Securing Data on Compromised Hardware

Matt Corallo

University of North Carolina at Chapel Hill, Computer Science Department
Chapel Hill, NC, USA

***Abstract*—Advancements in attacks with physical access to commodity hardware has resulted in a general consensus that, given physical access, all of the data on a machine should be considered compromised. In this talk, I will consider existing attacks and mitigations and propose a practical system under which certain data security guarantees can be made. Specifically, I propose a system which attempts to protecting encryption keys absolutely against physical attackers and uses them to protect both main memory and disk access.**

I. Introduction

Over the past many years, advancements in attacks allowing the exfiltration of data when given physical access to the hardware of a machine have progressed significantly faster than defences against such attacks. The goal of this research is to address the problem not by continuing the cat-and-mouse game of the past decade, but instead by reducing the Trusted Computing Base (TCB) to a limited, secureable device and ensure access to other parts of the device be limited in their ability to gain access to useful data. Specifically, by private encryption keys in processor registers where they are inaccessible to anything but software running in kernel mode, we can create a simple and very small root of trust. I then use this to encrypt both disk access and large sections of main memory. Additionally, this research attempts to remain entirely practical for use on a daily basis, drawing an important distinction between much of the academic discussion of securing against physical attackers and this work.

I will begin in Section II by defining the threat model used for my analysis. Section III by briefly surveys the current state of data exfiltration with physical access, including current state-of-the-art attacks which are relevant to this project, as well as the existing mitigations and their drawbacks. Section IV covers my approach in more detail, proposing a model under which it can be analysed. In Section V, I dive into the implementation details, discussing tradeoffs which were made at varying levels to balance usability and security of the system. Section VI analyses the practical use of the system, evaluating both its security guarantees and its practicality. I conclude the paper with a section on possible future research areas as well as a final conclusion.

II. Threat Model

There are many considerations to be taken into account when discussing exfiltration of data and compromise of a machine via physical attacks. This paper primarily concerns itself with the ability of an attacker to, using any reasonable methods, to gain access to data stored on a machine. Keep in mind that a sufficiently motivated and well-funded attacker could use some hypothetical hardware to modify the CPU's registers or data during runtime, however the overhead cost on such an operation against a running machine makes it expensive to the point of impracticality and possibly impossibility. Thus, I instead chose to analyse existing published research on data exfiltration. Additionally, I primarily focus on attacks other than side-channel attacks as such attacks are both specific in nature (ie in nearly all form cannot be used to exfiltrate any data, but instead focuses on eg the stealing of private keys by analysing CPU operations) and a very wide range and large number of attacks. Though this limits the security guarantees that my system can make, this is a limitation inherent to the type of system which I am proposing.

In order to adequately discuss the methods for data exfiltration which have been published today, we must clearly define what our attacker can and cannot do, ie our threat model. As a simplifying assumption, we begin with a “trusted loading” phase. Namely, we begin with a boot phase during which we have a trusted kernel loaded into memory

and a secure path from the keyboard to the kernel (to enter an encryption passphrase). Note that this phase must repeat itself when the system resumes from suspend and is not only required during system boot. Though this is a rather expensive requirement (in the sense that it is hard to achieve today with commodity hardware), I will revisit it late in Section III as well as as a possibility for future research. Additionally, we assume that some hardware is potentially malicious during this phase, specifically, we limit our trust to only the BIOS and related basic systems. Though this sounds nice, in practice this can include all PCI-Express devices attached to the machine (see, for example, [1] which is able to load firmware-level code into the BIOS using EFI extensions), potentially including externally-attached PCI Express devices. However, we are generally able to leave many devices, such as USB devices, entirely untrusted and the extension of trust to PCI Express and many other devices does not generally apply when the system resumes from suspend, only during initial boot.

After the "trusted loading" phase of the system completes, we assume that all hardware is malicious and is actively attempting to assist the attacker in exfiltrating data. Specifically, after the "trusted loading" phase completes, we reduce our TCB to only include the processor and must limit the access of peripheral devices (including PCI-Express devices) to only data which they should have access to.

