Lest We Remember: Cold Boot Attacks on Encryption Keys

J. Alex Halderman[†], Seth D. Schoen[‡], Nadia Heninger[†], William Clarkson[†], William Paul[§],
Joseph A. Calandrino[†], Ariel J. Feldman[†], Jacob Appelbaum, and Edward W. Felten[†]

[†] Princeton University
[‡] Electronic Frontier Foundation
[§] Wind River Systems

February 21, 2008[*]

Abstract

Contrary to popular assumption, DRAMs used in most modern computers retain their contents for seconds to minutes after power is lost, even at room temperature and even if removed from a motherboard. Although DRAMs become less reliable when they are not refreshed, they are not immediately erased, and their contents persist sufficiently for malicious (or forensic) acquisition of usable full-system memory images. We show that this phenomenon limits the ability of an operating system to protect cryptographic key material from an attacker with physical access. We use cold reboots to mount successful attacks on popular disk encryption systems using no special devices or materials. We experimentally characterize the extent and predictability of memory remanence and report that remanence times can be increased dramatically with simple techniques. We offer new algorithms for finding cryptographic keys in memory images and for correcting errors caused by bit decay. Though we discuss several strategies for partially mitigating these risks, we know of no simple remedy that would eliminate them.

1 Introduction

Most experts assume 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. Ordinary DRAMs typically lose their contents gradually over a period of seconds, even at standard operating temperatures and even if the chips are removed from the motherboard, 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 effects to recover cryptographic keys held in memory. They pose a particular threat to laptop users who rely on disk encryption products, since 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. We demonstrate this risk by defeating several popular disk encryption systems, including BitLocker, TrueCrypt, and FileVault, and we expect many similar products are also vulnerable.

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. Many other security systems are probably

[*] Revision 2 (This version includes substantial updates to our conference submission.)

Contact: {jhalderm, nadiah, wclarkso, jcalandr, ajfeldma, felten}@cs.princeton.edu, schoen@eff.org, wpaul@windriver.com, jacob@appelbaum.net

vulnerable. For example, we found that Mac OS X leaves the user's login passwords in memory, where we were able to recover it, and we have constructed attacks for extracting RSA private keys from Apache web servers.

As we discuss in Section 2, certain segments of the computer security and semiconductor physics communities have been conscious of DRAM remanence effects for some time, though strikingly little about them has been published. As a result, many who design, deploy, or rely on secure systems are unaware of these phenomena or the ease with which they can be exploited. To our knowledge, ours is the first comprehensive study of their security consequences.

1.1 Highlights and roadmap

In Section 3, we describe experiments that we conducted to characterize DRAM remanence in a variety of memory technologies. Contrary to the expectation that DRAM loses its state quickly if it is not regularly refreshed, we found that most DRAM modules retained much of their state without refresh, and even without power, for periods lasting thousands of refresh intervals. At normal operating temperatures, we generally saw a low rate of bit corruption for several seconds, followed by a period of rapid decay. Newer memory technologies, which use higher circuit densities, appeared to decay more quickly than older ones. In most cases, we observed that almost all bits decayed at predictable times and to predictable "ground states" rather than to random values.

We also confirmed that decay rates vary dramatically with temperature. We obtained surface temperatures of approximately $-50\,^{\circ}\mathrm{C}$ with a simple cooling technique: discharging inverted cans of "canned air" duster spray directly onto the chips. At these temperatures, we typically found that fewer than 1% of bits decayed even after 10 minutes without power. To test the limits of this effect, we submerged DRAM modules in liquid nitrogen (ca. $-196\,^{\circ}\mathrm{C}$) and saw decay of only 0.17% after 60 minutes out of the computer.

In Section 4, we present several attacks that exploit DRAM remanence to acquire memory images from which keys and other sensitive data can be extracted. 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 retained memory. A more advanced attack briefly cuts power to the machine, then restores power and boots a custom kernel; this deprives the operating system of any opportunity to scrub memory before shutting down. An even stronger attack cuts the power and then transplants the DRAM modules to a second PC prepared by the attacker, which extracts their state. This attack additionally deprives the original BIOS and PC hardware of any chance to clear the memory on boot. We have implemented imaging kernels for use with network booting or a USB drive.

If the attacker is forced to cut power to the memory for too long, the data will become corrupted. We propose three 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 reduces the error rate. The second is to apply algorithms we have developed for correcting errors in private and symmetric keys. The third is to replicate the physical conditions under which the data was recovered and experimentally measure the decay properties of each memory location; with this information, the attacker can conduct an accelerated error correction procedure. These techniques can be used alone or in combination.

In Section 5, we describe novel error correction algorithms that can reconstruct correct cryptographic keys even with relatively high bit-error rates. Rather than attacking the key directly, our methods focus on values derived from it, such as key schedules, that have a higher degree of redundancy. For performance reasons, many applications precompute these values and keep them in memory for as long as the key itself is in use. To reconstruct an AES key, we treat the decayed key schedule as an error correcting code and find the most likely values for the original key. Applying this method to keys with 10% of bits decayed, we can reconstruct nearly any 128-bit AES key within a few seconds. We have devised reconstruction techniques for AES, DES, and RSA keys, and we expect that similar approaches will be possible for other cryptosystems; we have correctly recovered keys from several popular disk encryption products. The vulnerability of precomputation products to such attacks suggests an interesting trade-off between efficiency and security.

Standard doctrine says that software should overwrite all copies of a key when it has finished using it, but there are important cases where this is impractical because the same key is used repeatedly. Two examples are an encrypted disk, where the root key must remain available to enable file access, and an SSL Web server, where an RSA private key must be kept available for establishing new sessions (if hardware offloading is not used). In Section 6, we present fully automatic techniques for identifying such keys from memory images, even in the presence of bit errors.

We demonstrate the effectiveness of these attacks in Section 7 by attacking several widely used disk encryption products, including BitLocker, TrueCrypt, and FileVault. We have developed a fully automated demonstration attack against BitLocker that allows access to the contents of the disk with only a few minutes of computation. Notably, using BitLocker with a Trusted Platform Module (TPM) sometimes makes it *less* secure, allowing an attacker to gain access to the data even if the machine is stolen while it is completely powered off.

