HyperCrypt: Hypervisor-based Encryption of Kernel and User Space

Johannes Götzfried, Nico Dörr, Ralph Palutke, and Tilo Müller
Department of Computer Science
Friedrich-Alexander University of Erlangen-Nuremberg
Email: {johannes.goetzfried,nico.doerr,ralph.palutke,tilo.mueller}@fau.de

***Abstract*—We present HyperCrypt, a hypervisor-based solution that encrypts the entire kernel and user space to protect against physical attacks on main memory, such as cold boot attacks. HyperCrypt is fully transparent for the guest operating system and all applications running on top of it. At any time, only a small working set of memory pages remains in clear while the vast majority of pages are constantly kept encrypted. By utilizing CPU-bound encryption, the symmetric encryption key is never exposed to RAM. We evaluated our prototype running a standard Linux system with an nginx web sever. With the default configuration of 1024 cleartext pages, successful cold boot attacks are rendered highly unlikely due to large caches of at least 4 MB in modern CPUs. The performance overhead of nginx is raised by factor 1.37 compared to a non-virtualized system.**

***Keywords*-memory disclosure attacks, cold boot, memory encryption**

I. Introduction

Today's computers are amongst many tasks used to process, distribute, and store sensitive information. Protecting these sensitive information has always been an important goal for many software and hardware solutions. While the problem of protecting sensitive data is largely solved for network transfers and permanent storage like hard disks, volatile memory still remains at risk. For SoCs and dedicated hardware, full memory encryption rarely exists [1], but standard hardware used by end users still cannot be protected with memory encryption.

When a computer is turned off, data in main memory is not lost immediately, but instead gradually fades away over time. This fact can be exploited by certain attacks such as cold boot attacks [2], attacks using the Firewire interface [3] and other DMA attacks that are capable of extracting RAM contents [4]. To effectively protect RAM, one of the most sensitive parts of modern computers, including private RSA keys, disk encryption keys, online banking credentials and user logins, we decided to encrypt RAM on hypervisor-level to be able to protect kernel space and hence, to ensure maximum compatibility and security.

Just recently Intel announced its *Software Guard Extensions* (SGX) [5] which amongst others also encrypt physical memory of software running within so-called *enclaves*, showing the importance of physical security for standard hardware today. Enclaves, however, are restricted to a static size and cannot dynamically grow once they have been created. Also software within enclaves needs to be written for SGX as the restricted enclave environment, for example, does not permit syscalls. Consequently, SGX cannot provide compatibility with existing applications that should be protected against physical memory disclosure. Furthermore, sensitive data is not only stored within the address space of a given application, but also partially within kernel space due to network and device buffers. Kernel space cannot be protected by Intel's SGX as enclaves cannot be entered in kernel mode but only in user mode [6]. Thus, SGX is only suitable to protect a limited set of rewritten applications but not to prevent physical data exposure of kernel and application memory in general.

A. Our Contribution

In this paper, we present HyperCrypt, a hypervisor-based encryption of the guest's main memory to protect against physical attacks such as cold boot and DMA attacks, while being transparent for applications and the OS kernel running on top of it. In detail our contribution is:

- At any time, the vast majority of main memory is kept encrypted. Only a small, configurable working set of pages remains unencrypted. These pages are kept within caches with a high probability.
- We demonstrate the practicability of our approach with a prototype implementation running a standard Linux system with an nginx webserver.
- The prototype of HyperCrypt leads to an overhead of 37% for the nginx webserver under heavy load.
- Utilizing CPU-bound encryption, the encryption keys used by HyperCrypt are never exposed to RAM.
- HyperCrypt, as a hypervisor-based solution, is fully transparent to all applications as well as the operating system kernel.

HyperCrypt is free software published under the GPL v2. It is available as an open source patch building on top of BitVisor [7] at https://www1.cs.fau.de/hypercrypt.

B. Related Work

In recent years, work that addresses memory disclosure attacks not only by keeping encrypted data but also by computing on the encrypted data led to the concept of cryptographic processors. Theoretic approaches such as homomorphic encryption [8], [9] and fully homomorphic encryption [10] are not yet practical, while other suggestions encrypt the communication channel to peripheral and storage devices [11], [12] which requires modified standard hardware. Futhermore, only tamper-resistance and authenticity has been covered and confidentiality has been excluded [13].

Solutions targeting specific problems, such as the protection of full disk encryption keys [14], can protect only a small fraction of sensitive data but not memory as a whole. Symmetric register-based encryption on operating system level [15], [16], [17] and hypervisor level [18], [19] as well as cache-based solutions [20] emerged in the field of CPU-bound encryption schemes. Solutions protecting asymmetric keys, which might be target to cold boot attacks as well [21], [22], include register-based implementations [23], [24] and implementations based on hardware transactional memory [25]. Nevertheless, all those solutions have in common that only a small portion of sensitive data, which is usually an encryption key, is protected.

A survey of full memory encryption solutions [26] shows theoretical approaches [27] as well as practical implementations like swap space [28]. For embedded hardware, full memory encryption solutions exist [1] as well as for the Linux kernel [29], [30]. To the best of our knowledge there is no hypervisor-based solution aiming at encrypting the whole physical address space of a guest yet.

