Cryptkeeper: Improving Security With Encrypted RAM

Peter A. H. Peterson
University of California, Los Angeles
3564 Boelter Hall, Los Angeles, CA 90095
pahp@cs.ucla.edu

***Abstract*—Random Access Memory (RAM) was recently shown to be vulnerable to physical attacks exposing the totality of memory, including user data and encryption keys. We present Cryptkeeper, a novel software-encrypted virtual memory manager that mitigates data exposure when used with a secure key-hiding mechanism. Cryptkeeper significantly reduces the amount of cleartext data in memory by dividing RAM into a smaller, cleartext working set and a larger, encrypted area. This extends the standard memory model and provides encrypted swap as a side effect. Despite a 9x slowdown in pathological cases, target applications such as Firefox are only 9% slower with our Linux-based prototype. We also identify several optimizations which can significantly improve performance. Cryptkeeper enables the expression of new security policies for memory, and demonstrates that modern personal computers can perform heavy-duty work on behalf of operating systems with surprisingly low overhead.**

***Index Terms*—Operating systems, Data security, Memory management.**

I. Introduction

In 2008, a widely publicized paper by Halderman, et al. [1] showed that DRAM is substantially less volatile than commonly believed. Using only a computer, software, and a can of "compressed air," so-called "cold boot attacks" can recover vast amounts of data from RAM. While the "cold boot" paper focuses on recovering Full Disk Encryption (FDE) [2] keys, RAM itself is likely to contain *gigabytes* of unencrypted private data due to aggressive caching and prefetching schemes. While cold boot attacks are unlikely to affect the average user, organizations for whom information security is truly critical cannot afford to ignore this threat.

We mitigate this vulnerability with Cryptkeeper (CK), a software-encrypted virtual memory manager. Traditional processors cannot operate on encrypted data, so CK segments RAM into a smaller working set called the Clear, and a larger encrypted RAM device called the Crypt. As the working space fills, pages are automatically swapped into the encrypted portion of memory, and are decrypted on demand.

This paper describes the CK model and demonstrates that it is practical today for real-world workloads. This is surprising because software-encrypted RAM seems unlikely to perform reasonably. CK is valuable because it has the potential to protect data in RAM from physical exposure (in concert with secure key storage). We discuss and analyze our prototype which is based on Linux. Finally, we discuss future and related work, performance optimizations, and methods for protecting encryption keys.

II. Cryptkeeper

The main idea behind CK is that splitting RAM into a smaller, faster, insecure segment, and a larger, slower, secure segment, can improve security for a reasonable (and adjustable) performance cost. In a CK system, the majority of RAM (the Crypt) contains encrypted data and functions like a high-priority encrypted swap which selectively passes older pages to disk. The smaller portion of RAM (the Clear) holds decrypted data for immediate use. Figure 1 describes a series of operations in an idealized version of CK.

The size of CK's Clear provides a direct "performance versus security" tradeoff. A smaller Clear represents more work, because the computer has a smaller "working set" of unencrypted pages. However, a larger Clear represents a greater security risk, because a larger portion of memory will be unencrypted at any given time.

All pages are initially allocated as Clear pages, so each allocation reduces the number of free Clear pages (FCP) in the system. When the number of FCP is low, Clear pages are freed by encrypting the least recently used Clear pages, which "moves" them to the Crypt. Similarly, pages in the Crypt swap to disk when memory pressure on the Crypt is high. As a result, CK provides encrypted swap as a side effect. Data in both the Crypt and swap are decrypted on demand, making room in the Clear if necessary.

CK *by itself* does not fully protect against physical attacks because personal computers do not have secure key storage. All software cryptosystems (such as CK and FDE) must keep keys in unencrypted RAM where they are vulnerable to physical attacks. We discuss using the Trusted Platform Module [15] or Exposure-Resilient Functions [16] as potential solutions to this problem in Section VI.

III. Maintaining System Performance

The idea of encrypting and decrypting main memory on the fly — in software — may seem positively quixotic. While there are those willing to pay high costs for security, there are still practical limits to usability and performance. In this section, we discuss the fundamental realities that enable CK to perform acceptably for common workloads.

A. Full Disk Encryption as a Foundation

We assume that a CK user would also use an FDE solution, because CK does not protect files. This is reasonable because many security-conscious users are already using FDE despite its performance impact. FDE is also an ideal base for CK because the two systems have partially overlapping workloads which can be combined. Nevertheless, we show that software FDE is not required for CK to be practical.

