Practical Timing Side Channel Attacks Against Kernel Space ASLR

Ralf Hund, Carsten Willems, Thorsten Holz
Horst-Goertz Institute for IT Security
Ruhr-University Bochum
{*firstname.lastname*}*@rub.de*

Abstract—Due to the prevalence of control-flow hijacking attacks, a wide variety of defense methods to protect both user space and kernel space code have been developed in the past years. A few examples that have received widespread adoption include stack canaries, non-executable memory, and Address Space Layout Randomization (ASLR). When implemented correctly (i.e., a given system fully supports these protection methods and no information leak exists), the attack surface is significantly reduced and typical exploitation strategies are severely thwarted. All modern desktop and server operating systems support these techniques and ASLR has also been added to different mobile operating systems recently.

In this paper, we study the limitations of kernel space ASLR against a local attacker with restricted privileges. We show that an adversary can implement a *generic* side channel attack against the memory management system to deduce information about the privileged address space layout. Our approach is based on the intrinsic property that the different caches are shared resources on computer systems. We introduce three implementations of our methodology and show that our attacks are feasible on four different x86-based CPUs (both 32- and 64-bit architectures) and also applicable to virtual machines. As a result, we can successfully circumvent kernel space ASLR on current operating systems. Furthermore, we also discuss mitigation strategies against our attacks, and propose and implement a defense solution with negligible performance overhead.

Keywords-Address Space Layout Randomization; Timing Attacks; Kernel Vulnerabilities; Exploit Mitigation

I. INTRODUCTION

Modern operating systems employ a wide variety of methods to protect both user and kernel space code against memory corruption attacks that leverage vulnerabilities such as stack overflows [1], integer overflows [2], and heap overflows [3]. Control-flow hijacking attempts pose a significant threat and have attracted a lot of attention in the security community due to their high relevance in practice. Even nowadays, new vulnerabilities in applications, drivers, or operating system kernels are reported on a regular basis. To thwart such attacks, many mitigation techniques have been developed over the years. A few examples that have received widespread adoption include stack canaries [4], non-executable memory (e.g., No eXecute (NX) bit and Data Execution Prevention (DEP) [5]), and Address Space Layout Randomization (ASLR) [6]–[8].

Especially ASLR plays an important role in protecting computer systems against software faults. The key idea behind this technique is to randomize the system's virtual memory layout either every time a new code execution starts (e.g., upon process creation or when a driver is loaded) or on each

system reboot. While the initial implementations focused on randomizing user mode processes, modern operating systems such as Windows 7 randomize *both* user and kernel space. ASLR introduces diversity and randomness to a given system, which are both appealing properties to defend against attacks: an attacker that aims to exploit a memory corruption vulnerability does not know *any* memory addresses of data or code sequences which are needed to mount a control-flow hijacking attack. Even advanced exploitation techniques like return-to-libc [9] and return-oriented programming (ROP) [10] are hampered since an attacker does not know the virtual address of memory locations to which she can divert the control flow. As noted above, all major operating systems such as Windows, Linux, and Mac OS X have adopted ASLR and also mobile operating systems like Android and iOS have recently added support for this defense method [7], [11]–[13].

Broadly speaking, successful attacks against a system that implements ASLR rely on one of three conditions:

1) In case not all loaded modules and other mapped memory regions have been protected with ASLR, an attacker can focus on these regions and exploit the fact that the system has not been fully randomized. This is an adoption problem and we expect that in the near future all memory regions (both in user space and kernel space) will be fully randomized [14], [15]. In fact, Windows 7/8 already widely supports ASLR and the number of applications that do not randomize their libraries is steadily decreasing. Legacy libraries can also be forced to be randomized using the Force ASLR feature.

2) If some kind of information leakage exists that discloses memory addresses [16]–[18], an attacker can obtain the virtual address of specific memory areas. She might use this knowledge to infer additional information that helps her to mount a control-flow hijacking attack. While such information leaks are still available and often used in exploits, we consider them to be software faults that will be fixed to reduce the attack surface [19], [20].

3) An attacker might attempt to perform a brute-force attack [21]. In fact, Shacham et al. showed that user mode ASLR on 32-bit architectures only leaves 16 bit of randomness, which is not enough to defeat brute-force attacks. However, such brute-force attacks are *not* applicable for kernel space ASLR. More specifically, if an attacker wants to exploit a vulnerability in kernel code, a wrong offset will

typically lead to a complete crash of the system and thus an attacker has only *one* attempt to perform an exploit. Thus, brute-force attacks against kernel mode ASLR are not feasible in practice.

In combination with DEP, a technique that enforces the $W \oplus X$ (**W**ritable **xor** e**X**ecutable) property of memory pages, ASLR significantly reduces the attack surface. Under the assumption that the randomization itself cannot be predicted due to implementation flaws (i.e., not fully randomizing the system or existing information leaks), typical exploitation strategies are severely thwarted.

In this paper, we study the limitations of kernel space ASLR against a local attacker with restricted privileges. We introduce a generic attack for systems running on the Intel *Instruction Set Architecture* (ISA). More specifically, we show how a local attacker with restricted rights can mount a timing-based side channel attack against the memory management system to deduce information about the privileged address space layout. We take advantage of the fact that the memory hierarchy present in computer systems leads to shared resources between user and kernel space code that can be abused to construct a side channel. In practice, timing attacks against a modern CPU are very complicated due to the many performance optimizations used by current processors such as hardware prefetching, speculative execution, multi-core architectures, or branch prediction that significantly complicate timing measurements [22]. Previous work on side-channels attacks against CPUs [23]–[25] focused on older processors without such optimization and we had to overcome many challenges to solve the intrinsic problems related to modern CPU features [22].

We have implemented three different attack strategies that are capable of successfully reconstructing (parts of) the kernel memory layout. We have tested these attacks on different Intel and AMD CPUs (both 32- and 64-bit architectures) on machines running either Windows 7 or Linux. Furthermore, we show that our methodology also applies to virtual machines. As a result, an adversary learns precise information about the (randomized) memory layout of the kernel. With that knowledge, she is enabled to perform control-flow hijacking attacks since she now knows where to divert the control flow to, thus overcoming the protection mechanisms introduced by kernel space ASLR. Furthermore, we also discuss mitigation strategies and show how the side channel we identified as part of this work can be prevented in practice with negligible performance overhead.

In summary, the contributions of this paper are the following:

- We present a generic attack to derandomize kernel space ASLR that relies on a side channel based on the memory hierarchy present in computer systems, which leads to timing differences when accessing specific memory regions. Our attack is applicable in scenarios where brute-force attacks are not feasible and we assume that no implementation flaws exist for ASLR. Because of the general nature of the approach, we expect that it can be applied to many operating systems and a variety of hardware architectures.

- We present three different approaches to implement our methodology. We successfully tested them against systems running Windows 7 or Linux on both 32-bit and 64-bit Intel and AMD CPUs, and also the virtualization software VMware. As part of the implementation, we reverse-engineered an undocumented hash function used in Intel Sandybridge CPUs to distribute the cache among different cores. Our attack enables a local user with restricted privileges to determine the virtual memory address of key kernel memory locations within a reasonable amount of time, thus enabling ROP attacks against the kernel.

- We discuss several mitigation strategies that defeat our attack. The runtime overhead of our preferred solution is not noticeable in practice and successfully prevents the timing side channel attacks discussed in this paper. Furthermore, it can be easily adopted by OS vendors.

II. TECHNICAL BACKGROUND

We review the necessary technical background information before introducing the methodology behind our attack.

A. Address Space Layout Randomization

As explained above, ASLR randomizes the system's virtual memory layout either every time a new code execution starts or every time the system is booted [6]–[8], [26]. More specifically, it randomizes the base address of important memory structures such as for example the code, stack, and heap. As a result, an adversary does not know the virtual address of relevant memory locations needed to perform a control-flow hijacking attack (i.e., the location of shellcode or ROP gadgets). All major modern operating systems have implemented ASLR. For example, Windows implements this technique since Vista in both user and kernel space [12], Linux implements it with the help of the PaX patches [7], and MacOS ships with ASLR since version 10.5. Even mobile operating systems such as Android [11] and iOS [13] perform this memory randomization nowadays.