C. Outline

The remainder of this paper is structured as follows: In Section II, we give necessary background information about the memory management of BitVisor and CPU-bound encryption. Readers familiar with these topics may safely skip this section. In Section III, we present the HyperCrypt architecture and provide some implementation details regarding the challenges that need to be solved with implementing HyperCrypt. In Section IV, we evaluate HyperCrypt regarding performance and security while considering benchmark tests as well as a real world application. In Section V, we deal with future research directions, and in Section VI, we finally conclude with a summary of our work.

II. Background

This section describes necessary building blocks that HyperCrypt relies on. In Section II-A, details of BitVisor and its memory management are presented, while in Section II-B, the concept of CPU-bound encryption is introduced.

A. BitVisor Memory Management

BitVisor [7] is a thin hypervisor based on Intel VT-x and AMD-V for enforcing I/O device security. It is a so called *parapass-through* hypervisor meaning that only a small set of hardware accesses, for which security should be enforced, are intercepted by the hypervisor while other accesses are passed through to the hardware. The advantage of this concept is a dramatically reduced code size in comparison to traditional VMMs such as XEN or KVM. BitVisor with enabled ATA parapass-through driver, for example, needs only 21,400 lines of code and is capable of performing transparent full disk encryption. Naturally, thin hypervisors have limited functionality compared to real VMMs; most notably, BitVisor allows running a single VM on top of it, eliminating the components for sharing and protecting system resources amongst different VMs.

To manage memory within a hypervisor, there are basically two widely known concepts for x86. The first concept, known as *Shadow Page Tables* (SPT), requires a hypervisor to deny the guest operating system write access to its own page tables. Instead, write accesses are trapped and handled in the hypervisor after applying certain mapping rules. The advantage of this approach is that, apart from basic virtualization support, no other hardware support is needed. The disadvantage, however, is that handling all page table changes within the hypervisor has a serious impact on performance.

We therefore decided to implement HyperCrypt based on the modern approach of implementing memory management within a hypervisor called *Second Layer Address Translation* (SLAT). SLAT requires support from the processor such as, for example, the *Extended Page Tables* (EPT) feature provided for Intel CPUs. With EPT, virtual addresses are first translated to guest physical addresses with the help of regular page tables managed by the guest. These guest physical addresses themselves are then translated to host physical addresses with the help of the EPT managed by the hypervisor. Consequently, the guest has a faked view of physical memory which can be provided by the hypervisor by manipulating the EPT. For our implementation of HyperCrypt, we mislead the guest operating system regarding the number of pages that are currently accessible in real physical memory by targeted EPT manipulations.

B. CPU-bound Encryption

To prevent cryptographic keys and key material, like intermediate states during encryption and decryption, from being stored in RAM, we use the principle of CPU-bound encryption. In particular, HyperCrypt is built upon BitVisor and uses some logic from TreVisor [18]. TreVisor is a patch for BitVisor that provides a cold-boot resistant AES implementation which is primarily designed for full disk encryption. Only CPU registers are used to store the encryption key as well as any intermediate state like the AES key schedule. To prevent the intermediate states from entering RAM, the AES cipher is executed inside an atomic section within the hypervisor, i.e., interrupts are disabled and the operating system is paused while the cipher runs on a particular CPU. According to the authors of TreVisor, tests have shown that the atomic sections are too short to affect the system interactivity by disabling interrupts.

TreVisor stores the AES encryption key inside the four x86 debug registers and prevents access to these registers for any other purpose. Although the cipher runs within the hypervisor, hardware breakpoints are not usable by the guest operating system. Every access to the debug registers by the guest is intercepted by the hypervisor. For write accesses, the value to be set is copied to a shadow area in memory and for read accesses, this value is returned back. To the guest operating system it seems the registers are functional, but the processor is not able to use real

hardware breakpoints as the registers are not set to the value stored within the shadow area. As most debuggers use software breakpoints by default, which are implemented by overwriting code with dedicated instructions, this is mostly a limitation for watchpoints.

For every input block, the AES computation is done within the SSE registers of the x86 architecture. Prior to the encryption or decryption of an input block, the AES key schedule is recalculated for this block. Due to the use of the AES-NI instruction set, however, the performance degrade is acceptable. After the encryption or decryption of an input block, all SSE and general purpose registers are cleared which effectively wipes any sensitive intermediate information of the AES computation. The wiping operation and the atomicity of the AES computation within the hypervisor ensure that parts of the key material or intermediate states never enter RAM.

To bootstrap TreVisor, the user has to type a password at an early stage during start-up right after the hypervisor is loaded by the boot loader. The password is used to derive the key which then is stored within the debug registers. We patched TreVisor such that a random key is generated during start-up instead of having to enter a password because for memory encryption, keys can be arbitrary. The key within the debug registers is protected against physical attacks like cold boot [31] because debug registers are cleared during a CPU reset. An attacker would require logical access to a machine and must be able to execute hypervisor code in order to read out the key.

III. Design and Implementation

In Section III-A, we present a high-level view of the HyperCrypt architecture and in Section III-B, we explain implementation details like the handling of DMA transfers and different configuration options for HyperCrypt.

A. HyperCrypt Architecture

HyperCrypt encrypts host physical pages and automatically decrypts those pages which are currently accessed by the guest OS or an application running on top of the guest OS. To this end, the EPT fault handler of BitVisor is extended to support the dynamic encryption and decryption of physical pages. As BitVisor is a thin hypervisor with only a single guest OS running, the EPT mapping within BitVisor is simple. Almost every guest physical address is mapped one-to-one to the corresponding host physical address with the exception of the physical pages containing the hypervisor itself. These pages are hidden from the guest operating system by hooking the BIOS call for getting the system memory map and re-mapping the addresses within the EPT is not allowed.

