similarly, other aspects of the system's design and implementation, such as the general lack of input validation and failure to apply defensive programming practices, are consistent with the hypothesis that it was constructed with little emphasis on security.

Third, our investigation into the company's engineering practices revealed that Diebold did not take the time to develop a formal threat model or security plan prior to implementing the system, and that they spent only limited time on testing. These practices probably cut the development time, but, in light of them, the security failures that appeared in the final

products are unsurprising. Diebold's principal goal seems to have been to ship the product under deadline, rather than to meet a particular quality standard.

> **Cause 3.1: Vendors had Artificially Heightened Incentives to Ship Products Quickly**
> The Help America Vote Act created tremendous incentives to bring voting machines to market in a short period of time. As a result, vendors such as Diebold implemented their products quickly, targeting tight deadlines rather than benchmark levels of security.

Implications

- This incentive may have been reinforced by the invisible nature of security. Since the vendors generally did not allow their customers to inspect the machines' source code, security problems could only be discovered after the devices were sold. Absent studies like this one, they would only be discovered if the machines were actually attacked (and the attacks were discovered). The vendors may have counted on their ability to improve security at a later date, after the machines were sold but before they were attacked.
- Using COTS may be more dangerous in voting machines than in many other contexts. To keep a Windows system secure, administrators must apply patches in a timely manner, but the special circumstances associated with voting systems make it difficult, or even dangerous, to do so. Software changes could introduce attacks against the voting system, so patches need to be certified before they are applied [28]. Rather than relying on COTS, voting machines should use software that is engineered to a stricter security standard and does not depend on freqent patching to stay secure.

3.6.2 Features and Complexity

Despite the intense time pressure that Diebold and other vendors faced, their products are remarkably complex, containing hundreds of thousands of lines of code. This apparent contradiction is explained by another aspect of the vendors' incentives: purchases of voting machines were driven primarily by features, not by security.

Analysis

Security was surely among the features that mattered to voting officials, but all the vendors claimed that their products were secure, and, as we discuss below, their customers had limited ability to evaluate these claims. The features on which the products primarily competed were those that were most visible, such as attractive user interfaces and sophisticated election management systems. Indeed, these were also responsible for a large part of the complexity of the voting systems and a significant portion of their codebases.

Other features that were attractive to the vendors' customers were low cost, ease of deployment, and rapid reporting of election results. These drove the design of the high level architecture of the Diebold system, particularly its bidirectional data flow. (Most other vendors use a similar architecture.) Returning electronic records to the central office for tabulation is not strictly necessary for a functioning DRE voting system—and it greatly amplifies the risk of viral attack—but it does reduce the time and cost of counting the vote.

Complicating matters further were the widely varying requirements of different jurisdictions. Since election laws vary from state to state, customers demanded a range of different ballot styles, reporting formats, and deployment styles, and, due to the HAVA deadline, all these requirements needed to be met in a short period of time. Much of the complexity of the Diebold system seems to have resulted from these demands. For example, the AccuBa-

sic interpreter was created to allow flexible ballot and report styles, but it also introduced several exploitable vulnerabilities [174].

HAVA itself imposed additional requirements, which, though they were uniform across all states, further complicated the products. To meet the law's accessibility requirements, vendors implemented features such as audio interfaces and larger fonts for the visually impaired. Implementing these features quickly motivated the use of complex COTS operating systems and device drivers. They also drove vendors towards complex PC-style system architectures, which could more easily support graphics and sound, rather than simpler special purpose hardware.

> **Cause 3.2: Complicated Requirements and Demand for Features Magnified Complexity**
> The vendors' customers demanded complicated and often diverse features that would be difficult to satisfy with a simple design. The result was a system that was too complicated to be implemented securely.

Implications

- Customers based their purchasing decisions mainly on visible features, giving vendors a disincentive to invest in security improvement "under the hood." This is reflected in Diebold's development practices, which appear to have emphasized features rather than security.
- Another reason that some voting official may not have demanded security features is such features can be costly and time consuming for officials to implement. Security measures such as VVPATs and automatic manual recounts could detect many kinds of attacks, but they also increase the workload of election staffers. By purchasing machines

without a VVPAT, officials could conveniently ensure that there would be no records to audit.

- Even if Diebold had invested more in security, the system was likely too complex to be secure. It may have been impossible to securely implement the features that customers wanted at a price they could afford.

3.6.3 Certification and Testing

In most states, voting machines are purchased by individual counties. Though most county voting officials are well intentioned, they usually lack the technical expertise necessary to evaluate the vendors' security claims. Even if they could obtain this expertise, the vendors typically refused to provide the machines' source code for examination before purchase, and they sometimes invoked NDAs to prevent counties from allowing outside experts to inspect their machines.

Due to these constraints, most counties have been forced to rely on federal and state voting machine certification processes to ensure that the machines offered for sale are secure. Yet, despite their very serious security flaws, the Diebold DREs were certified according to federal and state standards. This demonstrates that the certification processes are deficient.

Analysis

The machines were tested and certified under the Federal Election Commission's 2002 Voting System Standards [59]. These say relatively little about security and focus instead on the machine's reliability if used non-maliciously.

In general, the certification process seems to rely more on testing than analysis. Testing—i.e., reviewing the system's compliance with a checklist of specific measurable properties—is appropriate for some properties of interest, such as reliability in the face of heat, cold, and

vibration, but it is ill-suited for finding security problems. As discussed by Anderson [6] and many others, testing can only show that a system works under specific, predefined conditions; it generally cannot ensure that there is no way for an attacker to achieve some goal by violating these conditions. Only a competent and thorough security analysis can provide any confidence that the system can resist the full range of realistic attacks.

Some of the weaknesses we discovered highlight why this kind of testing is insufficient. Consider the mechanical lock used to protect the memory card on the AV-TS. If the required test was only the *presence* of a lock, the machine would have passed. Yet, as we discovered, the lock is easily picked, nullifying its security.

A better requirement would have been that the machine have a strong lock. Yet, even if we had not been able to pick the lock, it would still have been insecure, because the key used in every machine was widely sold on the Internet [25].

An even better requirement would have been that the machine have a strong lock with a proprietary key. However, even in this case, the AV-TS still would not have been secure. Diebold's online store displayed a picture of the key with sufficient clarity to allow it to be duplicated by an attacker [61]. Finally, even if there were a strong lock with control of the keys, one can just remove the screws in the base of the machine's case to gain access to the motherboard and replace the bootloader EPROM.

This example illustrates that testing to any sort of security checklist, however well specified, cannot anticipate every sort of attack. A far more effective way to identify problems is to subject the system to adversarial inspection by independent analysts. The many security problems identified in "certified " voting machines by the studies in Section 3.5 are a testament to this difference.

Cause 3.3: Established Testing and Certification Processes were Insufficient to Detect Security Problems
Though the Diebold machines were certified under Federal standards, the certification process relied on "checklist"-style security testing, which is insufficient to detect unanticipated kinds of attacks. A better certification regime would subject the machines to adversarial security analysis by truly independent third parties.

- Weak certification would be less of a problem if information about the system's design were more widely available to the public. Researchers and other experts would be able to provide valuable feedback on voting machine designs if they had the information to do so. Ideally, strong certification procedures would be coupled with public scrutiny to provide the highest assurance.
- No amount of analysis or testing can guarantee that a system is secure. Conducting trustworthy elections using computer voting machines may require designing the voting system to be *software independent* [143]; that is, so that an undetected change or error in the software cannot cause an undetectable change or error in an election outcome.
- The U.S. Election Assistance Commission issued voluntary voting system guidelines [168] in 2005. These are considerably more detailed, especially in the area of security, than the FEC's 2002 standards. Though it would not be entirely fair to apply the 2005 guidelines to the pre-2005 version of the AccuVote software we studied, we do note that the AccuVote hardware architecture may make it impossible to comply with the 2005 guidelines, in particular with the requirement to detect unauthorized modifications to the system software (see [168], Volume I, Section 7.4.6). In practice, a technology can be deployed despite noncompliance with certification requirements if the testing agencies fail to notice the problem.

3.7 Mitigation

The vulnerabilities that we have described can be mitigated, to some extent, by changing voting machine designs and election procedures. In this section we discuss several potential mitigation strategies and their limitations.

Logic and accuracy testing Logic and accuracy testing provides little defense against software-based attacks. Malicious software running on the machines can detect whether officials are performing logic and accuracy tests and can force the machine to behave normally until the testing completes.

Commercial virus scanners Commercially available virus scanners provide little defense against the kinds of attacks described in this report. They normally are only able to recognize PC viruses that have been observed in the wild on many computers. However, they cannot detect new attacks never seen before, and they are not designed to detect virally propagating malicious code that targets voting equipment and voting software.

Stricter chain-of-custody measures We are not optimistic that stricter chain-of-custody controls will prove effective in addressing the vulnerabilities identified in this report. We were not able to identify any realistic procedures that would ensure that voting equipment and memory cards remain under two-person control at all times. Leaving voting machines unattended overnight in a polling place breaks the chain of custody and creates an opportunity for an attacker to tamper with the machines. Sending voting equipment home with the chief poll worker allows that person unsupervised access to the equipment; since in many counties essentially any registered voter who volunteers can become a poll worker, it is difficult to prevent an attacker from becoming a poll worker. Since it might take only one compromised machine to spread a virus to all the county's voting machines, the prospects

for devising chain-of-custody rules that will meet the necessary level of perfection in practice seem dim.

Tamper-evident seals We do not expect that tamper-evident seals will be effective at detecting tampering with voting equipment while it has been left unattended. First, the Diebold polling place equipment does not appear to have been designed to meet this threat model. Investigators have identified several ways that a voter might be able to tamper with an AV-TSX machine while in the process of voting [2]. Second, most, if not all, tamper-evident seals have known vulnerabilities that could allow an attacker to break them and then replace them or restore them to a condition where the tampering is unlikely to be detected [88]. Third, it is challenging to devise protocols that make it likely that poll workers will detect and respond appropriately to tampering; few poll workers have prior training as a seal inspector, and it is not practical to provide the kind of training that would be needed in the already-rushed training that poll workers receive. Fourth, the false alarm rate (where seals are broken or unverifiable for innocuous, ordinary reasons) is so high that election workers may become inured to these issues; it may be difficult to ensure that broken seals are consistently taken seriously enough. Since it only takes one compromised machine to infect the entire county's voting machines, we do not believe that tamper-evident seals can prevent introduction of virally propagating malicious code.

Forensics Forensics performed after election day may be helpful to determine the cause and nature of attacks. However, procedures to govern forensic analysis should be in place before any problems are detected. Viruses and other malicious software could be designed to remove traces of their activities from the voting machines at the end of the election, so workers need to collect and preserve evidence even before they suspect an attack. Ideally,

some number of voting machines and memory cards should be randomly selected and set aside, unused, so that any attack software present will be preserved for analysis.

Parallel testing Parallel testing is another partial mitigation to consider. It involves selecting a random sample of DRE machines, taking them aside, and running a mock election on election day using the equipment. By preparing a known voting slate, one can compare the results from those machines against the inputs that mock voters entered. Parallel tests are one way to detect bugs or malice in DRE software, if the faulty software is widespread enough that the random sample is likely to pick at least one DRE that exhibits incorrect behavior.

The reliability of parallel testing at detecting malicious code appears to be open to debate. The effectiveness of parallel testing is heavily dependent upon the details of how the testing is done. If malicious software can distinguish when it is being tested from normal operations, for instance by looking for mistakes that inexperienced voters would make but officials performing tests would not, then the malicious software can evade detection by behaving correctly when it is under test.

Ultimately, parallel testing becomes an arms race between attack designers and officials who plan realistic parallel tests [24]. The defenders attempt to design testing procedures that mimic real elections as closely as possible, while we must assume the attackers will try to design methods to detect when they are being tested. It is not clear who has the advantage in this race. The problem with this kind of arms race is that it is difficult to know who is winning. Thus, there is a risk that an attacker might develop a secret way to defeat parallel testing, leaving the defenders with a false sense of security about election integrity.

Voter-verifiable paper records One of the critical security mechanisms for the Diebold voting system is the voter-verifiable paper trail created by some AV-TSX machines. The idea

is that, in case an attacker manages to replace the certified software on the AV-TSX with malicious software, the paper trail will provide a way to detect misbehavior by the malicious software. Any strategy to mitigate the Diebold system's technical problems must take into account the limitations of the paper trail system.

Voter-verifiable paper records (paper ballots and VVPATs) are perhaps the best defense against vote-stealing attacks; however, they may not be adequate to detect and recover from attacks that change only a small number of votes. The design of the paper audit trail greatly influences its effectiveness. Voters should be strongly encouraged to review the contents of the VVPAT record and to report any discrepancies to poll workers. Discrepancies should be logged and reported to election officials and centrally tracked on election day to monitor for signs of a widespread problem.

VVPATs provide little defense against most kinds of denial-of-service attacks, since the machines cannot print VVPAT records if they are not operational. Attackers may also target the VVPAT directly, for instance, by programming the machine to exhaust the supply of paper.

Changes to the high-level architecture Another potential approach that is worth investigating involves deploying two separate GEMS installations at county headquarters, a permanent GEMS and a sacrificial GEMS. This setup reduces the bidirectional data flow (see Section 3.4) that enables viruses to spread quickly through the voting system.

The permanent GEMS installation would be used for laying out the ballot, defining the election, and writing to memory cards before the election. The sacrificial GEMS installation would be used for reading memory cards, accumulating and tabulating results, and producing reports. The latter installation can be reformatted after the election and is never used to write memory cards, so if it is infected by a virus, at least the virus will not be able to spread to every other voting machine in the county.

This architecture is motivated by the observation that the key step in the propagation of the virus in Section 3.3.3 is when an infected central-office voting machine is used to write many memory cards destined for the field, infecting all of them. This step is what causes the virus to spread so rapidly. If we can ensure that no infected central-office voting machine is ever used to write memory cards, then we can prevent the rapid viral spread of Section 3.3.3.

Getting the details right is challenging. For instance, the memory cards must be erased before they are reused with the permanent GEMS system in the next election. Yet, as we will discuss in Section 4.7.3, removable memory cards are intelligent storage devices, containing their own microprocessors and firmware, that may be subject to attack. It might be possible for an adversary to reprogram a memory card—or create his own lookalike memory card—that would pretend to be erased but actually still contain a malicious payload. For a more complete analysis of the dual-GEMS approach, see [77].

Building a voting system that is secure by design Given the costs of designing a new voting system, leaving the Diebold software largely unmodified and relying on procedural changes to mitigate the threats that we describe may seem attractive to policymakers. We consider this to be a risky approach, however, because we are not convinced that it is possible to fully resolve the security problems in the Diebold system through procedural means. We are concerned that, because the Diebold system is vulnerable in so many ways, the procedures needed to protect it would be extensive, complex, and hard to follow. We worry that despite the best efforts and intentions of election officials, the procedures would not be followed perfectly every time and the system would sometimes be left open to attack. As a result, we believe that rather than attempting to retrofit security onto a flawed system, it is safer to re-engineer the Diebold system so that it is secure by design.

