

# Lest We Remember: Cold-Boot Attacks on Encryption Keys

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

## Abstract

**Contrary to widespread assumption, dynamic RAM (DRAM), the main memory in most modern computers, retains its contents for several seconds after power is lost, even at room temperature and even if removed from a motherboard. Although DRAM becomes less reliable when it is not refreshed, it is not immediately erased, and its 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 to a machine. It poses a particular threat to laptop users who rely on disk encryption: we demonstrate that it could be used to compromise several popular disk encryption products without the need for any special devices or materials. We experimentally characterize the extent and predictability of memory retention and report that remanence times can be increased dramatically with simple cooling 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 mitigating these risks, we know of no simple remedy that would eliminate them.**

## 1. INTRODUCTION

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. Dynamic RAM (DRAM), the hardware used as the main memory of most modern computers, loses its contents gradually over a period of seconds, even at normal operating temperatures and even if the chips are removed from the motherboard. This phenomenon is called *memory remanence*. Data will persist for minutes or even hours if the chips are kept at low temperatures, and 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. An attacker can exploit this to learn the encryption key and decrypt the disk. 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.

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

The full version of this paper was published in *Proceedings of the 17th USENIX Security Symposium*, August 2008, USENIX Association. The full paper, video demonstrations, and source code are available at <http://citp.princeton.edu/memory/>.

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.

Certain segments of the computer security and hardware communities have been conscious of DRAM remanence for some time, but strikingly little about it 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.

## 2. CHARACTERIZING REMANENCE

A DRAM cell is essentially a capacitor that encodes a single bit when it is charged or discharged.<sup>10</sup> 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 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.

To characterize DRAM decay, we performed experiments on a selection of recent computers, listed in Figure 1. 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

**Figure 1: Test Systems.** We experimented with six systems (designated A–F) that encompass a range of recent DRAM architectures and circuit densities.

	Density	Type	System	Year
A	128MB	SDRAM	Dell Dimension 4100	1999
B	512MB	DDR	Toshiba Portégé R100	2001
C	256MB	DDR	Dell Inspiron 5100	2003
D	512MB	DDR2	IBM Thinkpad T43p	2006
E	512MB	DDR2	IBM Thinkpad x60	2007
F	512MB	DDR2	Lenovo 3000 N100	2007

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.

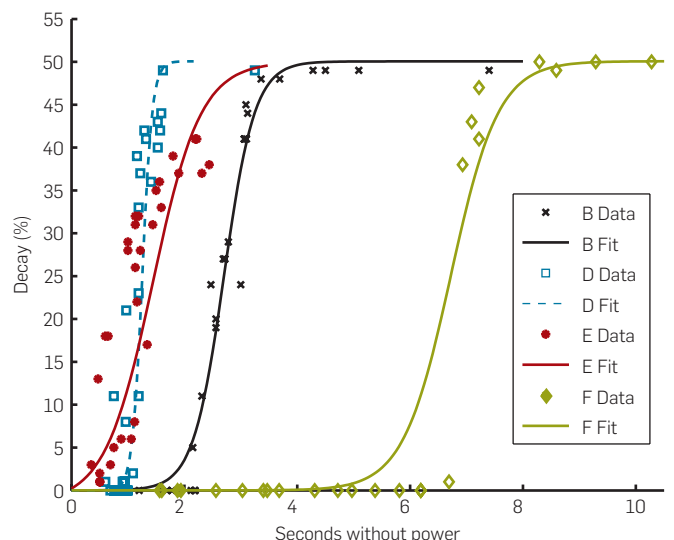
### 2.1. Decay at operating temperature

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 2.