Applications running on top of the guest OS, as well as the guest OS itself, can access pages only if a corresponding EPT entry exists. If no such entry is found, a trap into the hypervisor occurs and the fault is handled by the EPT fault handler. HyperCrypt leverages the one-to-one mapping mechanism by dynamically adding and removing entries from the EPT when physical pages are encrypted and decrypted, respectively.

1) Sliding Window: In an ideal world, only the single page that is currently in use by the guest OS is left decrypted while all remaining pages are encrypted. Hence, the EPT only contains a single entry and access attempts of other pages cause a fault followed by a trap into the hypervisor. The hypervisor could then encrypt the current page, remove the current EPT entry, decrypt the correspondent requested page, and add an EPT entry for the new page. Although this approach guarantees maximal security, because of minimal expose time of a given page, we decided against it for two reasons. First, one single page is not sufficient for the guest OS to run instruction and data fetches from two different pages in parallel, and second, trapping into the hypervisor each time a different page needs to be accessed decreases the performance dramatically such that a protected system needs days to boot up.

Instead of encrypting all but one page, we introduce a *sliding window* which keeps references to all pages that are currently kept in clear. Every page not referenced from within the sliding window is always kept encrypted. Consequently, instead of keeping only the currently used page in clear, the last n pages that have been accessed are kept in clear. The size n of the sliding window is a system wide constant and can be defined when building HyperCrypt; the default value is 1024. Note, that 1024 pages correspond to 4 MB in size which is less than most cache sizes of modern CPUs and therefore even the pages that are currently kept in clear are likely not exposed in clear within RAM.

2) HyperCrypt Workflow: As depicted in Figure 1, the basic HyperCrypt workflow consists of two main parts. First, the extended EPT fault handler which on demand decrypts pages for which a fault has occurred, and second, the sliding window mechanism which after each page fault checks whether there are more than n cleartext pages and potentially re-encrypts pages referenced by the sliding window if the limit of n pages is exceeded. In detail, the extended EPT fault handler decrypts a page and then makes it accessible to the guest by adding an entry for this physical page to the EPT. The sliding window mechanism first adds the newly decrypted page to the sliding window if the page fault was caused by an encrypted page. It then checks whether the limit is exceeded and which page to remove from the window. The chosen page is removed from the EPT table, taken out of the sliding window and finally encrypted.

Hooking the EPT fault handler within BitVisor is sufficient for implementing HyperCrypt, because the guest system is started with an empty EPT table in the hypervisor. Thus, for each first access of a page, the EPT fault handler is called and the sliding window mechanism inserts the page into the sliding window. To decide whether a page needs to be decrypted within the EPT fault handler or whether the fault just occurs because of being first accessed, a bit marks the page as currently being encrypted or in cleartext for each physical page.