Building a secure voting system requires making security a central part of the design process from the start. It also demands the involvement of election administrators, experienced software architects and developers, and experts in software security and physical security. Such a system would need to use design techniques appropriate for security-critical systems, such as threat modeling, attack surface reduction, defense in depth, and privilege separation. It would need to apply sound, generally accepted engineering practices for secure software, including input validation, defensive programming, and security testing and assessment. Designing a secure voting system is an expensive proposition that requires a long-term commitment, but the ultimate benefit of doing so is increased confidence in the electoral process.

3.8 Conclusion

Our study of the Diebold voting system found that it does not meet the requirements for a security-critical system. It is built upon an inherently fragile design and suffers from implementation flaws that can expose the entire voting system to attacks. These vulnerabilities, if exploited, could jeopardize the integrity of elections. An attack could plausibly be accomplished by a single skilled individual with temporary access to a single voting machine. The damage could be extensive—malicious code could spread to every voting machine in polling places and to county election servers. Even with a paper trail, malicious code might be able to subtly influence close elections, and it could disrupt elections by causing widespread equipment failure on election day.

Our analysis has led us to conclude that the design and implementation of the Diebold software does not meet the requirements for a security-critical system. We identified a number of systemic issues that were pervasive throughout the source code or that reflected

flaws in the design of the voting system. We also found that the Diebold system's code fails to consistently follow sound, generally accepted engineering practices for secure software.

Electronic voting machines have their advantages, but experience with the Diebold system and other DREs shows that they tend to be complex systems that are prone to very serious vulnerabilities. Making them safe, given the limitations of today's technology, will require safeguards beginning with software-independent design and truly independent security evaluation.

Chapter 4

Security Failures in On-the-Fly Disk Encryption Software

Contrary to popular assumption, dynamic RAM (DRAM), the main memory used in most modern computers, commonly retains its contents for several seconds after power is lost, even at room temperature and even when removed from a motherboard. In this chapter, we show how this phenomenon, called *memory remanence*, limits the ability of an operating system to protect cryptographic key material against an attacker with physical access to a machine, and we demonstrate how it can be exploited to defeat several popular on-the-fly disk encryption systems.

Most security practitioners have assumed that a computer's memory is erased almost immediately when it loses power, or that whatever data remains is difficult to retrieve without specialized equipment. We show that these assumptions are incorrect. Without power, DRAM loses its contents gradually over a period of seconds, and data will persist for minutes or even hours if the chips are kept at low temperatures. Residual data can be

recovered using simple, nondestructive techniques that require only momentary physical access to the machine.

We present a suite of attacks that exploit DRAM remanence to recover cryptographic keys held in memory. They pose a particular threat to laptop users who rely on disk encryption products. An adversary who steals a laptop while an encrypted disk is mounted could employ our attacks to access the contents, even if the computer is screen-locked or suspended when it is stolen.

On-the-fly disk encryption software operates between the file system and the storage driver, encrypting disk blocks as they are written and decrypting them as they are read. The encryption key is typically protected with a password typed by the user at login. The key needs to be kept available so that programs can access the disk; most implementations store it in RAM until the disk is unmounted.

The standard argument for disk encryption's security goes like this: As long as the computer is screen-locked when it is stolen, the thief will not be able to access the disk through the operating system; if the thief reboots or cuts power to bypass the screen lock, memory will be erased and the key will be lost, rendering the disk inaccessible. Yet, as we show, memory is not always erased when the computer loses power, and an attacker can exploit this to learn the encryption key and decrypt the disk. We demonstrate this risk by defeating five popular disk encryption systems—BitLocker, TrueCrypt, FileVault, LoopAES, and dm-crypt—and we expect many similar products are also vulnerable.

Our attacks come in three variants of increasing resistance to countermeasures. The simplest is to reboot the machine and launch a custom kernel with a small memory footprint that gives the adversary access to the residual memory. A more advanced attack is to briefly cut power to the machine, then restore power and boot a custom kernel; this deprives the operating system of any opportunity to scrub memory before shutting down. An even

stronger attack is to cut the power, transplant the DRAM modules to a second PC prepared by the attacker, and use it to extract their state. This attack additionally deprives the original BIOS and PC hardware of any chance to clear the memory on boot.

If the attacker is forced to cut power to the memory for too long, the data will become corrupted. We examine two methods for reducing corruption and for correcting errors in recovered encryption keys. The first is to cool the memory chips prior to cutting power, which dramatically prolongs data retention times. The second is to apply algorithms we have developed for correcting errors in private and symmetric keys. These techniques can be used alone or in combination.

While our principal focus is disk encryption, any sensitive data present in memory when an attacker gains physical access to the system could be subject to attack. For example, we found that Mac OS X leaves the user's login password in memory, where we were able to recover it. SSL-enabled web servers are vulnerable, since they normally keep in memory private keys needed to establish SSL sessions. DRM systems may also face potential compromise; they sometimes rely on software to prevent users from accessing keys stored in memory, but attacks like the ones we have developed could be used to bypass these controls.

It may be difficult to prevent all the attacks that we describe even with significant changes to the way encryption products are designed and used, but in practice there are a number of safeguards that can provide partial resistance. We suggest a variety of mitigation strategies ranging from methods that average users can employ today to long-term software and hardware changes. However, each remedy has limitations and trade-offs, and we conclude that there is no simple fix for DRAM remanence vulnerabilities.

We analyze the reasons why these failures occurred, and find two major causes. First, implementing disk encryption secure is difficult, because standard PC architectures provide

running software no safe place to keep secrets that are in active use. Second, disk encryption developers were unaware of the threats posed by the memory remanence phenomenon, because this behavior was concealed by widely used abstractions and mental models about computer operation. We also investigate whether the vendors' incentives played a role in reducing their products' security, with inconclusive results.

4.1 Previous Work

Though our investigation was, to our knowledge, the first security study to focus on DRAM data remanence and the first to demonstrate how it can be used to conduct practical attacks against real disk encryption systems, we were not the first to suggest that data in DRAM might survive reboots or that this might have security implications. Hints that memory behavior did not fit the widely held mental models can be found in the literature going back more than thirty years. Why did it take so long for the mainstream security community to assimilate these fundamental facts about how computers operate? We will return to this question in Section 4.7. For now, before presenting our new results, we will briefly review the state of knowledge about data remanence prior to February 2008.

Among electrical engineering circles, it has known since at least the 1970s that DRAM cell contents survive to some extent even at room temperature and that retention times can be increased by cooling. In a 1978 experiment [103], a DRAM showed no data loss for a full week without refresh when cooled with liquid nitrogen.

The first mention we can find in the computer security literature comes from Anderson [6], who briefly discusses remanence in his 2001 book:

> [A]n attacker can . . . exploit . . . memory remanence, the fact that many kinds of computer memory retain some trace of data that have been stored there. . . . [M]odern

> RAM chips exhibit a wide variety of memory remanence behaviors, with the worst of them keeping data for several seconds even at room temperature. . .

Anderson cites Skorobogatov [158], who found significant data retention times with *static* RAMs at room temperature. Our results for modern DRAMs show even longer retention in some cases.

Anderson's main focus is on "burn-in" effects that occur when data is stored in RAM for an extended period. Gutmann [70, 71] also examines "burn-in," which he attributes to physical changes that occur in semiconductor memories when the same value is stored in a cell for a long time. Accordingly, Gutmann suggests that keys should not be stored in one memory location for longer than several minutes. Our findings concern a different phenomenon: the remanence effects we have studied occur in modern DRAMs even when data is stored only momentarily. These effects do not result from the kind of physical changes that Gutmann described, but rather from the capacitance of DRAM cells.

We owe the suggestion that modern DRAM contents can survive cold boot to Pettersson [131], who seems to have obtained it from Chow, Pfaff, Garfinkel, and Rosenblum [34]. Pettersson suggested that remanence across cold boot could be used to acquire forensic memory images and obtain cryptographic keys, although he did not experiment with the possibility. Chow et al. discovered this property in the course of an experiment on data lifetime in running systems. While they did not exploit the property, they remark on the negative security implications of relying on a reboot to clear memory.

In a recent presentation, MacIver [106] stated that Microsoft considered memory remanence attacks in designing its BitLocker disk encryption system. He acknowledged that BitLocker is vulnerable to having keys extracted by cold-booting a machine when it is used in "basic mode" (where the encrypted disk is mounted automatically without requiring a user to enter any secrets), but he asserted that BitLocker is not vulnerable in "advanced

modes" (where a user must provide key material to access the volume). He also discussed cooling memory with dry ice to extend the retention time. MacIver apparently has not published on this subject.

Other methods for obtaining memory images from live systems include using privileged software running under the host operating system [172], or using DMA transfer on an external bus [52], such as PCI [32], mini-PCI, Firewire [22, 47, 48], or PC Card. Unlike these techniques, our attacks do not require access to a privileged account on the target system, they do not require specialized hardware, and they are resistant to operating system countermeasures. Sophisticated tools have been developed for analyzing memory images, regardless of the acquisition method [175].

The intelligence community may well be aware of the attacks we describe here, but we were unable to find any publications acknowledging this. A 1991 NSA report entitled "A Guide to Understanding Data Remanence in Automated Information Systems" (the "Forest Green Book") makes no mention of remanence in RAM, discussing only remanence on other storage media such as tapes and disks [118].

4.2 DRAM Remanence

A DRAM cell is essentially a capacitor that encodes a single bit when it is charged or discharged [149, 71]. Over time, charge leaks out, and eventually the cell will lose its state, or, more precisely, it will decay to its *ground state*, either zero or one depending on how the cell is wired. To forestall this decay, each cell must be *refreshed*, meaning that the capacitor must be recharged to hold its value—this is what makes DRAM "dynamic." Manufacturers specify a maximum *refresh interval*—the time allowed before a cell is recharged—that is typically on the order of a few milliseconds. These times are chosen conservatively to ensure extremely high reliability for normal computer operations where even infrequent bit errors

	Memory Type	Chip Maker	Memory Density	Make/Model	Year
A	SDRAM	Infineon	128Mb	Dell Dimension 4100	1999
B	DDR	Samsung	512Mb	Toshiba Portégé	2001
C	DDR	Micron	256Mb	Dell Inspiron 5100	2003
D	DDR2	Infineon	512Mb	IBM T43p	2006
E	DDR2	Elpida	512Mb	IBM x60	2007
F	DDR2	Samsung	512Mb	Lenovo 3000 N100	2007

Table 4.1—Test Systems We experimented with six test systems (designated A–F) that encompass a range of recent DRAM architectures and circuit densities.

can cause problems, but, in practice, a failure to refresh any individual DRAM cell within this time has only a tiny probability of actually destroying the cell's contents.

We conducted a series of experiments to characterize DRAM remanence effects and better understand the security properties of modern memories. We performed trials using PC systems with different memory technologies, as shown in Table 4.1. These systems included models from several manufacturers and ranged in age from 9 years to 6 months.

In each experiment, we filled representative memory regions with a pseudorandom test pattern, and read back the data after suspending refreshes for varying periods of time by cutting power to the machine. We measured the error rate for each sample as the number of bit errors (the Hamming distance from the pattern we had written) divided by the total number of bits. Fully decayed memory would have an error rate of approximately 50%, since half the bits would match by chance.

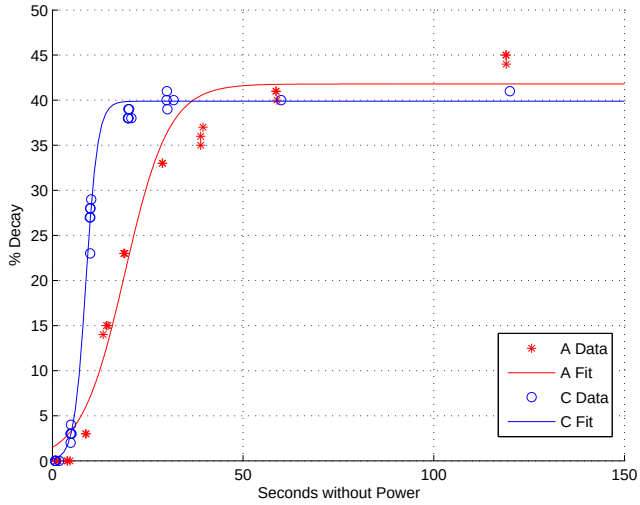
Decay at operating temperature Our first tests measured the decay rate of each machine's memory under normal operating temperature, which ranged from 25.5 °C to 44.1 °C. We found that the decay curves from different machines had similar shapes, with an initial period of slow decay, followed by an intermediate period of rapid decay, and then a final period of slow decay, as shown in Figure 4.1.

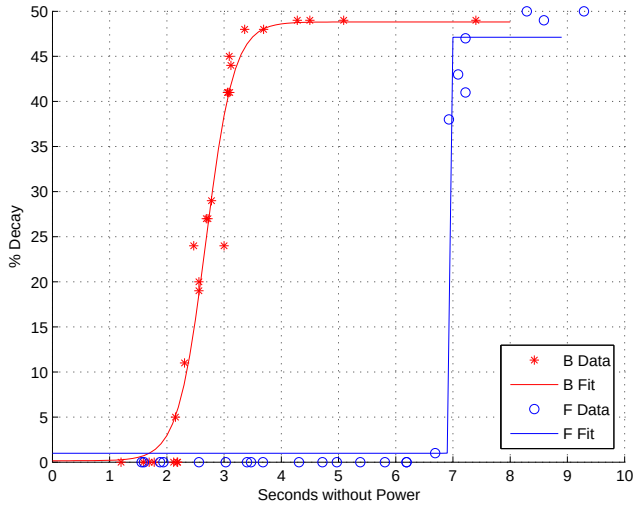
	Seconds Without Power	Average Error Rates (%)	
		(No Cooling)	(−50 °C)
A	60	41	[no errors]
	300	50	0.000095
B	360	50	[no errors]
	600	50	0.000036
C	120	41	0.00105
	360	42	0.00144
D	40	50	0.025
	80	50	0.18

Table 4.2—Cooling Prolongs Remanence We measured DRAM error rates for systems A–D after different intervals without power, first at normal operating temperatures (no cooling) and then at a reduced temperature of −50 °C. Decay occurred much more slowly under the colder conditions.

The dimensions of the decay curves varied considerably between machines, with the fastest exhibiting complete data loss in approximately 2.5 seconds and the slowest taking over a minute. Newer machines tended to exhibit a shorter time to total decay, possibly because newer chips have higher density circuits with smaller cells that hold less charge, but even the shortest times were long enough to enable some of our attacks. While some attacks will become more difficult if this trend continues, manufacturers may attempt to *increase* retention times to improve reliability or lower power consumption.

Decay at reduced temperature Colder temperatures are known to increase data retention times [103, 6, 183, 71, 159, 158]. We performed another series of tests to measure these effects. On machines A–D, we loaded a test pattern into memory, and, with the computer running, cooled the memory module to approximately −50 °C. We then cut power to the machine and maintained this temperature until power and refresh were restored. As expected, we observed significantly slower rates of decay under these reduced temperatures (see Table 4.2). On all of our test systems, the decay was slow enough that an attacker who cut power for 1 minute would recover at least 99.9% of bits correctly.