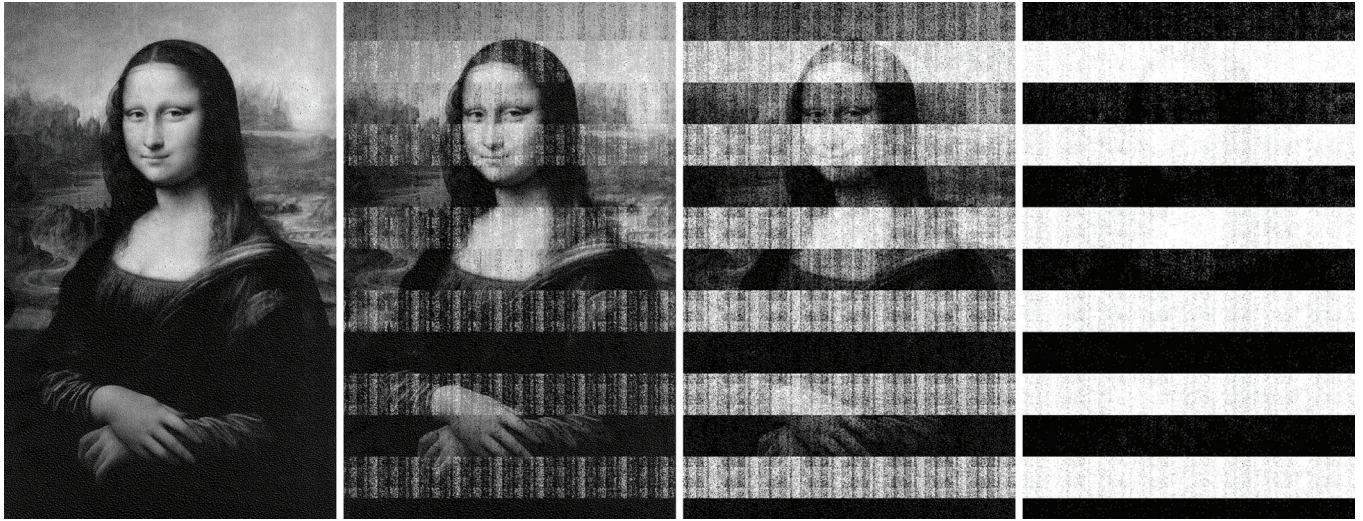
The dimensions of the decay curves varied considerably between machines, with the fastest exhibiting complete data loss in approximately 2.5 s and the slowest taking over a minute. Newer machines tended to exhibit a shorter time to total decay, possibly because newer chips have higher density circuits 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.

We observed that the DRAMs decayed in highly nonuniform patterns. While these varied from chip to chip, they were very stable across trials. The most prominent pattern is a gradual decay to the ground state as charge leaks out of the memory cells. In the decay illustrated in Figure 3, blocks of cells alternate between a ground state of zero and a ground state of one, resulting in the horizontal bars. The fainter vertical bands in the figure are due to manufacturing variations that cause cells in some parts of the chip to leak charge slightly faster than those in others.

**Figure 2: Measuring decay.** We measured memory decay after various intervals without power. The memories were running at normal operating temperature, without any special cooling. Curves for machines A and C would be off the scale to the right, with rapid decay at around 30 and 15 s, respectively.



**Figure 3: Visualizing memory decay.** We loaded a bitmap image into memory on test machine A, then cut power for varying intervals. After 5 s (left), the image is nearly indistinguishable from the original; it gradually becomes more degraded, as shown after 30, 60 s, and 5 min. The chips remained close to room temperature. Even after this longest trial, traces of the original remain. The decay shows prominent patterns caused by regions with alternating ground states (*horizontal bars*) and by physical variations in the chip (*fainter vertical bands*).



## 2.2. Decay at reduced temperature

Colder temperatures are known to increase data retention times. 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  $-50^{\circ}\text{C}$ . 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 Figure 4). On all of our test systems, the decay was slow enough that an attacker who cut power for 1 min would recover at least 99.9% of bits correctly.

We were able to obtain even longer retention times by cooling the chips with liquid nitrogen. After submerging the memory modules from machine A in liquid nitrogen for 60 min, we measured only 14,000 bit errors within a 1MB test region (0.13% decay). This suggests that data might be recoverable for hours or days with sufficient cooling.

## 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. Our 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

**Figure 4: Colder temperatures slow decay.** We measured memory errors for machines A–D after intervals without power, first at normal operating temperatures (no cooling) and then at a reduced temperature of  $-50^{\circ}\text{C}$ . Decay occurred much more slowly under the colder conditions.

	Seconds without Power	Average Bit Errors	
		No Cooling (%)	$-50^{\circ}\text{C}$ (%)
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

medium with minimal disruption to the original state.

Most modern PCs support network booting via Intel's Preboot Execution Environment (PXE), which provides rudimentary start-up and network services. We implemented a tiny (9KB) standalone application that can be booted directly via PXE and extracts the contents of RAM to another machine on the network. In a typical attack, a laptop connected to the target machine via an Ethernet crossover cable would run a client application for receiving the data. This tool takes around 30 s to copy 1GB of RAM.