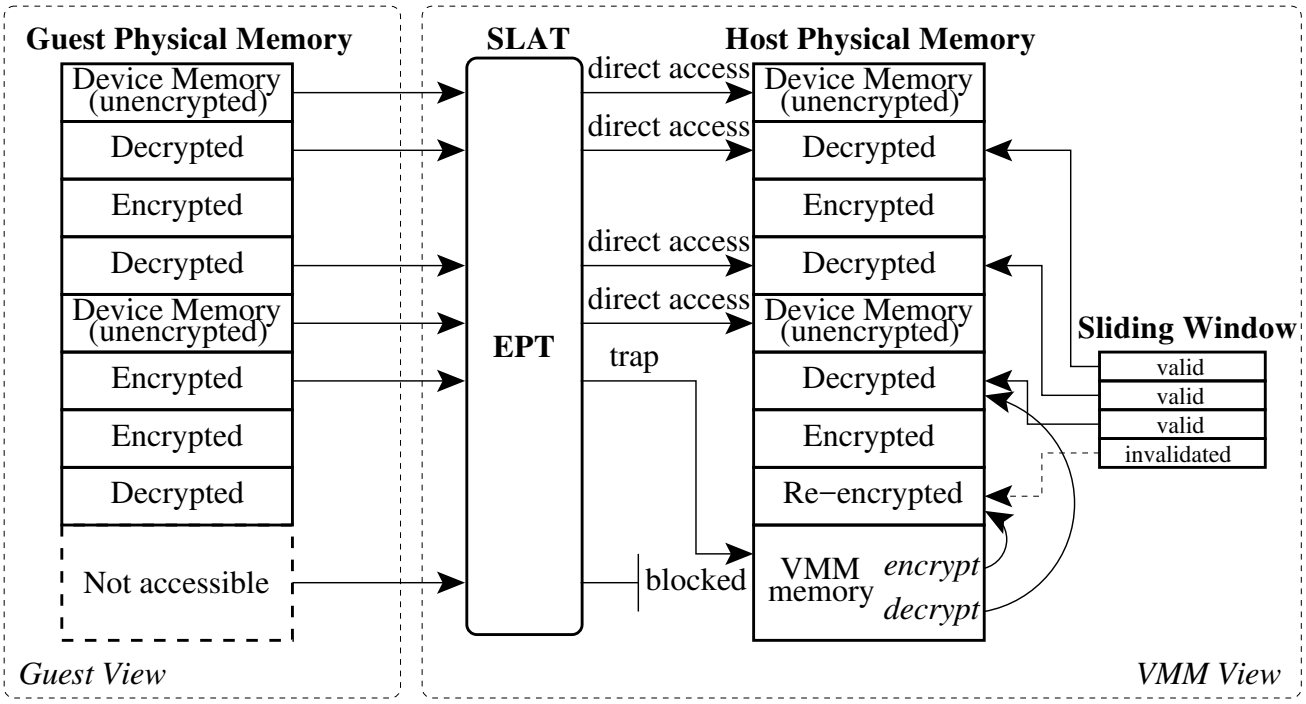

Figure 1. Overview of HyperCrypt's Workflow

3) HyperCrypt Challenges: While the concept of HyperCrypt may sound simple, a lot of challenges had to be solved to transparently encrypt main memory out of a thin hypervisor. Since BitVisor is a parapass-through hypervisor, it does not know about the guest's memory layout and how the guest OS uses certain memory regions. We cannot simply encrypt the whole physical address space, but instead device memory and DMA buffers need to be excluded, for example. To exclude device memory, we check each memory access that causes a fault against the system memory map provided by the BIOS. If a region is accessed which does not belong to usual system RAM, the access is mapped through with a one-to-one entry in the EPT and the page is not considered for encryption. Handling accesses to DMA buffers is more complicated, because the hypervisor does not know which areas have been allocated by the guest OS to be passed to devices as DMA regions. Encrypting DMA buffers causes problems, because the devices would read encrypted data. During writing, the devices would write cleartext data which the hypervisor then potentially decrypts for the operating system leading to unpredictable results. Our approach for mitigating this problem involves using device drivers provided by BitVisor and will be further explained in Section III-B.

For the encryption itself, we use a variant of the TRESOR cipher [15], [18] which behaves like AES-128. We further chose the XEX mode of operation with the initialization vector which is needed to generate the tweak set to the physical address of the page that is about to be encrypted. The physical page address is a reasonable choice because it is unique and cannot be forged by the guest operating system as host physical addresses are only used within the hypervisor.

B. HyperCrypt Implementation

The EPT fault handler is the point of choice to start implementing HyperCrypt. As the guest can only access pages that have an entry within the EPT and every other access results in a page fault, the EPTs can be manipulated to only hold the currently decrypted memory pages while encrypted pages are never inserted into an EPT. When the guest wants to access a memory page that is currently encrypted, the MMU cannot find an entry for this address in the EPT and issues a page fault which is handled by the hypervisor. The hypervsior then decrypts the page before adding it to the EPT and returns control to the guest which will repeat the previous memory access. This memory access succeeds because the appropriate mappings exist.

1) Page Management: In BitVisor, the EPT is empty at boot time and is subsequently filled page by page using the page fault handler. HyperCrypt stores the current status (encrypted or clear) of each physical page frame using a single bit within a bitmap. When a page fault occurs, the bitmap is used to check whether this fault was caused since the page was accessed for the first time or since it is currently encrypted and therefore not referenced by the EPT. In the latter case, the page is decrypted using the TRESOR cipher and added to the sliding window. If the page is accessed for the first time, it is added to the sliding window as well to ensure that it will be encrypted at some point in time. The sliding window contains structures of decrypted pages which store the physical addresses of these pages. In addition, these structures contain information whether or not the decrypted page is actually mapped

within the EPT. This information is needed because the hypervisor itself is allowed to access encrypted pages, too, which get first decrypted. These pages, however, are not part of the EPT as the hypervisor does not use the EPT but rather accesses the pages directly via its own page tables.

To ensure that only a certain previously defined maximum number of pages are decrypted at the same time, the number of decrypted pages, that is the number of pages referenced by the sliding window, is compared to this limit each time after a page has been decrypted. If the limit has been exceeded, the first page within the sliding window is encrypted again and removed from the EPT if necessary. As in standard BitVisor no entry is ever removed from the EPT, the functionality for removing pages from the EPT had to be implemented first.

The current strategy for choosing the page which first gets encrypted again is basically first-in-first-out (FIFO). However, for more recent CPUs, we also implemented a better strategy. Recent Intel CPUs now support *accessed* and *dirty* bits within EPTs which allow to use a second chance algorithm. With second chance, before encrypting a page, the pages within the sliding window are checked for the accessed bit. If the bit is set, the page was accessed recently and therefore gets a second chance. The accessed bit is then cleared and the page is added to the back of the list again. This process continues until the first page without an access bit set within the sliding window is found. This page is then encrypted instead of just using the very first page within the sliding window. The second chance algorithm performs better if there are pages that are accessed frequently, because these pages get almost never encrypted. On the other hand, if a lot of different pages are accessed constantly, the second chance algorithm introduces additional overhead by iterating over the sliding window.

BitVisor has the capabilities to map and access pages within the guest memory, and thus, it must be dealt with that encrypted pages can be accessed by the hypervisor. Consequently, the mapping functions used by BitVisor are modified in a way such that mapped pages are checked for encryption before adding them to the hypervisor page table. This is particularly important for the BitVisor drivers as they usually map guest pages to copy content from the guest to pass it to devices.