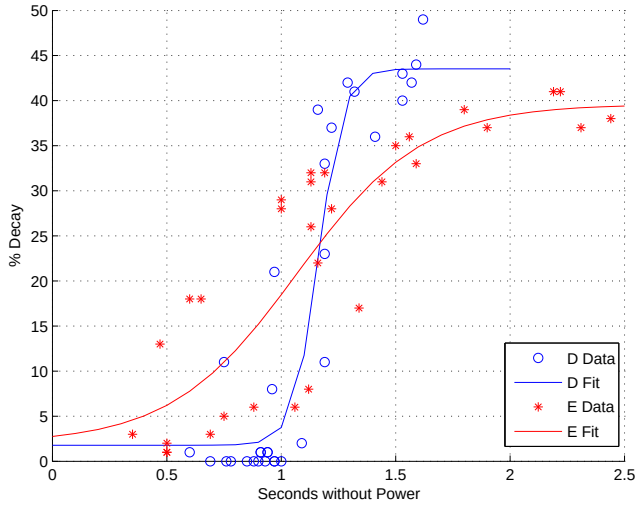


Figure 4.1—DRAM Decay Curves We measured DRAM decay curves for the six test systems. These reflect the average number of bits in a pseudorandom test pattern that changed value after a given interval without power. Data and fits are shown here for machines A and C *(top)*, B and F *(middle)*, and D and E *(bottom)*. All memories were running at normal operating temperature—i.e., without any special cooling. Note that graphs are at different scales.

As an extreme test of memory cooling, we performed another experiment using liquid nitrogen as an additional cooling agent. We first cooled the memory module of machine A to $-50\,^{\circ}$C using the "canned air" product. We then cut power to the machine, and quickly removed the DRAM module and placed it in a canister of liquid nitrogen. We kept the memory module submerged in the liquid nitrogen for 60 minutes, then returned it to the machine. We measured only 14,000 bit errors within a 1 MB test region (0.17% decay). This suggests that, even in modern memory modules, data may be recoverable for hours or days with sufficient cooling.

Decay patterns and predictability We observed that the DRAMs we studied tended to decay in highly nonuniform patterns. While these patterns varied from chip to chip, they were very predictable in most of the systems we tested. Figure 4.2 shows the decay in one memory region from machine A after progressively longer intervals without power.

There seem to be several components to the decay patterns. The most prominent is a gradual decay to the "ground state" as charge leaks out of the memory cells. In the DRAM shown in Figure 4.2, blocks of cells alternate between a ground state of 0 and a ground state of 1, resulting in the series of horizontal bars. Other DRAM models and other regions within this DRAM exhibited different ground states, depending on how the cells are wired.

We observed a small number of cells that deviated from the "ground state" pattern, possibly due to manufacturing variation. In experiments with 20 or 40 runs, a few "retrograde" cells (typically $\sim 0.05\%$ of memory cells, but larger in a few devices) always decayed to the opposite value of the one predicted by the surrounding ground state pattern. An even smaller number of cells decayed in different directions across runs, with varying probabilities.

Apart from their eventual states, the *order* in which different cells decayed also appeared to be highly predictable. At a fixed temperature, each cell seems to decay after a consistent

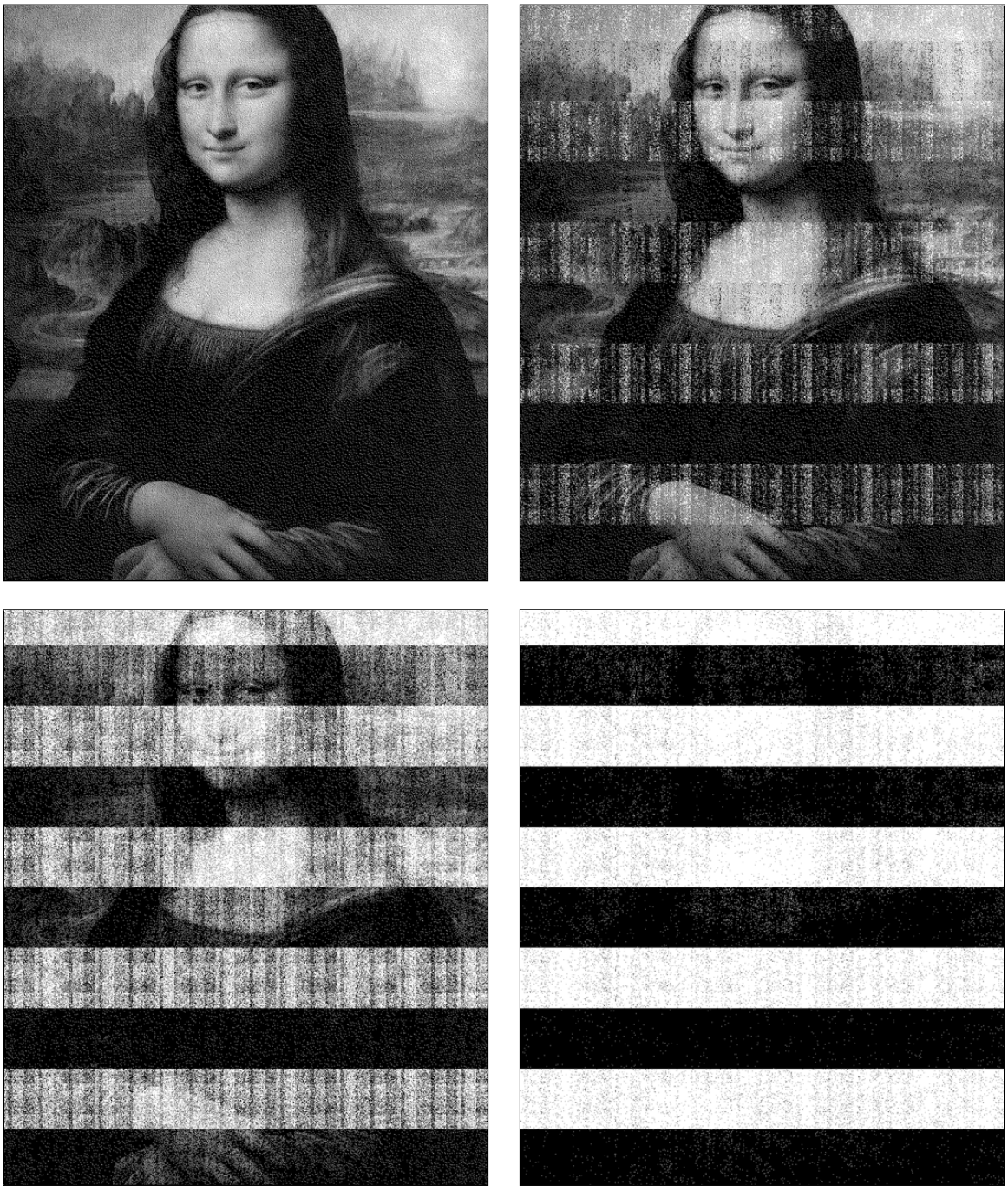


Figure 4.2—Visualizing Remanence and Decay We loaded a bitmap image into memory on test machine A, then cut power for varying intervals. After 5 seconds *(top left)*, the image is nearly indistinguishable from the original; it gradually becomes more degraded, as shown after 30 seconds, 60 seconds, and 5 minutes. Even after this longest trial, traces of the original remain. Note patterns due to ground states (horizontal bands) and physical variations in the chip (fainter vertical bands).

length of time without power. The relative order in which the cells decayed was largely fixed, even as the decay times were changed by varying the temperature. This may also be a result of manufacturing variations, which result in some cells leaking charge faster than others.

To visualize this effect, we captured degraded memory images, including those shown in Figure 4.2, after cutting power for intervals ranging from 1 second to 5 minutes, in 1 second increments. We combined the results into a video. Each test interval began with the original image freshly loaded into memory. We might have expected to see a large amount of variation between frames, but instead, most bits appear stable from frame to frame, switching values only once, after the cell's decay interval. The video also shows that the decay intervals themselves follow higher order patterns, likely related to the physical geometry of the DRAM.

BIOS footprints and memory wiping Even if memory contents remain intact while power is off, the system BIOS may overwrite portions of memory when the machine boots. In the systems we tested, the BIOS overwrote only relatively small fractions of memory with its own code and data, typically a few megabytes concentrated around the bottom of the address space.

On many machines, the BIOS can perform a destructive memory check during its Power-On Self Test (POST). Most of the machines we examined allowed this test to be disabled or bypassed (sometimes by enabling an option called "Quick Boot").

On other machines, mainly high-end desktops and servers that support ECC memory, we found that the BIOS cleared memory contents without any override option. ECC memory must be set to a known state to avoid spurious errors if memory is read without being initialized [15], and we believe many ECC-capable systems perform this wiping operation whether or not ECC memory is installed.

ECC DRAMs are not immune to retention effects, and an attacker could transfer them to a non-ECC machine that does not wipe its memory on boot. Indeed, ECC memory could turn out to *help* the attacker by making DRAM more resistant to bit errors.

4.3 Tools and Attacks

Extracting residual memory contents requires no special equipment. When the system is powered on, the memory controller immediately starts refreshing the DRAM, reading and rewriting each bit value; at this point, the values are fixed, decay halts, and programs running on the system can read any residual data using normal memory-access instructions.

One challenge is that booting the system will necessarily overwrite some portions of memory. While we observed in our tests that the BIOS typically overwrote only a small fraction of memory, loading a full operating system would be very destructive. One solution is to use tiny special-purpose programs that, when booted from either a warm or cold reset state, copy the memory contents to some external medium with minimal disruption to the original state.

Our memory-imaging tools make use of several different attack vectors to boot a system and extract the contents of its memory. For simplicity, each saves memory images to the medium from which it was booted.

PXE network boot Most modern PCs support network booting via Intel's Preboot Execution Environment (PXE) [84], which provides rudimentary startup and network services. We implemented a tiny (9 KB) standalone application that can be booted via PXE and whose only function is streaming the contents of system RAM via a UDP-based protocol. Since PXE provides a universal API for accessing the underlying network hardware, the same binary image will work unmodified on any PC system with PXE support. In a typical attack setup,

a laptop connected to the target machine via an Ethernet crossover cable runs DHCP and TFTP servers as well as a simple client application for receiving the memory data. We have extracted memory images at rates up to 300 Mb/s (around 30 seconds for a 1 GB RAM) with gigabit Ethernet cards.

USB drives Alternatively, most PCs can boot from an external USB device such as a USB hard drive or flash device. We implemented a small (10 KB) plug-in for the SYSLINUX bootloader [9] that can be booted from an external USB device or a regular hard disk. It saves the contents of system RAM into a designated data partition on this device. We succeeded in dumping 1 GB of RAM to a flash drive in approximately 4 minutes.

EFI boot Some recent computers, including all Intel-based Macintosh computers, implement the Extensible Firmware Interface (EFI) instead of a PC BIOS. We have also implemented a memory dumper as an EFI netboot application. We have achieved memory extraction speeds up to 136 Mb/s, and we expect it will be possible to increase this throughput with further optimizations.

iPods We have installed memory imaging tools on an Apple iPod (which behaves like a USB disk) so that it can be used to covertly capture memory dumps without impacting its functionality as a music player. This provides a plausible way to conceal the attack in the wild.

An attacker could use tools like these in a number of ways, depending on his level of access to the system and the countermeasures employed by hardware and software. The simplest attack is to reboot the machine and configure the BIOS to boot the memory extraction tool. A warm boot, invoked with the operating system's restart procedure, will normally ensure that refresh is not interrupted and the memory has no chance to decay,

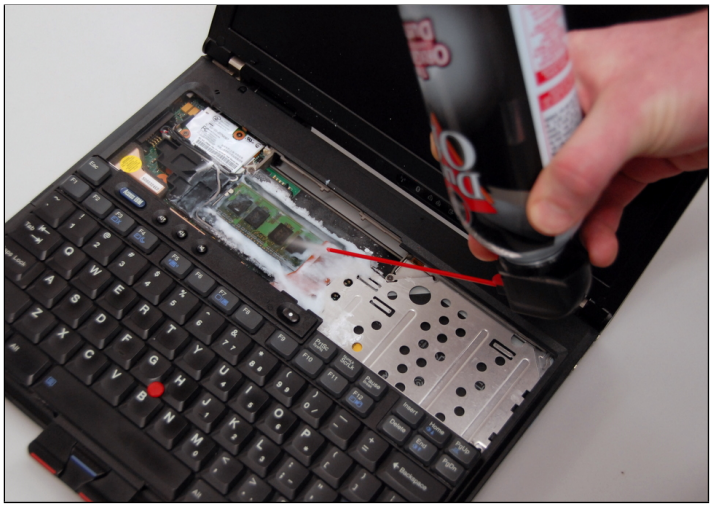


Figure 4.3—Advanced Cold-Boot Attack An advanced cold-boot attack involves reducing the temperature of the memory chips while the computer is still running, then physically moving them to another machine that the attacker has configured to read them without overwriting any data. Before powering off the computer, the attacker can spray the chips with "canned air," holding the container in an inverted position so that it discharges cold liquid refrigerant instead of gas *(top)*. This cools the chips to around −50 °C *(middle)*. At this temperature, the data will persist for several minutes after power loss with minimal error, even if the memory modules are removed from the computer *(bottom)*.

though software will have an opportunity to wipe sensitive data. A cold boot, initiated using the system's restart switch or by briefly removing power, may result in a small amount of decay, depending on the memory's retention time, but denies software any chance to scrub memory before shutting down.

Even if an attacker cannot force a target system to boot memory extraction tools, or if the target employs countermeasures that erase memory contents during boot, an attacker with sufficient physical access can transfer the memory modules to a computer he controls and use it to extract their contents. Cooling the memory before powering it off slows the decay sufficiently to allow it to be transplanted with minimal data loss. Widely-available "canned air" dusters, usually containing a compressed fluorohydrocarbon refrigerant, can easily be used for this purpose. When the can is discharged in an inverted position, as shown in Figure 4.3, it dispenses its contents in liquid form instead of as a gas. The rapid drop in pressure inside the can lowers the temperature of the discharge, and the subsequent evaporation of the refrigerant causes a further chilling. By spraying the contents directly onto memory chips, we can cool their surfaces to $-50\,^{\circ}\mathrm{C}$ and below. If the DRAM is cooled to this temperature before power is cut, and kept cold, we can achieve nearly lossless data recovery even after the chip is out of the computer for several minutes.

4.4 Attacking Cryptographic Keys

The attacker's task is more complicated when the memory is partially decayed, since there may be errors in the cryptographic keys he extracts, but we find that attacks can remain practical. We have developed algorithms for correcting errors in symmetric and private keys that can efficiently reconstruct keys when as few as 27% of the bits are known, depending on the type of key. We have also developed methods for automatically locating keys in potentially decayed memory.

The naïve approach to key error correction, a brute-force search over keys with a low Hamming distance from the decayed key that was retrieved from memory, is computationally burdensome even with a moderate amount of unidirectional error. For 128-bit keys at a 10% unidirectional error, the expected number of keys to be searched is more than 2^{56}.