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. In Section 8, we suggest a variety of mitigation strategies ranging from methods that average users can apply today to long-term software and hardware changes. Each remedy has limitations and trade-offs. As we conclude in Section 9, it seems there is no simple fix for DRAM remanence vulnerabilities.

2 Previous Work

Previous researchers have suggested that data in DRAM might survive reboots, and that this fact might have security implications, but to our knowledge, ours is the first security study to focus on this phenomenon, the first to consider how to reconstruct symmetric keys in the presence of errors, the first to apply such attacks to real disk encryption systems, and the first to offer a systematic discussion of countermeasures.

We owe the suggestion that modern DRAM contents can survive cold boot to Pettersson [34], who seems to have obtained it from Chow, Pfaff, Garfinkel, and Rosenblum [13]. 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 [31] 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.

It has been known since 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 [29], a DRAM showed no data loss for a full week without refresh when cooled with liquid nitrogen. Anderson [3] briefly discusses the issue 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 [39], who found significant data retention times with *static* RAMs at room temperature. Our results for modern DRAMs show even longer retention.

Anderson's main focus is on "burn-in" effects that occur when data is stored in RAM for an extended period. Gutmann [23, 24] 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, and we believe they do not result from the kind of physical changes that Gutmann described, but rather from the capacitance of DRAM cells.

Other methods for obtaining memory images from live systems include using privileged software running under the host operating system [42, 43], or using DMA transfer on an external bus [19], such as PCI [12], mini-PCI, Firewire [9, 16, 17], 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.

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 [33].

3 Characterizing Remanence Effects

A DRAM cell is essentially a capacitor. Each cell encodes a single bit by either charging or not charging one of the capacitor's conductors. The other conductor is hard-wired either to power or to ground, depending on the cell's address within the chip [37, 24].

Over time, charge will leak out of the capacitor, and the cell will lose its state or, more precisely, it will decay to its *ground state*, either zero or one depending on whether the fixed conductor of the capacitor is hard-wired to ground or power. To forestall this decay, the cell must be *refreshed*, meaning that the capacitor must be re-charged to hold its value. Specifications for DRAM chips give a *refresh time*, which is the maximum interval that is supposed to pass before a cell is refreshed. The standard refresh time (on the order of milliseconds for contemporary devices) is meant to achieve extremely high reliability for normal computer operations where even infrequent bit errors could cause serious problems; however, 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 the table below. These systems included models from several manufacturers and ranged in age from 9 years to 6 months.

	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

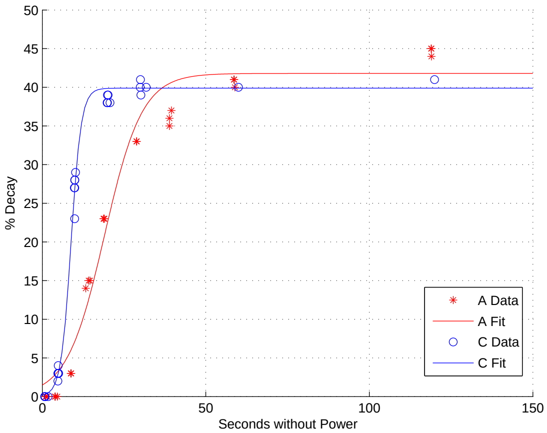

Figure 1: Machines A and C

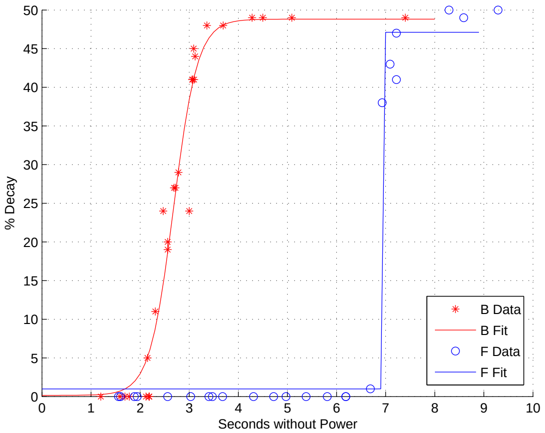

Figure 2: Machines B and F

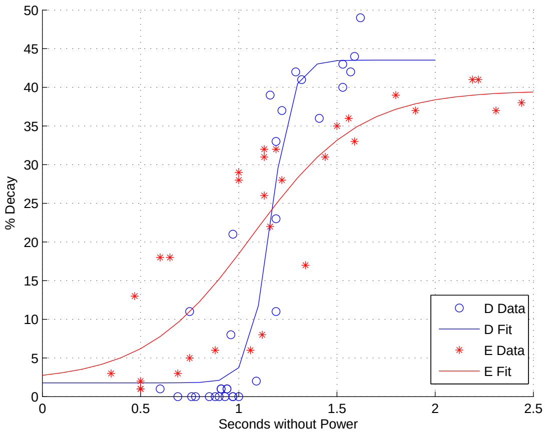

Figure 3: Machines D and E

	Seconds w/o power	Error % at operating temp.	Error % at −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

Figure 4: Effect of cooling on error rates

3.1 Decay at normal operating temperature

Using a modified version of our PXE memory imaging program (see Section 4.1), we filled representative memory regions with a pseudorandom pattern. We read back these memory regions after varying periods of time without refresh and under different temperature conditions, and measured the error rate of each sample. The error rate is the number of bit errors in each sample (the Hamming distance from the pattern we had written) divided by the total number of bits we measured. Since our pseudorandom test pattern contained roughly equal numbers of zeros and ones, we would expect fully decayed memory to have an error rate of approximately 50%.

Our first tests measured the decay rate of each memory module under normal operating temperature, which ranged from 25.5 °C to 44.1 °C, depending on the machine (see Figures 1, 2, and 3). We found that 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 an average of 35 seconds. However, the decay curves all display a similar shape, with an initial period of slow decay, followed by an intermediate period of rapid decay, and then a final period of slow decay.