2) Device Memory: When dealing with RAM encryption, an important part is to consider which different kinds of memory regions there are. Besides user data, program code, and kernel level data, parts of the memory are also used to communicate with devices like hard disks and keyboards. Some devices are, for example, controlled using memory mapped I/O, where the operating system writes to specified memory areas which are then read by a device and interpreted as a command. If the entire memory used by the guest OS should be encrypted, these memory mapped regions have to be excluded. Otherwise, the encryption routine would write to these memory areas and send randomized commands to the devices. To check which regions to exclude, we use the memory map provided by the BIOS. This map is obtained by BitVisor at an early stage to check for available memory regions.

Before checking for encryption within the EPT fault handler, it is additionally checked if the address that causes a fault is part of a memory region marked as *available* in the memory map. Available means that the respective page can actually be used by the system and is not already reserved for any device. If the faulted address points to memory within a reserved region, the extended EPT fault handler will not be called and the page will be added to the EPT directly. For future accesses, this device page is excluded from the encryption process altogether and remains permanently mapped within the EPT.

3) DMA Buffers: The most critical part of memory that has to be considered when encrypting data stored in RAM are DMA buffers. BitVisor allows the guest to directly communicate with devices and therefore the guest directly initiates DMA transfers. The guest just passes an address of a DMA buffer to a device and then proceeds with other operations. By executing other instructions, other pages are mapped and decrypted by the hypervisor which causes pages from the DMA buffer to be encrypted, potentially before the DMA transfer is completed. A device would then, for example, write unencrypted data to encrypted memory which results in unpredictable data once the page gets decrypted.

To deal with this issue, the BitVisor drivers have been used to mitigate this problem. BitVisor provides drivers to intercept commands issued to devices to transparently implement full disk encryption and tunnel ethernet packets through a virtual private network, for example. These drivers perform the actual DMA transfer, meaning that if data should be provided for a device, the driver uses BitVisor's internal mapping functions to map the guest memory, copy the data from the guest and finally send the data to the device. With HyperCrypt, we hook all available mapping functions and ensure that data gets decrypted properly before being sent to the device. Receiving data from a device works similar, but instead the data is encrypted by HyperCrypt before being written to guest memory.

4) Booting HyperCrypt: For booting HyperCrypt, we basically provide two options: A user password to derive the encryption key, or automatically generating a random key on each boot. The *userpassword* option can be used to define if the user wants to enter a particular key for encryption. In that case, the user is asked to enter a password at boot time, which is used to generate the TRESOR encryption key stored in the debug registers. Otherwise, the key used for the encryption is generated randomly. The random number generator of the *Trusted Platform Module* (TPM) is used to generate 256 bits of key material consisting of the 128 bit AES key and the 128 bit XEX tweak key. The default setting is to use a random key as the key only has to be remembered until a system is rebooted. If the user, however, does not trust the random generator of the TPM, the password method can still be used.

When configuring HyperCrypt for the first time, the size of the sliding window, which defines how many pages are unencrypted at the same time, has to be fixed. This is a "security vs. performance" parameter because a higher number causes more of the RAM to be exposed while the decryption routine has to be called less frequently. As one page is 4096 bytes is size, setting the value to 512 results in 2MB of RAM being exposed, 1024 results in 4MB and so forth. Although, 4MB of exposed data could contain a lot of sensitive data, it is very unlikely that this data can be acquired by physical attacks on main memory due to large CPU caches nowadays. The size of the sliding window should be always chosen to expose less data than the size of the cache, and hence we set the default value to 1024 pages as a conservative choice and a reasonable trade-off between security and performance.

IV. Evaluation

In this Section, HyperCrypt is evaluated regarding performance (Section IV-A) and security (Section IV-B). For our evaluation, we consider widely accepted benchmarks as well as a real world application.

A. Runtime Performance

In this Section, HyperCrypt is evaluated regarding performance. First, we run the SPECINT2006 benchmark on top of HyperCrypt using a standard Linux distribution and secondly, we measure the average reply rate of an nginx webserver using HyperCrypt. All evaluations have been performed on a standard desktop PC with Intel Core i7-2600 CPU and 8GB of RAM running a Debian GNU/Linux 7.8 operating system.

1) SPECINT2006 Benchmarks: Naturally, a huge performance overhead is created by encrypting memory pages, and additionally HyperCrypt suffers from a significant performance drawback due to context switching between the hypervisor and the guest OS when a page is encrypted or decrypted, respectively. To have verifiable performance results, we decided to use the standardized SPECINT2006 benchmark suite.

These tests were performed with our default sliding window size of 1024. Additionally the same tests were executed with a plain Linux system and standard BitVisor running Linux. However, the overhead of Linux on top of standard BitVisor over Linux without hypervisor is neglectable and therefore in Figure 2, we show the overhead of HyperCrypt over a standard Linux system running SPECINT2006.

The overhead varies a lot depending on the chosen test which most likely depends on the data locality involved in the different tests. Altogether the overhead factor of HyperCrypt varies between 15 and 148 with three outliers with an overhead factor of 1.15, 388, and 718 respectively. The bzip2 test which is more memory intense than I/O intense has an overhead of 15.8 which is the second least of all tests. The second chance page replacement algorithm gave a 19% performance boost over the standard FIFO algorithm.