Our algorithms achieve significantly better performance by considering data other than the raw form of the key. Most encryption programs speed up computation by storing data precomputed from the encryption keys—for block ciphers, this is most often a key schedule, with subkeys for each round; for RSA, this is an extended form of the private key which includes the primes p and q and several other values derived from d. This data contains much more structure than the key itself, and we can use this structure to efficiently reconstruct the original key even in the presence of errors.

These results imply an interesting trade-off between efficiency and security. All of the disk encryption systems we studied (see Section 4.5) precompute key schedules and keep them in memory for as long as the encrypted disk is mounted. While this practice saves some computation for each disk block that needs to be encrypted or decrypted, we find that it greatly simplifies key recovery attacks.

Our approach to key reconstruction has the advantage that it is completely self-contained, in that we can recover the key without having to test the decryption of ciphertext. The data derived from the key, and not the decoded plaintext, provides a certificate of the likelihood that we have found the correct key.

We have found it useful to adopt terminology from coding theory. We may imagine that the expanded key schedule forms a sort of *error correcting code* for the key, and the problem of reconstructing a key from memory may be recast as the problem of finding the closest *code word* (valid key schedule) to the data once it has been passed through a channel that has introduced bit errors.

Modeling the decay Our experiments showed that almost all memory bits tend to decay to predictable ground states, with only a tiny fraction flipping in the opposite direction. In describing our algorithms, we assume, for simplicity, that all bits decay to the same ground state. (They can be implemented without this requirement, assuming that the ground state of each bit is known.)

If we assume we have no knowledge of the decay patterns other than the ground state, we can model the decay with the *binary asymmetric channel*, in which the probability of a 1 flipping to 0 is some fixed δ_0 and the probability of a 0 flipping to a 1 is some fixed δ_1.

In practice, the probability of decaying to the ground state approaches 1 as time goes on, while the probability of flipping in the opposite direction remains relatively constant and tiny (less than 0.1% in our tests). The ground state decay probability can be approximated from recovered key schedules by counting the fraction of 1s and 0s, assuming that the original key schedule contained roughly equal proportions of each value.

We also observed that bits tended to decay in a predictable order that could be learned over a series of timed decay trials, although the actual order of decay appeared fairly random with respect to location. An attacker with the time and physical access to run such a series of tests could easily adapt any of the approaches in this section to take this order into account and improve the performance of the error-correction. Ideally such tests would be able to replicate the conditions of the memory extraction exactly, but knowledge of the decay order combined with an estimate of the fraction of bit flips is enough to give a very good estimate of an individual decay probability of each bit. This probability can be used in our reconstruction algorithms to prioritize guesses.

For simplicity and generality, we will analyze the algorithms assuming no knowledge of this decay order.

4.4.1 Reconstructing DES Keys

We begin with a relatively simple application of these ideas: an error-correction technique for DES keys. Before software can encrypt or decrypt data with DES, it must expand the secret key K into a set of *round keys* that are used internally by the cipher. The set of round keys is called the *key schedule*; since it takes time to compute, programs typically cache it in memory as long as K is in use. The DES key schedule consists of 16 round keys, each a permutation of a 48-bit subset of bits from the original 56-bit key. Every bit from the key is repeated in about 14 of the 16 round keys.

In coding theory terms, we can treat the DES key schedule as a repetition code: the message is a single bit, and the corresponding codeword is a sequence of n copies of this bit.

We begin with a partially decayed DES key schedule. For each bit of the key, we consider the n bits extracted from memory that were originally all identical copies of that key bit. Since we know roughly the probability that each bit decayed $0 \rightarrow 1$ or $1 \rightarrow 0$, we can calculate whether the extracted bits were more likely to have resulted from the decay of reptitions of 0 or repetitions of 1.

If 5% of the bits in the key schedule have decayed to the ground state, the probability that this technique will get any of the 56 bits of the key wrong is less than 10^{-8}. Even if 25% of the bits in the key schedule are in error, the probability that we can correctly reconstruct the key without resorting to a brute force search is more than 98%.

This technique can be trivially extended to correct errors in Triple DES keys. Triple DES applies the same key schedule algorithm to two or three 56-bit key components (depending on the version of Triple DES). With 50% decay under our model, we can correctly decode a 112-bit Triple DES key with at least 97% probability and a 168-bit key with at least 96% probability.

4.4.2 Reconstructing AES Keys

AES is a more modern cipher than DES, and it uses a key schedule with a more complex structure, but nevertheless we can efficiently reconstruct keys. For 128-bit keys, the AES key schedule consists of 11 round keys, each made up of four 32-bit words. The first round key is equal to the key itself. Each subsequent word of the key schedule is generated either by XORing two earlier words, or by performing an operation called the key schedule core (in which the bytes of a word are rotated and each byte is mapped to a new value) on an earlier word and XORing the result with another earlier word.

Instead of trying to correct an entire key at once, we can examine a smaller set of the bits at a time and then combine the results. This separability is enabled by the high amount of linearity in the key schedule. Consider a "slice" of the first two round keys consisting of byte i from words 1–3 of the first two round keys, and byte $i-1$ from word 4 of the first round key (see Figure 4.4). This slice is 7 bytes long, but is uniquely determined by the 4 bytes from the first round key.

Our algorithm exploits this fact as follows. For each possible set of 4 key bytes, we generate the relevant 3 bytes of the next round key, and we order these possibilities by the likelihood that these 7 bytes might have decayed to the corresponding bytes extracted from memory. Now we may recombine four slices into a candidate key, in order of decreasing likelihood. For each candidate key, we calculate the key schedule. If the likelihood of this key schedule decaying to the bytes we extracted from memory is sufficiently high, we output the corresponding key.

When the decay is largely unidirectional, this algorithm will almost certainly output a unique guess for the key. This is because a single flipped bit in the key results in a cascade of bit flips through the key schedule, half of which are likely to flip in the "wrong" direction.

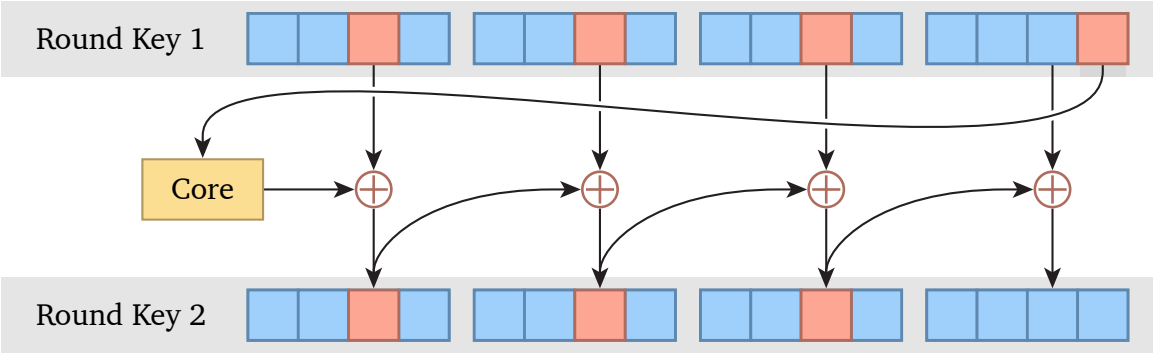


Figure 4.4—Error Correction for AES Keys In the AES-128 key schedule, four bytes from each round key completely determine three bytes of the next round key, as shown here. Our error correction algorithm "slices" the key into four groups of bytes with this property. It computes a list of likely candidate values for each slice, then checks each combination to see if it is a plausible key.

Our implementation of this algorithm is able to reconstruct keys with 7% of the bits decayed in a fraction of a second. It succeeds within 30 seconds for about half of keys with 15% of bits decayed.

This idea can be extended to 256-bit keys by dividing the words of the key into two sections—words 1–3 and 8, and words 4–7, for example—then comparing the words of the third and fourth round keys generated by the bytes of these words and combining the result into candidate round keys to check.

4.4.3 Reconstructing Tweak Keys

The same methods can be applied to reconstruct keys for tweakable encryption modes [104], which are commonly used in disk encryption systems.

LRW LRW augments a block cipher E (and key K_1) by computing a "tweak" X for each data block and encrypting the block using the formula $E_{K_1}(P \oplus X) \oplus X$. A tweak key K_2 is used to compute the tweak, $X = K_2 \otimes I$, where I is the logical block identifier. The operations $\oplus$ and $\otimes$ are performed in the finite field $GF(2^{128})$.

In order to speed tweak computations, implementations commonly precompute multiplication tables of the values $K_2 x^i \mod P$, where x is the primitive element and P is an irreducible polynomial over $GF(2^{128})$ [90]. In practice, $Qx \mod P$ is computed by shifting the bits of Q left by one and possibly XORing with P.

Given a value $K_2 x^i$, we can recover nearly all of the bits of K_2 simply by shifting right by i. The number of bits lost depends on i and the nonzero elements of P. An entire multiplication table will contain many copies of nearly all of the bits of K_2, allowing us to reconstruct the key in much the same way as the DES key schedule.

As an example, we apply this method to reconstruct the LRW key used by the TrueCrypt 4 disk encryption system. TrueCrypt 4 precomputes a 4048-byte multiplication table consisting of 16 blocks of 16 lines of 4 words of 4 bytes each. Line 0 of block 14 contains the tweak key.

The multiplication table is generated line by line from the LRW key by iteratively applying the shift-and-XOR multiply function to generate four new values, and then XORing all combinations of these four values to create 16 more lines of the table. The shift-and-XOR operation is performed 64 times to generate the table, using the irreducible polynomial $P = x^{128} + x^7 + x^2 + x + 1$. For any of these 64 values, we can shift right i times to recover $128 - (8 + i)$ of the bits of K_2, and use these recovered values to reconstruct K_2 with high probability.

XEX and XTS For XEX [144] and XTS [82] modes, the tweak for block j in sector I is $X = E_{K_2}(I) \otimes x^j$, where I is encrypted with AES and x is the primitive element of $GF(2^{128})$. Assuming the key schedule for K_2 is kept in memory, we can use the AES key reconstruction techniques to reconstruct the tweak key.

4.4.4 Reconstructing RSA Private Keys

An RSA public key consists of the modulus N and the public exponent e, while the private key consists of the private exponent d and several optional values: prime factors p and q of N, $d \bmod (p-1)$, $d \bmod (q-1)$, and $q^{-1} \bmod p$. Given N and e, any of the private values is sufficient to efficiently generate the others. In practice, RSA implementations store some or all of these values to speed computation.

In this case, the structure of the key information is the mathematical relationship between the fields of the public and private key. It is possible to iteratively enumerate potential RSA private keys and prune those that do not satisfy these relationships. Subsequent to our initial publication, Heninger and Shacham [78] showed that this leads to an algorithm that is able to recover in seconds an RSA key with all optional fields when only 27% of the bits are known.

4.4.5 Automatically Identifying Keys in Memory

After extracting the memory from a running system, an attacker needs some way to locate the cryptographic keys. This is like finding a needle in a haystack, since the keys might occupy only tens of bytes out of gigabytes of data. Simple approaches, such as attempting decryption using every block of memory as the key, are intractable if the memory contains even a small amount of decay.

We have developed fully automatic techniques for locating encryption keys in memory images, even in the presence of errors. We target the key schedule instead of the key itself, searching for blocks of memory that satisfy the properties of a valid key schedule.

Although previous approaches to key recovery do not require a key schedule to be present in memory, they have other practical drawbacks that limit their usefulness for our purposes. Shamir and van Someren [154] conjecture that keys have higher entropy than the

other contents of memory and claim that they should be distinguishable by a simple visual test. However, even perfect copies of memory often contain large blocks of random-looking data (e.g., compressed files). Pettersson [131] suggests locating program data structures containing key material based on the range of likely values for each field. This approach requires the manual derivation of search heuristics for each cryptographic application, and it is not robust to memory errors.

We propose the following algorithm for locating scheduled AES keys in extracted memory:

1. Iterate through each byte of memory. Treat that address as the start of an AES key schedule.
2. Calculate the Hamming distance between each word in the potential key schedule and the value that would have been generated from the surrounding words in a real, undecayed key schedule.
3. If the sum of the Hamming distances is sufficiently low, the region is close to a correct key schedule; output the key.

We implemented this algorithm for 128- and 256-bit AES keys in an application called `keyfind`. The program receives extracted memory and outputs a list of likely keys. It assumes that key schedules are contiguous regions of memory in the byte order used in the AES specification; this can be adjusted for particular cipher implementations. A threshold parameter controls how many bit errors will be tolerated. We apply a quick test of entropy to reduce false positives.

As described Section 4.5, we successfully used `keyfind` to recover keys from closed-source disk encryption programs without having to reverse engineer their key data structures. In other tests, we even found key schedules that were partially overwritten after the memory where they were stored was reallocated.

This approach can be applied to many other ciphers, including DES. A similar method works to identify the precomputed multiplication tables used for advanced cipher modes like LRW (see Section 4.4.3). To locate RSA keys, we can search for known key data or for characteristics of the standard data structure used for storing RSA private keys; we successfully located the SSL private keys in memory extracted from a computer running Apache 2.2.3 with `mod_ssl`. For details, see [75].

4.5 Attacking Disk Encryption Software

Encrypting hard drives is an increasingly common countermeasure against data theft, and many users assume that disk encryption products will protect sensitive data even if an attacker has physical access to the machine. A California law adopted in 2002 [30] requires disclosure of possible compromises of personal information, but offers a safe harbor whenever data was "encrypted." Though the law does not include any specific technical standards, many observers have recommended the use of full-disk or file system encryption to obtain the benefit of this safe harbor. (At least 38 other states have enacted data breach notification legislation [119].) Our results below suggest that disk encryption, while valuable, is not necessarily a sufficient defense.

We have applied some of the tools developed in this chapter to attack popular on-the-fly disk encryption systems. The most time-consuming parts of these tests were generally developing system-specific attacks and setting up the encrypted disks. Actually imaging memory and locating keys took only a few minutes and were almost fully automated by our tools. We expect that most disk encryption systems are vulnerable to such attacks.

BitLocker BitLocker, which is included with some versions of Windows Vista, operates as a filter driver that resides between the file system and the disk driver, encrypting and

decrypting individual sectors on demand. The keys used to encrypt the disk reside in RAM, in scheduled form, for as long as the disk is mounted.

In a paper released by Microsoft, Ferguson [64] describes BitLocker in enough detail for us both to discover the roles of the various keys and to program an independent implementation of the BitLocker encryption algorithm without reverse engineering any software. BitLocker uses the same pair of AES keys to encrypt every sector on the disk: a sector pad key and a CBC encryption key. These keys are, in turn, indirectly encrypted by the disk's master key. To encrypt a sector, the plaintext is first XORed with a pad generated by encrypting the byte offset of the sector under the sector pad key. Next, the data is fed through two diffuser functions, which use a Microsoft-developed algorithm called Elephant. The purpose of these un-keyed functions is solely to increase the probability that modifications to any bits of the ciphertext will cause unpredictable modifications to the entire plaintext sector. Finally, the data is encrypted using AES in CBC mode using the CBC encryption key. The initialization vector is computed by encrypting the byte offset of the sector under the CBC encryption key.