We calculated best fit curves to the data using the logistic function because MOSFETs, the basic components of a DRAM cell, exhibit a logistic decay curve. We found that machines using newer memory technologies tend to exhibit a shorter time to total decay than machines using older memory technologies, but even the shorter times are long enough to facilitate most of our attacks. We ascribe this trend to the increasing density of the DRAM cells as the technology improves; in general, memory with higher densities have a shorter window where data is recoverable. (While this trend might make DRAM retention attacks more difficult in the future, manufacturers also generally seek to *increase* retention times, because DRAMs with long retention require less frequent refresh and have lower power consumption.)

3.2 Decay at reduced temperature

It has long been known that low temperatures can significantly increase memory devices' retention times [29, 3, 46, 24, 40, 39]. We performed a second series of tests to measure this effect.

In each trial, we loaded a pseudorandom test pattern into memory, and, with the computer running, cooled the memory module to approximately $-50\,^{\circ}\mathrm{C}$. We then powered off the machine and maintained this temperature until power was restored. We achieved these temperatures using commonly available "canned air" duster products (see Section 4.2), which we discharged, with the can inverted, directly onto the chips.[1]

As expected, we observed a significantly lower rate of decay under these reduced temperatures (see Figure 4). On all of our sample DRAMs, the decay rates were low enough that an attacker who cut power for 60 seconds would recover 99.9% of bits correctly.

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 $-50\,^{\circ}\mathrm{C}$ 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.

3.3 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 5 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 5, 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 in different 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 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.

[1] We thank Gregory Sutter for this straightforward and effective cooling technique.

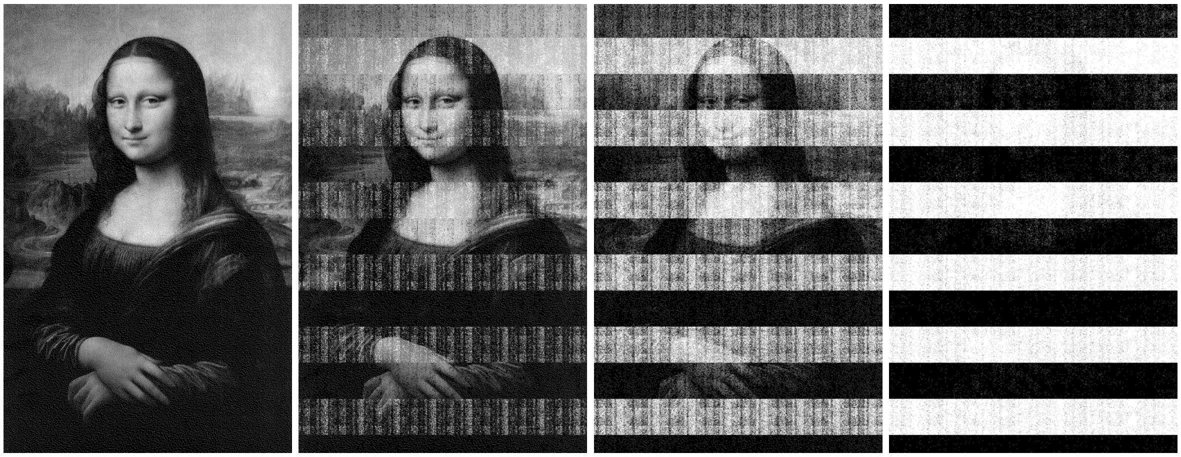

Figure 5: We loaded a bitmap image into memory on Machine A, then cut power for varying lengths of time. After 5 seconds (left), the image is indistinguishable from the original. It gradually becomes more degraded, as shown after 30 seconds, 60 seconds, and 5 minutes.

To visualize this effect, we captured degraded memory images, including those shown in Figure 5, after cutting power for intervals ranging from 1 second to 5 minutes, in 1 second increments. We combined the results into a video (available on our web site). 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.

3.4 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 [7], 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 Imaging Residual Memory

Imaging residual memory contents requires no special equipment. When the system boots, the memory controller begins 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 whatever data is present using normal memory-access instructions.

4.1 Imaging tools

One challenge is that booting the system will necessarily overwrite some portions of memory. Loading a full operating system would be very destructive. Instead, our approach is to use tiny special-purpose programs that, when booted from either a warm or cold reset state, produce accurate dumps of memory contents to some external medium. These programs use only trivial amounts of RAM, and their memory offsets used can be adjusted to some extent to ensure that data structures of interest are unaffected.

Our memory-imaging tools can 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) [15], 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 one minute 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[2]) plug-in for the SYSLINUX bootloader [4] 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 memory extraction speeds up to 136 Mb/s, and we expect it will be possible to increase this throughput with further optimizations.

4.2 Imaging attacks

An attacker could use imaging tools like ours in a number of ways, depending on his level of access to the system and the countermeasures employed by hardware and software.

- **Simple reboots** The simplest attack is to reboot the machine and configure the BIOS to boot the imaging tool. A warm boot, invoked with the operating system's restart procedure, will normally ensure that the memory has no chance to decay, though software will have an opportunity to wipe sensitive data prior to shutdown. A cold boot, initiated using the system's restart switch or by briefly removing and restoring power, will result in little or no decay depending on the memory's retention time. Restarting the system in this way denies the operating system and applications any chance to scrub memory before shutting down.

[2] SYSLINUX itself is about 12 KB, and around 64 KB is occupied by a buffer for disk I/O.

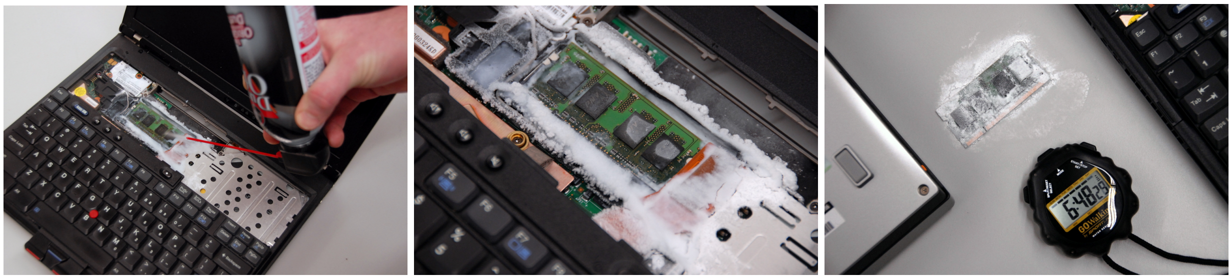

Figure 6: Before powering off the computer, we spray an upside-down canister of multipurpose duster directly onto the memory chips, cooling them to $-50\,^{\circ}\mathrm{C}$. At this temperature, the data will persist for several minutes after power loss with minimal error, even if we remove the DIMM from the computer.