The security gain of the randomization is twofold: First, it can protect against remote attacks, such as hardening a networking daemon against exploitation. Second, it can also protect against local attackers by randomizing the privileged address space of the kernel. This should hinder exploitation attempts of implementation flaws in kernel or driver code that allow a local application to elevate its privileges, a prevalent problem [27], [28]. Note that since a user mode application has no means to *directly* access the kernel space, it cannot determine the base addresses kernel modules are loaded to: every attempt to access kernel space memory from user mode results in an access violation, and thus kernel space ASLR effectively hampers local exploits against the OS kernel or drivers.

Windows Kernel Space ASLR: In the following we describe the kernel space ASLR implementation of Windows (both 32-bit and 64-bit). The information presented here applies to Vista, Windows 7, and Windows 8. We obtained this information by
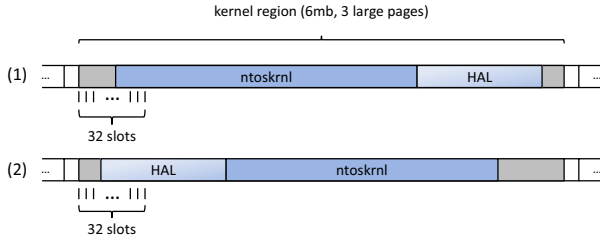
Figure 1. ASLR for Windows $kernel\_region$ (not proportional). Slot and load order (either (1) or (2)) are chosen randomly

reverse-engineering the corresponding parts of the operating system code.

During the boot process, the Windows loader is responsible for loading the two core components of the OS, the kernel image and the hardware abstraction layer (HAL), which is implemented as a separate module. At first, the Windows loader allocates a sufficiently large address region (the $kernel\_region$) for the kernel image and the HAL. The base address of this region is constant for a given system. Then, it computes a random number ranging from 0 to 31. This number is multiplied by the page size (0x1000) and added to the base address of the reserved region to form a randomized load address. Furthermore, the order in which the kernel and the HAL are loaded is also randomized. Both components are always loaded consecutively in memory, there is no gap in between. This effectively yields 64 different slots to which the kernel image and the HAL each can be loaded (see also Figure 1). In summary, the formula for computing the kernel base address is as follows:

$$k\_base = kernel\_region + (r_1 * 0x1000) + (r_2 * hal\_size),$$

where $r_1 \in \{0 \dots 31\}$ and $r_2 \in \{0, 1\}$ are random numbers within the given ranges. Kernel and HAL are commonly mapped using so called *large pages* (2 MB) which improves performance by reducing the duration of page walks; both components usually require three large pages (= 6 MB). An interesting observation is that the randomization is already applied to the *physical* load addresses of the image and that for the $kernel\_region$, the formula

$$virtual\_address = 0x80000000 + physical\_address$$

holds. The lower 31 bits of virtual kernel addresses are thus identical to the physical address. Again, this is only true for addresses in the $kernel\_region$ and does not generally apply to kernel space addresses. For the rest of the paper, note that we assume that the system is started without the /3GB boot option that restricts the kernelspace to 1 GB. In this case, the kernelspace base address would be $0xC0000000$ instead.

Once the kernel is initialized, all subsequent drivers are loaded by the kernel's driver load routine MmLoadSystemImage. This mechanism contains a different ASLR implementation to randomize the base address of drivers

in the subroutine MiReserveDriverPtes. The process works as follows: the kernel first reserves a memory region of 2 MB using standard 4 KB sized pages (a $driver\_region$). It then randomly chooses one out of 64 page-aligned start slots in this region where the first driver is loaded to. All subsequent drivers are then appended, until the end of the 2 MB region is hit, which is when the next driver is mapped to the beginning of the region (i.e., a wrap-around occurs). In case a region is full, a new 2MB $driver\_region$ with a random start slot is allocated. For session-wide drivers such as win32k.sys, a similar randomization with 64 slots for each driver image is applied in a dedicated $session\_driver\_region$. We observed that the loading order of drivers is always the same in practice.

B. Memory Hierarchy

There is a natural trade-off between the high costs of *fast* computer memory and the demand for *large* (but inexpensive) memory resources. Hence, modern computer systems are operating on hierarchical memory that is built from multiple stages of different size and speed. Contemporary hierarchies range from a few very fast CPU registers over different levels of cache to a huge and rather slow main memory. Apparently, with increasing distance to the CPU the memory gets slower, larger, and cheaper.

We focus on the different *caches* that are used to speed up address translation and memory accesses for code and data. As illustrated in Figure 2, each CPU core typically contains one dedicated *Level 1 (L1)* and *Level 2 (L2)* cache and often there is an additional *Level 3 (L3)* shared cache (also called *last level cache (LLC)*). On level 1, instructions and data are cached into distinct facilities (*ICACHE* and *DCACHE*), but on higher stages unified caches are used. The efficiency of cache usage is justified by the *temporal* and *spatial locality* property of memory accesses [29]. Hence, not only single bytes are cached, but always chunks of adjacent memory. The typical size of such a *cache line* on x86/x64 is 64 bytes.

One essential question is where to store certain memory content in the caches and how to locate it quickly on demand. All described caches operate in an *n-way set associative* mode. Here, all available slots are grouped into sets of the size *n* and each memory chunk can be stored in all slots of one particular set. This target set is determined by a bunch of *cache index* bits that are taken from the memory address. As an example, consider a 32-bit address and a typical L3 cache of 8 MB that is 16-way set associative. It consists of $(8,192*1,024)/64 = 131,072$ single slots that are grouped into $131,072/16 = 8,192$ different sets. Hence, 13 bits are needed to select the appropriate set. Since the lower 6 bits (starting with bit 0) of each address are used to select one particular byte from each cacheline, the bits 18 to 6 determine the set. The remaining upper 13 bits form the *address tag*, that has to be stored with each cache line for the later lookup.

One essential consequence of the *set associativity* is that memory addresses with identical index bits compete against the available slots of one set. Hence, memory accesses may

evict and replace other memory content from the caches. One common replacement strategy is *Least Recently Used (LRU)*, in which the entry which has not been accessed for the longest time is replaced. Since managing real timestamps is not affordable in practice, the variant *Pseudo-LRU* is used: an additional *reference bit* is stored with each cacheline that is set on each access. Once all reference bits of one set are enabled, they are all cleared again. If an entry from a set has to be removed, an arbitrary one with a cleared reference bit is chosen.

Virtual Memory and Address Translation: Contemporary operating systems usually work on *paged virtual memory* instead of physical memory. The memory space is divided into equally sized pages that are either *regular pages* (e.g., with a size of 4 KB), or *large pages* (e.g., 2 or 4 MB). When accessing memory via virtual addresses (VA), they first have to be translated into physical addresses (PA) by the processor's *Memory Management Unit* (MMU) in a *page walk*: the virtual address is split into several parts and each part operates as an array index for certain levels of *page tables*. The lowest level of the involved *paging structures* (PS), the *Page Table Entry* (PTE), contains the resulting physical *frame number*. For large pages, one level less of PS is needed since a larger space of memory requires less bits to address. In that case, the frame number is stored one level higher in the *Page Directory Entry* (PDE). In case of *Physical Address Extension* (PAE) [30] or 64-bit mode, additional PS levels are required, i.e. the *Page Directory Pointer* (PDP) and the *Page Map Level 4* (PML4) structures. Appendix A provides more information and examples of such address resolutions for PAE systems.

In order to speed up this address translation process, resolved address mappings are cached in *Translation Lookaside Buffers* (TLBs). Additionally, there often are dedicated caches for the involved higher level PS [31]. Depending on the underlying system, the implementation of these translation caches differs a lot. Current x86/x64 systems usually have two different levels of TLB: the first stage *TLB0* is split into one for data (DTLB) and another for instructions (ITLB), and the second stage TLB1 is used for both. Further, the TLBs are often split into one part for regular pages and another for large pages.

Even with TLBs and PS caches, the address translation takes some clock cycles, during which the resulting physical address is not available yet. As an effect, the system has to wait for the address translation *before* it can check the tag values of the caches. Therefore, lower caches (mostly only the L1 cache) are *virtually indexed, but physically tagged*. This means that the cache index is taken from the virtual address but the stored tag values from the physical one. With that approach, the corresponding tag values already can be looked up and then quickly compared once the physical address is available.