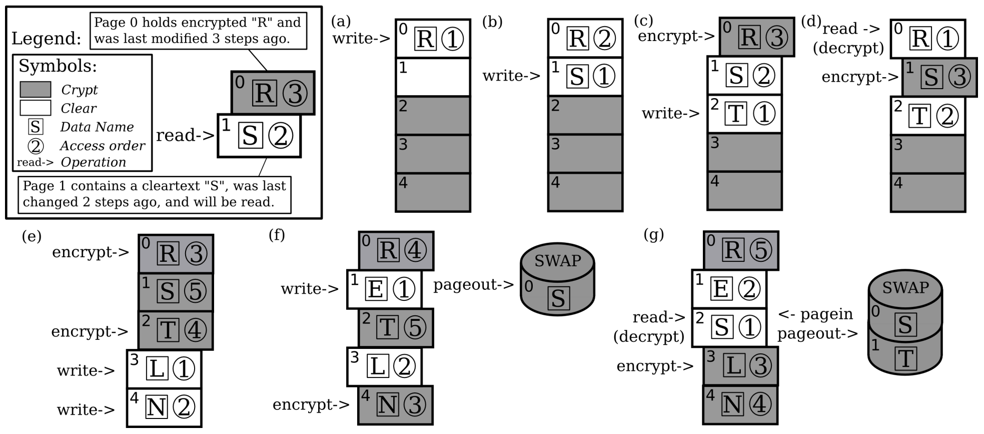

Fig. 1. Example operation. In this scenario, Cryptkeeper has a capacity of five physical memory pages which are split into two Clear pages and three Crypt pages. **(a):** All memory is free. Data R is written into physical page 0. **(b):** S is written into page 1. **(c):** T is to be written into page 2, but this would result in three Clear pages. This forces the oldest page, 0, to be encrypted and sent to the Crypt. **(d):** 0 is read, which decrypts it and returns it to the Clear. However, the Clear is full, so page 1 (the oldest clear page), is encrypted and sent to the Crypt. **(e):** Data L and N are written into 3 and 4, which causes 0 and 2 to be sent to the Crypt. RAM is now full. **(f):** Data E is being written. However, space must be created in both physical memory and the Clear before this can happen. Page 4 is sent to the Crypt because it is the oldest Clear page. Next, a page must be swapped to make room in RAM. Page 1 is swapped out because it is the oldest Crypt page. E is now written into page 1. **(g):** The data in swap entry 0 (an encrypted version of S) is requested. Once again, we make space by encrypting page 3 (the oldest Clear page), and by swapping out page 2 (the oldest Crypt page). The encrypted version of T from page 2 is written into swap entry 1, and S is finally decrypted into page 2.

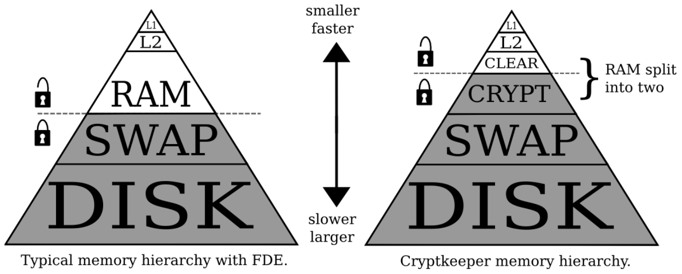

Fig. 2. Comparison of traditional memory hierarchy with CK. Levels below the dotted line are encrypted, levels above are unencrypted.

B. Performance Bounds and Hierarchical Memory

CK's architecture extends the traditional hierarchical memory model shown in Figure 2, which requires each level in the hierarchy to be slower and larger than the one above it. This architecture is advantageous for many real-world usage patterns and is standard in virtually all computers. CK does not break this model, and as a result can use its properties. We can also use this knowledge to reason about CK's performance:

Our best-case scenario is that *CK will run as fast as an equivalent system with normal RAM and encrypted swap.* Since CK will need to do *at least* this much work, CK *could not be faster than such a system*. However, that would still be quite satisfactory; encrypted swap is a popular option for Linux systems and is provided by many FDE solutions.

We can also easily define the point at which CK would cease to provide any benefit. Given a CK system with a Clear of M megabytes, we can compare it to a Linux system with M megabytes of RAM and encrypted swap. If CK is *slower* than such a system, a user who wanted CK's security guarantees would achieve better performance by limiting physical RAM and using encrypted swap to protect inactive data.

C. Processing Power and Parallelism

Moore's Law [3] has made many things possible that were previously almost unimaginable, such as FDE on personal computers (PCs). Recently, multiprocessing has been a prime mover in performance growth, and most new PCs have multiple cores. Multiple processors can trivially run a set of applications in parallel, but the difficulty of writing parallel versions of individual programs has limited our ability to fully utilize multiple cores. However, block cryptography can often be parallelized — which significantly improves CK's performance by reducing the wait time associated with cryptographic operations. We discuss algorithm choices and other related issues in Section IV-C.