Some recent computers, including Intel-based Macintosh systems, implement the Extensible Firmware Interface (EFI) instead of a PC BIOS. We implemented a second memory extractor as an EFI netboot application. Alternatively, most PCs can boot from an external USB device such as a USB hard drive or flash device. We created a third implementation in the form of a 10KB plug-in for the SYSLINUX



bootloader. It can be booted from an external USB device or a regular hard disk.

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, 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. As shown in Figure 5, widely available “canned air” dusting spray can be used to cool the chips to  $-50^{\circ}\text{C}$  and below. At these temperatures data can be recovered with low error rates even after several minutes.

#### 4. KEY RECONSTRUCTION

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.

Our algorithms achieve significantly better performance than brute force by considering information other than the actual key. Most cryptographic software is optimized by storing data precomputed from the key, such as a key schedule for block ciphers or an extended form of the private key for RSA. This data contains much more structure than the key

itself, and we can use this structure to perform efficient error correction.

These results imply a trade-off between efficiency and security. All of the disk encryption systems we studied 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 access, we find that it also facilitates attacks.

Our algorithms make use of the fact that most decay is *unidirectional*. In our experiments, almost all bits decayed to a predictable ground state with only a tiny fraction flipping in the opposite direction. 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 tiny—less than 0.1% in our tests. We further assume that the ground state decay probability is known to the attacker; it can be approximated by comparing the fractions of zeros and ones in the extracted key data and assuming that these were roughly equal before the data decayed.

##### 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.

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 repetitions 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

**Figure 5: Advanced cold-boot attack.** In our most powerful attack, the attacker reduces the temperature of the memory chips while the computer is still running, then physically moves them to another machine 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 (left). This cools the chips to around  $-50^{\circ}\text{C}$  (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 (right).



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%.

## 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 to 3 of the first two round keys, and byte  $i - 1$  from word 4 of the first round key (see Figure 6). This slice is 7 bytes long, but it is uniquely determined by the 4 bytes from the first round key.

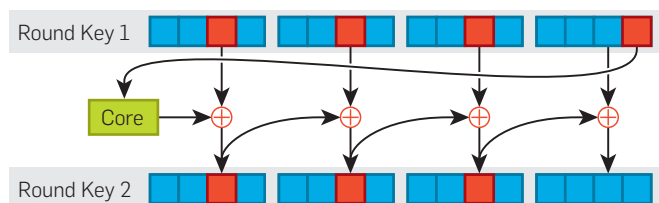
Our algorithm exploits this fact as follows. For each possible set of 4 key bytes, we generate the relevant 3 bytes of the next round key, and we order these possibilities by the likelihood that these 7 bytes might have decayed to the corresponding bytes extracted from memory. Now we may recombine four 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.

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 s for about half of keys with 15% of bits decayed.

We have extended this idea to 256-bit AES keys and to other ciphers. See the full paper for details.

**Figure 6: Error correction for AES keys. In the AES-128 key schedule, 4 bytes from each round key completely determine 3 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.**



## 4.3. 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<sup>11</sup> 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.

## 5. 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<sup>16</sup> 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<sup>15</sup> 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.

As described in Section 6, 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. 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 the full version of this paper.

## 6. ATTACKING ENCRYPTED DISKS

We have applied the tools developed in this paper to defeat several popular on-the-fly disk encryption systems, and we suspect that many similar products are also vulnerable. Our results suggest that disk encryption, while valuable, is not necessarily a sufficient defense against physical data theft.

### 6.1. BitLocker

BitLocker is a disk encryption feature included with some versions of Windows Vista and Windows 7. It operates as a filter driver that resides between the file system and the disk driver, encrypting and decrypting individual sectors on demand. As described in a paper by Niels Ferguson of Microsoft,<sup>8</sup> the BitLocker encryption algorithm encrypts data on the disk using a pair of AES keys, which, we discovered, reside in RAM in scheduled form for as long as the disk is mounted.