Figure 2 illustrates all the different caching facilities of the *Intel i7* processor. The vertical arrows are labeled with the amount of clock cycles that are normally required to access the particular stages [32], [33]. The dashed arrow (pointing from the TLB1 to the DCACHE) indicates that PS are not only cached
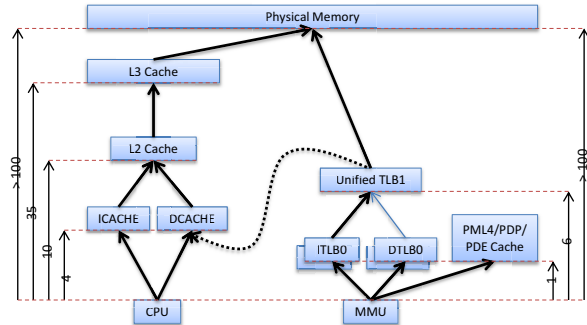

Figure 2. Intel i7 memory hierarchy plus clock latency for the relevant stages (based on [32], [33])

in the TLB or PML4/PDP/PDE caches, but may also reside as regular data within the DCACHE or higher level unified caches.

An essential part of each virtual memory system is the *page fault handler* (PFH). It is invoked if a virtual address cannot be resolved, i.e., the page walk encounters invalid PS. This may happen for several reasons (e.g., the addressed memory region has been swapped out or the memory is accessed for the first time after its allocation). In such cases, the error is handled completely by the PFH. Although this happens transparently, the process induces a slight time delay. Besides translation information, the PS also contain several protection flags (e.g., to mark memory as *non-executable* or to restrict access to privileged code only). After successful translation, these flags are checked against the current system state and in case of a protection violation, the PFH is invoked as well. In that case an access violation exception is generated that has to be caught and handled by the executing process. Again, a slight time delay may be observable between accessing the memory and the exception being delivered to the exception handler.

III. TIMING SIDE CHANNEL ATTACKS

Based on this background information, we can now explain how time delays introduced by the memory hierarchy enable a side channel attack against kernel-level ASLR.

A. Attacker Model

We focus in the following on local attacks against kernel space ASLR: we assume an adversary who already has restricted access to the system (i.e., she can run arbitrary applications) but does not have access to privileged kernel components and thus cannot execute privileged (kernel mode) code. We also assume the presence of a user mode-exploitable vulnerability within kernel or driver code, a common problem [27]. The exploitation of this software fault requires to know (at least portions of) the kernel space layout since the exploit at some point either jumps to an attacker controlled location or writes to an attacker controlled location to divert the control flow.

Since the entire virtual address space is divided in both user and kernel space, an attacker might attempt to directly jump to a

user space address from within kernel mode in an exploit, thus circumventing any kernel space ASLR protections. However, this is not always possible since the correct user space might not be mapped at the time of exploitation due to the nature of the vulnerability [14]. Furthermore, this kind of attack is rendered impossible with the introduction of the *Supervisor Mode Execution Protection* (SMEP) feature of modern CPUs that disables execution of user space addresses in kernel mode [34].

We also assume that the exploit uses ROP techniques due to the $W \oplus X$ property enforced in modern operating systems. This requires to know a sufficiently large amount of executable code in kernel space to enable ROP computations [10], [35]. Schwartz et al. showed that ROP payloads can be built automatically for 80% of Linux programs larger than 20 KB [36]. Further, we assume that the system fully supports ASLR and that no information leaks exist that can be exploited. Note that a variety of information leaks exist for typical operating systems [18], but these types of leaks stem from shortcomings and inconsequences in the actual implementation of the specific OS. Developers can fix these breaches by properly adjusting their implementation. Recently, Giuffrida et al. [37] argued that kernel information leakage vulnerabilities are hard to fix. While we agree that it is not trivial to do so, we show that even in the absence of any leak, we can still derandomize kernel space ASLR.

One of our attacks further requires that the userland process either has access to certain APIs or gains information about the physical frame mapping of at least one page in user space. However, since this prerequisite holds only for one single attack – which further turns out to be our least effective one – we do not consider it in the general attacker model but explain its details only in the corresponding Section IV-A.

In summary, we assume that the system correctly implements ASLR (i.e., the complete system is randomized and no information leaks exist) and that it enforces the $W \oplus X$ property. Hence, all typical exploitation strategies are thwarted by the implemented defense mechanisms.

B. General Approach

In this paper, we present *generic* side channels against processors for the Intel ISA that enable a restricted attacker to deduce information about the privileged address space by timing certain operations. Such side channels emerge from intricacies of the underlying hardware and the fact that parts of the hardware (such as caches and physical memory) are *shared* between both privileged and non-privileged code. Note that all the approaches that we present in this paper are *independent* of the underlying operating system: while we tested our approach mainly on Windows 7 and Linux, we are confident that the attacks also apply for other versions of Windows or even other operating systems. Furthermore, our attacks work on both 32- and 64-bit systems.

The methodology behind our timing measurements can be generalized in the following way: At first, we attempt to set the system in a specific state from user mode. Then we measure the

duration of a certain memory access operation. The time span of this operation then (possibly) reveals certain information about the kernel space layout. Our timing side channel attacks can be split into two categories:

- **L1/L2/L3-based Tests:** These tests focus on the L1/L2/L3 CPU caches and the time needed for fetching data and code from memory.
- **TLB-based Tests:** These tests focus on TLB and PS caches and the time needed for address translation.

To illustrate the approach, consider the following example: we make sure that a privileged code portion (such as the operating system's system call handler) is present within the caches by executing a system call. Then, we access a designated set of user space addresses and execute the system call again. If the system call takes longer than expected, then the access of user space addresses has evicted the system call handler code from the caches. Due to the structure of modern CPU caches, this reveals parts of the physical (and possibly virtual) address of the system call handler code as we show in our experiments.

Accessing Privileged Memory: As explained in Section II-B, different caching mechanisms determine the duration of a memory access:

- The TLB and PS caches speed up the translation from the virtual to the physical address.
- In case no TLB exists, the PS entries of the memory address must be fetched during the page walk. If any of these entries are present in the normal L1/L2/L3 caches, then the page walk is accelerated in a significant (i.e., measurable) way.
- After the address translation, the actual memory access is faster if the target data/code can be fetched from the L1/-L2/L3 caches rather than from the RAM.

While it is impossible to access kernel space memory directly from user mode, the nature of the cache facilities still enables an attacker to *indirectly* measure certain side-effects. More precisely, she can directly access a kernel space address from user mode and measure the duration of the induced exception. The page fault will be faster if a TLB entry for the corresponding page was present. Additionally, even if a permission error occurs, this still allows to launch address translations and, hence, generate valid TLB entries by accessing privileged kernel space memory from user mode.

Further, an attacker can (to a certain degree) control which code or data regions are accessed in kernel mode by forcing fixed execution paths and known data access patterns in the kernel. For example, user mode code can perform a system call (`sysenter`) or an interrupt (`int`). This will force the CPU to cache the associated handler code and data structures (e.g., IDT table) as well as data accessed by the handler code (e.g., system call table). A similar effect can be achieved to cache driver code and data by indirectly invoking driver routines from user mode.

Note that the x86/x64 instruction set also contains a number of instructions for explicit cache control (e.g., `invlpg`,

Method	Requirements	Results	Environment	Success
Cache Probing	*large pages* or PA of *eviction buffer*, partial information about $kernel\_region$ location	`ntoskrnl.exe` and `hal.sys`	all	✓
Double Page Fault	none	allocation map, several drivers	all but AMD	✓
Cache Preloading	none	`win32k.sys`	all	✓

Table I

SUMMARY OF TIMING SIDE CHANNEL ATTACKS AGAINST KERNEL SPACE ASLR ON WINDOWS.

`invd/wbinvd`, `clflush`, or `prefetch`) [30]. However, these instructions are either privileged and thus cannot be called from user mode, or they cannot be used with kernel space addresses from user mode. Hence, none of these instructions can be used for our purposes. As a result, we must rely on *indirect* methods as explained in the previous paragraphs.