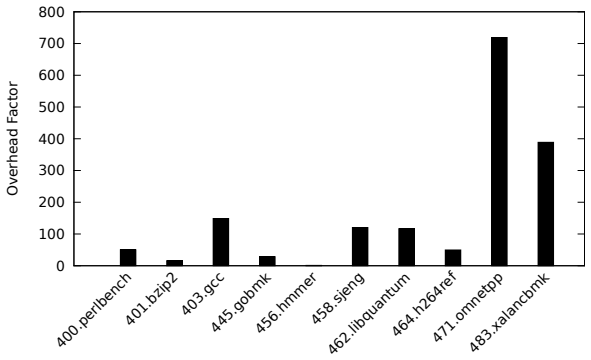

Figure 2. Performance overhead of HyperCrypt compared to a native Linux system using SPECINT2006.

2) HTTP Server Performance: While the SPECINT2006 suite is a standardized benchmark suite that makes a comparison between other setups easy, we also want to show how HyperCrypt performs in a real world scenario. One example, where HyperCrypt could be used is a secure web server that should be protected against physical attacks on main memory. We benchmarked the widely used nginx web server [32] on top of HyperCrypt using a standard Linux. To measure the performance, we used the program Httperf [33] on a computer directly connected to our test machine. Httperf sends various requests to a given host address and evaluates the response time that the web server needs to send back the response. Httperf was configured to send 10,000 requests with a timeout of five seconds.

Table I shows the absolute average reply rates and connection times for HyperCrypt configured with different sliding window sizes. As expected, reply rates are lager if more pages are allowed to reside within RAM in clear. For our default sliding window size of 1024, the reply rate is not much lesser than for a plain Linux system. The overall overhead for different sliding window sizes is visualized in Figure 3.

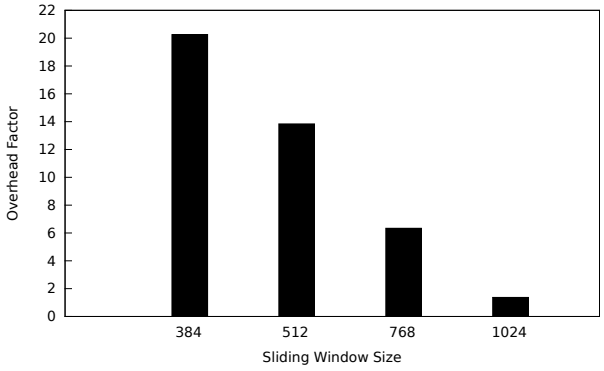

Figure 3. Overall performance overhead of the nginx web server running on top of HyperCrypt with different sliding window sizes.

The overhead for our default sliding window size of 1024 is just 37% and has been calculated based on the runtime. For even more secure settings with less decrypted pages, the factor grows to 6.34 for 768 pages and 13.83 for

Sliding Window Size	No Encryption	384	512	768	1024
Test duration (s)	81.0	1,641.3	1,120.8	513.5	110.8
Avg. reply rate (replies/s)	123.4	6.1	8.9	19.5	90.2
Avg. connection time (ms)	8.1	164.1	112.1	51.3	11.1

Table I
AVERAGE REPLY RATES AND CONNECTION TIMES FOR AN NGINX WEB SERVER RUNNING ON TOP OF HYPERCRYPT WITH DIFFERENT SLIDING WINDOW SIZES.

512 pages. 384 pages caused the web server to reply to the requests 20.26 times slower. As modern caches are usually much larger than 4MB, the default sliding window size of 1024 is a good trade-off between security and performance.

With an overhead of only 37% for the default sliding window size of 1024, the evaluation shows that a real world example, like a web server, performs a lot better than the integer benchmark suite SPECINT2006. A web server is probably far more I/O intense than most tests within SPECINT2006. As our primary goal was to provide physical security for severs, this result shows a practical scenario for using HyperCrypt.

B. Practical Security Evaluation

Besides performance, the most important aspect of HyperCrypt is of course whether it guarantees the security we claim. In this section, we show the effectiveness of HyperCrypt and how the sliding window size affects data exposure in general.

1) Effectiveness: To verify that HyperCrypt mitigates data exposure, we filled a certain memory region with a repeated randomly generated pattern and afterwards searched for this pattern within physical memory. If HyperCrypt works as intended, the pattern should at maximum be only found as often as it fits within all pages referenced by the sliding window. All other pages should be either encrypted or should not have been accessed at all, thus not containing the pattern.

We wrote a small program which runs on top of standard Linux and uses `/dev/urandom` to generate a random 128 bit pattern each time it is started. The program then allocates four gigabytes of memory and fills the allocated memory area with the just generated pattern. Afterwards, the program issues a `vmcall` and passes the pattern as a parameter to the hypervisor. The hypervisor searches for the 128 bit pattern within the whole physical address space and counts the matches. Performing the search from the hypervisor guarantees atomicity, meaning during the search, the guest memory contents do not change.

Despite atomicity, however, the identical number of matches with a given sliding window size for multiple runs of the test program is not always found. This is due to the fact that not only the test program but also other programs are executing and that the guest OS is responsible for scheduling. We therefore ran the test program ten times for each sliding window size starting with 256 pages (1MB) to 1024 pages (4MB) with a step size of 64 pages.

The minimum, maximum, and average numbers of pattern matches for different sliding window sizes are shown in Table II. Without encryption enabled, the number of matches always stays the same for each run, because the allocated area just resides in RAM in clear. With encryption, the difference between the maximum and the minimum is quite large, which depends on, for example, whether another process was scheduled in the meantime. The maximum number of matches, however, never exceeds the potential possible number given by the size of the sliding window which confirms that the security claims are really guaranteed.