- **Transferring DRAM modules** Even if an attacker cannot force a target system to boot memory-imaging tools, or if the target employs countermeasures that erase memory contents during boot, DIMM modules can be physically removed and their contents imaged using another computer selected by the attacker.

  Some memory modules exhibit far faster decay than others, but as we discuss in Section 3.2 above, cooling a module before powering it off can slow decay sufficiently to allow it to be transferred to another machine with minimal decay. 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 6, 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.

  Removing the memory modules can also allow the attacker to image memory in address regions where standards BIOSes load their own code during boot. The attacker could remove the primary memory module from the target machine and place it into the secondary DIMM slot (in the same machine or another machine), effectively remapping the data to be imaged into a different part of the address space.

- **Remote attacks** Machines that are connected to Internet-accessible networks and that attempt a network boot by default may be vulnerable to *remote* memory-dumping attacks under certain circumstances. Suppose that one computer on a network segment (for example, at a colocation facility) has been remotely compromised and configured to act as a PXE boot server. If a software flaw or denial-of-service attack can cause other machines on the same network segment to be rebooted, these target machines might boot a PXE memory dumper and reveal their memory contents to the compromised machine. Alternatively, this attack may be prepared in advance of software updates that are legitimately applied by the system operator and that require a reboot. Similar attacks have been possible for as long as network-boot capabilities have existed, but system administrators may have mistakenly believed that the use of disk encryption was sufficient to prevent an attacker from obtaining useful data. As a countermeasure, authentication could be added as an extension to the PXE and EFI network boot standards [32].

5 Key Reconstruction

Our experiments (see Section 3) show that it is possible to recover memory contents with few bit errors even after cutting power to the system for a brief time, but the presence of even a small amount of error complicates the process of extracting correct cryptographic keys. In this section we present algorithms for correcting errors in symmetric and private keys. These algorithms can correct most errors quickly and uniquely even in the presence of relatively high bit error probabilities in the range of 5% to 50%, depending on the type of key.

A naïve approach to key error correction is to brute-force search over keys with a low Hamming distance from the decayed key that was retrieved from memory, but this is computationally burdensome even with a moderate amount of unidirectional error. As an example, if only 10% of the ones have decayed to zeros in our memory image, the data recovered from a 256-bit key with an equal number of ones and zeroes has an expected Hamming distance of 12 from the actual key, and the number of such keys is $\binom{128+12}{12} > 2^{56}$.

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 7) 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've 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 induced 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 experiments). The ground state decay probability can be approximated from recovered key data by counting the fraction of 1s and 0s, assuming that the original key data 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 exactly replicate the conditions of the memory extraction, but note that 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.

5.1 Reconstructing DES keys

We first apply these methods to develop an error correction technique for DES. The DES key schedule algorithm produces 16 subkeys, each a permutation of a 48-bit subset of bits from the original 56-bit key. Every bit from the original key appears in about 14 of the 16 subkeys.

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. If $\delta_0 = \delta_1 < \frac{1}{2}$, the optimal decoding of such an n-bit codeword is 0 if more than $n/2$ of the recovered bits are 0, and 1 otherwise. For $\delta_0 \neq \delta_1$, the optimal decoding is 0 if more than nr of the recovered bits are 0 and 1 otherwise, where

$$r = \frac{\log(1-\delta_0) - \log\delta_1}{\log(1-\delta_0) + \log(1-\delta_1) - \log\delta_1 - \log\delta_0}$$

.

For $\delta_0 = .1$ and $\delta_1 = .001$ (that is, we are in a block with ground state 0), $r = .25$ and this approach will fail to correctly decode a bit only if more than 3 of the 14 copies of a 0 decay to a 1, or more than 11 of the 14 copies of a 1 decay to 0. The probability of this event is less than 10^{-9}. Applying the union bound, the probability that any of the 56 key bits will be incorrectly decoded is at most $56 \times 10^{-9} < 6 \times 10^{-8}$; even at 50% error, the probability that the key can be correctly decoded without having to brute force any bits at all is more than 98%.

This technique can be trivially extended to correct errors in Triple DES keys. Since Triple DES applies the same key schedule algorithm to two or three 56-bit key components, the probability of correctly decoding each key bit is the same as for regular DES. With a decay rate of $\delta_0 = .5$ and probability $\delta_1 = .001$ of bit flips in the opposite direction, we can correctly decode a 112-bit Triple DES key with at 97% probability and a 168-bit key with at least 96% probability.

5.2 Reconstructing AES keys

The AES key schedule is generated through a more complex mechanism than the DES key schedule, but we can still use it to efficiently reconstruct a key in the presence of errors.

In principle we could search all 2^{128} valid key schedules and find the one that best matches the observed contents of memory given the decay direction; but this is clearly infeasible. A slightly better approach would be to search keys in order of distance to the recovered key and output any key whose schedule is sufficiently close to the recovered schedule. Our implementation of this algorithm took twenty minutes to search 10^9 candidate keys in order to reconstruct a key in which 7 zeros had flipped to ones. At this rate it would take ten days to reconstruct a key with 11 bits flipped.

We can do significantly better by taking advantage of the structure of the AES key schedule. Instead of trying to decode an entire key schedule at once, we can examine a smaller set of bytes at a time. The high amount of linearity in the key schedule is what permits this separability–we can take advantage of pieces that are small enough to brute force optimal decodings for, and still large enough that these decodings are useful to reconstruct the overall key. Once we have a list of possible decodings for these smaller pieces of the key in order of likelihood, we can combine them into an entire key to check against the key schedule.

Each of the decoding steps is quite fast, so the running time is ultimately limited by the number of combinations we need to check. The number of combinations is still roughly exponential in the number of errors, but it is a vast improvement over brute force searching and is practical in many realistic cases.