We have created a fully-automated demonstration attack called BitUnlocker. It consists of an external USB hard disk containing Linux, a custom SYSLINUX-based bootloader, and a FUSD [56] filter driver that allows BitLocker volumes to be mounted under Linux. To use BitUnlocker, one first cuts the power to a running Windows Vista system, connects the USB disk, and then reboots the system off of the external drive. BitUnlocker then automatically dumps the memory image to the external disk, runs `keyfind` on the image to determine candidate keys, tries all combinations of the candidates (for the sector pad key and the CBC encryption key), and, if the correct keys are found, mounts the BitLocker encrypted volume. Once the encrypted volume has been mounted, one can browse it like any other volume

in Linux. On a modern laptop with 2 GB of RAM, we found that this entire process took approximately 25 minutes.

BitLocker differs from other disk encryption products in the way that it protects the keys when the disk is not mounted. In its default "basic mode," BitLocker protects the disk's master key solely with the Trusted Platform Module (TPM) found on many modern PCs. This configuration, which may be quite widely used [64], is particularly vulnerable to our attack, because the disk encryption keys can be extracted with our attacks even if the computer is powered off for a long time. When the machine boots, the keys will be automatically loaded into RAM from the TPM before the login screen, without the user having to enter any secrets.

It appears that Microsoft is aware of this problem [106] and recommends configuring BitLocker in "advanced mode," where it protects the disk key using the TPM along with a password or a key on a removable USB device. However, even with these measures, BitLocker is vulnerable if an attacker gets to the system while the screen is locked or the computer is asleep (though not if it is hibernating or powered off).

FileVault Apple's FileVault disk encryption software has been examined and reverse-engineered in some detail [177]. In Mac OS X 10.4, FileVault uses 128-bit AES in CBC mode. A user-supplied password decrypts a header that contains both the AES key and a second key k_2 used to compute IVs. The IV for a disk block with logical index I is computed as HMAC-SHA1$_{k_2}(I)$.

We used our EFI memory imaging program to extract a memory image from an Intel-based Macintosh system with a FileVault volume mounted. Our `keyfind` program automatically identified the FileVault AES key, which did not contain any bit errors in our tests.

With the recovered AES key but not the IV key, we can decrypt 4080 bytes of each 4096 byte disk block (all except the first AES block). The IV key is present in memory. Assuming

no bits in the IV key decay, an attacker can identify it by testing all 160-bit substrings of memory to see whether they create a plausible plaintext when XORed with the decryption of the first part of the disk block. The AES and IV keys together allow full decryption of the volume using programs like `vilefault` [178].

In the process of testing FileVault, we discovered that Mac OS X 10.4 and 10.5 keep multiple copies of the user's login password in memory, where they are vulnerable to imaging attacks. Login passwords are often used to protect the default keychain, which may protect passphrases for FileVault disk images.

TrueCrypt TrueCrypt is a popular open-source disk encryption product for the Windows, Mac OS, and Linux platforms. It supports a variety of ciphers, including AES, Serpent, and Twofish. In version 4, all ciphers used LRW mode; in version 5, they use XTS mode (see Section 4.4.3). TrueCrypt stores a cipher key and a tweak key in the volume header for each disk, which is then encrypted with a separate key derived from a user-entered password.

We tested TrueCrypt versions 4.3a and 5.0a running on a Linux system. We mounted a volume encrypted with a 256-bit AES key, then briefly cut power to the system and used our memory imaging tools to record an image of the retained memory data. In both cases, our `keyfind` program was able to identify the 256-bit AES encryption key, which did not contain any bit errors. For TrueCrypt 5.0a, `keyfind` was also able to recover the 256-bit AES XTS tweak key without errors.

To decrypt TrueCrypt 4 disks, we also need the LRW tweak key. We observed that TrueCrypt 4 stores the LRW key in the four words immediately preceding the AES key schedule. In our test memory image, the LRW key did not contain any bit errors. (Had errors occurred, we could have recovered the correct key by applying the techniques we developed in Section 4.4.3.)

dm-crypt Linux kernels starting with 2.6 include built-in support for dm-crypt, an on-the-fly disk encryption subsystem. The dm-crypt subsystem handles a variety of ciphers and modes, but defaults to 128-bit AES in CBC mode with non-keyed IVs.

We tested a dm-crypt volume created and mounted using the LUKS (Linux Unified Key Setup) branch of the `cryptsetup` utility and kernel version 2.6.20. The volume used the default AES-CBC format. We briefly powered down the system and captured a memory image with our PXE kernel. Our `keyfind` program identified the correct 128-bit AES key, which did not contain any bit errors. After recovering this key, an attacker could decrypt and mount the dm-crypt volume by modifying the `cryptsetup` program to allow input of the raw key.

Loop-AES Loop-AES is an on-the-fly disk encryption package for Linux systems. In its recommended configuration, it uses a so-called "multi-key-v3" encryption mode, in which each disk block is encrypted with one of 64 encryption keys. By default, it encrypts sectors with AES in CBC mode, using an additional AES key to generate IVs.

We configured an encrypted disk with Loop-AES version 3.2b using 128-bit AES encryption in "multi-key-v3" mode. After imaging the contents of RAM, we applied our `keyfind` program, which revealed the 65 AES keys. An attacker could identify which of these keys correspond to which encrypted disk blocks by performing a series of trial decryptions. Then, the attacker could modify the Linux `losetup` utility to mount the encrypted disk with the recovered keys.

Loop-AES attempts to guard against the long-term memory burn-in effects described by Gutmann [71] and others. For each of the 65 AES keys, it maintains two copies of the key schedule in memory, one normal copy and one with each bit inverted. It periodically swaps these copies, ensuring that every memory cell stores a 0 bit for as much time as it stores a 1 bit. Not only does this fail to prevent the memory remanence attacks that we describe

here, but it also makes it easier to identify which keys belong to Loop-AES and to recover the keys in the presence of memory errors. After recovering the regular AES key schedules using a program like `keyfind`, the attacker can search the memory image for the inverted key schedules. Since very few programs maintain both regular and inverted key schedules in this way, those keys are highly likely to belong to Loop-AES. Having two related copies of each key schedule provides additional redundancy that can be used to identify which bit positions are likely to contain errors.

4.6 Countermeasures and their Limitations

Memory remanence attacks are difficult to prevent because cryptographic keys in active use must be stored *somewhere*. Potential countermeasures focus on discarding or obscuring encryption keys before an adversary might gain physical access, preventing memory extraction software from executing on the machine, physically protecting the DRAM chips, and making the contents of memory decay more readily.

Suspending a system safely Simply locking the screen of a computer (i.e., keeping the system running but requiring entry of a password before the system will interact with the user) does not protect the contents of memory. Suspending a laptop's state to RAM ("sleeping") is also ineffective, even if the machine enters a screen-locked state on awakening, since an adversary could simply awaken the laptop, power-cycle it, and then extract its memory state. Suspending to disk ("hibernating") may also be ineffective unless an externally held secret is required to decrypt the disk when the system is awakened.

With most disk encryption systems, users can protect themselves by powering off the machine completely when it is not in use then guarding the machine for a minute or so until the contents of memory have decayed sufficiently. Though effective, this countermeasure

is inconvenient, since the user will have to wait through the lengthy boot process before accessing the machine again.

Suspending can be made safe by requiring a password or other external secret to reawaken the machine and encrypting the contents of memory under a key derived from the password. If encrypting all of memory is too expensive [34], the system could encrypt only those pages or regions containing important keys. An attacker might still try to guess the password and check his guesses by attempting decryption (an offline password-guessing attack), so systems should encourage the use of strong passwords and employ password strengthening techniques [23] to make checking guesses slower. Some existing systems, such as Loop-AES, can be configured to suspend safely in this sense, although this is usually not the default behavior [14].

Storing keys differently Our attacks show that using precomputation to speed cryptographic operations can make keys more vulnerable, because redundancy in the precomputed values helps the attacker reconstruct keys in the presence of memory errors. To mitigate this risk, implementations could avoid storing precomputed values, instead recomputing them as needed and erasing the computed information after use. This improves resistance to memory remanence attacks but can carry a significant performance penalty. (These performance costs are negligible compared to the access time of a hard disk, but disk encryption is often implemented on top of disk caches that are fast enough to make them matter.)

Implementations could transform the key as it is stored in memory in order to make it more difficult to reconstruct in the case of errors. This problem has been considered from a theoretical perspective; Canetti et al. [31] define the notion of an *exposure-resilient function* (ERF) whose input remains secret even if all but some small fraction of the output

is revealed. This carries a performance penalty because of the need to reconstruct the key before using it.

Physical defenses It may be possible to physical defend memory chips from being removed from a machine, or to detect attempts to open a machine or remove the chips and respond by erasing memory. In the limit, these countermeasures approach the methods used in secure coprocessors [51] and could add considerable cost to a PC. However, a small amount of memory soldered to a motherboard would provide moderate defense for sensitive keys and could be added at relatively low cost.

Architectural changes Some countermeasures involve changes to the computer's architecture that might make future machines more secure. DRAM systems could be designed to lose their state quickly, though this might be difficult given the need to keep the probability of decay within a DRAM refresh interval vanishingly small. Key-store hardware could be added—perhaps inside the CPU—to store a few keys securely while erasing them on power-up, reset, and shutdown. Some proposed architectures would routinely encrypt the contents of memory for security purposes [102, 100, 50]; these would prevent the attacks we describe as long as the keys are reliably destroyed on reset or power loss.

Encrypting in the disk controller Another approach is to perform encryption in the disk controller rather than in software running on the main CPU and to store the key in the controller's memory instead of the PC's DRAM. In a basic form of this approach, the user supplies a secret to the disk at boot, and the disk controller uses this secret to derive a symmetric key that it uses to encrypt and decrypt the disk contents [151].

For this method to be secure, the disk controller must erase the key from its memory whenever the computer is rebooted. Otherwise, an attacker could reboot into a malicious

kernel that simply reads the disk contents. For similar reasons, the key must also be erased if an attacker attempts to transplant the disk to another computer.

While we leave an in-depth study of encryption in the disk controller to future work, we did perform a cursory test of two hard disks with this capability, the Seagate Momentus 5400 FDE.2 and the Hitachi 7K200. We found that they do not appear to defend against the threat of transplantation. We attached both disks to a PC and confirmed that every time we powered on the machine, we had to enter a password via the BIOS in order to decrypt the disks. However, once we had entered the password, we could disconnect the disks' SATA cables from the motherboard (leaving the power cables connected), connect them to another PC, and read the disks' contents on the second PC without having to re-enter the password.

Trusted computing Trusted Computing hardware, in the form of Trusted Platform Modules (TPMs) [166] is now deployed in some personal computers. Though useful against some attacks, most TPMs deployed in PCs today do not prevent the attacks described here.

Such hardware generally does not perform bulk data encryption itself; instead, it monitors the boot process to decide (or help other machines decide) whether it is safe to store a key in RAM. If a software module wants to safeguard a key, it can arrange that the usable form of that key will not be stored in RAM unless the boot process has gone as expected [106]. However, once the key is stored in RAM, it is subject to our attacks. Today's TPMs can prevent a key from being loaded into memory for use, but they cannot prevent it from being captured once it is in memory.

In some cases using a TPM can make the problem worse. BitLocker, in its default "basic mode," protects the disk keys solely with Trusted Computing hardware. When the machine boots, BitLocker automatically loads the keys into RAM from the TPM without requiring the user to enter any secrets. Unlike other disk encryption systems we studied, this configuration

is at risk even if the computer has been shut down for a long time—the attacker needs only power on the machine to have the keys loaded back into memory, where they are vulnerable to our attacks.

4.7 Causes of Disk Encryption Security Failures

At this point we understand the threat posed to disk encryption by memory remanence attacks. What remains is to understand the reasons why these vulnerabilities occurred and why they went undiscovered. Our analysis is informed by the remarkable facts that every disk encryption system we studied was vulnerable to such attacks, and that nearly every vendor we contacted was caught off guard by the severity of the problem; the causes must run deeper than any particular implementation.

This section examines several contributing causes that apply to all disk encryption software. These fall into three groups. First, we discuss deficiencies in the architecture of most PCs, which fail to give software any place to securely store secrets while they are in use. Second, we focus on the incentives that motivated the different vendors. Third, we consider the role of abstractions, which, though an indispensable engineering tool, have the ability to conceal details of the computer's operation that can impact its security. These causes point to other kinds of systems that may be vulnerable to similarly widespread but undiscovered attacks, and they reveal important lessons for broader security contexts.

4.7.1 Architectural Deficiencies

One design feature shared by all the systems we studied is that they retain the encryption key or keys in main memory for as long as the computer is running with the encrypted disk mounted. Discarding the key would be undesirable, since it would have to be reconstructed from user input the next time any software needed to access the encrypted data. Under this

constraint, there is no obviously better place to store the key than in main memory, since most PCs provide no secure place for software to store secrets that are in active use.

Analysis

Other than main memory, there are only a few places in typical PCs where software might store the key. We will consider several of them: registers, the CPU cache, other external memories, a TPM, the disk itself, or remotely on a network. Each has significant drawbacks.

Placing the key on the hard disk is not an adequate defense, of course. One of the threats we wish to protect against is an attacker with physical access to the disk.

Peripheral memories, such as the hard disk's internal cache or the graphics card's on-board RAM, are another possibility, but when these devices are removable, we face the threat that an attacker could transfer them to a machine he controls without cutting power. The key will be vulnerable unless the devices detect that they have been removed and immediately erase it.[1] Furthermore, these memories can be vulnerable to attacks using DMA, which allows them to be read over Firewire and other buses [22].

Storing the key on a remote network device also seems insufficient. The computer needs some way to authenticate to the device, and whatever authentication mechanism is used faces the same challenge of safeguarding a secret from the attacker.

TPMs are a more promising possibility, since they have the ability to store a key so that it will be accessible only to the original operating system kernel running on the original machine. However, while TPM hardware is becoming widespread, it is not ubiquitous. Furthermore, as we have already discussed, today's TPMs do not provide bulk encryption capabilities, so the key must be loaded from the TPM before it can be used; this is a very slow operation compared to main memory accesses, taking on the order of a second [166]. One

1 Using graphics memory (or other RAM) soldered to the motherboard would make this attack much harder, though an attacker might still cool and desolder the chips.

solution would be to transfer the key from the TPM when it is needed and to discard it after some period of inactivity. Unfortunately, this creates a window of vulnerability, particularly if the attacker can force the key to be loaded into memory by triggering disk activity (e.g., by sending network traffic or attaching a new peripheral).

CPU registers and caches are more both promising solutions, since they are fast, tightly integrated into the processor, and erased on boot. With registers, the challenge is to isolate the key from unprivileged software running on the machine, which might leak it or overwrite it. With caches, the system would need to prevent the key from ever being evicted or written into main memory. A cache-based solution is under development by Jürgen Pabel [126]. Unfortunately, it requires disabling normal operation of the CPU cache for one core, which may cause too severe a performance penalty for some applications.