III. Brief Survey on The State of Hardware Compromise and its Impact on My System

Because of a large body of research amassed over the past decade, security common sense generally dictates that if an attacker has physical access to a machine, there is nothing that can be done to protect the data stored on and the behaviour of running programs on that machine. In this section, I analyse this claim and step through a selection of the various defences and attacks which have appeared over time.

A. First steps in protecting data

When attempting to protect the data on a machine from a physical attacker, the first step is usually full disk encryption, which does not allow any unencrypted data from ever reaching physical nonvolatile memory. In recent years, full disk encryption has become both incredibly performant (with the overhead to encrypt and decry data stored on disk being far less than the overhead required for disk reads/writes) and incredibly popular (with all mainstream operating systems having support for it "baked-in", often with incredibly simple user-facing Graphical User Interfaces, or GUIs). However, in step with the rise in the use of full disk encryption, attackers have made significant strides in defeating it. Specifically, because the encryption keys used to encrypt and decrypt data on the drive are stored in main memory, attacks which allow the reading and/or writing of system memory are incredibly successful. Specifically, both cold-boot attacks and DMA-based attack schemes have proven very successful and often quite easy to perform.

B. DMA-Based Attacks

Many modern peripheral devices require direct memory access (DMA) to communicate with the CPU in order to increase efficiency. In these systems, devices, such as graphics cards and network interface cards, are able to read and write arbitrarily to main memory and are then able to trigger interrupts on the processor to notify it that new data has been read or written. Under normal operation, such a device will only read or write to memory allocated to to it by the kernel. However, malicious DMA devices have been proposed many times in various formats over the years. Originally, much research focused on the use of FireWire (IEEE 1394) for its DMA capability and status as an external device, making exploitation without the need to install a device internal to a computer easy [2]. There was then much research into the use of similar DMA attacks by PCI-Express devices, often by exploiting a vulnerable device and installing persistent malware on it such that the device can provide persistent exploitation of the host system [3] by modifying the running kernel or other code present in main memory. Also note that more recent hardware interfaces, such as ThunderBolt, have exposed a host to DMA-based attacks via external devices as they expose a PCI-Express interface externally to an attacker. As a generic line of defence against against DMA-based exploitation,

modern commodity x86 hardware often includes I/OMMU capability. This device performs a similar function to the virtual memory translation operations performed for user-space applications, but applies to DMA access instead. A properly configured kernel and device driver will set the DMA translation tables for each device such that it is only able to access the areas of memory allocated to it, and such that any access outside of those areas is rejected by the I/OMMU hardware. Note that many kernels do not well support proper I/OMMU configuration [4] and there have been many attacks targeting the kernel to trick it into misconfiguration of the I/OMMU [5] or attack errors in the configuration of the I/OMMU as a result of bugs in the kernel or device drivers.

Instead of attempting to prevent DMA attacks by creating a new system of defence, my work instead builds on existing I/OMMU work, strongly encouraging the user to enable existing security features which are often disabled by default. For example, the I/OMMU features of x86 hardware are often disabled by default in Linux, which generates clearly worded warnings for users during the initialization of the system under my work. Additionally, as a part of my work, I have thoroughly studied existing research in DMA defence on commodity x86 hardware and either provided the user with warnings if they are vulnerable to known exploits. Specifically, by encouraging the user to initialize the kernel boot using Intel Trusted Execution Technology (TXT) the user is protected against attacks which exploit DMA during early boot (ie before the I/OMMU is configured) to hide their presence from the kernel, preventing them from being placed under I/OMMU protection [5].