C. Handling Noise

While performing our timing measurements we have to deal with different kinds of noise that diminish the quality of our data if not addressed properly. Some of this noise is caused by the architectural peculiarities of modern CPUs [22]: to reach a high parallelism and work load, CPU developers came up with many different performance optimizations like hardware prefetching, speculative execution, multi-core architectures, or branch prediction. We have adapted our measuring code to take the effects of these optimizations into account. For example, we do not test the memory in consecutive order to avoid being influenced by memory prefetching. Instead, we use access patterns that are not influenced by these mechanisms at all. Furthermore, we have to deal with the fact that our tool is not the only running process and there may be a high CPU load in the observed system. The thread scheduler of the underlying operating system periodically and, if required, also preemptively interrupts our code and switches the execution context. If we are further running inside a virtual machine, there is even more context switching when a transition between the virtual machine monitor and the VM (or between different VMs) takes place. Finally, since all executed code is operating on the same hardware, also the caches have to be shared to some extent.

As mentioned above, our approach is based on two key operations: (a) set the system into a specific state and (b) measure the duration of a certain memory access operation. Further, these two operations are performed for each single memory address that is probed. Finally, the complete experiment is repeated multiple times until consistent values have been collected. While it is now possible — and highly probable — that our code is interrupted many times while probing the complete memory, it is also very likely that the low-level two step test operations can be executed without interruption. The mean duration of these two steps depends on the testing method we perform, but even in the worst case it takes no more than 5,000 clock cycles. Since modern operating systems have time slices of at least several milliseconds [38], [39], it is highly unlikely that the scheduler interferes with our measurements. Accordingly, while there may be much noise due to permanent interruption of our

experiments, after a few iterations we will eventually be able to test each single memory address without interruption. This is sufficient since we only need minimal measurement values, i.e., we only need one measurement without interruption.

IV. IMPLEMENTATION AND RESULTS

We now describe three different implementations of timing side channel attacks that can be applied independently from each other. The goal of each attack is to precisely locate some of the currently loaded kernel modules from user mode by measuring the time needed for certain memory accesses. Note that an attacker can already perform a ROP-based attack once she has derandomized the location of a few kernel modules or the kernel [35], [36].

Depending on the randomness created by the underlying ASLR implementation, the first attack might still require partial information on the location for the kernel area. For the Windows ASLR implementation (see Section II-A), this is not the case since only 64 slots are possible of the kernel. The first attack requires either the presence of two large pages or the knowledge of the physical address of a single page in user space. Our second attack has no requirements. However, due to the way the AMD CPU that we used during testing behaves in certain situations, this attack could not be mounted on this specific CPU. The third attack has no requirements at all.

We have evaluated our implementation on the 32-bit and 64-bit versions of *Windows 7 Enterprise* and *Ubuntu Desktop 11.10* on the following (native and virtual) hardware architectures to ensure that they are commonly applicable on a variety of platforms:

1) Intel i7-870 (Nehalem/Bloomfield, Quad-Core)
2) Intel i7-950 (Nehalem/Lynnfield, Quad-Core)
3) Intel i7-2600 (Sandybridge, Quad-Core)
4) AMD Athlon II X3 455 (Triple-Core)
5) VMWare Player 4.0.2 on Intel i7-870 (with VT-x)

Table I provides a high-level overview of our methods, their requirements, and the obtained results. We implemented an exploit for each of the three attacks.

For the sake of simplicity, all numbers presented in the remainder of this section were taken using Windows 7 Enterprise 32-bit. Note that we also performed these tests on Windows 7 64-bit and Ubuntu Desktop (32-bit and 64-bit) and can confirm that they work likewise. The Ubuntu version we used did not employ kernel space ASLR yet, but we were able to determine the location of the kernel image from user space. In general,

this does not make any difference since the attacks also would have worked in the presence of kernel space ASLR.

In the following subsections, we explain the attacks and discuss our evaluation results.

A. First Attack: Cache Probing

Our first method is based on the fact that multiple memory addresses have to be mapped into the same cache set and, thus, compete for available slots. This can be utilized to infer (parts of) virtual or physical addresses indirectly by trying to evict them from the caches in a controlled manner. More specifically, our method is based on the following steps: first, the searched code or data is loaded into the cache indirectly (e.g., by issuing an interrupt or calling `sysenter`). Then certain parts of the cache are consecutively replaced by accessing corresponding addresses from a user-controlled *eviction buffer*, for which the addresses are known. After each replacement, the access time to the searched kernel address is measured, for example by issuing the system call again. Once the measured time is significantly higher, one can be sure that the previously accessed eviction addresses were mapped into the same cache set. Since the addresses of these *colliding locations* are known, the corresponding cache index can be obtained and obviously this is also a part of the searched address.

Several obstacles have to be addressed when performing these timing measurements in practice. First, the correct kind of memory access has to be performed: higher cache levels are unified (i.e., there are no separate data and instruction caches), but on lower levels either a memory read/write access or an execution has to be used in order to affect the correct cache type. Second, accessing the colliding addresses only once is not enough. Due to the Pseudo-LRU algorithm it may happen that not the searched address is evicted, but one from the eviction buffer. Therefore, it is necessary to access each of the colliding addresses twice. Note that it is still possible that code within another thread or on other CPUs concurrently accesses the search address in the meantime, setting its reference bit that way. To overcome this problem, all tests have to be performed several times to reduce the influence of potential measuring errors and concurrency.

More serious problems arise due to the fact that the cache indexes on higher levels are taken from the physical instead of the virtual addresses. In our experiments, the eviction buffer is allocated from user mode and, hence, only its virtual address is known. While it is still possible to locate the colliding cacheset, no information can be gathered about the corresponding physical addresses. In general, even if the physical address of the searched kernel location is known, this offers no knowledge about its corresponding virtual address. However, the relevant parts of the virtual and physical address are identical for the $kernel\_region$ of Windows (see Section II-A). Hence, all the relevant bits of the virtual address can be obtained from the physical address.

Cache probing with the latest Intel CPUs based on the Sandybridge [30] architecture is significantly harder, even if the attacker has a contiguous region of memory for which all corresponding physical addresses are known. These processors employ a *distributed last level cache* [30] that is split into equally sized *cache slices* and each of them is dedicated to one CPU core. This approach increases the access bandwidth since several L3 cache accesses can be performed in parallel. In order to uniformly distribute the accesses to all different cache slices, a hash function is used that is not publicly documented. We thus had to reconstruct this hash function in a black-box manner before cache probing can be performed, since otherwise it is unknown which (physical) addresses are mapped into which cache location. We explain our reverse-engineering approach and the results in a side note before explaining the actual evaluation results for our first attack.

1) Side Note: Sandybridge Hash Function: In order to reconstruct the Sandybridge hash function, we utilized the Intel i7-2600 processor. This CPU has an 8 MB L3 cache and 4 cores, resulting in 2 MB L3 slices each. Hence, the hash function has to *decide* between 4 different slices (i.e., resulting in 2 output bits). Since our testing hardware had 4 GB of physical memory, we have reconstructed the hash function for an input of 32 bits. In case of larger physical memory, the same method can be applied to reverse the influence of the upper bits as well.

We started with the reasonable assumption that L3 cachelines on this CPU still consist of 64 bytes. Hence, the lowest 6 bits of each address operate as an offset and, therefore, do not contribute as input to the hash function. Accordingly, we assumed a function $h : \{0,1\}^{32-6} \rightarrow \{0,1\}^2$.

In order to learn the relationship between the physical addresses and the resulting cache slices, we took one arbitrary memory location and an additional eviction buffer of 8 MB and tried to determine the colliding addresses within (i.e., those which are mapped into the same cacheset of the same cache slice). Since the L3 cache operates on physical addresses, the eviction buffer had to be contiguous. Therefore, we used our own custom driver for this experiment.

Performance optimization features of modern CPUs like hardware prefetching, speculative execution, and branch prediction make it impossible to directly identify single colliding addresses. Therefore, we performed a two-step experiment: (1) we identified eviction buffer *regions* of adjacent memory addresses that collide with the probe address and then (2) we located the particular colliding addresses within these regions. We have performed these tests several hundred times with different physical addresses in order to gain variety in the test data. As a result of each single test we got a tuple $(p, CA = \{ca_1, ca_2, ...\})$ whereas p is the used probe address and each ca_i is a colliding address from our eviction buffer. By manually comparing those tuples (p, CA) and (p', CA') with a hamming distance of *one* between p and p', we were able to learn the influence of particular bits on the colliding addresses from CA and CA'.