All these potential solutions face another threat that may be more difficult to surmount than poor performance. In this chapter so far, we have focused on attackers who seek to extract the contents of main memory, but attackers may also be able to *alter* data stored there while the system is running. Govindavajhala and Appel introduced attacks against virtual machines based on introducing random memory faults by exposing DRAM chips to intense heat and light [69]. An attacker might be able to make controlled changes using methods similar to our cold boot attack—suspending the system to RAM, cooling the DRAM modules, transplanting them to another machine, writing to them there, then returning them to the original PC and resuming execution. A simpler approach, but one that may be easier for software to defend against, is to rewrite parts of RAM using DMA requests from an external device. Boileau has demonstrated this attack using Firewire [22].

An adversary who can rewrite the contents of RAM could replace parts of the system kernel and other privileged code. He could thereby extract any secret accessible to the

running kernel—including keys stored in TPMs, registers, or CPU caches—or he could simply unlock the system and access the disk directly.

> **Cause 4.1: Software has no place to securely store secrets while they are in use**
> In today's PCs, an attacker with physical access to a machine can read (and potentially modify) the contents of main memory. By leveraging these capabilities, the attacker can access any state that is accessible to applications or the operating system.

Implications

- If we relax our threat model and suppose that attackers can read but not write main memory, we can provide a secure data store with the addition of a single special-purpose CPU register. The register would need to be large enough to store an encryption key, accessible only to the OS kernel, and guaranteed to be discarded when the processor resets. The OS could leverage this capability to provide an API for applications to store their secrets, which, encrypted by the OS with this key, could be stored in regular registers and RAM. This would be an effective countermeasure against attackers who can read, but not write, main memory; however, an attacker who could modify the memory of the running system could inject code that would cause the OS to decrypt secrets thus protected.
- A more complete solution would be to provide memory confidentiality and integrity by adding cryptography to the processor, as we discussed in Section 4.6. In some ways, this would be simple. Key management would be trivial, since the CPU could choose a random key on boot and discard it on reset. It would then use this key to MAC and encrypt each cache line as it was evicted from the cache, and to decrypt and verify

each cache line read back in. Providing full encryption and integrity verification at the speed of the memory bus might be a challenging performance problem, but it could be accomplished at some cost with the addition of dedicated cryptographic circuitry.

4.7.2 Vendor Incentives

In previous chapters, we observed that differences in incentives between system vendors and other parties contributed to security failures. Were similar factors at work here? To answer this question, we compare the behavior of disk encryption vendors who faced different kinds of incentives. We divide the vendors into two groups: (1) Microsoft and Apple, which are large companies that integrated disk encryption features into their operating system products; (2) the developers of Loop-AES, dm-crypt, and TrueCrypt, which are open source projects, primarily run by volunteers and given away for free, with an exclusive focus on providing security. These groups are motivated by greatly dissimilar incentives, so if incentives contributed to some of the security failures, we would expect to see differences between the impact of the problem on the products from each groups of vendors.

Analysis

One hypothesis is that incentives are most strongly aligned between users and developers when the product's primary purpose is security. When this is the case, there is intense competitive pressure for the vendor to implement its product securely. Customers can easily switch to a competing product, and many will do so if there is a major security failure. On the other hand, the hypothesis continues, in cases where the disk encryption system is simply a component of an operating system, the incentives for security are weaker. Security failures will not necessarily drive customers to use a different OS. They select an operating system on the basis of a basket of features, and once they have selected one OS, lock-in

makes it expensive for them to move to an alternative. Instead, most dissatisfied customers will buy a third-party disk encryption product rather than a different OS.

We find some evidence to support this theory in differences between the way Microsoft approached memory remanence attacks and the way the Loop-AES developers did. Microsoft was aware of memory remanence when it designed BitLocker (though they seem not to have realized how easily the phenomenon could be exploited). Nevertheless, they implemented "basic mode" in a way that amplifies its vulnerability, and they designed the more secure "advanced modes" in a way that is difficult for users to configure. Furthermore, the company did not communicate the relative risks of each mode to their customers clearly. On the other hand, the developers of Loop-AES took steps to defend against residual key data being recovered from memory, but they had an incomplete understanding of the physical effects involved. As we discussed in Section 4.6, their defense ended up aggravating the problem rather than mitigating it.

Arguably, Loop-AES did their best to defend against the memory attacks as they understood them, while Microsoft knew better but made things worse. Though differences in incentives might explain these behaviors, the remaining products do not seem to fit the pattern. Comparing the failures of TrueCrypt and dm-crypt with the failure of Apple's FileVault, we find that they failed in essentially identical ways; none of the vendors involved seem to have been aware of memory vulnerabilities, and none attempted to defend against them. Thus, we are hesitant to draw a general conclusion about the role of incentives in disk encryption security failures.

A competing explanation for Microsoft's failure to communicate the risks of these attacks to its customers has to do with widely held perceptions about physical attacks. When we initially disclosed these vulnerabilities to Microsoft, their support representative dismissed our concerns, pointing to the company's "10 Immutable Laws of Security": "Law #3: If

a bad guy has unrestricted physical access to your computer, it's not your computer anymore." [110]. Of course, protecting data from theft by an adversary who has physical access to the storage medium is the primary purpose of disk encryption! Perhaps this "Immutable Law" led Microsoft's user interface designers and documentation writers to take defenses against remanence attacks less seriously, even though other employees at the company did understand their importance.

This knee-jerk reaction reflects a bias in parts of the security community against dealing with physical threats. This attitude is certainly not universal; there have been efforts to produce physically tamper-resistant systems in a number of contexts, including military and financial applications [160], DRM [166], smart cards [95], and mobile devices. Nevertheless, many people seem to regard efforts to defend against physical attacks as futile. Further evidence comes from Slashdot commentors' responses to a story about our memory remanence attacks: "It's a common tenant [sic.] in systems security that anyone with physical access and sufficient time can disable or otherwise bypass any security system. ... Likewise, software security means nothing if the hardware is vulnerable" [124]. "If the attacker has physical control over the machine, the game is ... lost" [117]. Physical attacks seem to be discounted because defending against them is hard, but this is not an excuse to ignore them.

4.7.3 Abstractions and Security

These memory remanence attacks came a surprise to many in the computer security community, including many experts in the disk encryption field. I personally disclosed them to the CTO of a major security firm that produced a disk encryption product; his jaw dropped. The developers of all the products we tested appear to have been caught off guard—even

if they were cognizant of the remanence phenomenon, they seem not to have been aware of the ease with which it can be exploited.

Yet not everyone was surprised. On another occasion, I mentioned these findings to the president emeritus of a semiconductor firm, who happened to have been one of the first DRAM designers. *Of course* memory behaves this way, he said, scolding me for suggesting that remanence and the means to recovery residual data had been unknown. With marked frustration, he told me that manufacturers could easily have made memory that decayed almost immediately, but that nobody had ever specified this requirement.

What happens to RAM during a reboot is a basic aspect of computer behavior. How could it have been unknown for so long to so many people who needed to know—especially when it was well known in other parts of the computing field? The answers have to do with the way engineering fields are structured and the tools that engineers use to reason about complex behaviors.

Analysis

Abstraction is a critical tool in many engineering disciplines, including computer science. We use it to compartmentalize the complex behavior of different parts of a system. Some abstractions are formal specifications of behavior passed up from lower levels (e.g. APIs) or passed down from higher levels (e.g. system requirements). Others are informal mental models that more loosely describe component behaviors. Without such abstractions, it would be impossible to make sense of a computer's behavior.

Abstractions, formal and informal, are useful expedients for understanding complex systems and necessary tools for achieving cooperation among different system components. Unfortunately, they sometimes mask attributes of behavior that turn out to have dire security implications. For example, we often think about TCP connections as if they were telephone-

like circuits to a remote end point. However, low-level details of TCP's operation make it susceptible to attacks like packet injection [16], which poorly fit the telephone analogy. Hash functions are another example. We often think of them abstractly as if they were random oracles, but hash functions based on the popular Merkle-Damgård construction are subject to length extension attacks [115], whereby the attacker who knows the hash of a message can derive the hash of the message concatenated with a chosen suffix. In both cases, anticipating and defending against the attacks requires opening the "black box" of the abstraction and understanding the lower level implementational details.

The idea that a PC's memory is erased as soon as the machine is powered off is a similar kind of informal abstraction. Users quickly come to understand that a power interruption destroys their unsaved data. Like the prevalent mental models for TCP and hash functions, this understanding is accurate enough for many scenarios, where developers are chiefly concerned about reliability and data loss. As we have seen, it is not quite accurate, but it fails only in a few cases such as security against physical attacks. As a result, most people never investigated the details of memory's behavior that made our attacks possible, and the inaccurate intuitions they carried over from everyday experience with data loss mislead them into believing their systems were secure.

Cause 4.2: Abstractions concealed security-critical behavior
Disk encryption developers built systems that depended on DRAM contents being erased during a cold reboot because they relied on an abstraction about memory behavior. While accurate enough for most software development scenarios, the intuition that memory is erased as soon as it loses power ignores phenomena that turn out to be critical in some security contexts.

Implications

- Abstraction-related security failures highlight a lack of communication between the developers of different layers of a system. It may be tempting to blame developers of lower level components for not adequately documenting behaviors that are important to security at higher levels, but the problem runs in both directions. Application designers need to do a better job specifying the behaviors that the OS and hardware must provide in order for the entire system to be secure.
- These communication failures are exacerbated by a lack of communication between engineering disciplines. Memory remanence has been understood in the semiconductor community for decades, but it proved shocking to many security experts. Likewise, the semiconductor community seems to have been caught off guard by applications' reliance on memory volatility for security. We tend to erect abstractions at the boundaries between disciplines in order to separate the responsibilities of experts in each field. These divisions are reinforced by differences in terminology, course work, and publication venues within different fields. The best antidote may be to require a broad education for engineering students. Merely understanding programming or circuit design is not enough—students need to understand the behavior of higher and lower level systems in order to achieve security.
- Other abstractions may also conceal vulnerabilities. By generalizing this lesson, we can anticipate security failures in different contexts where widely held abstractions mask important aspects of component behavior.

  For example, programmers usually think of the CPU as executing commands in the instruction set (e.g., x86) in which it is programmed, but many modern CPUs first translate the instructions into a lower level microcode. Attacks may be possible that

exploit errors in the translation process or that reprogram the CPU via microcode updates to induce insecure behavior.

Another example concerns nonvolatile storage. Many in the computer field think of modern storage devices as if they were floppy disk drives—unintelligent hardware supporting simple seeks, reads, and writes. In reality, many of today's storage devices, such as hard disks, solid-state drives, and compact flash cards are separate computers, containing their own processors and programmable firmware. Subverting this firmware could lead to a variety of attacks. A number of potential attacks involve programming the compromised disk to respond differently to commands under different circumstances. For instance, they might pretend to obey erasure commands while secretly retaining sensitive data where an attacker could later access it; they might return a benign file when read by a virus scanner but return malicious code when read by a program loader; or they might insert the adversary's login information when the password file is read at a predetermined attack time. Many additional attacks are conceivable.

We hope to investigate these potential attacks and other abstraction-related failures in future work.

- These findings highlight the value of security as an independent subfield within computer science. Attackers are not confined by divisions between academic interest areas, and mounting a strong defense requires a top-to-bottom view. As its own young subfield, computer security has the potential to bridge the discipline's entrenched abstractions and seek to understand computers and their use at all levels, from silicon to social institutions.

4.8 Conclusions

Contrary to common belief, DRAMs hold their values for surprisingly long intervals without power or refresh. We show that this fact enables attackers to extract cryptographic keys and other sensitive information from memory despite the operating system's efforts to secure memory contents. The attacks we describe are practical—for example, we have used them to defeat several popular disk encryption systems. These results imply that disk encryption on laptops, while beneficial, does not guarantee protection.

In recent work, Chan et al. [33] demonstrate a dangerous extension to our attacks. They show how to reset a running computer, surgically alter its memory, and then restore the machine to its previous running state. This allows the attacker to defeat a wide variety of security mechanisms—including disk encryption, screen locks, and antivirus software—by tampering with data in memory before reanimating the machine. This attack can potentially compromise data beyond the local disk; for example, it can be executed quickly enough to bypass a locked screen before any active VPN connections time out. Though it appears that this attack would be technically challenging to execute, it illustrates that memory's vulnerability to physical attacks presents serious threats that security researchers are only beginning to understand.

There seems to be no easy remedy for memory remanence attacks. Ultimately, it might become necessary to treat DRAM as untrusted and to avoid storing sensitive data there, but this will not be feasible until architectures are changed to give running software a safe place to keep secrets.

Chapter 5

Conclusion

In this dissertation, we investigated the security of a variety of different systems and discovered that they all suffered from serious vulnerabilities. The severity and prevalence of these problems paint a grim portrait of security in contemporary computer systems. They might lead some to conclude, pessimistically, that attack research is destined to remain little more than a game of security whack-a-mole, correcting an endless stream of flaws one bug at a time.

Though security problems are likely to remain a fact of life, we find reasons to be hopeful about research's potential to mitigate them. Rather than merely exposing flaws, our investigations carry lessons much broader than the individual faults we discovered. In this way, the study of security failures can have a multiplicative impact, improving security far beyond the systems it examines.

We have already seen several classes of broader benefits from our research. It has brought about significant changes in the ways the systems we studied are used. Record labels recalled the dangerous CD-DRM systems, several states decertified the brittle DRE voting machines, and many users reconsidered the trust they placed in disk encryption systems. It has altered implementers' understanding of the threats they need to defend against,

establishing DRM as a potential security risk to users' PCs, proving that e-voting machines are susceptible to viral attacks, and demonstrating that physical attacks on memory are a practical attack vector. It has motivated new research directions in response to these threats, such as the development of machine-assisted election auditing [29], which applies technology to strengthen elections without forcing us to trust that technology, and the creation of cryptographic primitives that are more secure against cold-boot attacks [4]. Clearly, the flaws we exposed have resulted in security gains that are more fundamental than mere bug fixes.

Studying security failures can also shed light on the reasons why these problems occur. Our analyses have pointed to a number of generalizable factors—facts about technologies, patterns of behavior, etc.—that apply to systems well beyond those we studied. Examples include CD-DRM's losing battle against the convergence of the PC and the CD player, systemic problems with electronic voting machine certification, and the PC architecture's lack of a secure place for software to store secrets while they are in use. Future systems must avoid these pitfalls in order to be secure.

Though drawing lessons like these is a time-honored practice in security research, we have argued that the results have been weakened by the lack of a uniform approach to learning from failures. We have proposed a new methodology to serve this role, which we call the analytic approach to security research. Our approach coordinates the search for security failures with the search for informative causes, factors that explain why failures occurred and can help head off other problems in the future. This conceptual framework helps to partially systematize (perhaps, to "civilize") the process of learning from failures. In this dissertation we applied it to our past work, and we found that it organized and honed the lessons we drew and helped us expose new lessons.