C. *The ColdBoot Attack*

The ColdBoot attack [6], published in 2008, works by using the property of modern DRAM chips that retain their charge for seconds after power is lost (ie during a system shutdown). Thus, when a system is shut down, an attacker can read the majority of contents of main memory for several seconds before it fades. Additionally, if the memory is frozen before the system is shut down, its contents remains readable significantly longer, hence the name. Though it has other uses for data exfiltration (potentially recovering the memory of running applications at the time of shutdown), this attack captured popular attention as a method for drive-encryption key recovery and the most attention has been paid to it as such. The ColdBoot paper also discussed the possibility of recovering an encryption key by searching memory for the structure of AES and other encryption keys after they have undergone key expansion. Using this method allows an attacker to efficiently search memory without needing to fully interpret its contents as well as recover from some errors in key recovery. Additionally, note that even if some parts of an encryption key has faded and is not recoverable, any remaining parts may help to decrease the search space for a brute force approach significantly.

One of the proposed solutions to storing encryption keys such that a ColdBoot attack can not recover keys is to store them in processor registers, outside the reach of main memory-stealing techniques. One particular implementation of this idea is TRESOR [7]. There are a few sets of registers which are available in x86 which may be potential key storage locations (they must be largely unused and be accessible only to the kernel, ie not accessible to applications running in user mode). TRESOR suggested the use of the debug registers present in x86 as they offer a total of 256 bits of key storage space in registers which are used largely only by debugging applications and for which debugging applications nearly always have software-only alternatives to (for example, a 4-line patch to GDB will force it to always use software breakpoints instead of hardware ones, avoiding the debug registers). Though this approach largely solves the issue of an attacker using ColdBoot to steal encryption keys, it has several flaws in other areas. Specifically, an attacker exploiting and DMA-based attack with write access to system memory may rewrite part of the kernel code to simply copy the encryption keys from debug registers into main memory, allowing them to be read out using DMA or a ColdBoot attack, as shown by the TRESOR-HUNT [8] paper published a year later. Additionally, note that TRESOR does not solve the issue of an attacker gaining access to

any other data stored in memory, including application memory and disk cache, including caches of decrypted copies of files.

D. Notes on Types of Targets and Additional Vulnerabilities

It is important to note that physical attacks are generally only applicable with certain hardware present. For example, exploiting a host using DMA may be more difficult if there are no DMA-capable busses available externally and access to the machine's motherboard directly is impossible (ie locked, case-open-detection enabled, etc). Thus, the type of exploitation you are facing is important to consider. As noted above, Thunderbolt hardware is DMA-capable as it exposes the PCI-Express interface. However, Thunderbolt is far more exploitable as it is not only exposed externally without an attacker needing to install an internal PCI-Express device, but also supports hotplugging on more operating systems, resulting in an attacker not being required to reboot the machine before exploitation. Additionally, in many modern mobile computers, physical space within the device is incredibly limited, limiting access for attackers to install malicious PCI-Express devices. However, in a desktop, workstation, or server environment, installing a malicious PCI-Express device becomes very possible as most users do not check for the presence of unknown devices on each system start.

An additional issue with mobile, and often workstation, computing is the need to load the system kernel from an unencrypted device, often susceptible to modification by an “evil-maid” attacker. This issue necessitates my “trusted-loading” phase, and is both incredibly easy to exploit and difficult to detect. Note that this issue is largely not present on server and other long-running systems are there is little need to reload the kernel after boot but very rarely. A common defence against such attacks is the use of SecureBoot to prevent the booting of a kernel other than a pre-programmed one. However, note that there is little preventing an “evil-maid” from simply replacing the TPM/motherboard/entire computer which has the trusted kernel programmed in it with an identical one which has no such limitation.

In addition to the lack of need to repeat the “trusted-loading” phase, servers have a potentially different threat model due to their storing of data for multiple users. For a server to be considered secure, the users who store their data on it may want some kind of assurance that not only a physical attacker with no connection to the system administrator can steal their data, but also a physical attack by a rogue system administrator should as well.

IV. Approach

In order to analyse the security of the proposed system, I will begin by introducing a model for the security of various data-storage locations.