In the end we were able to fully reconstruct the hashing function h that decides which cache slice is used for a given

$$h = (h_1, h_2)$$
$$h_1 = b_{31} \oplus b_{30} \oplus b_{29} \oplus b_{27} \oplus b_{25} \oplus b_{23} \oplus b_{21} \oplus b_{19} \oplus b_{18}$$
$$h_2 = b_{31} \oplus b_{29} \oplus b_{28} \oplus b_{26} \oplus b_{24} \oplus b_{23} \oplus b_{22} \oplus b_{21} \oplus b_{20} \oplus b_{19} \oplus b_{17}$$
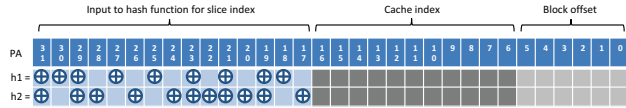
Figure 3. Results for the reconstruction of the undocumented Sandybridge hash function
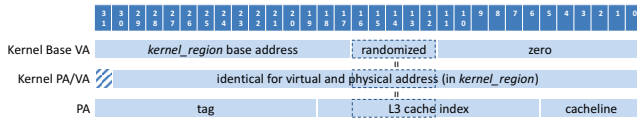
Figure 4. Correlation of different memory addresses

address. It turned out that only the bits 31 to 17 are considered as input values. Each cache slice operates as a separate smaller 2 MB cache, whereas the address bits 16 to 6 constitute as the cache index (11 bits are necessary to address all sets of such a 2 MB cache). Figure 3 shows how the 32 bits of a physical address are used as inputs to the hash function, cache index, and block offset.

2) Evaluation Results: We evaluated cache probing on all of our testing machines. We assume that the base address of the $kernel\_region$ (see $kernel\_base$ from Section II-A) is known. This is a reasonable assumption in practice since this information can be reliably extracted using the method presented in Section IV-B. In Windows this address actually is constant for a particular system.

Figure 4 shows the correlation of the different parts of the virtual and physical address inside the $kernel\_region$. In essence, bits 16 to 12 of the kernel's base address are randomized in Windows' ASLR implementation and must be known by an attacker. Since the PA and VA for bits 30 to 0 are identical in the $kernel\_region$, it is also sufficient to know bits 16 to 12 of the physical address. This bit range overlaps with the L3 cache index. In other words: if the L3 cache index is known, then an attacker can tell the virtual base address of the kernel.

We used cache probing to extract parts of the physical address of the system call handler `KiFastCallEntry`. The offset from this function to the kernel's base address is static and known. If we know the address of this function, then we also know the base address of the kernel (and HAL).

We performed the following steps for all cache sets i:
1) Execute `sysenter` with an unused syscall number.
2) Evict cache set i using the eviction buffer.
3) Execute `sysenter` again and measure the duration.

The unused syscall number minimizes the amount of executed kernel mode code since it causes the syscall handler to immediately return to user mode with an error. Step 1 makes sure that the syscall handler is present in the caches. Step 2 tries to evict the syscall handler code from the cache. Step 3 measures if the eviction was successful. If we hit the correct set i, then the second `sysenter` takes considerably longer and from i we can deduce the lower parts of the physical address of the syscall handler. Along with the address of the $kernel\_region$, this yields the *complete virtual address* of the syscall handler, and thus the base of the entire kernel and the HAL.

We performed extensive tests on the machine powered by an Intel i7-870 (Bloomfield) processor. We executed the cache probing attack 180 times; the machine was rebooted after each test and we waited for a random amount of time before the measurements took place to let the system create artificial noise. Figure 5 shows the cache probing measurements. The x-axis consists of the different L3 cache sets ($8,192$ in total) and the y-axis is the duration of the second system call handler invocation in CPU clocks, after the corresponding cache set was evicted. The vertical dashed line indicates the correct value where the system call handler code resides. There is a clear cluster of high values at this dashed line, which can be used to extract the correct cache set index and thus parts of the physical (and possibly virtual) address. We were able to successfully determine the correct syscall handler address in each run and there were no false positives. The test is fast and generally takes less than one second to finish.

3) Discussion: For successful cache probing attacks, an adversary needs to know the physical addresses of the eviction buffer, at least those bits that specify the cache set. Furthermore, she somehow has to find out the corresponding virtual address of the kernel module from its physical one. This problem is currently solved by using *large pages* for the buffer, since under Windows those always have the lowest bits set to 0. Therefore, their first byte always has a cache index of 0 and all following ones can be calculated from that. However, this method does not work with Sandybridge processors, since there we need to know the complete physical address as input to the hash function that decides on which cache slice an address is mapped. Furthermore, allocating large pages requires a special right under Windows (`MEM_LARGE_PAGES`), which first has
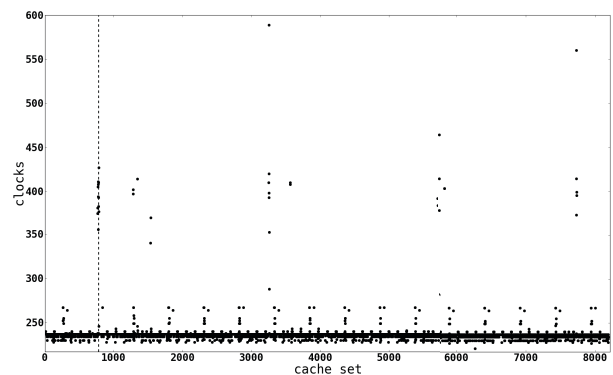

Figure 5. Cache probing results for Intel i7-870 (Bloomfield)

to be acquired somehow. One possible way to overcome this problem is to exploit an application that already possesses this right.

In case of non-Sandybridge processors, large pages are not needed per se. It is only necessary to know the physical start address of the eviction buffer. More generically, it is only necessary to know parts of the physical base address of *one* user space address, since this can then be used to align the eviction buffer. Our experiments have shown that these parts of the physical base address of the common module `ntdll`, which is always mapped to user space, is always constant (even after reboots). Though the concrete address varies depending on the hardware and loaded drivers and is thus difficult to compute, the value is deterministic.

B. Second Attack: Double Page Fault

The second attack allows us to reconstruct the allocation of the entire kernel space from user mode. To achieve this goal, we take advantage of the behavior of the TLB cache. When we refer to an *allocated* page, we mean a page that can be accessed without producing an address translation failure in the MMU; this also implies that the page must not be paged-out.

The TLB typically works in the following way: whenever a memory access results in a successful page walk due to a TLB miss, the MMU replaces an existing TLB entry with the translation result. Accesses to non-allocated virtual pages (i.e., the present bit in the PDE or PTE is set to zero) induce a page fault and the MMU does *not* create a TLB entry. However, when the page translation was successful, but the *access permission* check fails (e.g., when kernel space is accessed from user mode), a TLB entry is indeed created. Note that we observed this behavior only on Intel CPUs and within the virtual machine. In contrast, the AMD test machine acted differently and *never* created a TLB entry in the mentioned case. The double page fault method can thus not be applied on our AMD CPU.

The behavior on Intel CPUs can be exploited to reconstruct the entire kernel space from user mode as follows: for each kernel space page p, we first access p from user mode. This results in a page fault that is handled by the operating system and forwarded to the exception handler of the process. One of the following two cases can arise:

- p refers to an *allocated* page: since the translation is successful, the MMU creates a TLB entry for p although the succeeding permission check fails.
- p refers to an *unallocated* page: since the translation fails, the MMU does *not* create a TLB entry for p.

Directly after the first page fault, we access p again and measure the time duration until this second page fault is delivered to the process's exception handler. Consequently, if p refers to an allocated kernel page, then the page fault will be delivered *faster* due to the inherent TLB hit.

Due to the many performance optimizations of modern CPUs and the concurrency related to multiple cores, a single measurement can contain noise and outliers. We thus probe the kernel space multiple times and only use the observed minimal access time for each page to reduce measurement inaccuracies. Figure 6 shows measurement results on an Intel i7-950 (Lynnfield) CPU for eight measurements, which we found empirically to yield precise results. The dots show the minimal value (in CPU clocks) observed on eight runs. The line at the bottom indicates which pages are actually allocated in kernel space; a black bar means the page is allocated. As one can see, there is a clear correlation between the timing values and the allocation that allows us to infer the kernel memory space.