The greatest potential benefit of applying a more uniform approach to learning from failures in different systems is that it can expose patterns of causes that are common across dissimilar systems. Stepping back and examining these clusters of causes can carry deep lessons for all security. Three overarching themes are apparent:

- Incentives play a powerful role in shaping security-critical behaviors—DRM vendors took risky and aggressive actions because they were insulated from liability, voting machine vendors focused on features instead of strong security because of heightened pressure to bring products to market quickly. Understanding incentives may suggest nontechnical approaches for improving security—imposing liability for failures, reducing risky mismatches in incentives among parties, avoiding creating inflated incentives to skimp on security.
- Intuitions about the behavior of systems can be misleading, masking security dangers—contrary to widespread beliefs, audio CDs can be an attack vector, electronic voting machines are more similar to PCs than to mechanical voting machines, and memory does not lose its contents immediately upon losing power. This may be a valuable template for discovering other security failures; searching for similar misunderstandings in unrelated systems will likely reveal other kinds of vulnerabilities.
- Tension between conflicting goals leads to security trade-off. Sometimes different kinds of security are in conflict (e.g., stronger DRM versus stronger PC security), sometimes security conflicts with system complexity (e.g., strengthened voting machines versus administrative convenience and attractive UIs), and sometimes security conflicts with performance (e.g., faster disk encryption versus heightened vulnerability to remanence attacks). Security is almost never free, but understanding the costs involved helps us make informed decisions about how much security to buy, and it helps focus our efforts on reducing those costs.

The analytic methodology that we have proposed and applied in this dissertation provides a starting point for systematizing the understanding of security failures, but much remains to be done. Despite numerous attempts to devise systematic approaches for discovering failures and causes, none is widely accepted, and these remain largely creative exercises. While it seems unlikely that they can ever become fully mechanical, they would surely be practiced more widely and consistently if formal techniques proved effective. For the most part, our methodology operates at a higher level, providing a structure for directing and organizing investigations rather than a roadmap. Though we have found it to be a promising tool, it still needs to be validated through further experience in other investigations. The proof of the pudding, as they say, is in the eating. If we can demonstrate the usefulness of these techniques in our own future work, we hope that they will be the basis for a stronger consensus within the research community about how we should learn from security failures.

Thus, we conclude optimistically that we are on the road to better security. Investigating how systems fail helps us understand the reasons why they fail, and this ultimately teaches us how to build stronger systems. By developing and applying new approaches to learning from failures, we can transform security problems into opportunities for discovery.

Bibliography

[1] Analysis. In *Wiktionary*, May 2009.

[2] R. P. Abbot, M. Davis, J. Edmonds, L. Florer, E. Proebstel, B. Porter, and J. Stauffer. Security evaluation of the Diebold voting system. Part of the California Secretary of State's Top-to-Bottom Voting Systems Review, July 2007.

[3] R. Abbott, J. Chin, J. Donnelley, W. Konigsford, S. Tokubo, and D. Webb. Security analysis and enhancements of computer operating systems. NBSIR 76–1041, ICET, National Bureau of Standards, Washington, DC, Apr. 1976.

[4] A. Akavia, S. Goldwasser, and V. Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In *TCC*, volume 5444 of *Lecture Notes in Computer Science*, pages 474–495. Springer, 2009.

[5] R. Anderson. Why cryptosystems fail. In *Proc. 1st ACM conference on Computer and Communications Security*, pages 215–227, 1993.

[6] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, Jan. 2001.

[7] R. Anderson. Why information security is hard: An economic perspective. In *Proc. 17th Annual Computer Security Applications Conference*, Washington, DC, 2001.

[8] R. Anderson and M. Kuhn. Tamper resistance – A cautionary note. In *Proc. Second Usenix Workshop on Electronic Commerce*, pages 1–11, Nov. 1996.

[9] H. P. Anvin. SYSLINUX. Online at `http://syslinux.zytor.com/`.

[10] A. W. Appel, M. Ginsburg, H. Hursti, B. W. Kernighan, C. D. Richards, and G. Tan. Insecurities and inaccuracies of the Sequoia AVC Advantage 9.00H DRE voting machine, Oct. 2008.

[11] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *Proc. IEEE Symp. on Security and Privacy*, pages 65–71, May 1997.

[12] T. Aslam and E. H. Spafford. Use of a taxonomy of security faults. In *Proc. 19th National Information Systems Security Conference*, Baltimore, Maryland, 1996.

[13] J. Baker. An Enhanced CD survival guide. Online at `http://www.cdman.com/pdf/ecd.pdf`.

[14] A. Bar-Lev. Linux, Loop-AES and optional smartcard based disk encryption, Nov. 2007. Online at `http://wiki.tuxonice.net/EncryptedSwapAndRoot`.

[15] P. Barry and G. Hartnett. *Designing Embedded Networking Applications: Essential Insights for Developers of Intel IXP4XX Network Processor Systems*, page 47. Intel Press, first edition, May 2005.

[16] S. Bellovin. A look back at "Security problems in the TCP/IP protocol suite". In *Proc. 20th Computer Security Applications Conference*, pages 229–249, 2004.

[17] S. M. Bellovin. Security problems in the TCP/IP protocol suite. *SIGCOMM Comput. Commun. Rev.*, 19(2):32–48, 1989.

[18] P. Biddle, P. England, M. Peinado, and B. Willman. The Darknet and the future of content distribution. In *Proc. 2002 ACM Workshop on Digital Rights Management*, Nov. 2002.

[19] R. Bisbey and D. Hollingworth. Protection analysis: Final report. Technical Report ISI/SR-75–4, University of Southern California Information Sciences Institute, Marina Del Rey, CA, Dec. 1978.

[20] J. Bishop and R. LaRhette. Managing human performance—INPO's human performance evaluation system. In *Proc. IEEE Conference on Human Factors and Power Plants*, pages 471–474, Jun 1988.

[21] M. Blaze, A. Cordero, S. Engle, C. Karlof, N. Sastry, M. Sherr, T. Stegers, and K.-P. Yee. Source code review of the Sequoia voting system. Part of the California Secretary of State's Top-to-Bottom Voting Systems Review, July 2007.

[22] A. Boileau. Hit by a bus: Physical access attacks with Firewire. Presentation, Ruxcon, 2006.

[23] X. Boyen. Halting password puzzles: Hard-to-break encryption from human-memorable keys. In *Proc. 16th USENIX Security Symposium*, Aug. 2008.

[24] Brennan Center Task Force on Voting System Security. The machinery of democracy: Protecting elections in an electronic world, 2006. Online at `http://www.brennancenter.org/programs/downloads/Full%20Report.pdf`.

[25] A. Broache. Diebold reveals 'key' to e-voting? *CNet News.com*, Jan. 2007. Online at http://news.com.com/2061-10796_3-6153328.html.

[26] J. Brunner. Evaluation and validation of election-related equipment, standards and testing (EVEREST), Dec. 2007.

[27] J. Burns and A. Stamos. MediaMax access control vulnerability, Nov. 2005. Online at http://www.eff.org/IP/DRM/Sony-BMG/MediaMaxVulnerabilityReport.pdf.

[28] J. A. Calandrino, A. J. Feldman, J. A. Halderman, D. Wagner, H. Yu, and W. Zeller. Source code review of the Diebold voting system. Part of the California Secretary of State's Top-to-Bottom Voting Systems Review, July 2007.

[29] J. A. Calandrino, J. A. Halderman, and E. W. Felten. Machine-assisted election auditing. In *Proc. 2007 USENIX/ACCURATE Electronic Voting Technology Workshop (EVT 07)*, Aug. 2007.

[30] California Statutes. Cal. Civ. Code §1798.82, created by S.B. 1386, Aug. 2002.

[31] R. Canetti, Y. Dodis, S. Halevi, E. Kushilevitz, and A. Sahai. Exposure-resilient functions and all-or-nothing transforms. In *Advances in Cryptology – EUROCRYPT 2000*, volume 1807/2000, pages 453–469, 2000.

[32] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1:50–60, Dec. 2003.

[33] E. M. Chan, J. C. Carlyle, F. M. David, R. Farivar, and R. H. Campbell. Bootjacker: compromising computers using forced restarts. In *Proc. 15th ACM Conference on Computer and Communications Security*, pages 555–564, Oct. 2008.

[34] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. 14th USENIX Security Symposium*, pages 331–346, Aug. 2005.

[35] S. Christey and R. A. Martin. Vulnerability type distributions in CVE, version 1.1, May 2007. Online at http://cwe.mitre.org/documents/vuln-trends/index.html.

[36] Class action complaint. In *Hull et al. v. Sony BMG et al.*, 2005.

[37] Columbia Accident Investigation Board. Final report, Aug. 2003.

[38] Compuware. Direct recording electronic (DRE) technical security assessment report, Nov. 2003. Online at http://www.sos.state.oh.us/sos/hava/compuware112103.pdf.

[39] I. Cox, J. Kilian, T. Leighton, and T. Shamoon. Secure spread spectrum watermarking for multimedia. *IEEE Transactions on Image Processing*, 6(12):1673–1687, 1997.

[40] S. A. Craver, M. Wu, B. Liu, A. Stubblefield, B. Swartzlander, D. S. Wallach, D. Dean, and E. W. Felten. Reading between the lines: Lessons from the SDMI challenge. In *Proc. 10th USENIX Security Symposium*, Aug. 2001.

[41] DataRescue sa/nv. IDA Pro Disassembler. Online at `http://www.datarescue.com/idabase`.

[42] D. Dean, E. W. Felten, and D. S. Wallach. Java security: From HotJava to Netscape and beyond. In *Proc. 1996 IEEE Symposium on Security and Privacy*, pages 190–200, 1996.

[43] D. Dill and D. Wallach. Stones unturned: Gaps in the investigation of Sarasota's disputed congressional election, Apr. 2007. Online at `http://www.cs.rice.edu/~dwallach/pub/sarasota07.html`.

[44] C. Doctorow. The 3-minute guide to the broadcast flag. Online at `http://w2.eff.org/IP/broadcastflag/three_minute_guide.php`.

[45] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM Journal on Computing*, 38(1):97–137, 2008.

[46] D. Dorner. *The Logic of Failure: Recognizing and Avoiding Error in Complex Situations*. Metropolitan Books, 1996.

[47] M. Dornseif. 0wned by an iPod. Presentation, PacSec, 2004.

[48] M. Dornseif. FireWire – All your memory are belong to us. Presentation, CanSecWest/core05, May 2005.

[49] S. Drimer, S. Murdoch, and R. Anderson. Thinking inside the box: System-level failures of tamper proofing. In *IEEE Symp. on Security and Privacy*, pages 281–295, May 2008.

[50] J. Dwoskin and R. B. Lee. Hardware-rooted trust for secure key management and transient trust. In *Proc. 14th ACM Conference on Computer and Communications Security*, pages 389–400, Oct. 2007.

[51] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 secure coprocessor. *Computer*, 34:57–66, Oct. 2001.

[52] K. Eckstein and M. Dornseif. On the meaning of 'physical access' to a computing device: A vulnerability classification of mobile computing devices. Presentation, NATO C3A Workshop on Network-Enabled Warfare, Apr. 2005.

[53] ECMA. Data interchange on read-only 120 mm optical data discs (CD-ROM). ECMA standard 130, June 1996.

[54] Election Data Services. 2006 voting equipment study. Online at http://www.edssurvey.com/images/File/ve2006_nrpt.pdf.

[55] Election Science Institute. DRE analysis for May 2006 primary, Cuyahoga County, Ohio, Aug. 2006. Online at http://bocc.cuyahogacounty.us/GSC/pdf/esi_cuyahoga_final.pdf.

[56] J. Elson and L. Girod. FUSD – a Linux framework for user-space devices. Online at http://www.circlemud.org/~jelson/software/fusd/.

[57] Essay: Cactus Data Shield 200. CDR-Info, Jan. 2002. Online at http://www.cdrinfo.com/Sections/News/Details.asp?RelatedID=1926.

[58] Federal Building and Fire Investigation of the World Trade Center Disaster. Final report of the National Construction Safety Team on the Collapses of the World Trade Center Tower. Technical Report NCSTAR 1, NIST, Sept. 2005.

[59] Federal Election Commission. Voting system standards, 2002. Online at http://www.eac.gov/election_resources/vss.html.

[60] J. Feld and K. Carper. *Construction Failure*. Wiley, 2nd edition, 1997.

[61] A. J. Feldman, J. A. Halderman, and E. W. Felten. Security analysis of the Diebold AccuVote-TS voting machine. In *Proc. 2007 USENIX/ACCURATE Electronic Voting Technology Workshop (EVT 07)*, Aug. 2007.

[62] E. W. Felten. NJ election day: Voting machine status, June 2008. http://www.freedom-to-tinker.com/blog/felten/nj-election-day-voting-machine-status.

[63] E. W. Felten and J. A. Halderman. Digital rights management, spyware, and security. *IEEE Security and Privacy*, 4(1):18–23, January/February 2006.

[64] N. Ferguson. AES-CBC + Elephant diffuser: A disk encryption algorithm for Windows Vista, Aug. 2006. Online at http://www.microsoft.com/downloads/details.aspx?FamilyID=131dae03-39ae-48be-a8d6-8b0034c92555.

[65] Feurio version history, 1.64. Online at http://www.feurio.com/English/history_1_64.shtml.

[66] A. Friedman, R. Baliga, D. Dasgupta, and A. Dreyer. Understanding the broadcast flag: A threat analysis model. In *Telecommunications Policy*, volume 28, pages 503–521, 2004.

[67] D. Geer, C. Pfleeger, B. Schneier, J. Quarterman, P. Metzger, R. Bace, and P. Gutmann. CyberInsecurity: The cost of monopoly, 2003.

[68] R. Gonggrijp and W.-J. Hengeveld. Studying the Nedap/Groenendaal ES3B voting computer: A computer security perspective. In *Proc. 2007 USENIX/ACCURATE Electronic Voting Technology Workshop*.

[69] S. Govindavajhala and A. Appel. Using memory errors to attack a virtual machine. In *IEEE Symp. on Security and Privacy*, 2003.

[70] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proc. 6th USENIX Security Symposium*, pages 77–90, July 1996.

[71] P. Gutmann. Data remanence in semiconductor devices. In *Proc. 10th USENIX Security Symposium*, pages 39–54, Aug. 2001.

[72] J. A. Halderman. Evaluating new copy-prevention techniques for audio CDs. In *Proc. 2002 ACM Workshop on Digital Rights Management*, Nov. 2002.

[73] J. A. Halderman. Analysis of the MediaMax CD3 copy-prevention system. Technical Report TR-679-03, Princeton University Computer Science Department, Princeton, NJ, Oct. 2003.

[74] J. A. Halderman and E. W. Felten. Lessons from the Sony CD DRM episode. In *Proc. 15th USENIX Security Symposium*, pages 77–92, Aug. 2006.

[75] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proc. 17th USENIX Security Symposium*, July 2008.

[76] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.

[77] J. A. Halderman, H. Shacham, E. Rescorla, and D. Wagner. You go to elections with the voting system you have: Stop-gap mitigations for deployed voting systems. In *Proc. 2008 USENIX/ACCURATE Electronic Voting Technology Workshop (EVT 08)*, July 2008.

[78] N. Heninger and H. Shacham. Improved RSA private key reconstruction for cold boot attacks. Cryptology ePrint Archive, Report 2008/510, Dec. 2008.