D. CPU vs. I/O Throughput

The performance gap between CPU and hard drive throughput is wide and growing wider [4]. One outcome of this is that the throughput of CPUs encrypting data can easily surpass the throughput of hard drives writing data. Our test machine (described in Section V) is able to encrypt approximately 208M/s using OpenSSL's AES-256 CBC speed test. On the other hand, the test machine's hard disk[1] can only write at a sustained rate of about 67M/s[2]. As a result, CK can perform a substantial amount of cryptography while waiting for file I/O to complete.

E. Memory Management Techniques

CK must use the limited space of the Clear efficiently to reduce cryptographic costs. Because CK fits neatly into the standard memory hierarchy, we can use traditional techniques for efficient memory management. For example, CK supports

[1]Hitachi DeskStar SATA, 250GB, 7200rpm

[2]For reference, the state-of-the-art mechanical hard drive as of September 2009 [5] is able to write at a sustained 138M/s.

demand paging to ensure that allocated Clear pages will actually be used. CK also supports Copy On Write (COW) shared memory, reducing overall memory consumption. Finally, just as Linux uses a Least Recently Used (LRU) algorithm to send underused RAM pages to swap, CK uses a similar algorithm when sending Clear pages to the Crypt.

IV. Cryptkeeper Prototype

We built a prototype based on Linux 2.6.24 to demonstrate CK's performance. This required adding to and extending many elements of the virtual memory manager. While our prototype is Linux-based, the CK model is general and should be portable to other operating systems.

A. Page Accounting and Management

CK must be able to accurately track the capacity of the Clear, as well as trap and decrypt encrypted pages. To track the free Clear pages (FCP), we added an atomic counter. In order to be able to set or check the status of a physical page outside of any process, we extended the kernel `page` structure with flags to indicate Crypt/Clear status. To trap accesses to encrypted pages, we mark the page table entry (PTE) for Crypt pages with a protection bit and a new "crypted" bit. When a page fault occurs, if the PTE indicates that the page is encrypted (or in swap), it is decrypted, the FCP count is updated, and the page is returned to the process. If there are not enough FCP, the faulting process will be descheduled until more are available.

B. Page Reclamation

Each memory zone in the Linux kernel has its own thread which sleeps until the zone is running out of free pages. This thread, `kswapd`, is designed to keep "just enough" pages free so that allocations are never blocked because of too few pages, but not so many that pages sit unused. To accomplish this, the kernel defines three "watermarks" for each zone, *high*, *low*, and *min*. When the count of free pages in a zone is below *low*, `kswapd` is awoken and attempts to free inactive pages back up to the *high* watermark. If less than *min* pages are available, the kernel goes into "direct reclaim," where the page allocator manually removes pages from active processes. CK defines its own *high*, *low*, and *min* watermarks, creates `kcryptd` threads for each CPU, and uses these elements in a similar fashion to keep "just enough" FCP available at all times.

C. Cryptography

While any standard block cipher could be used, our CK prototype uses the Advanced Encryption Standard (AES) algorithm [6], with code from the Nettle cryptographic library [7]. We used Electronic Code Book (ECB) mode because it is easier to debug, although this simplicity also allows a considerable amount of information about the underlying plaintext to be visible [8]. A fielded CK would use a stronger mode such as Cyclic Block Chaining (CBC), which XORs each plaintext block with the previous ciphertext block [9]. CBC clearly requires more work than ECB. However, the additional cost of CBC is small due to the XOR's low cost compared to the cryptography [10]. Additionally, chaining would not prohibit access to arbitrary sequences of pages, because we could use unique keys *per page* and only chain the cipher blocks in each page. This allows each page to be encrypted and decrypted independently while benefiting from the stronger protection of CBC mode. Ironically, this may improve parallelism by reducing lock contention for cryptographic keys.

V. Evaluation

Ideally, CK would be integrated into a Full Disk Encryption system, and thus we would compare CK+FDE to Linux with FDE. Both systems must encrypt and decrypt a large portion of the data they process, although CK does this in RAM and FDE does this between RAM and disk (Figure 2). Because of these overlapping workloads, a combined system would gain the benefits of both without paying for them twice. However, the additional cost of building a fully-integrated system is not necessary to demonstrate the performance of the CK model. Instead, we can factor out the common workload. To do this, we compare CK (which provides encrypted swap) to Linux with encrypted swap enabled.

A. Benchmark and Test System

Our memory microbenchmark, `memeater`, allocates, writes, reads, and frees n pages of memory, and returns the total test time in seconds and the average number of microseconds each step takes per page. As n grows, `memeater` takes longer to complete, especially as n grows beyond the size of the Clear. `memeater` has two read modes, *stripe* and *random*. *Stripe* performs a "worst case" test for CK by reading pages from oldest to youngest to increase the chance that pages being read have been moved to the Crypt or swap. *Random* mode is less pathological and reads n randomly chosen pages.