We developed an algorithm that reconstructs the allocation from the timing values. In the simplest case, we can introduce a threshold value that differentiates allocated from unallocated pages. In the above example, we can classify all timing values below $5,005$ clocks as allocated and all other values as unallocated as indicated by the dashed line. This yields a high percentage of correct classifications. Depending on the actual CPU model, this approach might induce insufficient results due to inevitable overlap of timing values and thus other reconstruction algorithms are necessary. We implemented a second approach that aims at detecting transitions from allocated to unallocated memory by looking at the pitch of the timing curve, a straightforward implementation of a *change point detection* (CPD) algorithm [40]. Further measurement results and figures displaying the results are shown in Appendix B.

1) Evaluation Results: We evaluated our double page fault based approach on the three Intel CPUs and the virtual machine, Table 2 shows a summary of the results. We employed the threshold algorithm on CPU (1) and the CPD algorithm on platforms (2)–(4). The numbers shown in the table are the average out of ten runs for each machine. Between each run, we rebooted the operating system to make sure that the kernel space allocation varies. We took a snapshot of the allocation with the help of a custom driver *before* we started the measurements to obtain a ground truth of the memory layout. Since the allocation might change while the measurements are running, the correct-
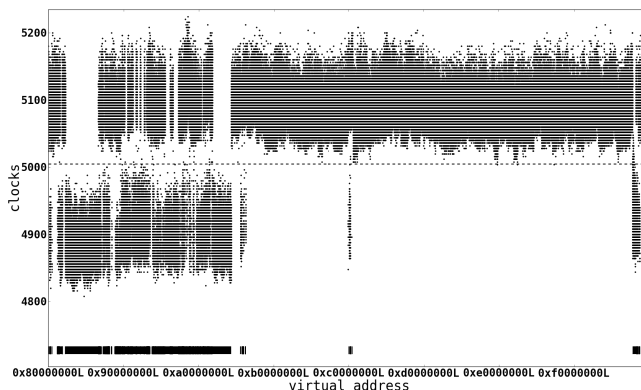

Figure 6. Example of double page fault measurements for an Intel i7-950 (Lynnfield) CPU

CPU model	Correctness	Runtime
(1) i7-870 (Bloomfield)	96.42%	17.27 sec (8 it.)
(2) i7-950 (Lynnfield)	99.69%	18.36 sec (8 it.)
(3) i7-2600 (Sandybr.)	94.92%	65.41 sec (32 it.)
(4) VMware on (1)	94.98%	72.93 sec (32 it.)

Table II
RESULTS FOR DOUBLE PAGE FAULT TIMINGS

CPU model	Matches	Code size
(1) i7-870 (Bloomfield)	21	7,431 KB
(2) i7-950 (Lynnfield)	9	4,184 KB
(3) i7-2600 (Sandybridge)	5	1,696 KB
(4) VMware on (1)	18	7,079 KB
(1) with signatures of (2)	9	2,312 KB

Table III
EVALUATION OF ALLOCATION SIGNATURE MATCHING

ness slightly decreases because of this effect. Nevertheless, we were able to successfully reconstruct the state of at least 94.92% of all pages in the kernel space on each machine. With the help of *memory allocation signatures* (a concept we introduce next) such a precision is easily enough to exactly reconstruct the location of many kernel components.

The average runtime of the measurements varies between 18 and 73 seconds and is therefore within reasonable bounds. One iteration is one probe of the entire kernel space with one access per page. As noted above, we empirically found that more than eight iterations on Nehalem CPUs do not produce better results. For Sandybridge and VMware, more iterations yielded more precise results, mainly due to the fact that there was more noise in the timings.

2) Memory Allocation Signatures: The double page fault timings yield an estimation for the allocation map of the kernel space, but do not determine at which concrete base addresses the kernel and drivers are loaded to. However, the allocation map can be used, for example, to spot the $kernel\_region$ (i.e., the memory area in which the kernel and HAL are loaded) due to the large size of this region, which can be detected easily.

One could argue that, since the virtual size of each driver is known, one could find driver load addresses by searching for allocated regions which are exactly as big as the driver image. This does not work for two reasons: first, Windows kernel space ASLR appends most drivers in specific memory regions and thus there is usually no gap between two drivers (see Section II-A). Second, there are gaps of unallocated pages inside the driver images as we explain next.

In contrast to the $kernel\_region$, Windows drivers are not mapped using large pages but using the standard 4 KB page granularity. Code and data regions of drivers are unpageable by default. However, it is possible for developers to mark certain sections inside the driver as pageable to reduce the memory usage of the driver. Furthermore, drivers typically have a discardable INIT section, that contains the initialization code of the driver which only needs to be executed once. All code pages in the INIT section are freed by Windows after the driver is initialized. Code or data in pageable sections that are never or rarely used are likely to be unallocated most of the time. Along with the size and location of the INIT section, this creates a *memory allocation signature* for each driver in the system. We can search for these signatures in the reconstructed kernel space allocation map to determine the actual load addresses of a variety of drivers.

We evaluated the signature matching on all three Intel CPUs

and the virtual machine. At first, we took a snapshot of the kernel space with the help of a custom driver. Then we created signatures for each loaded driver. A signature essentially consists of a vector of boolean values that tell whether a page in the driver was allocated (true) or paged-out (false). Note that this signature generation step can be done by an attacker in advance to build a database of memory allocation signatures.

In the next step, we rebooted the machine, applied the double page fault approach, and then matched the signatures against the reconstructed kernel space allocation map. To enhance the precision during the signature matching phase, we performed two optimizations: first, we rule out signatures that contain less than five transitions from allocated to paged-out memory to avoid false positives. Second, we require a match of at least 96% for a signature, which we empirically found to yield the best results.

The results are shown in Table 3. On machine (1), the signature matching returns the exact load addresses of 21 drivers (including big common drivers such as win32k.sys and ndis.sys); 141 drivers are loaded in total and 119 signatures were ruled out because they held too few transitions. Hence only one signature had a too low match ratio. All identified base addresses are correct, there are no false positives. Most of the other drivers could not be located since they are too small and their signatures thus might produce false positives. The 21 located drivers hold $7,431$ KB of code, which is easily enough to mount a full ROP attack as explained previously [35], [36]. Similar results hold for the other CPUs.

To assess whether the signatures are also portable across different CPUs, we took the signatures generated on machine (2) and applied them to machine (1). The operating system and driver versions of both machines are identical. This yields 9 hits with $2,312$ KB of code. This experiment shows that the different paging behavior in drivers is not fundamentally affected by differing hardware configurations.

3) Discussion: Although the double page fault measurements only reveal which pages are allocated and which are not, this still can be used to derive precise base addresses as we have shown by using the memory allocation signature matching. Furthermore, the method can be used to find large page regions (especially the $kernel\_region$).

C. Third Attack: Address Translation Cache Preloading

In the previous section we have described an approach to reconstruct the allocation map of the complete kernel space.

While it is often possible to infer the location of certain drivers from that, without driver signatures it only offers information about the fact that there is *something* located at a certain memory address and not *what*. However, if we want to locate a certain driver (i.e., obtain the virtual address of some piece of code or data from its loaded image), we can achieve this with our third implementation approach: first we flush all caches (i.e., address translation and instruction/data caches) to start with a *clean* state. After that, we preload the address translation caches by indirectly calling into kernel code, for example by issuing a `sysenter` operation. Finally, we intentionally generate a page fault by jumping to some kernel space address and measure the time that elapses between the jump and the return of the page fault handler. If the faulting address lies in the same memory range as the preloaded kernel memory, a *shorter* time will elapse due to the already cached address translation information.

Flushing all caches from user mode cannot be done directly since the `invlpg` and `invd/wbinvd` are privileged instructions. Thus, this has to be done indirectly by accessing sufficiently many memory addresses to evict all other data from the cache facilities. This is trivial for flushing the address translation and L1 caches, since only a sufficient number of virtual memory addresses have to be accessed. However, this approach is not suitable for L2/L3 caches, since these are *physically* indexed and we do not have any information about physical addresses from user mode. Anyway, in practice the same approach as described above works if the eviction buffer is chosen large enough. We have verified for Windows operating systems that large parts of the physical address bits of consecutively allocated pages are in successive order as well. Presumably this is done for performance reasons to optimally distribute the data over the caches and increase the effectiveness of the hardware prefetcher. As our experiments have shown, even on Sandybrige CPUs one virtually consecutive memory buffer with a size twice as large as the L3 cache is sufficient to completely flush it.