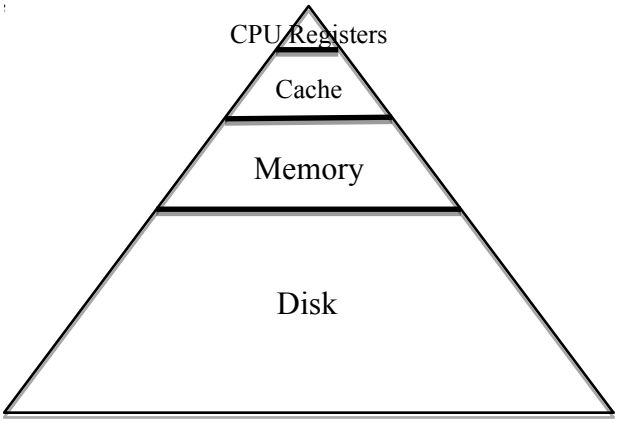

Fig. 1. The hierarchy of data security used to analyse the data an attacker can access

At the lowest level of data security and highest total capacity (see Fig. 1) we have the disk. Because it is nonvolatile memory, an attacker with physical access to a machine can read the disk's contents at any time by simply removing the drive and plugging it into one of their own systems. If its contents are unencrypted, this allows the attacker to gain access to any private data stored on it. Directly above the disk is main memory. Although it is not technically nonvolatile, ColdBoot attacks place it somewhere between nonvolatile and completely volatile memory. In addition to an attacker being able to read the contents of memory after shutdown via a ColdBoot attack, an attacker can potentially read much of main memory using DMA-based attacks. Thus, its security against data being read or even modified is fairly low. Additionally, main

memory is fairly large and generally stores significant amounts of disk cache, making it an ideal target for the stealing of private data.

Above disk and memory comes the CPU's internal memory cache as well as its registers. Unlike main memory, the CPU's cache is cleared upon system restart (it is even cleared in ACPI S1 sleep), making it truly volatile memory. Additionally, neither CPU cache nor CPU registers are available to an attacker unless they are able to run software directly on the machine, and CPU registers of other applications are only readable by the application which set them or the kernel, which is in our TCB. Thus, it is our goal to reduce our private data storage as much as possible to be only in CPU Registers and CPU cache. Note that no peripherals are in this hierarchy as they are all assumed to be either compromised or entirely malicious.

The first step to protecting ourselves from physical attackers is the creation of a secure TCB which can be used to store encryption keys to protect the less secure areas of data storage in our model. Though this is possible by entering a trusted, measured environment each time we wish to perform a secure operation, because we anticipate regularly using this TCB to encrypt and decrypt data, doing this would be prohibitively expensive. Instead in order to create this TCB, we rely on the secure loading of a trusted kernel and our ability to protect the kernel against malicious modification. Thus we are required to properly configure the I/OMMU to protect against malicious devices which might modify the running kernel to perform any action which the attacker wishes, including divulge secrets or other confidential data. Though this is rather simple in statement, it turns out to be rather complicated in practice, requiring several existing solutions be used in combination to protect against various attack scenarios. Additionally, we are required to prevent users from being able to load kernel modules at runtime, as those modules could divulge secrets by running in kernel mode. This has an additionally nice property as inserting a malicious device can no longer trigger the loading of any device driver, some of which may have know bugs leading to exploitation.

After we have sufficiently protected ourselves such that we have a trusted kernel which can be trusted even in the face of physical attackers, we must set up encryption keys within secure areas of data security such that only our kernel can access them. Luckily, x86 provides us with many registers which are only readable/writeable by code running in kernel mode. Thus, during our trusted loading phase we require the user exploit the trusted path from keyboard into CPU to enter a passphrase which will be used for key derivation and stored only in CPU registers. In order to limit the effect of a loss of the trusted path, we enable the user to use a disk drive, such as a USB flash drive, as an additional key storage mechanism, the selected part of which will be included in the key derivation process. Thus, an attacker would have to compromise both paths in order to compromise the system, not just a single path.

In addition to the single-key method described above, I provide a multi-key method for servers. By requiring multiple keys distributed using a standard secret-sharing method, users who store their data on a remote server can ensure that no single, rogue system administrator with significant resources and physical access can compromise private data.