The kernels shown are CK, with 256M of Clear and 583M of Crypt, and Linux+Crypt, which is a non-CK Linux kernel with encrypted swap. We also compared three other kernels: CK Single is a single-core version of CK, CK SMP1KCD is a multiprocessing kernel with only one `kcryptd` thread, and Linux NoCrypt which is the Linux kernel *without* encrypted swap. Apart from the tested options, all kernels are built from the same source and are run on the same Intel CoreDuo 2.66Ghz with 839M of RAM. Averages shown have 95% confidence.

B. Cryptographic and Bookkeeping Overhead

CK imposes a small non-cryptographic overhead due to bookkeeping, but this is dwarfed by the cryptographic costs. We measured the average cryptographic cost that CK imposes by examining the data for the page allocations between 96,000 and 196,000 4K pages, which correspond to the range of allocations that significantly use the Crypt but do not use swap. If we average the differences in cost between CK NoCrypt and CK over these ranges, we find that the average cost to encrypt a page when writing is 20μs, and the average cost for decryption while reading is 76μs. With encryption, the average

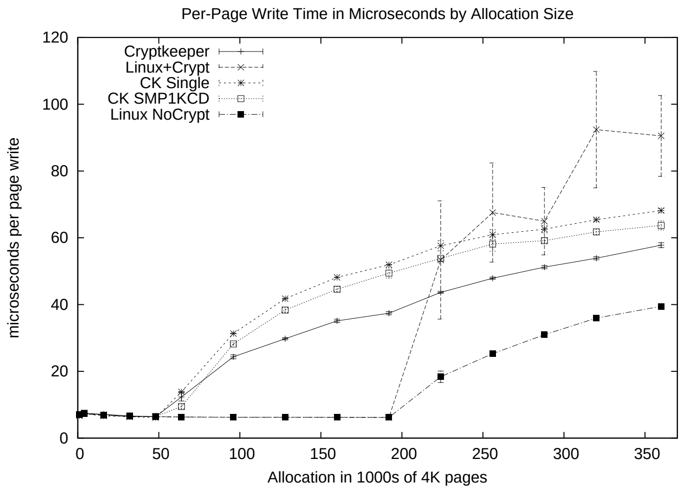

Fig. 3. Write performance comparison of five kernels as allocation sizes increase.

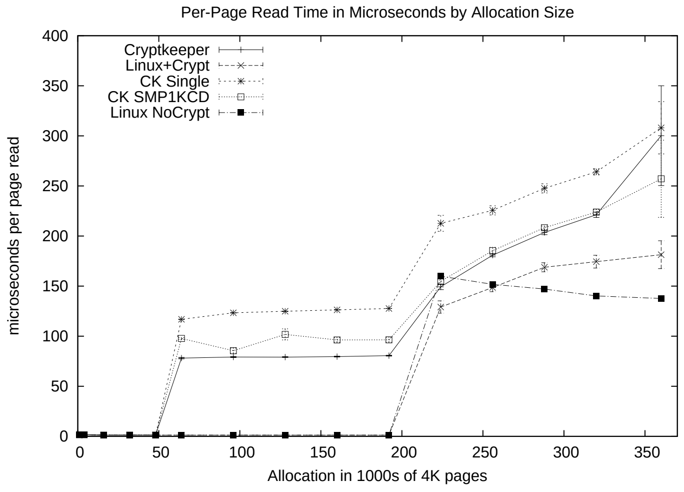

Fig. 4. Read performance comparison of five kernels as allocation sizes increase.

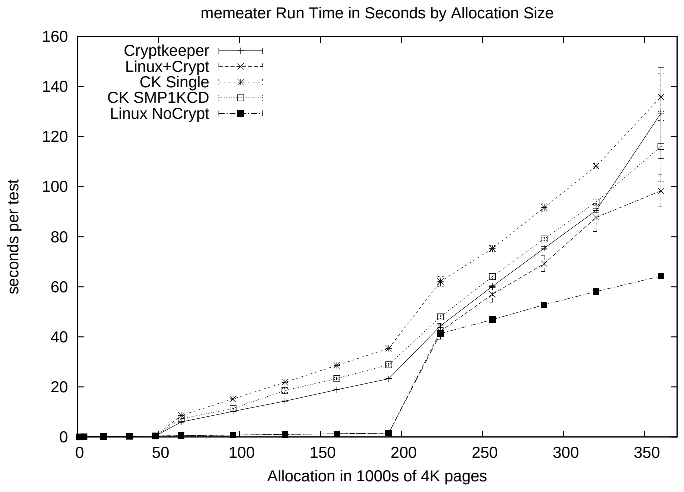

Fig. 5. Wall clock time comparison of five kernels as allocation sizes increase.

test runtime is about 8 times slower in this range (13 seconds versus 1.6 seconds). Cryptographic improvements could dramatically improve performance, such as using OpenSSL's implementation of AES (which is about 20% faster than the Nettle AES implementation [11]).