Overview of the algorithm For 128-bit keys, an AES key expansion consists of 11 four-word (128-byte) round keys. The first round key is equal to the key itself. Each remaining word of the key schedule is generated either by XORing two words of the key schedule together, or by performing the key schedule core (a combination of the RotWord, SubWord, and Rcon operations) on a word of the key schedule and XORing the result with another word of 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. This slice is 7 bytes long, but is uniquely determined by the four bytes from the key. In theory, there are still 2^{32} possibilities to examine for each slice, but we can do quite well by examining them in order of distance to the recovered key. For each possible set of 4 key bytes, we generate the relevant three bytes of the next round key and calculate the probability, given estimates of δ_0 and δ_1, that these seven bytes might have decayed to the corresponding bytes of the recovered round keys.

Now we proceed to guess candidate keys, where a candidate consists of a value for each slice of bytes. We consider the candidates in order of decreasing total likelihood as calculated above. For each candidate key we consider, we calculate the expanded key schedule and ask if the likelihood of that expanded key schedule decaying to our recovered key schedule is sufficiently high. If so, then we output the corresponding key as a guess.

When one of δ_0 or δ_1 is very small, this algorithm will almost certainly output a unique guess for the key. To see this, observe that a single bit flipped 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.

Our implementation of this algorithm is able to reconstruct keys with 15% error (that is, $p_0 = .15$ and $p_1 = .001$) in a fraction of a second, and about half of keys with 30% error within 30 seconds.

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.

5.3 Reconstructing tweak keys

The same methods can be applied to reconstruct keys for tweakable encryption modes [30], 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 $GF(2^{128})$ [25]. 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. 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 [36] and XTS [2] 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.

5.4 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, optionally, the prime factors p and q of N, $d \mod (p-1)$, $d \mod (q-1)$, and $q^{-1} \mod p$. Given N and e, any of the private values is sufficient to generate the others using the Chinese remainder theorem and greatest common divisor algorithms. In practice, RSA implementations store some or all of the optional private values to speed up computation.

There have been a number of results on efficiently reconstructing RSA private keys given a fraction of the bits of private key data. Let $n = \lg N$. It is known that N can be factored in polynomial time given the $n/4$ least significant bits of p (Coppersmith [14]), given the $n/4$ least significant bits of d (Boneh, Durfee, and Frankel [10]), or given the $n/4$ least significant bits of $d \mod (p-1)$ (Blömer and May [8]).

These previous results are all based on Coppersmith's method of finding bounded solutions to polynomial equations using lattice basis reduction; the number of contiguous bits recovered from the most or least significant bits of the private key data determines the additive error tolerated in the solution. In our case, however, the errors may be distributed across all bits of the key data, so we are instead searching for solutions with low Hamming weight, and these previous approaches do not seem to be directly applicable.

Given the public modulus N and the values $\tilde{p}$ and $\tilde{q}$ recovered from memory, we can deduce values for the original p and q by iteratively reconstructing them from the least-significant bits. For unidirectional decay with probability δ, bits p_i and q_i are uniquely determined by N_i and our guesses for the $i - 1$ lower-order bits of p and q (observe that $p_0 = q_0 = 1$) except in the case when $\tilde{p_i}$ and $\tilde{q_i}$ are both in the ground state. This yields a branching process with expected degree $\frac{(3+\delta)^2}{8}$. If decay is not unidirectional, we may follow a similar approach, and use the estimated probabilities to weight the branches at each bit.

Combined with a few heuristics—for example, choose the most likely state first and prune nodes that contradict bounds on the solution—this results in a practical algorithm for reasonable error rates. We expect that this process can be improved substantially using additional data recovered from the private key. For example, given $\tilde{d_p}$ recovered from $d \mod (p-1)$, we know that $ed_p = 1 + h(p-1)$ where $h < e$ and similarly for d_q, so for small values of e (in practice 65537) we can further prune the trees for p_i and q_i based on the feasible solutions to these equations.

We tested a simple implementation of the algorithm on a fast modern machine. For fifty trials with 1024-bit primes and $\delta = 4\%$, the median number of nodes visited was 16,499, the mean was 621,707, and the standard deviation was 2,136,870. The median reconstruction time was 4.492 seconds. For $\delta = 6\%$, the median number of nodes visited was 227,763 and median reconstruction time was 2.582 minutes.

For 512-bit primes and a $\delta = 10\%$, the median number of nodes visited was 188,702, and the median reconstruction time was 1.181 minutes.

For larger error rates, we can attempt to reconstruct only the first $n/4$ bits of the key using this process and use the lattice techniques to reconstruct the rest of the key; these computations generally take several hours in practice. For a 1024-bit RSA key, we would need to recover 256 bits of a factor. The expected depth of the tree from our branching reconstruction process would be $(\frac{1}{2}+\delta)^2 256$ (assuming an even distribution of 0s and 1s) and the expected fraction of branches that would need to be examined is $1/2+\delta^2$.

6 Identifying Keys in Memory

Extracting encryption keys from memory images requires a mechanism for locating the target keys. A simple approach is to test every sequence of bytes to see whether it correctly decrypts some known plaintext. Applying this method to a 1 GB memory image known to contain a 128-bit symmetric key aligned to some 4-byte machine word implies at most 2^{28} possible key values. However, this is only the case if the memory image is perfectly accurate. If there are bit errors in the portion of memory containing the key, the search quickly becomes intractable.

We have developed fully automatic techniques for locating symmetric encryption keys in memory images, even in the presence of bit errors. Our approach is similar to the one we used to correct key bit errors in Section 5. We target the key schedule instead of the key itself, searching for blocks of memory that satisfy (or are close to satisfying) the combinatorial properties of a valid key schedule. Using these methods we have been able to recover keys from closed-source encryption programs without having to disassemble them and reconstruct their key data structures, and we have even recovered partial key schedules that had been overwritten by another program when the memory was reallocated.

Although previous approaches to key recovery do not require a scheduled key to be present in memory, they have other practical drawbacks that limit their usefulness for our purposes. Shamir and van Someren [38] propose visual and statistical tests of randomness which can quickly identify regions of memory that might contain key material, but their methods are prone to false positives that complicate testing on decayed memory images. Even perfect copies of memory often contain large blocks of random-looking data that might pass these tests (e.g., compressed files). Pettersson [34] suggests a plausibility test for locating a particular program data structure that contains key material based on the range of likely values for each field. This approach requires the operator to manually derive search heuristics for each particular encryption application, and it is not very robust to memory errors.