[79] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[80] H. Hursti. Critical security issues with Diebold TSx (unredacted), May 2006. Online at http://www.wheresthepaper.org/reports/BBVreportIIunredacted.pdf.

[81] H. Hursti. Diebold TSx evaluation: Critical security issues with Diebold TSx, May 2006. Online at http://www.bbvdocs.org/reports/BBVreportIIunredacted.pdf.

[82] IEEE 1619 Security in Storage Working Group. IEEE P1619/D19: Draft standard for cryptographic protection of data on block-oriented storage devices, July 2007.

[83] S. Inguva, E. Rescorla, H. Shacham, and D. S. Wallach. Source code review of the Hart InterCivic voting system. Part of the California Secretary of State's Top-to-Bottom Voting Systems Review, July 2007.

[84] Intel. Preboot Execution Environment (PXE) specification version 2.1, Sept. 1999.

[85] International Electrotechnical Commission. Compact disc digital audio system. IEC standard 60908, Feb. 1999.

[86] Interview with Talbot Iredale, Software Development Manager, Diebold Election Systems.

[87] K. Itabashi. Trojan.Welomoch technical description, Dec. 2005. Online at http://securityresponse.symantec.com/avcenter/venc/data/trojan.welomoch.html.

[88] R. G. Johnston. Tamper-indicating seals. *American Scientist*, 94:515–523, November-December 2006. Online at http://ephemer.al.cl.cam.ac.uk/~rja14/johnson/newpapers/American%20Scientist%20(2006).pdf.

[89] K. Julisch. Clustering intrusion detection alarms to support root cause analysis. *ACM Transactions on Information and System Security*, 6:443–471, 2002.

[90] C. Kent. Draft proposal for tweakable narrow-block encryption, 2004. Online at https://siswg.net/docs/LRW-AES-10-19-2004.pdf.

[91] A. Kerckhoffs. La cryptographie militaire. *J. des Sciences Militaires*, 9:161–191, 1883.

[92] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *IEEE Symp. on Security and Privacy*, May 2006.

[93] D. Kirovski and F. A. Petitcolas. Replacement attack on arbitrary watermarking systems. In *Proc. ACM Workshop on Digital Rights Management*, 2002.

[94] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach. Analysis of an electronic voting system. In *IEEE Symp. on Security and Privacy*, May 2004.

[95] O. Kömmerling and M. Kuhn. Design principles for tamper-resistant smartcard processors. In *Proc. USENIX Workshop on Smartcard Technology*, pages 9–20, 1999.

[96] I. Krsul, E. Spafford, and M. Tripunitara. Computer vulnerability analysis. Technical Report COAST TR 98-07, Purdue University, 1998.

[97] I. V. Krsul. *Software Vulnerability Analysis*. PhD thesis, West Lafayette, IN, 1998.

[98] M. Kuhn and R. Anderson. Soft tempest: Hidden data transmission using electromagnetic emanations. *Lecture Notes in Computer Science*, 1525:124–142, 1998. Online at `citeseer.ist.psu.edu/article/kuhn98soft.html`.

[99] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM Comput. Surv.*, 26(3):211–254, 1994.

[100] R. B. Lee, P. C. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Proc. Intl. Symp. on Computer Architecture*, pages 2–13, 2005.

[101] A. Leventhal. Mac OS X and the missing probes, Jan. 2008. Online at `http://blogs.sun.com/ahl/entry/mac_os_x_and_the`.

[102] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[103] W. Link and H. May. Eigenschaften von MOS-Ein-Transistorspeicherzellen bei tiefen Temperaturen. *Archiv für Elektronik und Übertragungstechnik*, 33:229–235, June 1979.

[104] M. Liskov, R. L. Rivest, and D. Wagner. Tweakable block ciphers. In *Advances in Cryptology – CRYPTO 2002*, pages 31–46, 2002.

[105] Y. Liu. Backdoor.Ryknos.B technical description, Nov. 2005. Online at `http://securityresponse.symantec.com/avcenter/venc/data/backdoor.ryknos.b.html`.

[106] D. MacIver. Penetration testing Windows Vista BitLocker drive encryption. Presentation, Hack In The Box, Sept. 2006.

[107] A. McFadden. comp.publish.cdrom FAQ, June 2002. Online at http://www.cdrfaq.org/.

[108] MediaMax Technology Corp. Annual report (S.E.C. Form 10-KSB/A), Sept. 2005.

[109] J. B. Michael, S. E. Roberts, J. M. Voas, and T. C. Wingfield. The role of policy in balancing outsourcing and homeland security. *IT Professional*, 7(4):19–23, 2005.

[110] Microsoft. 10 immutable laws of security. Online at http://technet.microsoft.com/en-us/library/cc722487.aspx.

[111] Microsoft. Enabling and disabling AutoRun. Online at http://msdn.microsoft.com/en-us/library/cc144204.aspx.

[112] Microsoft. What's the difference between AutoPlay and autorun? Windows Vista Help. Online at http://windowshelp.microsoft.com/Windows/en-us/help/a19ac945-1007-4638-9615-e2c3bfd92b751033.mspx.

[113] Microsoft. Windows Media data session toolkit. Online at http://download.microsoft/com/download/a/1/a/a1a66a2c-f5f1-450a-979b-ddf790756f1d/Data_Session_Datasheet.pdf.

[114] Microsoft. Security bulletin MS07-067 – Important: Vulnerability in Macrovision driver could allow local elevation of privilege, Dec. 2007. Online at http://www.microsoft.com/technet/security/Bulletin/MS07-067.mspx.

[115] I. Mironov. Hash functions: Theory, attacks, and applications. Technical Report MSR-TR-2005-187, Microsoft Research, Nov. 2005.

[116] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22:594–597, 1979.

[117] mxs. Re: Clear the DRAM?, Feb. 2008. Online at http://it.slashdot.org/comments.pl?sid=461784&cid=22578400.

[118] National Computer Security Center. A guide to understanding data remanence in automated information systems, Sept. 1991.

[119] National Conference of State Legislatures. State security breach notification laws, Jan. 2008. Online at http://www.ncsl.org/programs/lis/cip/priv/breachlaws.htm.

[120] National Transportation Safety Board. Collapse of I-35W highway bridge, Minneapolis, Minnesota, August 1, 2007. NTSB Accident Report HAR-08/03, Dec. 2008.

[121] NCITS. SCSI multimedia commands 3 (MMC-3). Working draft, revision 10g, Nov. 2001.

[122] Nero AG. Nero Burning ROM. Online at http://ww2.nero.com/enu/Products.html.

[123] M. Nikki. Muzzy's research about Sony's XCP DRM system, Dec. 2005. Online at http://hack.fi/~muzzy/sony-drm/.

[124] orclevegam. Re: Clear the DRAM?, Feb. 2008. Online at http://it.slashdot.org/comments.pl?sid=461784&cid=22504308.

[125] X. Ou, S. Govindavajhala, and A. W. Appel. MulVAL: A logic-based network security analyzer. In *Proc. 14th USENIX Security Symposium*, 2005.

[126] J. Pabel. Frozen cache, Jan. 2009. Online at http://frozencache.blogspot.com/.

[127] C. Perrow. *Normal Accidents: Living with High-Risk Technologies*. Basic Books, 1984.

[128] F. A. Petitcolas, R. J. Anderson, and M. G. Kuhn. Attacks on copyright marking systems. In *Information Hiding*, pages 218–238, 1998.

[129] H. Petroski. *To Engineer Is Human: The Role of Failure in Successful Design*. Basic Books, 1984.

[130] H. Petroski. *Success through Failure: The Paradox of Design*. Princeton University Press, 2006.

[131] T. Pettersson. Cryptographic key recovery from Linux memory dumps. Presentation, Chaos Communication Camp, Aug. 2007.

[132] *Proc. 14th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2007.

[133] *Proc. 15th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2008.

[134] *Proc. 15th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2007.

[135] *Proc. 16th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2008.

[136] *Proc. 16th USENIX Security Symposium*. USENIX, 2007.

[137] *Proc. 17th USENIX Security Symposium*. USENIX, 2008.

[138] *Proc. 2007 IEEE Symp. on Security and Privacy (Oakland)*. IEEE, 2007.

[139] *Proc. 2008 IEEE Symp. on Security and Privacy (Oakland)*. IEEE, 2008.

[140] RABA Technologies. Trusted agent report: Diebold AccuVote-TS voting system, Jan. 2004. Online at http://www.raba.com/press/TA_Report_AccuVote.pdf.

[141] J. Reason. *Human Error*. Cambridge University Press, 1990.

[142] R. W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proc. 2000 IEEE Symp. on Security and Privacy*, Washington, DC, 2000.

[143] R. L. Rivest and J. P. Wack. On the notion of "software independence" in voting systems, July 2006. Online at http://vote.nist.gov/SI-in-voting.pdf.

[144] P. Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In *Advances in Cryptology – ASIACRYPT 2004*, pages 16–31, 2004.

[145] Rogers commission. Presidential commission report on Space Shuttle Challenger accident. Technical report, June 1986.

[146] M. Russinovich. More on Sony: Dangerous decloaking patch, EULAs and phoning home, Nov. 2005. Online at http://www.sysinternals.com/blog/2005/11/more-on-sony-dangerous-decloaking.htm.

[147] M. Russinovich. Sony, rootkits and digital rights management gone too far, Oct. 2005. Online at http://www.sysinternals.com/blog/2005/10/sony-rootkits-and-digital-rights.html.

[148] M. Russinovich. Sony's rootkit: First 4 Internet responds, Nov. 2005. Online at http://www.sysinternals.com/blog/2005/11/sonys-rootkit-first-4-internet.html.

[149] L. Z. Scheick, S. M. Guertin, and G. M. Swift. Analysis of radiation effects on individual DRAM cells. *IEEE Transactions on Nuclear Science*, 47:2534–2538, Dec. 2000.

[150] Science Applications International Corporation. Risk assessment report: Diebold AccuVote-TS voting system and processes (unredacted version), Sept. 2003. Online at http://www.bradblog.com/?p=3731.

[151] Seagate. Drivetrust technology: A technical overview. Online at http://www.seagate.com/docs/pdf/whitepaper/TP564_DriveTrust_Oct06.pdf.

[152] SecuROM: Why they hate Process Explorer. Sysinternals Forum Posting, June 2007. Online at http://forum.sysinternals.com/forum_posts.asp?TID=11000.

[153] A. Serjantov and R. Anderson. On dealing with adversaries fairly. In *Third Workshop on the Economics of Information Security*, 2004.

[154] A. Shamir and N. van Someren. Playing "hide and seek" with stored keys. *Lecture Notes in Computer Science*, 1648:118–124, 1999. Online at citeseer.ist.psu.edu/vansomeren98playing.html.

[155] R. Shirley. Internet security glossary, version 1. RFC 2828, Internet Engineering Task Force, May 2000.

[156] R. Shirley. Internet security glossary, version 2. RFC 4949, Internet Engineering Task Force, Aug. 2007.

[157] P. Sinquin, P. Selve, and R. Alcalay. Anti-counterfeit compact disc. US Patent 6,208,598, Mar. 2001. Assignee: Midbar Tech Ltd.; filed Jan 13, 1999.

[158] S. Skorobogatov. Low-temperature data remanence in static RAM. University of Cambridge Computer Laboratory Technical Report No. 536, June 2002. Online at http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-536.pdf.

[159] S. W. Smith. *Trusted Computing Platforms: Design and Applications*. Springer, first edition, 2005.

[160] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. In *Computer Networks*, pages 831–860, 1998.

[161] L. Solum. Legal theory lexicon 020: Causation, Jan. 2004. Online at http://lsolum.typepad.com/legal_theory_lexicon/2004/01/legal_theory_le.html.

[162] M. Songini. E-voting security under fire in San Diego lawsuit. *Computerworld*, Aug. 2006.

[163] E. H. Spafford. The internet worm program: An analysis. Technical Report CSD-TR-823, Purdue University Computer Science Department, West Lafayette, Indiana, 1988.

[164] State of Maryland. Code of Maryland regulations, title 33, State Board of Elections. Online at http://www.dsd.state.md.us/comar/subtitle_chapters/33_Chapters.htm.

[165] P. Stephenson. Modeling of post-incident root cause analysis. *International Journal of Digital Evidence*, 2, 2003.

[166] Trusted Computing Group. Trusted Platform Module specification version 1.2, July 2007. Online at https://www.trustedcomputinggroup.org/specs/TPM/.

[167] N. Ulaby. Sony music CDs under fire from privacy advocates. NPR News interview, Nov. 2005. Online at http://www.npr.org/templates/story/story.php?storyId=4989260.

[168] United States Election Assistance Commission. Voluntary voting systems guidelines, 2005. Online at http://www.eac.gov/vvsg_intro.htm.

[169] US Congress. Help America Vote Act of 2002 (HAVA). Public Law No. 107-252, 116 Stat. 1666.

[170] US Department of Veterans Affairs, National Center for Patient Safety. Root Cause Analysis (RCA). Online at http://www.va.gov/NCPS/rca.html.

[171] H. R. Varian. Managing online security risks. *The New York Times*, June 2000.

[172] T. Vidas. The acquisition and analysis of random access memory. *Journal of Digital Forensic Practice*, 1:315–323, Dec. 2006.

[173] P. Vixie. DNS and BIND security issues. In *Proc. 5th USENIX Security Symposium*, 1995.

[174] D. Wagner, D. Jefferson, and M. Bishop. Security analysis of the Diebold AccuBasic interpreter, Feb. 2006. Online at http://www.ss.ca.gov/elections/voting_systems/security_analysis_of_the_diebold_accubasic_interpreter.pdf.

[175] A. Walters and N. L. P. Jr. FATKit: The forensic analysis toolkit. Online at http://www.4tphi.net/fatkit/.

[176] B. Warner. 'Copy-proof' CDs cracked with 99-cent marker pen. Reuters; May 24, 2002.

[177] R.-P. Weinmann and J. Appelbaum. Unlocking FileVault. Presentation, 23rd Chaos Communication Congress, Dec. 2006. Online at http://events.ccc.de/congress/2006/Fahrplan/events/1642.en.html.

[178] R.-P. Weinmann and J. Appelbaum. VileFault, Jan. 2008. Online at http://vilefault.googlecode.com/.

[179] A. Wiethoff. Exact Audio Copy. Online at http://www.exactaudiocopy.de/.

[180] B. J. Williams. Security in the Georgia voting system, Apr. 2003. Online at http://macht.arts.cornell.edu/wrm1/gov317/diebold/georgia.pdf.

[181] R. Wood and R. Sweginnis. *Aircraft Accident Investigation*. Endeavor Books, 2nd edition, 2006.

[182] A. W. Wu, A. K. M. Lipshutz, and P. J. Pronovost. Effectiveness and efficiency of root cause analysis in medicine. *J. Am. Med. Assoc.*, 299(6):685–687, 2008.

[183] P. Wyns and R. L. Anderson. Low-temperature operation of silicon dynamic random-access memories. *IEEE Transactions on Electron Devices*, 36:1423–1428, Aug. 1989.