Once encryption keys are safely stored away in CPU registers, we can begin using them to encrypt the less-secure data storage areas. First, and most simply, we encrypt the disk. Because full disk encryption is a popular mechanism on many computers today, doing so using existing infrastructure is quite simple. Additionally, recall that modern disk encryption systems have nearly no performance overhead as the cost to encrypt/decrypt a single block is incredibly low and orders of magnitude lower than the cost to access the disk itself. Secondly, we encrypt main memory. Because modern commodity hardware does not include any mechanism for encrypting main memory, we have to rely on partial encryption and page faults to decrypt memory as it is needed. Some modern processors have significant space in CPU cache, which we can attempt to use to avoid storing much, if any, unencrypted data in main memory. Specifically, some Intel Haswell processors include 128MB of on-die L4 CPU cache. Sadly, there is no way for the kernel to control this cache, or even

gain insight into which data it is storing using performance counters. To make matters worse, this cache is shared between the CPU and GPU, making it even harder to limit unencrypted memory to that which is in cache. Thus, I instead propose a probabilistic system under which a large percentage of active memory is encrypted. Because there is a large tradeoff between encrypting more memory (and thus gaining more security) and the performance of the system as a whole, the ratio between encrypted memory and unencrypted memory is left configurable for the user.

V. Implementation Details

The implementation of my system can be largely categorized into two sections, the disk encryption and the encryption of main memory.

A. Disk Encryption

In order to build on existing research, I chose to use TRESOR as a starting point for implementation of my system. In this way, much of the disk encryption work was already complete, specifically the storage of keys in the processor's debug registers and prevent applications from accessing them and integrating with Linux' existing disk encryption system were largely implemented. However, the implementation of TRESOR as published was severely lacking in its practicality and needed to be extended to more fully support the set of features described in Section IV. Specifically, the implementation of TRESOR published with the paper included only simple key derivation from one passphrase instead of also allowing the user to input key material using drives and support for secret-sharing between multiple system administrators.

1) Keydevice Support: In order to allow the user to distribute their trusted-path trust among multiple paths, I added support for “keysectors”. A user who wishes to add additional key material to the key derivation function may choose to fill a device with random data data and select a single sector which will be loaded as a part of the key derivation processes. This functions similarly to keyfiles which are employed by many encryption systems, however work at a lower level, removing the need for filesystem handling in the key derivation system. Though the reading of a user-specified sector from a device is not complicated during system boot, doing the same when the encryption keys must be re-derived when resuming from sleep requires some additional legwork. In order to prevent I/O threads from reading or writing to disk while encryption keys were unavailable, TRESOR originally simply prevented any kernel threads from resuming before the key could be re-derived. However, because we need to use those I/O threads for keysector loading during key-derivation, we need a mechanism to wait for the key derivation process to complete. This necessitated adding locks around the key being set in memory and simply using Linux' scheduling features to sleep the processes which attempt filesystem I/O before keys are available. Because of the way the Linux cryptography subsystem is structured, this results in an inability to use Encrypted salt-sector initialization vector (ESSIV) encryption mode, which encrypts the sector number before using it as the Initialization Vector (IV) in the encryption/decryption process. Thus, a second TRESOR cipher is exposed to the kernel cryptography subsystem which automatically encrypts the IV before running through standard CBC encryption/decryption.

In order to use a keysector, a user needs to simply specify the device or partition's identifier (using the same method and allowed device identifiers which are used to specify the root device) to the kernel's boot arguments. Ideally the user will use a device with a UEFI/GPT partition table, allowing them to use a partition's Universally Unique Identifier (UUID), allowing the device to be removed and inserted at will as it will be automatically identified as the correct device by the device-searching code. Then, after being asked for a passphrase using the original TRESOR passphrase dialog code, the user is asked to enter a sector number within the partition or device which will be loaded as the keysector.