6.1 Identifying AES keys

In order to identify scheduled AES keys in a memory image, we propose the following algorithm:

1. Iterate through each byte of memory. Treat the following block of 176 or 240 bytes of memory as an AES key schedule.

2. For each word in the potential key schedule, calculate the Hamming distance from that word to the key schedule word that should have been generated from the surrounding words.

3. If the total number of bits violating the constraints on a correct AES key schedule is sufficiently small, output the key.

We created an application called `keyfind` that implements this algorithm for 128- and 256-bit AES keys. The program takes a memory image as input and outputs a list of likely keys. It assumes that key schedules are contained in contiguous regions of memory and in the byte order used in the AES specification [1]; this can be adjusted to target particular cipher implementations. A threshold parameter controls how many bit errors will be tolerated in candidate key schedules. We apply some simple heuristics, such as ignoring empty or repetitive blocks of memory, to cut down on false positives.

We expect that this approach can be applied to many other ciphers. For example, to identify DES keys based on their key schedule, calculate the distance from each potential subkey to the permutation of the previous subkey. A similar method works to identify the precomputed multiplication tables used for advanced cipher modes like LRW (see Section 5.3).

6.2 Identifying RSA Keys

Methods proposed for identifying RSA private keys range from the purely algebraic (Shamir and van Someren suggest, for example, multiplying adjacent key-sized blocks of memory [38]) to the *ad hoc* (searching for the RSA Object Identifiers found in ASN.1 key objects [35]). The former ignores the widespread use of standard key formats, and the latter seems insufficiently robust.

The most widely used format for an RSA private key is specified in PKCS #1 [26] as an ASN.1 object of type RSAPrivateKey with the following fields: version, modulus n, publicExponent e, privateExponent d, prime1 p, prime2 q, exponent1 $d \mod (p-1)$, exponent2 $d \mod (q-1)$, coefficient $q^{-1} \mod p$, and optional other information. This object, packaged in DER encoding, is the standard storage and interchange format for private keys.

This format suggests two techniques we might use for identifying RSA keys in memory: we could search for known *contents* of the fields, or we could look for memory that matches the *structure* of the DER encoding. We tested both of these approaches on a computer running Apache 2.2.3 with `mod_ssl`.

One value in the key object that an attacker is likely to know is the public modulus. (In the case of a web server, the attacker can obtain this and the rest of the public key by querying the server.) We tried searching for the modulus in memory and found several matches, all of them instances of the server's public or private key.

We also tested a key finding method described by Ptacek [35] and others: searching for the RSA Object Identifiers that should mark ASN.1 key objects. This technique yielded only false positives on our test system.

Finally, we experimented with a new method, searching for identifying features of the DER-encoding itself. We looked for the sequence identifier (0x30) followed a few bytes later by the DER encoding of the RSA version number and then by the beginning of the DER encoding of the next field (02 01 00 02). This method found several copies of the server's private key, and no false positives. To locate keys in decayed memory images, we can adapt this technique to search for sequences of bytes with low Hamming distance to these markers and check that the subsequent bytes satisfy some heuristic entropy bound.

7 Attacking Encrypted Disks

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 [41] 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 filesystem encryption to obtain the benefit of this safe harbor. Our results below suggest that this faith in the strength of disk encryption may be misplaced. We find that a moderately skilled attacker can circumvent many widely used disk encryption products if a laptop is stolen while it is powered on or suspended. The use of encryption is not, by itself, necessarily an adequate defense, and data in stolen laptops may be compromised even when encryption is used.

We have applied some of the tools developed in this paper to attack popular on-the-fly disk encryption systems. The most time-consuming parts of these tests were generally setting up the encrypted disks and verifying that we had actually found the correct decryption keys. 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.

7.1 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.

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 FUSE 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.

Notably, it is possible to implement this attack without reverse engineering any software. In a paper released by Microsoft, Ferguson [20] describes BitLocker in enough detail both to discover the roles of the sector pad key and the CBC encryption key and to program an independent implementation of the BitLocker sector encryption algorithm.

BitLocker differs from other disk encryption products mainly 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 [20], is particularly vulnerable to our attack, because it allows the disk encryption keys to be extracted with our attacks even if the computer is powered off for a long time. When the machine boots, the keys will be loaded into RAM automatically (before the login screen) without the entry of any secrets.

It appears that Microsoft is aware of this problem [31] 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).

7.2 FileVault

Apple's FileVault disk encryption software has been examined and reverse-engineered in some detail [44]. 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). We verified that the IV key is also present in our recovered memory. Assuming no bits in the IV key decay, an attacker could identify it by testing all 160-bit substrings of the memory image 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` [45].

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.

7.3 TrueCrypt

TrueCrypt is a popular open-source disk encryption product for 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 the current version, version 5, they use XTS mode (see Section 5.3). 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 5.3.)

7.4 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 the raw key to be input.

8 Countermeasures and their Limitations

Memory imaging attacks are difficult to defend against because cryptographic keys that are in active use need to be stored *somewhere*. Our suggested countermeasures focus on discarding or obscuring encryption keys before an adversary might gain physical access, preventing memory-dumping software from being executed on the machine, physically protecting DRAM chips, and possibly making the contents of memory decay more readily.

Scrubbing memory Countermeasures begin with efforts to avoid storing keys in memory. Software should overwrite keys when they are no longer needed, and it should attempt to prevent keys from being paged to disk. Runtime libraries and operating systems should clear memory proactively; Chow *et al.* show that this precaution need not be expensive [13]. Of course, these precautions cannot protect keys that must be kept in memory because they are still in use, such as the keys used by encrypted disks or secure web servers.

Systems can also clear memory at boot time. Some PCs can be configured to clear RAM at startup via a destructive Power-On Self-Test (POST) before they attempt to load an operating system. If the attacker cannot bypass the POST, he cannot image the PC's memory with locally-executing software, though he could still physically move the memory chips to different computer with a more permissive BIOS.