C. *Impact of Parallelism*

We examined the effect of parallelism on CK's performance. Figures 3 and 4 show comparisons of write and read throughput of three different CK kernels as allocation sizes change. The graphs show that CK usually benefits from parallelism. One surprise is that CK SMP1KCD (with one `kcryptd`) briefly outperforms the standard version of CK (with two `kcryptd`s) for writes when allocation sizes are low (Figure 3). This is due to CPU contention; `memeater` is single-threaded and there are no other significant processes running on the test machine. In that environment, CK SMP1KCD can win by using one core for `memeater` and the other core for `kcryptd` when CK must balance `memeater` and two `kcryptd` threads. Another surprise is that reads for CK and CK SMP1KCD converge once the allocation exceeds physical RAM (Figure 4). This is because when swap dominates the tests, the bottleneck becomes disk throughput. At that point, one `kcryptd` is as good or better than two. This is made clear during 360K page allocations, when two `kcryptd` threads degrade CK's performance while CK SMP1KCD's growth stays constant. A fielded CK could optimize performance by managing the number of `kcryptd` threads on the fly.

D. *Read vs. Write Performance*

CK's maximum tested per-page write time is around 58μs (Figure 3), while the maximum tested per-page read time reaches 300μs (Figure 4). It would be tempting to ascribe the poor read performance to the well-known latency of swap disks. However, read times are uniformly higher than write times well before swap is touched. This is actually because processes are almost never blocked waiting to write into memory since pages are sent to the Crypt in large batches. However, page faults caused by reads must be handled *sequentially* because a fault blocks the process until it is resolved. What's more, if one page must be brought in from the Crypt, it is possible that the next page may also need to be decrypted, but this will not be discovered until the current fault is resolved. We will discuss some strategies for mitigating this in Section VI.

E. *Performance vs. Clear Size*

We can now highlight the performance tradeoff between the size of the Clear and the workload. Using the *random* mode of `memeater` for more general performance, we measured the runtime performance of 196K page tests (768M) over Clear sizes from 64M to 768M. 64M/196K tests completed in about 25 seconds, while 768M/196K tests completed in about 5 seconds, with roughly linear growth in between. This shows that CK's performance can be maximized based on the required level of security. More sophisticated tradeoffs are discussed in Section VI.

F. *Cryptkeeper vs. Reduced Memory*

To provide any benefit, CK must outperform the "low memory plus encrypted swap" configuration from Section III-B. To show that CK is indeed faster than such a system, we tested CK versus Linux+LowMem, which is Linux+Crypt with non-swappable ramdisks filling all but approximately 256M of RAM. The two systems performed equivalently until 64K page tests, when Linux+LowMem's RAM was exhausted. At that point, Linux+LowMem's performance erratically degrades until it is roughly twice as slow as CK during 360K page tests.

G. Cryptkeeper vs. Linux

Figures 3, 4, and 5 show a comparison between CK and Linux+Crypt (Linux with encrypted swap) on `memeater`'s pathological, worst-case workload. Before the Clear is full, performance times for Linux and CK are almost statistically equivalent. The worst period for CK is when the Crypt is in use but swap is not (64K to 196K page allocations). In this range, which essentially shows the raw cryptographic overhead of CK, Linux writes take 6.2μs versus CK's 28μs, for a 352% penalty. Predictably, reads are worse; Linux takes 1.3μs per read versus CK's 79.5μs, for a 6015% penalty. While this is a huge increase, it is not at all surprising. Linux pays "nothing" for the read, but CK must encrypt a page, decrypt a page, and update the page tables of all affected processes. Figure 5 shows that wall clock time is somewhat better because the overall system behavior hides the microbenchmark costs; tests in this range take an average of 1 second to complete on Linux+Crypt versus 14.6 seconds on CK, for a "mere" 1360% penalty. As shown in Section V-B, this is by far the single biggest bottleneck for CK, and thus is where optimizations could have the greatest impact.

The performance gap shrinks dramatically during the range where swap is used (229K to 360K pages), because Linux is slowed both by disk latency and encryption overhead. CK is actually *faster* than Linux for writes in this range — 51μs on CK versus 74.1μs per write on Linux for a 31% **improvement** over Linux. Reads on Linux take 161μs in this range versus only 212μs for CK for a 32% penalty (which is 188 times better than the non-swap penalty of 6015%). Wall clock time in Figure 5 is again much better; for most of this range, CK is *statistically equivalent* to Linux in runtime, although 288K page tests take 1.3 second longer than Linux on average, and 360K page tests are 5.3 seconds longer than Linux (32.5 instead of 27.2 seconds).

Finally, it is interesting as a curiosity to note that 229K page reads on Linux NoCrypt are actually *slower* (about 30μs per page) than Linux+Crypt. This could be for any number of reasons; for example, encrypted swap might allocate buffers for encryption which result in better early swap performance. Regardless, CK should not affect this in any way.