Though supporting keysectors themselves is a huge addition for limiting trust in the derivation process, they serve an additional role which is far more important. Namely, they provide users with a two-factor authentication system which allows them flexibility in the key storage mechanism they wish to use. Note that any device which reports itself as a

block device can be used, allowing the user to use any type of custom block device they wish. As a part of implementing the system for myself, I opted to use an entirely custom USB drive which appears to be a simple USB mass storage device to the kernel, but, in fact, has additional logic baked in allowing it to be more clever about when it will and will not divulge its keys. Specifically, if an attacker is able to steal a user's passphrase (eg by looking over their shoulder as they enter it), they can rather trivially brute-force their key by stealing their key storage device and simply testing each sector on the device (to an offline attacker, tens or hundreds of thousands of attempts at a few hundred milliseconds per attempt is not significant). Thus, my key storage device attempts to prevent any kind of mass reading by wiping itself in the event of a read or write to a sector in one of several ranges. Specifically, by storing its data on an encrypted SD card and keeping its encryption keys in on-die nonvolatile memory that cannot be read by anything except the ROM programmed into the processor, and which are cleared when the ROM is modified, the device can protect its data from any reads except those that go through its normal sector-reading handling. As a part of this code, the device checks the sector number and, if it is one of the disallowed ranges, simply clears its encryption keys, locking the data permanently. Though my device is rather simple, a more advanced device could be constructed as there are some basic fingerprints that could be performed to identify the type of machine which the system was plugged into. For example, a system running Microsoft Windows will re-read the MBR sectors multiple times when the device is first attached, allowing the device to identify the host and wipe itself before any other reads can complete. Also note that my particular device is designed to appear to an uninformed observer to be a simple flash drive, who, in the case of an attacker attempting to steal private information, might simply attempt to image the device, clearing it in the process.

2) Keydevice Secret-Sharing Support: As mentioned, in order to support servers which store data for users who wish to trust more than a single key-holding system administrator, my modifications to TRESOR include the ability to read secrets which are secret-shared across multiple key devices. In order to accomplish this, I searched for a trivially auditable and trusted implementation of Sharmir's Secret Sharing (SSS), however, unable to find one, I implemented a simple one myself, splitting each secret one byte at time over the simple GF(2^8) curve used in Rijndael (AES). This has the effect of limiting the total number of shares possible to 256-1 = 255, however this seems to be a reasonable tradeoff in exchange for the simplicity and, thus, auditability of the secret sharing implementation. To use secret sharing, administrators need to use a simple userspace program to split a sector into parts using SSS and then write the resulting parts to separate devices at sectors each administrator can choose. Note that if each part has a distinct non-0 first byte and the administrators wish to require all devices to recreate the secret instead of a quroum n-of-m (with n < m), the administrators may simply use random values they each choose independently. During kernel boot, the kernel then needs to be informed of all of the possible devices which may contain keysectors. It will then simply repeat the keysector-loading code until it has enough sectors to recreate the secret. Because each share requires an additional byte (the X coordinate of the share on the polynomial), each secret is one byte smaller than regular shares as they must both fit inside exactly one sector on disk. Thus, in order to allow administrators to split keys when a system has already been encrypted, an extra, arbitrary, byte can be input as hex during key derivation.

3) General Updates: In addition to the above changes, I made a few small tweaks to TRESOR to make it more practical and to slightly protect the user from themselves. First of all, TRESOR allowed arbitrarily low passphrase length, which I limited to at least 8 characters. Secondly, as noted in Section III, I inform the user in case they are booting without sufficient protection from DMA attacks by requiring both a Intel TXT measured boot environment as well as the enabling of the I/OMMU period.