Limiting booting from network or removable media Many of our attacks involve booting a system via the network or from removable media. Computers can be configured to require an administrative password to boot from these sources. We note, however, that even if a system will boot only from the primary hard drive, an attacker could still swap out this drive, or, in many cases, reset the computer's NVRAM to re-enable booting from removable media.

Suspending a system safely Our results show that 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 ("sleeping") is also ineffective, even if the machine enters screen-lock 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 resume normal operations.

With most disk encryption systems, users can protect themselves by powering off the machine completely when it is not in use. (BitLocker in "basic" TPM mode remains vulnerable, since the system will automatically mount the disk when the machine is powered on.) Memory contents may be retained for a short period after power-off, so the owner should keep an eye on the machine for a minute or so after removing power. Though effective, this countermeasure is very inconvenient given the long boot time of current machines.

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. The password must be strong (or strengthened), as an attacker who can extract memory contents can then try an offline password-guessing attack. If encrypting all of memory is too expensive, the system could encrypt only those pages or regions containing important keys. Some existing systems can be configured to suspend safely in this sense, although this is often not the default behavior [6].

Avoiding precomputation Our attacks show that using precomputation to speed cryptographic operations can make keys more vulnerable. Precomputation tends to lead to redundant storage of key information, which helps an attacker reconstruct keys in the presence of bit errors. For example, the round keys in AES and DES are highly redundant and very useful to attackers, as described above in Section 5.

Avoiding precomputation will hurt performance, as potentially expensive computations will have to be repeated. Compromises are possible; for example, precomputed values could be cached for a predetermined period of time and discarded if not re-used within that interval. This approach would accept some vulnerability in exchange for reducing computation, a sensible tradeoff in some situations.

Key expansion Another defense against key reconstruction is to apply some transform to 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.* [11] define the notion of an *exposure-resilient function* whose input remains secret even if all but some small fraction of the output is revealed, and show that the existence of this primitive is equivalent to the existence of one-way functions.

In practice, suppose we have a 256-bit AES key K which is not currently in use but will be needed later. We cannot overwrite the key but we want to make it more resistant to reconstruction attacks. One way to do this is to allocate a large B-bit buffer, fill the buffer with random data R, then store $K \oplus H(R)$ where H is a hash function such as SHA-256.

Now suppose there is a power-cutting attack which causes d of the bits in this buffer to be flipped. Assuming the hash function is strong, the attacker's only hope in finding K is to guess which bits were flipped, of the roughly $B/2$ that could have decayed. If d bits are flipped, the adversary will have to search a space of size $\binom{B/2+d}{d}$ to find the correct value of R and hence the correct key K. If B is large, this search will be prohibitive even when d is relatively small.

In principle, we could store all keys like this, recompute each key K when we need to use it, and delete K immediately after use. Alternatively, we could sometimes keep K in memory, introducing the precomputation tradeoff discussed above.

Physical defenses Some of our attacks rely on physical access to DRAM chips or modules. These attacks can be mitigated by physically protecting the memory. For example, DRAM modules could be locked in place inside the machine, or encased in a material such as epoxy to frustrate attempts to remove or access them. Similarly, the system could respond to low temperatures or opening of the computer's case by attempting to overwrite memory, although these defenses require active sensor systems with their own backup power supply. Many of these techniques are associated with specialized tamper-resistant hardware such as the IBM 4758 coprocessor [18, 40] and could add considerable cost to a PC. However, a small amount of memory soldered to a motherboard could be added at relatively low cost.

Architectural changes Some countermeasures try to change the machine's architecture. This will not help on existing machines, but it might make future machines more secure.

One approach is to find or design DRAM systems that lose their state quickly. This might be difficult, given the tension between the desire to make memory decay quickly and the desire to keep the probability of decay within a DRAM refresh interval vanishingly small.

Another approach is to add key-store hardware that is guaranteed to erase its state on power-up, reset, and shutdown. This would provide a safe place to put a few keys, though precomputation of derived keys would still pose a risk.

Others have proposed architectures that would routinely encrypt the contents of memory for security purposes [28, 27]. These would apparently prevent the attacks we describe, as long as the encryption keys were destroyed on reset or power loss.

Trusted Computing Trusted Computing hardware, in the form of Trusted Platform Modules (TPMs) [22] is now deployed in some personal computers. Though useful against some attacks, today's Trusted Computing hardware does not appear to prevent the attacks we describe here.

Deployed TCG TPMs do not implement bulk encryption. Instead, they monitor boot history in order to decide (or help other machines decide) whether it is safe to store a key in RAM. If a software module wants to use 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 [31]. However, once the key is stored in RAM, it is subject to our attacks. 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.

9 Conclusions

Contrary to popular belief, DRAMs hold their values for surprisingly long intervals without power or refresh. Our experiments show that this fact enables a variety of security attacks that can extract sensitive information such as cryptographic keys from memory, despite the operating system's best efforts to protect memory contents. The attacks we describe are practical—for example, we have used them to defeat several popular disk encryption systems.

Other types of software may be similarly vulnerable. DRM systems often rely on symmetric keys stored in memory, which may be recoverable using the techniques outlined in our paper. As we have shown, SSL-enabled web servers are vulnerable, since they often keep in memory private keys needed to establish SSL sessions. Furthermore, methods similar to our key-finder would likely be effective for locating passwords, account numbers, or other sensitive data in memory.

There seems to be no easy remedy for these vulnerabilities. Simple software changes are likely to be ineffective; hardware changes are possible but will require time and expense; and today's Trusted Computing technologies appear to be of little help because they cannot protect keys that are already in memory. The risk seems highest for laptops, which are often taken out in public in states that are vulnerable to our attacks. These risks imply that disk encryption on laptops may do less good than widely believed.

Ultimately, it might become necessary to treat DRAM as untrusted, and to avoid storing sensitive confidential data there, but this will not be feasible until architectures are changed to give software a safe place to keep its keys.

Acknowledgments

We thank Andrew Appel, Jesse Burns, Grey David, Laura Felten, Christian Fromme, Dan Good, Benjamin Mako Hill, David Hulton, Brie Ilenda, Scott Karlin, David Molnar, Tim Newsham, Chris Palmer, Audrey Penven, David Robinson, Kragen Sitaker, N.J.A. Sloane, Gregory Sutter, Sam Taylor, Ralf-Philipp Weinmann, and Bill Zeller for their helpful comments, suggestions, and contributions. Without them, this paper would not have been possible.