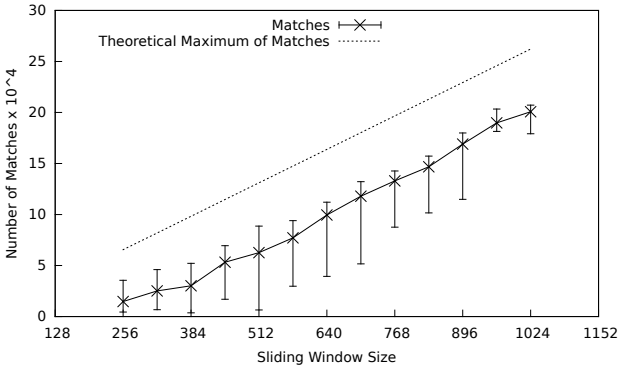

Figure 4. Maximum and average matches for different numbers of decrypted pages.

Figure 4 visualizes the minimum, maximum, and average numbers of matches with error lines. Furthermore, the potential possible number of matches determined by the sliding window size is shown as separate graph. It is clearly visible that the maximum is always below the potential maximum of matches. Note that because the `vmcall` is issued by the very same program, the number of matches is generally rather high, while it is usually much lower in real world scenarios where more processes are scheduled between the storing and acquiring of sensitive data.

2) Cache Sizes: Generally the size of the sliding window should be chosen smaller than the cache size to ensure that sensitive information is not exposed in clear to main memory. During our evaluation, the CPU cache was 8MB which is twice the size needed for our default window size of 1024. It is unlikely that sensitive data is ever exposed to RAM with HyperCrypt running with a sliding window size of half the cache size. If, however, additional security guarantees are needed, existing measures like *Cache as RAM* [34] which is used in CoreBoot, must be used to additionally control the cache. Cache as RAM has also been used to mitigate cold boot attacks against full disk encryption keys in FrozenCache [20], [35] which is capable

Sliding Window Size	No Encryption	256	512	768	1024
Minimum	249,023,439	4,465	6,505	87,595	179,140
Maximum	249,023,439	35,575	88,615	142,675	207,190
Average	249,023,439	14,767	62,605	132,959.5	200,713

Table II
NUMBER OF PATTERN MATCHES WITHIN PHYSICAL ADDRESS SPACE FOR DIFFERENT SLIDING WINDOW SIZES.

of running a full Linux system. A commercial virtualization-based solution which leverages Cache as RAM to protect against physical attacks is called *vCage* [36]. Of course, including cache as RAM as a security guarantee would also come at an additional performance cost. As cache as RAM is not necessary for HyperCrypt to protect sensitive data in general, we decided against it and did not include it into our current implementation for performance reasons.

V. FUTURE WORK

HyperCrypt's performance slowdown is caused by both the encryption mechanism and the overhead of the VMM itself. Limiting the times the VMM performs the actual page encryption or decryption would of course lead to a major increase in performance. Therefore, a future version of HyperCrypt could only enable the encryption when it deals with sensitive data and disable it for performance critical operations. This implies, however, that the physical memory must be decrypted and sensitive data must be erased from memory before the encryption is disabled. Further performance gains could be achieved by switching the VMM on and off during system execution, thereby entirely disabling the VMM's trapping mechanism. Stopping the VMM from running without shutting down the guest would require a different underlying design of the VMM though, as BitVisor is implemented as a bare metal hypervisor. Like Rutkowska's proof of concept *Bluepill* [37], the VMM would need to support on-the-fly system virtualization and devirtualization.

VI. CONCLUSION

In this paper we presented HyperCrypt, a hypervisor-based solution to transparently encrypt guest memory, including both kernel and user space, to effectively protect against physical memory disclosure. We designed and implemented our solution on top of BitVisor using the cold-boot resistant AES implementation from TreVisor and took care to handle device memory and DMA buffers securely. The security of HyperCrypt can be adjusted by configuring the size of the sliding window to determine how many pages are left unencrypted within main memory at a time. We evaluated HyperCrypt regarding performance and security and showed that data exposure is effectively decreased. With the default sliding window size of 1024 pages, cold boot attacks are rendered unlikely due to large caches while the reply rate of an nginx web server is 37% lower than for an unprotected system.

ACKNOWLEDGEMENTS

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

REFERENCES

[1] M. Henson and S. Taylor, "Beyond full disk encryption: Protection on security-enhanced commodity processors," in *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, 2013, pp. 307–321.

[2] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold Boot Attacks on Encryptions Keys," in *Proceedings of the 17th USENIX Security Symposium*, Princeton University. San Jose, CA: USENIX Association, Aug. 2008, pp. 45–60.

[3] B. Boeck, *Firewire-based Physical Security Attacks on Windows 7, EFS and BitLocker*, Secure Business Austria Research Lab, Aug. 2009.

[4] Robert David Graham, "Thunderbolt: Introducing a new way to hack Macs," http://erratasec.blogspot.com/2011/02/thunderbolt-introducing-new-way-to-hack.html, Feb. 2011, Errata Security.

[5] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy*. Tel-Aviv, Israel: ACM, 2013.

[6] Intel Cooperation, "Intel Software Guard Extensions Programming Reference (329298-002US)," Oct. 2014.

[7] T. Shinagawa, H. Eiraku, K. Omote, S. Hasegawa, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "Bitvisor: a thin hypervisor for enforcing i/o device security," in *Proceedings of the 5th International Conference on Virtual Execution Environments (VEE'09)*. Washington, DC: University of Tsukuba, Mar. 2009.