H. Application Tests

CK's target platform is a PC running typical applications containing sensitive information. In order to get a more accurate picture of CK performance on day-to-day tasks, we performed three kinds of application tests on CK and Linux+Crypt.

First, we built the Linux kernel on our test machine. While most "road warriors" don't compile code, they often run multiple processes which use varying amounts of memory, CPU, and disk resources. Linux kernel compilation is multi-threaded, multi-process, utilizes disk, RAM, and CPU.

For a more realistic user scenario, we used the Linux Desktop Testing Project [12] to automate a session with Firefox and the X Window System running on Ubuntu Linux. In this test, the simulated user reads 100 archived Facebook and CNN pages over a 100-megabit link. The script opens each page (with a few seconds delay for "think time"), and randomly opens new tabs and switches between tabs to simulate user behavior. Firefox alone uses between 200M and 270M of RAM during the test, while the X Window System and other various system processes use over 400M of RAM. With a Clear size of 256M, this workload will overflow the Clear and exercise CK. Finally, we ran this Firefox test and the kernel test concurrently in order to create increased disk, memory, and CPU contention on the desktop.

I. Application Results

Table I shows that the performance of the Linux kernel build is hardly penalized, so we can conclude that its requirements barely exercise CK at all. This is probably because the total workload does not expand much beyond the Clear, and the build generates considerable I/O.

The Firefox results are much more informative. Predictably, the system time measurement is significantly different between CK and Linux+Crypt. CK uses 1.07 seconds of system time versus Linux+Crypt, which uses only 0.04 seconds. This 1.03 seconds represents the additional CK kernel workload relating to Firefox. In wall clock time, CK uses 410 seconds on average per test and Linux+Crypt uses 379 seconds for a 31-second or 8% penalty.

For the Firefox+kernel results, it is important to explain that the tests only trace the resources used by Firefox; the kernel build only adds to the ambient workload. Thus, when the Firefox and the kernel build tests are combined, the system time penalty is only 0.94 seconds (which is statistically equivalent to the Firefox-only value of 1.03 seconds). However, the kernel build resource usage is visible in the increased wall clock time. In the end, CK uses 466 seconds of wall clock time while Linux+Crypt uses 428 seconds for a difference of 38 seconds or a 9% increase. This is unsurprising; the greater the application load on the system, the greater the overall cost of CK.

Although swap activity hides CK's overhead, today's machines rarely experience swap churn. However, our Firefox and kernel tests essentially do not involve swap, demonstrating that swap is not responsible for their reasonable performance. They do involve disk I/O, either as a part of their own operation or as part of the general operation of the system. This is one reason why testing real-world applications is important; by their very nature, they introduce I/O-related latency, which can actually improve user-visible performance. Regardless, CK may not be appropriate for large, memory-dominated tasks such as data mining, but should perform within the demonstrated range for typical user applications.

TABLE I
CRYPTKEEPER PENALTIES BY APPLICATION.

	user		sys		real	
Test	sec.	%	sec.	%	sec.	%
Kernel	NA	NA	0.45	2.6	0.07	0.04
Firefox	NA	NA	1.03	2575	31	8
Combined	NA	NA	0.94	2238	38	9

VI. FUTURE WORK

A. Speculative Decryption and Parallelization

CK could make better use of parallelism and speculation. For example, CK could "pre-decrypt" pages that are mapped in near a faulting page in an attempt to avoid future page faults. We could also potentially decrease turnaround time for sequences of page faults by decrypting discrete pages on different cores, which would reduce the discrepancy between per-page write and read times. Finally, swap read latency could be masked by disk I/O if 128-bit cipher blocks were decrypted as soon as possible, rather than after a full swap page is read.

B. Dynamically Sized Clear

The size of the Clear is currently hard-coded, but could be adjustable during run time, allowing CK to respond dynamically. For example, a policy might mandate a small Clear in a untrusted environment, while in a secure environment, the Clear could be increased until there is no performance cost. Another approach would set the number of Clear pages as a percentage of the number of used pages. In this scheme, if only 100 pages were in use and the Clear percentage was set to 25%, only 25 pages would be in the Clear. As the number of used pages grows, the capacity of the Clear would likewise grow. This would provide better protection than our prototype, where no encryption is performed until the Clear is almost full. Finally, a more finely grained approach would define Clear size limits *per process* rather than globally. Sensitive processes could be heavily restricted — adding security to the application through the operating system — without modifying the program or penalizing other processes.

C. Kernel Memory

Kernel memory in Linux is never swapped out and kernel memory cannot be page-protected [13] due to the complexity required to support it [14]. As a result, our prototype only protects userspace data. This is a limitation of the prototype, not the model. Still, a fielded CK system must at least be able to protect, trap and decrypt kernel memory in order to secure encryption keys or other critical data. It may be possible to implement the necessary parts in Linux without a complete redesign. If not, other operating systems (such as Windows) can swap kernel memory, so an alternative solution would be to build CK on top of a different operating system.