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

Calandrino performed this research while under appointment to the Department of Homeland Security (DHS) Scholarship and Fellowship Program under DOE contract number DE-AC05-06OR23100. All opinions expressed in this paper are the authors' and do not necessarily reflect the policies and views of DHS or DOE.

References

[1] Advanced Encryption Standard. National Institute of Standards and Technology, FIPS-197, November 2001.

[2] IEEE P1619/D19 Draft standard for cryptographic protection of data on block-oriented storage devices. IEEE 1619 Security in Storage Working Group, July 2007.

[3] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*, page 282. Wiley, first edition, January 2001.

[4] H. Peter Anvin. SYSLINUX. http://syslinux.zytor.com/.

[5] William Arbaugh, David Farber, and Jonathan Smith. A secure and reliable bootstrap architecture. In *Proc. IEEE Symp. on Security and Privacy*, pages 65–71, May 1997.

[6] Alon Bar-Lev. Linux, Loop-AES and optional smartcard based disk encryption. http://wiki.tuxonice.net/EncryptedSwapAndRoot, November 2007.

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

[8] Johannes Blömer and Alexander May. New partial key exposure attacks on RSA. In *Proc. CRYPTO 2003*, pages 27–43, 2003.

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

[10] Dan Boneh, Glenn Durfee, and Yair Frankel. An attack on RSA given a small fraction of the private key bits. In *Advances in Cryptology – ASIACRYPT '98*, pages 25–34, 1998.

[11] Ran Canetti, Yevgeniy Dodis, Shai Halevi, Eyal Kushilevitz, and Amit Sahai. Exposure-resilient functions and all-or-nothing transforms. In *Advances in Cryptology – EUROCRYPT 2000*, volume 1807/2000, pages 453–469, 2000.

[12] Brian D. Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1:50–60, December 2003.

[13] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. 14th USENIX Security Symposium*, pages 331–346, August 2005.

[14] Don Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *J. Cryptology*, 10(4):233–260, 1997.

[15] Intel Corporation. Preboot Execution Enviroment (PXE) specification version 2.1, September 1999.

[16] Maximillian Dornseif. 0wned by an iPod. Presentation, PacSec, 2004.

[17] Maximillian Dornseif. Firewire – all your memory are belong to us. Presentation, CanSecWest/core05, May 2005.

[18] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart. Building the IBM 4758 secure coprocessor. *Computer*, 34:57–66, October 2001.

[19] Knut Eckstein and Maximillian 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, April 2005.

[20] Niels Ferguson. AES-CBC + Elephant diffuser: A disk encryption algorithm for Windows Vista. http://www.microsoft.com/downloads/details.aspx?FamilyID=131dae03-39ae-48be-a8d6-8b0034c92555, August 2006.

[21] S. Govindavajhala and A. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, 2003.

[22] Trusted Computing Group. Trusted Platform Module specification version 1.2, July 2007.

[23] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proc. 6th USENIX Security Symposium*, pages 77–90, July 1996.

[24] Peter Gutmann. Data remanence in semiconductor devices. In *Proc. 10th USENIX Security Symposium*, pages 39–54, August 2001.

[25] Clement Kent. Draft proposal for tweakable narrow-block encryption. https://siswg.net/docs/LRW-AES-10-19-2004.pdf, 2004.

[26] RSA Laboratories. PKCS #1 v2.1: RSA cryptography standard. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf.

[27] Ruby B. Lee, Peter C.S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *Proc. Intl. Symposium on Computer Architecture*, pages 2–13, 2005.

[28] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[29] Walter Link and Herbert May. Eigenschaften von MOS-Ein-Transistorspeicherzellen bei tiefen Temperaturen. *Archiv für Elektronik und Übertragungstechnik*, 33:229–235, June 1979.

[30] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In *Advances in Cryptology – CRYPTO 2002*, pages 31–46, 2002.

[31] Douglas MacIver. Penetration testing Windows Vista BitLocker drive encryption. Presentation, Hack In The Box, September 2006.

[32] Wesley G. Miller. U.S. Patent #7313690: Three way validation and authentication of boot files transmitted from server to client, 2007.

[33] National Computer Security Center. A guide to understanding data remanence in automated information systems, September 1991.

[34] Torbjörn Pettersson. Cryptographic key recovery from Linux memory dumps. Presentation, Chaos Communication Camp, August 2007.

[35] Thomas Ptacek. Recover a private key from process memory. http://www.matasano.com/log/178/recover-a-private-key-from-process-memory/.

[36] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In *Advances in Cryptology – ASIACRYPT 2004*, pages 16–31, 2004.

[37] Leif Z. Scheick, Steven M. Guertin, and Gary M. Swift. Analysis of radiation effects on individual DRAM cells. *IEEE Transactions on Nuclear Science*, 47:2534–2538, December 2000.

[38] Adi Shamir and Nicko van Someren. Playing “hide and seek” with stored keys. *Lecture Notes in Computer Science*, 1648:118–124, 1999.

[39] Sergei Skorobogatov. Low-temperature data remanence in static RAM. University of Cambridge Computer Laborary Technical Report No. 536, June 2002.

[40] Sean W. Smith. *Trusted Computing Platforms: Design and Applications*. Springer, first edition, 2005.

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

[42] Timothy Vidas. The acquisition and analysis of random access memory. *Journal of Digital Forensic Practice*, 1:315–323, December 2006.

[43] AAron Walters and Nick L. Petroni Jr. FATKit: The forensic analysis toolkit. http://www.4tphi.net/fatkit/.

[44] Ralf-Philip Weinmann and Jacob Appelbaum. Unlocking FileVault. Presentation, 23rd Chaos Communication Congress, December 2006.

[45] Ralf-Philip Weinmann and Jacob Appelbaum. VileFault. http://vilefault.googlecode.com/, January 2008.

[46] Philippe Wyns and Richard L. Anderson. Low-temperature operation of silicon dynamic random-access memories. *IEEE Transactions on Electron Devices*, 36:1423–1428, August 1989.