[8] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," in *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, 2013, pp. 40–49.

[9] M. Brenner, J. Wiebelitz, G. von Voigt, and M. Smith, "Secret program execution in the cloud applying homomorphic encryption," in *Digital Ecosystems and Technologies Conference (DEST), 2011 Proceedings of the 5th IEEE International Conference on*, May 2011, pp. 114–119.

[10] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC, Bethesda, MD, USA*, 2009, pp. 169–178.

[11] P. T. Breuer and J. P. Bowen, "A fully homomorphic crypto-processor design: correctness of a secret computer," in *Proceedings of the 5th international conference on Engineering Secure Software and Systems (ESSoS'13)*, J. Jürjens, B. Livshits, and R. Scandariato, Eds. Springer-Verlag, Berlin, Heidelberg, 2013, pp. 123–138. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36563-8__9

[12] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[13] G. E. Suh, C. W. Fletcher, D. E. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Author retrospective AEGIS: architecture for tamper-evident and tamper-resistant processing," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, 2014, pp. 68–70.

[14] Patrick Simmons, "Security Through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption," *CoRR*, vol. abs/1104.4843, 2011, University of Illinois at Urbana-Champaign.

[15] Tilo Müller and Felix Freiling and Adreas Dewald, "TRESOR Runs Encryption Securely Outside RAM," in *20th USENIX Security Symposium*. San Francisco, California: University of Erlangen-Nuremberg, Aug. 2011.

[16] E. Blass and W. Robertson, "TRESOR-HUNT: attacking cpu-bound encryption," in *28th Annual Computer Security Applications Conference, ACSAC, Orlando, FL, USA*, 2012, pp. 71–78.

[17] J. Götzfried and T. Müller, "ARMORED: cpu-bound encryption for android-driven ARM devices," in *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, 2013, pp. 161–168.

[18] T. Müller, B. Taubmann, and F. C. Freiling, "Trevisor - os-independent software-based full disk encryption secure against main memory attacks," in *Applied Cryptography and Network Security - 10th International Conference, ACNS 2012, Singapore, June 26-29, 2012. Proceedings*, 2012, pp. 66–83.

[19] J. Götzfried and T. Müller, *Mutual Authentication and Trust Bootstrapping towards Secure Disk Encryption*, 2nd ed., New York, NY, USA, 2014, vol. 17.

[20] Jürgen Pabel, "Frozen Cache," http://frozenchache.blogspot.com/, Jan. 2009.

[21] N. Heninger and H. Shacham, "Reconstructing RSA private keys from random key bits," in *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, 2009, pp. 1–17.

[22] T. P. Parker and S. Xu, "A method for safekeeping cryptographic keys from memory disclosure attacks," in *Trusted Systems, First International Conference, INTRUST 2009, Beijing, China, December 17-19, 2009. Revised Selected Papers*, 2009, pp. 39–59.

[23] B. Garmany and T. Müller, "PRIME: private RSA infrastructure for memory-less encryption," in *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, 2013, pp. 149–158.

[24] L. Guan, J. Lin, B. Luo, and J. Jing, "Copker: Computing with private keys without RAM," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2013*, 2014.

[25] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang, "Protecting private keys against memory disclosure attacks using hardware transactional memory," in *36th IEEE Symposium on Security and Privacy*, 2015.

[26] M. Henson and S. Taylor, "Memory encryption: A survey of existing techniques," *ACM Comput. Surv.*, vol. 46, no. 4, p. 53, 2013.

[27] G. Duc and R. Keryell, "Cryptopage: An efficient secure architecture with memory encryption, integrity and information leakage protection," in *22nd Annual Computer Security Applications Conference ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*, 2006, pp. 483–492.

[28] N. Provos, "Encrypting virtual memory," in *9th USENIX Security Symposium, Denver, Colorado, USA, August 14-17, 2000*, 2000.

[29] P. Peterson, "Cryptkeeper: Improving security with encrypted RAM," in *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*, Nov 2010, pp. 120–126.

[30] J. Götzfried, T. Müller, G. Drescher, S. Nürnberger, and M. Backes, "Ramcrypt: Kernel-based address space encryption for user-mode processes," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIACCS'16. New York, NY, USA: ACM, 2016, pp. 919–924.

[31] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold Boot Attacks on Encryptions Keys," in *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, Aug. 2008.

[32] NGINX Inc., "NGINX: High Performance Load Balancer, Web Server, and Reverse Proxy," http://nginx.com/, Jan. 2016.

[33] T. Bullock, "httperf," http://sourceforge.net/projects/httperf, Jan. 2016.

[34] Y. Lu, L.-T. Lo, G. R. Watson, and R. G. Minnich, "CAR: Using Cache as RAM in LinuxBIOS," http://rere.qmqm.pl/ mirq/cache_as_ram_lb_09142006.pdf, 2006.

[35] Jürgen Pabel, "FrozenCache – Mitigating cold-boot attacks for Full-Disk-Encryption software," in *27th Chaos Communication Congress*. Berlin, Germany: CCC, Dec. 2010.

[36] PrivateCore, "Trustworthy Cloud Computing with vCage," https://privatecore.com/vcage, Aug. 2014.

[37] J. Rutkowska, "Subverting Vista Kernel For Fun And Profit," in *Black Hat Briefings, Las Vegas, Nevada, USA, August 3rd, 2006*, 2006.