We created a fully automated demonstration attack tool called `BitUnlocker`. It consists of an external USB hard disk containing a Linux distribution, a custom SYSLINUX-based bootloader, and a custom driver that allows BitLocker volumes to be mounted under Linux. To use it against a running Windows system, one cuts power momentarily to reset the machine, then connects the USB disk and boots from the external drive. `BitUnlocker` automatically dumps the memory image to the external disk, runs `keyfind` to locate candidate keys, tries all combinations of the candidates, and, if the correct keys are found, mounts the BitLocker encrypted volume. Once the encrypted volume has been mounted, one can browse it using the Linux distribution just like any other volume.

We tested this attack on a modern laptop with 2GB of RAM. We rebooted it by removing the battery and cutting power for less than a second; although we did not use any cooling, `BitUnlocker` successfully recovered the keys with no errors and decrypted the disk. The entire automated process took around 25 min, and optimizations could greatly reduce this time.

### 6.2. FileVault

Apple's FileVault disk encryption software ships with recent versions of Mac OS X. A user-supplied password decrypts a header that contains both an AES key used to encrypt stored data and a second key used to compute IVs (initialization vectors).<sup>18</sup>

We used our EFI memory extraction program on an Intel-based Macintosh system running Mac OS X 10.4 with a FileVault volume mounted. Our `keyfind` program automatically identified the FileVault AES encryption key, which did not contain any bit errors in our tests.

As for the IV key, it is present in RAM while the disk is mounted, and if none of its bits decay, an attacker can identify it by attempting decryption using all appropriately sized substrings of memory. FileVault encrypts each disk block in CBC (cipher-block chaining) mode, so even if the attacker cannot recover the IV key, he can decrypt 4080 bytes of each 4096 byte disk block (all except the first cipher block) using only the AES key. The AES and IV keys together allow full decryption of the volume using programs like `vilefault`.<sup>18</sup>

### 6.3. TrueCrypt, dm-crypt, and Loop-AES

We tested three popular open-source disk encryption systems, TrueCrypt, dm-crypt, and Loop-AES, and found that they too are vulnerable to attacks like the ones we have described. In all three cases, once we had extracted a memory image with our tools, we were able to use `keyfind` to locate the encryption keys, which we then used to decrypt and mount the disks.

## 7. COUNTERMEASURES

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.

### 7.1. 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 key 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 the memory is too expensive, 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<sup>2</sup> 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.

### 7.2. 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.<sup>3</sup> 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.

### 7.3. Physical defenses

It may be possible to physically 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<sup>7</sup> 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.

### 7.4. 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<sup>6, 12</sup>; these would prevent the attacks we describe as long as the keys are reliably destroyed on reset or power loss.

### 7.5. 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.

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.

### 7.6. Trusted computing

Though useful against some attacks, most Trusted Computing hardware deployed in PCs today does 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. However, once the key is stored in RAM, it is subject to our attacks. Today's Trusted Computing devices 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, Trusted Computing makes 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 Trusted Computing hardware 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 attacks only need to power on the machine to have the keys loaded back into memory, where they are vulnerable to our attacks.

## 8. PREVIOUS WORK

We owe the suggestion that DRAM contents can survive cold boot to Pettersson,<sup>15</sup> who seems to have obtained it from Chow et al.<sup>5</sup> Pettersson suggested that remanence across cold boot could be used to acquire forensic memory images and cryptographic keys. Chow et al. discovered the property during an unrelated experiment, and they remarked on its security implications. Neither experimented with those implications.

MacIver stated in a presentation<sup>14</sup> that Microsoft considered memory remanence in designing its BitLocker disk encryption system. He acknowledged that BitLocker is vulnerable to having keys extracted by cold-booting a machine when used in a "basic mode," but he asserted that BitLocker is not vulnerable in "advanced modes" (where a user must

provide key material to access the volume). MacIver apparently has not published on this subject.

Researchers have 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.<sup>13</sup> In 2002, Skorobogatov<sup>17</sup> found significant retention times with *static* RAMs at room temperature. Our results for DRAMs show even longer retention in some cases.

Some past work focuses on “burn-in” effects that occur when data is stored in RAM for an extended period. Gutmann<sup>9,10</sup> attributes burn-in to physical changes in memory cells, and he suggests that keys be relocated periodically as a defense. Our findings concern a different phenomenon. The remanence effects we studied occur even when data is stored only momentarily, and they result not from physical changes but from the electrical capacitance of DRAM cells.

A number of methods exist for obtaining memory images from live systems. Unlike existing techniques, our attacks do not require access to specialized hardware or a privileged account on the target system, and they are resistant to operating system countermeasures.