During our experiments we tried to locate certain system service handler functions within `win32k.sys`. To avoid cache pollution and obtain the best measuring results, we chose the system service `bInitRedirDev`, since it only executes 4 bytes of code before returning. As a side effect, we also located the *System Service Dispatch/Parameter Tables* (SSDT and SSPT) within that module, since these tables are accessed internally on each service call.

In our implementation we first allocated a 16 MB eviction buffer and filled it with `RET` instructions. Then for each page p of the complete kernel space memory (or a set of selected candidate regions), we performed three steps:

1) Flush all (address translation-, code- and unified) caches by calling into each cacheline (each 64th byte) of the eviction buffer.
2) Perform `sysenter` to preload address translation caches.
3) Call into some arbitrary address of page p and measure time until page fault handler returns.
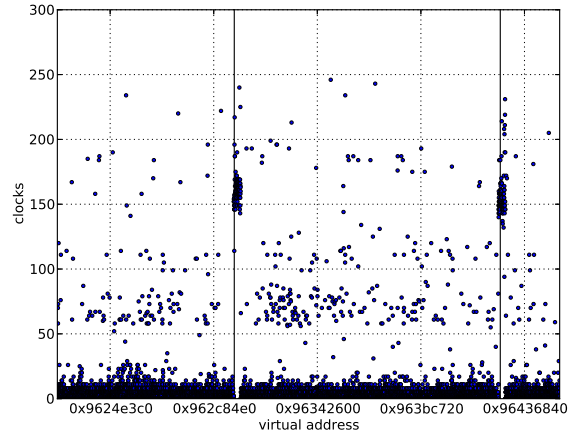

Figure 7.   Extract of cache preloading measurements

1) Evaluation Results: The steps described above have to be repeated several times to diminish the effects of noise and measuring inaccuracies. It turned out that the necessary amount of iterations strongly depends on the underlying hardware. Empirically we determined that around 100 iterations are needed on Nehalem, 60 on AMD, and only 30 on Sandybridge to reliably produce precise results. Inside the virtual machine, we had to further increase the number of iterations due to the noise that was generated by the virtual machine monitor. Nevertheless, by increasing it to 100 (the VM operated on the Sandybridge processor) this timing technique also worked successfully inside a virtualized environment.

We learned that the noise could be additionally reduced by taking different addresses randomly from each probed page for each iteration. In addition, we found out that using relative time differences was less error-prone than using absolute values. Therefore, we enhanced our testing procedure by performing the measuring twice for each page: the first time like shown above and the second time *without* performing the syscall in between. By calculating the relative time difference between both timing values, we were able to measure the speedup of address translation caches for our particular scenario. Figure 7 shows an extract of our measuring results for the Intel i7-950 (Lynnfield) CPU. The x-axis displays the probed virtual address, while the y-axis displays the relative time difference in clock cycles. The two vertical lines indicate those locations where the searched system service handler function resp. the SSDT/SSPT were located. As one can easily see those memory regions have much higher timing difference values than the others. Though there was a lot of noise within the data, our algorithms were able to locate those regions correctly on all of our testing environments.

While this method only reveals the memory *page* of the searched kernel module, it is still possible to reconstruct its

full virtual address. This can be achieved by obtaining the relative address offset of the probed code/data by inspecting the image file of the module. As the measuring operates on a *page granularity*, it is best suited to locate kernel modules that reside in regular pages. Nevertheless, with the described *difference* technique, also large page memory regions can be identified that contain certain code or data. Obviously, the exact byte locations within such regions cannot be resolved and, therefore, we have used it to locate `win32k.sys` in our experiments. Due to its size, this module is sufficient to perform arbitrary ROP attacks [35], [36].

2) Discussion: Our third proposed method has no remarkable limitations. However, depending on the size of the probed memory range and the amount of necessary test iterations, it may take some time to complete. The probing of a 3 MB region (this is the size of `win32k.sys`) for *one* iteration takes around 27 seconds. Therefore, if an attacker has employed the *double page fault* method to identify an appropriate candidate region and then performs 30 iterations on a Sandybridge processor, it takes 13 minutes to perform the complete attack. However, since the relative offset of the searched kernel function can previously be obtained from the image file, the probed memory region can be reduced drastically, enabling to perform the test in a minute or less. If the location of candidate regions is not possible, our attack will still work but take longer time. Furthermore, the technique operates on page granularity. Hence, drivers residing in large pages can be located, but their exact byte offset cannot be identified without additional techniques.

V. MITIGATION APPROACHES

Since the methods presented in the previous section can be used to break current ASLR implementations, mitigation strategies against our attacks are necessary. To that end, there are several options for CPU designers and OS vendors.

The root cause of our attacks is the *concurrent usage* of the same caching facilities by privileged and non-privileged code and data, i.e., the memory hierarchy is a shared resource. One solution to overcome this problem would be to split all caches and maintain isolated parts for user and kernel mode, respectively. Obviously, this imposes several performance drawbacks since additional checks had to be performed in several places and the maximum cache size would be cut in half for both separate caches (or the costs increase).

A related mitigation attempt is to forbid user mode code to resolve kernel mode addresses. One way to achieve this is to modify the *global descriptor table* (GDT), setting a *limit* value such that the segments used in non-privileged mode only span the user space. However, doing so would render some CPU optimization techniques useless that apply when the *flat memory model* is used (in which all segments span the complete memory). Furthermore, the complete disabling of segmentation on 64-bit architectures makes this mitigation impossible. Another option would be to suppress the creation of TLB entries on successful address translation if an access

violation happens, like it is done with the tested AMD CPU. Nevertheless, the indirect loading of kernel code, data, or address mappings through system calls still cannot be avoided with this method.

Current ASLR implementations (at least under Windows) do not fully randomize the address space, but randomly choose from 64 different memory slots. By utilizing the complete memory range and distributing all loaded modules to different places, it would be much harder to perform our attacks. Especially when dealing with a 64-bit memory layout, the time needed for measuring is several magnitudes higher and would increase the time needed to perform some of our attacks. Nevertheless, scattering allocated memory over the full address range would significantly degrade system performance since much more paging structures would be needed and spatial locality would be destroyed to a large extent. Furthermore, we expect that our *double page fault* attack even then remains practical. Due to the huge discrepancy between the 64-bit address space and the used physical memory, the page tables are very sparse (especially those one higher levels). Since page faults can be used to measure the depth of the valid page tables for a particular memory address, only a very small part of the complete address space actually has to be probed.

We have proposed a method to identify mapped kernel modules by comparing their memory allocation patterns to a set of known signatures. This is possible because parts of these modules are marked *pageable* or *discardable*. If no code or data could be paged-out (or even deallocated) after loading a driver, it would be impossible to detect them with our signature approach. Again, applying this protection would decrease the performance, because unpageable memory is a scarce and critical system resource.

One effective mitigation technique is to modify the execution time of the page fault handler: if there is no correlation between the current allocation state of a faulting memory address and the observable time for handling that, the timing side channel for address translation vanishes. This would hinder our attacks from Sections IV-B and IV-C. We have implemented one possible implementation for this method and verified that our measuring no longer works. To that end, we have hooked the page fault handler and *normalized* its execution time if unprivileged code raises a memory access violation on kernel memory. In that case we enforce the execution to always return back to user mode after a *constant* amount of clock cycles. For that purpose we perform a bunch of timing tests in advance to measure the timing differences for memory accesses to unallocated and allocated (for both regular and large) pages. Inside the hooked page fault handler we delay execution for the appropriate amount of time, depending on the type of memory that caused the exception. Since this happens only for software errors – or due to active probing – there is no general impact on system performance. Note that modifying the page fault handler renders our attack infeasible, but there might be other side channels an attacker can exploit to learn more about the memory layout of the kernel.

Even with normalizing the page fault handling time, our *cache probing attack* remains feasible. However, cache probing has one fundamental shortcoming: it only reveals information about *physical* addresses. If the kernel space randomization is only applied to virtual addresses, then knowing physical addresses does not help in defeating ASLR.