B. *Main Memory Encryption*

In addition to upgrading the security of the TRESOR disk encryption significantly using various software methods in combination with custom hardware, I implemented memory encryption in Linux. This allows the kernel to encrypt the majority of main memory and rely on page faults to be notified when individual pages need to be decrypted for access. While the theory of simply encrypting pages and marking them non-resident in page tables is quite simple, the reality is that the Linux Memory Management subsystem is incredibly complicated and accomplishing this without subtle race conditions resulting in memory corruption is incredibly difficult. The only reference to main memory encryption in academic literature which runs on commodity hardware instead of relying on custom hardware to perform the encryption is in a paper entitled CryptKeeper [9]. This paper focuses largely on exploring the possibility of memory encryption in Linux by benchmarking an implementation thereof and analysing the performance tradeoff in herit in memory encryption on modern commodity hardware. As a proof-of-concept paper, the source to CryptKeeper was never released and disabled a number of practical features (such as PAE and 64-bit mode) which further make the use of CryptKeeper on a real-world system impractical. In non-academic literature, there is also some reference to using a memory-backed encrypted SWAP partition to force the kernel to encrypt a user-configurable amount of memory. Note that in the second case the manipulation of the kernel is even worse as the kernel's normal attempts at avoiding the overuse of SWAP are then applied to a large section of main memory, hurting the performance of disk caches and potentially other applications. Both of these implementations, however, suffered from a common flaw, namely the storage of encryption keys in the same main memory which they were being used to encrypt. Thus, in the case of either a DMA-based or a ColdBoot attack the encrypted memory was trivially decryptable with some additional effort on the part of the attacker to interpret the decrypted memory.

My implementation relies on existing kernel infrastructure as much as possible, limiting both the number of lines of code required to implement it (in the process decreasing chance for bugs, decreasing the size of the TCB) and exploiting the existing tuning of the kernel's performance to increase performance of the system with encrypted memory. By relying on the kernel's active/inactive lists, which it uses to determine which pages to deallocate by writing them to disk or swapping them out to disk to determine which pages to encrypt, I build on all of the existing work which has gone into tuning Linux' memory management subsystem. Additionally, because the Linux inactive/active list rebalancing is based on the process' Control Group (cgroup) and utilizing a simple patch which places each parent process in its own cgroup, pages of processes which have more total pages allocated are encrypted more than pages which are allocated to others. This, in turn, ensures that even processes which have very little memory also get encrypted, ensuring that small processes which store secret data are not simply ignored by the page encrypter due to large processes that allocate but do not use large sections of memory.

In order to perform the encryption of memory itself, I repurposed the kswapd process, which normally rebalances the active/inactive lists and swaps out pages as main memory becomes full. In addition to its usual tasks, under my system it also encrypts pages as it moves them from active to inactive lists. Normally, kswapd sleeps until there is limited available free memory, allowing it to not cause a performance impact needlessly moving pages from active to inactive lists. Under my system this would be unacceptable, so it instead attempts to encrypt 1/4th of the total pages which would be required to have the user's desired ratio between unencrypted and encrypted memory every 1/10th of a second. Note that this approach, instead of ensuring strict adherence to the user's desired ratio, allows the security provided by the memory encryption to float as necessary when the system enters higher or lower load (as processes which need their entire address space will decrypt their 1/4th every 1/10th of a second as their memory is encrypted). Additionally, it places an upper bound on the overhead of the memory encryption. This allows the system to be far more performant than other, stricter, alternatives while still providing the

desired security on a desktop system where CPU load is generally very low.

VI. Analysis

In this section I will largely analyze the performance and practicality of the memory encryption part of my work. Because the disk encryption section of the work functions nearly identically to existing disk encryption systems and the security model thereof has already been thoroughly explored in previous sections, it does not merit further analysis.

For the memory encryption to be considered practical, it has to perform well. Specifically, it cannot have significant overhead when compared to a system running with fully unencrypted main memory. In order to adequately benchmark the performance of applications as their memory is encrypted, I designed a simple benchmark which allocates a large section of memory, initializes it to known values, and then verifies its value repeatedly. In this way, memory can be encrypted as the process steps through its memory and then will have to be decrypted as the process enters its next iteration. The results of the benchmark can be seen in Fig. 2. As can be seen, the total time taken does not increase greatly as the majority of the program's time is spent verifying the contents of memory and not encrypting encrypted pages, a huge win for performance. To better analyze the time spent decrypting pages, we look at the time spent in system mode (including handling page faults and decrypting pages). After the initial jump in system time to have any encrypted memory, the time spent in system mode between a ratio of 0.2 to 5 only increased 83%.