D. Key Hiding

There is a chicken-and-egg problem that affects all typical software cryptographic systems, including CK and FDE. Cryptography is the solution for data protection, but software cryptography requires that keys be in RAM where they are vulnerable to physical attacks. In order to fix this general vulnerability, computers *need* a method to keep at least a small amount of data truly secure.

The Trusted Platform Module [15] can provide protected storage by "binding" data — encrypting it with keys from its root of trust. The TPM can also "seal" data, ensuring that it cannot be unsealed unless the TPM's Platform Control Registers are in a particular state. Binding or sealing could be used to protect encryption keys in CK when not in use. However, key protection would have to be judiciously balanced with performance because TPMs are not designed for speed. Additionally, a CK+TPM design would need to consider any scenarios where a TPM could be tricked into decrypting memory keys through physical attacks on memory or the TPM itself.

A fascinating potential solution which requires no special hardware was suggested by Halderman, et al. [1] They discuss using Exposure-Resilient Functions (ERF) [16] developed by Canetti, et al. to protect keys in RAM. ERFs are cryptographic functions that can make encrypted data unrecoverable given some guaranteed number of unknown bits in the key. While cold boot attacks slow bit decay, they cannot completely stop it. If properly implemented, this inevitable decay combined with ERF's key expansion could ensure key destruction in a power-cutting attack.

Keys protected by either ERFs or a TPM cannot be used for decryption; to be useful, they must be decrypted and thus made vulnerable. However, by encrypting segments of RAM with different keys and protecting least recently used keys with some mechanism, we can limit exposure to the data in the Clear and data encrypted with unprotected keys. Regardless of the mechanism, we need a reasonably performant solution to the "key hiding problem," or software cryptosystems like CK and FDE will continue to be insecure against physical memory attacks.

VII. PRIOR WORK

Previous work on physical attacks, cryptography, and key hiding motivates and informs CK. The paper on "cold boot attacks" by Halderman, et al. [1] describes physical RAM attacks, key recovery methods, and the use of limited decay and "key expansion" as a potential means of mitigating key recovery. In typical cryptography, exposing *any* bits of the key destroys security guarantees. Canetti, et al. [16] describes Exposure-Resilient Functions (ERF) which are a method of making cryptographic keys susceptible to decay.

Blaze introduced the Crypto File System [17] and Provos discusses the (now) well-known reasons for using encrypted swap [18]. These methods serve to protect sensitive data in "non-volatile" storage such as hard disks. Yee introduced "crypto-paging" [19] [20], which is the encryption of cryptoproccessor memory before it is "swapped" to less secure media. In particular, Yee shows how a cryptoprocessor with limited memory can increase its secure space through encrypted virtual memory techniques on its own memory, swapping encrypted pages out to the host system's RAM. In this way, crypto-paging, like CK, exchanges a performance penalty for a larger amount of secure memory. CK is closely related to these ideas. However, CK is unique as a proof-of-concept and as a step towards improving the security of general purpose memory on personal computers.

There has been work on defenses to physical attacks. Boneh and Lipton describe a "Revocable Backup" system [21], and Di Crescenzo, et al. [22] discuss "Erasable Memory," an idea which is similar in spirit to the end result of CK. Erasable Memory is more formally defined than Revocable Backups but both rely on secure key destruction to render encrypted data "erased" even if the media is persistent. Erasable Memory has some interesting security properties that CK lacks, and it shares an architecture similar to CK and crypto-paging wherein a small special area is cryptographically leveraged to create a larger secure area. However, Eraseable Memory does not consider decay retardation methods such as cold boot attacks and it is not designed to fit into the hierarchical memory model. Furthermore, the specialized guarantees of Eraseable Memory impose obstacles to efficient use of large memory spaces, and the authors accordingly acknowledge that their design is probably not suitable for large data sets. CK does not require these guarantees, so it is able to protect a larger space with a more efficient use of resources.

Execution Only Memory (XOM) [23], [24] is a memory encryption architecture which supports copy and tamper resistant software. With cryptographic hardware built into the processor, XOM encrypts and decrypts all sensitive data as it enters or leaves the CPU. XOM and CK can use predecryption and caching to mitigate the resulting overhead. However, while XOM's guarantees are stronger, CK runs on commodity hardware.

VIII. Conclusion

By dividing RAM into a smaller cleartext (the Clear) and a larger encrypted segment (the Crypt), and by moving pages between these segments on the fly, Cryptkeeper extends the traditional memory hierarchy and can help to significantly reduce the exposure of sensitive data in RAM. CK represents a direct and powerful performance versus security tradeoff and opens the door for new security policies, such as encrypting memory on a per-process basis without modifying the applications.