## 9. CONCLUSION


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.<sup>4</sup> demonstrate a dangerous extension to our attacks. They show how to cold-reboot 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.

## Acknowledgments

We thank Andrew Appel, Jesse Burns, Grey David, Laura Felten, Christian Fromme, Dan Good, Peter Gutmann, Benjamin Mako Hill, David Hulton, Brie Ilanda, 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 contributions. This work was supported in part by a National Science Foundation Graduate Research Fellowship and by the Department of Homeland Security Scholarship and Fellowship Program; it does not necessarily reflect the views of NSF or DHS. 

## References

- Arbaugh, W., Farber, D., Smith, J. A secure and reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 1997), 65–71.
- Boyer, X. Halting password puzzles: Hard-to-break encryption from human-memorable keys. In *Proceedings of the 16th USENIX Security Symposium* (August 2008).
- Canetti, R., Dodis, Y., Halevi, S., Kushilevitz, E., Sahai, A. Exposure-resilient functions and all-or-nothing transforms. In *EUROCRYPT 2000*, volume 1807/2000 (2000), 453–469.
- Chan, E.M., Carlyle, J.C., David, F.M., Farivar, R., Campbell, R.H. Bootjacker: Compromising computers using forced restarts. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (October 2008), 555–564.
- Chow, J., Pfaff, B., Garfinkel, T., Rosenblum, M. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the 14th USENIX Security Symposium* (August 2005), 331–346.
- Dwoskin, J., Lee, R.B. Hardware-rooted trust for secure key management and transient trust. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (October 2007), 389–400.
- Dyer, J.G., Lindemann, M., Perez, R., Sailer, R., van Doorn, L., Smith, S.W., Weingart, S. Building the IBM 4758 secure coprocessor. *Computer 34* (Oct. 2001), 57–66.
- Ferguson, N. AES-CBC + Elephant diffuser: A disk encryption algorithm for Windows Vista. (August 2006).
- Gutmann, P. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th USENIX Security Symposium* (July 1996), 77–90.
- Gutmann, P. Data remanence in semiconductor devices. In *Proceedings of the 10th USENIX Security Symposium* (August 2001), 39–54.
- Heninger, N., Shacham, H. Improved RSA private key reconstruction for cold boot attacks. *Cryptology ePrint Archive*, Report 2008/510, December 2008.
- Lie, D., Thekkath, C.A., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., Horowitz, M. Architectural support for copy and tamper resistant software. In *Symposium on Architectural Support for Programming Languages and Operating Systems* (2000).
- Link, W., May, H. Eigenschaften von MOS-Ein-Transistorspeicherzellen bei tiefen Temperaturen. *Archiv für Elektronik und Übertragungstechnik 33* (June 1979), 229–235.
- MacIver, D. Penetration testing Windows Vista BitLocker drive encryption. Presentation, Hack In The Box (September 2006).
- Pettersson, T. Cryptographic key recovery from Linux memory dumps. Presentation, Chaos Communication Camp (August 2007).
- Shamir, A., van Someren, N. Playing “hide and seek” with stored keys. *LNCS 1648* (1999), 118–124.
- Skorobogatov, S. Low-temperature data remanence in static RAM. University of Cambridge Computer Laboratory Technical Report 536, June 2002.
- Weinmann, R.-P., Appelbaum, J. Unlocking FileVault. Presentation, 23rd Chaos Communication Congress, December 2006.

**J. Alex Halderman**  
(jhalderm@eecs.umich.edu)  
University of Michigan.

**Seth D. Schoen**  
(schoen@eff.org)  
Electronic Frontier Foundation.

**Nadia Heninger**  
(nadiah@cs.princeton.edu)  
Princeton University.

**William Clarkson**  
(wclarkso@cs.princeton.edu)  
Princeton University.

**William Paul**  
(wpaul@windriver.com)  
Wind River Systems.

**Joseph A. Calandrino**  
(jcalandr@cs.princeton.edu)  
Princeton University.

**Ariel J. Feldman**  
(ajfeldma@cs.princeton.edu)  
Princeton University.

**Jacob Appelbaum**  
(jacob@appelbaum.net)  
The Tor Project.

**Edward W. Felten**  
(felten@cs.princeton.edu)  
Princeton University.



Copyright of Communications of the ACM is the property of Association for Computing Machinery and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.