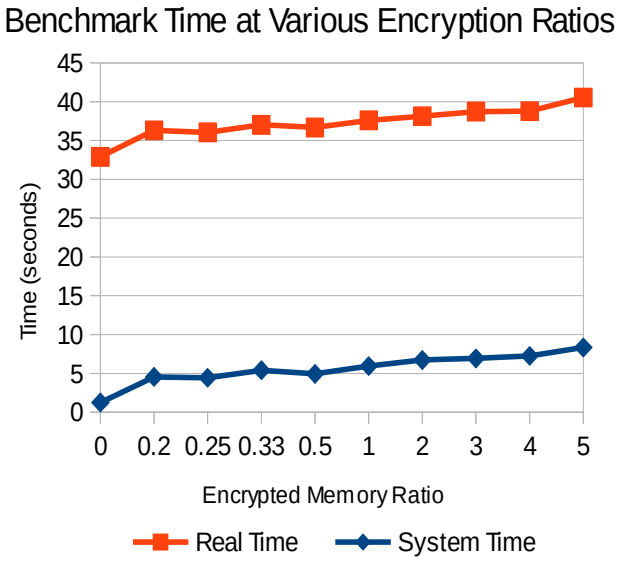

Fig. 2. The hierarchy of data security used to analyze the data an attacker can access

VII. Future Work

While this research moved the idea of data confidentiality on compromised commodity hardware forward significantly, it should still be considered early research and there is a lot of work left to be done in the area. First of all, additional research into the effectiveness of I/OMMUs needs to be continued. There have been numerous examples of serious bugs in implementation of I/OMMU and related technologies [10]. While these bugs are addressed as they are discovered, their presence on a regular basis indicates a lack of security in the system as a whole. More importantly to this research, however, would be research into the decrease in trust required during the initial loading phase. One potential approach to this would be to utilize the TPM's ability to measure the running system to authenticate the kernel to the key storage device before the key storage device will divulge any information. After the kernel has been authenticated, one could create a secure, encrypted channel from the kernel to the key storage device. By exchanging the keys only over this channel, an attacker would no longer be

able to intercept the keys by monitoring the channel (ie the USB bus).

VIII. Conclusion

In this work, I propose a new system which is both practical and drastically increases data security against an attacker with significant resources. After extending the security of existing disk encryption systems drastically against DMA-based attackers, I implemented a novel memory encryption system for commodity x86 hardware, greatly increasing data privacy for users against physical attackers. In addition, I provided a system for a group of system administrators to effectively share the disk encryption keys of a shared system.

Acknowledgment

Mike Reiter provided significant input to the design and discussion of security tradeoffs used in this system.

Peter A. H. Peterson provided the details of his CryptKeeper implementation, providing advice on the implementation of memory encryption in Linux.

References

[1] For example, see the presentation given at BlackHat which available at he presentation on a similar topic for PCs available at https://www.blackhat.com/presentations/bh-usa-07/Heasman/Presentation/bh-usa-07-heasman.pdf

[2] For example, see the implementation of FireWire-based authentication skipping bypassing available at http://www.breaknenter.org/projects/inception/.

[3] For example, see the presentation on Mac EFI rootkits, given in 2012 available at http://ho.ax/downloads/De_Mysteriis_Dom_Jobsivs_Ruxcon.pdf

[4] Bystrov, Stewin. “Understanding DMA Malware”

[5] Sang, Lacombe, Nicomette, Deswarte. “Exploiting an I/OMMU Vulnerability”

[6] Halderman, Schoen, Heninger, et al. “Lest We Remember: Cold Boot Attacks on Encryption Keys”

[7] Mueller, Freiling, Dewald. “TRESOR Runs Encryption Securely Outside RAM”

[8] Blass, Robertson. “TRESOR-HUNT: Attacking CPU-Bound Encryption”

[9] Peterson. “Cryptkeeper: Improving Security With Encrpted RAM”

[10] There are, for example, multiple examples of Intel TXT (a requirement for our secure MMU) being broken by a research team at http://theinvisiblethings.blogspot.com/