CK's performance is reasonable for desktop applications, even though it requires more computation than Full Disk Encryption or encrypted swap alone. While exposing less than a third of RAM, our prototype imposes only a 9% penalty on real-world applications in a desktop environment. Although CK performs nine times worse on memory microbenchmarks, these tests were engineered to demonstrate worst-case performance. We have also discussed several optimizations that could further improve results.

CK and systems such as FDE require a method to secure cryptographic keys so that they are not vulnerable to physical attacks. While they would increase overhead, the Trusted Platform Module and Exposure-Resilient Functions provide promising means of protecting cryptographic keys in RAM.

Ultimately, CK pushes the boundaries of what software can do to protect physical memory on commodity systems; it is surprising that a personal computer can achieve reasonable performance while encrypting and decrypting its memory on the fly. CK clearly demonstrates that modern hardware trends — in particular parallelism — provide systems with unprecedented resources that can be used to improve security, performance, or usability at a small cost to the user.

References

[1] J. A. Halderman *et al.*, "Lest We Remember: Cold Boot Attacks on Encryption Keys," *Proc. 17th USENIX Security Symposium*, 2008.
[2] A. Czeskis *et al.*, "Defeating Encrypted and Deniable File Systems: TrueCrypt v5.1a and the Case of the Tattling OS and Applications," *3rd USENIX HotSec*, 2008.
[3] G. Moore, "Intel Museum – Moore's Law," http://www.intel.com/museum/archives/history_docs/mooreslaw.htm, 2009, accessed Tuesday, October 20th, 2009.
[4] P. Schmid, "15 Years of Hard Drive History: Capacities Outran Performance," http://www.tomshardware.com/reviews/15-years-of-hard-drive-history,1368-7.html, 2008, accessed Sunday, October 18th 2009.
[5] *SG Barracuda XT Series SATA Product Manual, Rev. A*, Seagate Technology, 2009, http://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda%20XT/100586689a.pdf.
[6] J. Daemen and V. Rijmen, *The Design of Rijndael:AES - The Advanced Encryption Standard*. Springer, 2002.
[7] N. Moller, "Nettle – a low-level cryptographic library," http://www.lysator.liu.se/~nisse/nettle/nettle.html, 2009, accessed Tuesday, October 20th, 2009.
[8] N. El-Fishawy and O. M. A. Zaid, "Quality of Encryption Measurement of Bitmap Images with RC6, MRC6, and Rijndael Block Cipher Algorithms," *International Journal of Network Security*, 2007.
[9] W. F. Ehrsam *et al.*, "Message verification and transmission error detection by block chaining," U.S. Patent No. 4,074,066, 1978.
[10] M. Roe, "Performance of symmetric ciphers and one-way hash functions," in *Fast Software Encryption*, ser. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 1995, pp. 359–362.
[11] T. Bingmann, "Speedtest and Comparsion of Open-Source Cryptography Libraries and Compiler Flags," http://idlebox.net/2008/0714-cryptography-speedtest-comparison/, 2008.
[12] Various, "The Linux Desktop Testing Project," http://ldtp.freedesktop.org/wiki/, accessed Tuesday, October 20th, 2009.
[13] M. Gorman, *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004, ch. 4, Process Address Space, p. 82.
[14] Various, http://lkml.indiana.edu/hypermail/linux/kernel/0104.2/0282.html, 2001, accessed Monday, October 19th, 2009.
[15] Trusted Computing Group, "TPM Main Specification 1.2," http://www.trustedcomputinggroup.org/resources/tpm_main_specification.
[16] R. Canetti *et al.*, "Exposure-Resilient Functions and All-or-Nothing Transforms," *EUROCRYPT 2000, LNCS 1807*, 2000.
[17] M. Blaze, "A cryptographic file system for UNIX," *Proc. of the 1st ACM conference on Computer and communications security*, 1993.
[18] N. Provos, "Encrypting virtual memory," *Proc. of the 9th USENIX Security Symposium*, 2000.
[19] B. Yee, "Secure coprocessors in electronic commerce applications," *First USENIX Workshop on Electronic Commerce*, 1995.
[20] ——, "Using Secure Coprocessors (PhD Thesis)," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, 1994.
[21] D. Boneh and R. J. Lipton, "A Revocable Backup System (Extended Abstract)," *USENIX Security Symposium*, 1996.
[22] G. D. Crescenzo *et al.*, "How to forget a secret (extended abstract)," *STACS'99, LNCS 1563*, 1999.
[23] D. Lie, "Architectural Support for Copy and Tamper Resistant Software," Ph.D. dissertation, Stanford University, 2003.
[24] B. Rogers *et al.*, "Memory predecryption: hiding the latency overhead of memory encryption," *SIGARCH Comput. Archit. News*, vol. 33, no. 1, pp. 27–33, 2005.