The kernel (or an underlying hypervisor) may also try to detect suspicious access patterns from usermode to kernelspace, for example by limiting the amount of usermode page faults for kernel space addresses. Such accesses are necessary for two of the previously described methods. While our current implementations of these attacks could be detected without much effort that way, we can introduce artificial sleep times and random access patterns to mimicry benign behavior. In the end, this would lead to an increased runtime of the exploits.

In case the attacks are mounted from within a VMM, the hypervisor might also provide the VM with incorrect information on the true CPU model and features, for example by modifying the `cpuid` return values. However, this might have undesirable side-effects on the guest operating system which also needs this information for optimizing cache usage. Furthermore, the architectural parameters of the cache (such as size, associativity, use of slice-hashing, etc.) can be easily determined from within the VM using specific tests.

Finally, the most intuitive solution would be to completely disable the `rdtsc` instruction for usermode code, since then no CPU timing values could be obtained at all. However, many usermode applications actually rely on this operation and, hence, its disabling would cause significant compatibility issues.

VI. RELATED WORK

Timing and side channel attacks are well-know concepts in computer security and have been used to attack many kinds of systems, among others cryptographic implementations [41]–[43], OpenSSL [44], [45], SSH sessions [46], web applications [47], [48], encrypted VoIP streams [49], [50], and virtual machine environments [51]–[53].

Closely related to our work is a specific kind of these attacks called *cache games* [24], [25], [54], [55]. In these attacks, an adversary analyzes the cache access of a given system and deduces information about current operations taking place. The typical target of these attacks are cryptographic implementations: while the CPU performs encryption or decryption operations, an adversary infers memory accesses and uses the obtained information to derive the key or related information. In a recent work, Gullasch et al. showed for example how an AES key can be recovered from the OpenSSL 0.9.8n implementation [24] and Zhang et al. introduced similar attacks in a cross-VM context [53].

We apply the basic principle behind cache attacks in our work and introduce different ways how this general approach can be leveraged to obtain information about the memory layout of a given system. Previous work focused on attacks against the instruction/data caches and not on the address translation cache, which is conceptually different. We developed novel approaches to attack this specific aspect of a computer system. Furthermore, all documented cache attacks were implemented either for embedded processors or for older processors such as Intel Pentium M (released in March 2003) [24], Pentium 4E (released in February 2004) [25], or Intel Core Duo (released in January 2006) [23]. In contrast, we focus on the latest processor architectures and need to solve many obstacles related to modern performance optimizations in current CPUs [22]. To the best of our knowledge, we are the first to present timing attacks against ASLR implementations and to discuss limitations of kernel space ASLR against a local attacker.

VII. CONCLUSION AND FUTURE WORK

In this paper, we have discussed a generic, timing-based side channel attack against kernel space ASLR. Such side channels emerge from intricacies of the underlying hardware and the fact that parts of the hardware (such as caches and physical memory) are shared between both privileged and non-privileged mode. We have presented three different instances of this methodology that utilize timing measures to precisely infer the address locations of mapped kernel modules. We successfully tested our implementation on four different CPUs and within a virtual machine and conclude that these attacks are feasible in practice. As a result, a local, restricted attacker can infer valuable information about the kernel memory layout and bypass kernel space ASLR.

As part of our future work, we plan to apply our methods to other operating systems such as Mac OS X and more kinds of virtualization software. We expect that they will work without many adoptions since the root cause behind the attacks lies in the underlying hardware and not in the operating system. We further plan to test our methods on other processor architectures (e.g., on ARM CPUs to attack ASLR on Android [11]). Again, we expect that timing side channel attacks are viable since the memory hierarchy is a shared resource on these architectures as well.

Another topic for future work is the identification of methods to obtain the physical address of a certain memory location from user mode. One promising method would be to identify certain data structures that are always mapped to the same physical memory and use the technique of cache probing with them. First experiments have shown that certain parts of mapped system modules are constant for a given system (e.g., the physical base address of `ntdll.dll`). Another possibility is to instrument the characteristics of the *Sandybridge* hash function to locate colliding memory locations and infer the bits of their physical address.

VIII. ACKNOWLEDGEMENTS

REFERENCES

[1] Aleph One, "Smashing the Stack for Fun and Profit," *Phrack Magazine*, vol. 49, no. 14, 1996.

[2] blexim, "Basic Integer Overflows," *Phrack Magazine*, vol. 60, no. 10, 2002.

[3] M. Conover, "w00w00 on Heap Overflows," 1999.

[4] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *USENIX Security Symposium*, 1998.

[5] Microsoft, "Data Execution Prevention (DEP)," http://support. microsoft.com/kb/875352/EN-US/, 2006.

[6] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," in *USENIX Security Symposium*, 2003.

[7] PaX Team, "Address Space Layout Randomization (ASLR)," http://pax.grsecurity.net/docs/aslr.txt.

[8] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent Runtime Randomization for Security," in *Symposium on Reliable Distributed Systems (SRDS)*, 2003.

[9] Solar Designer, ""return-to-libc" attack," Bugtraq, 1997.

[10] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[11] H. Bojinov, D. Boneh, R. Cannings, and I. Malchev, "Address Space Randomization for Mobile Devices," in *ACM Conference on Wireless Network Security (WiSec)*, 2011.

[12] M. Russinovich, "Inside the Windows Vista Kernel: Part 3," http://technet.microsoft.com/en-us/magazine/2007.04.vistakernel. aspx, 2007.

[13] Charles Miller and Dion Blazakis and Dino Dai Zovi and Stefan Esser and Vincenzo Iozzo and Ralf-Phillipp Weinmann, *iOS Hacker's Handbook*. John Wiley & Sons, Inc., 2012, p. 211.

[14] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight Kernel Protection Against return-to-user Attacks," in *USENIX Security Symposium*, 2012.

[15] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *USENIX Security Symposium*, 2012.

[16] T. Durden, "Bypassing PaX ASLR Protection," *Phrack Magazine*, vol. 59, no. 9, 2002.

[17] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the Memory Secrecy Assumption," in *European Workshop on System Security (EuroSec)*, 2009.

[18] M. Jurczyk, "Windows Security Hardening Through Kernel Address Protection," http://j00ru.vexillium.org/?p=1038, 2011.

[19] P. Akritidis, "Cling: A Memory Allocator to Mitigate Dangling Pointers," in *USENIX Security Symposium*, 2010.

[20] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors," in *USENIX Security Symposium*, 2009.

[21] H. Shacham, M. Page, B. Paff, E. jin Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-Space Randomization," in *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[22] K. Mowery, S. Keelveedhi, and H. Shacham, "Are AES x86 Cache Timing Attacks Still Feasible?" in *ACM Cloud Computing Security Workshop (CCSW)*, 2012.

[23] O. Aciiçmez, B. B. Brumley, and P. Grabher, "New Results on Instruction Cache Attacks," in *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2010.

[24] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice," in *IEEE Symposium on Security and Privacy*, 2011.

[25] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient Cache Attacks on AES, and Countermeasures," *J. Cryptol.*, vol. 23, no. 2, Jan. 2010.

[26] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software," in *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[27] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[28] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the Reliability of Commodity Operating Systems," *ACM Trans. Comput. Syst.*, vol. 23, no. 1, 2005.

[29] W. A.-K. Abu-Sufah, "Improving the Performance of Virtual Memory Computers," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1979.

[30] Intel Corporation, "Intel: 64 and IA-32 Architectures Software Developer's Manual," 2007, http://www.intel.com/products/ processor/manuals/index.htm.

[31] Intel, "TLBs, Paging-Structure Caches, and Their Invalidation," http://www.intel.com/content/www/us/en/ processors/architectures-software-developer-manuals.html.

[32] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, Inc., 2012, p. 118.

[33] D. Levinthal, "Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors," http://software.intel.com/sites/products/collateral/hpc/vtune/ performance_analysis_guide.pdf.

[34] Invisible Things Lab, "From Slides to Silicon in 3 Years!" http://theinvisiblethings.blogspot.de/2011/06/ from-slides-to-silicon-in-3-years.html, 2011.

[35] R. Hund, T. Holz, and F. C. Freiling, "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms," in *USENIX Security Symposium*, 2009.

[36] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *USENIX Security Symposium*, 2011.

[37] Giuffrida, Cristiano and Kuijsten, Anton and Tanenbaum, Andrew S., "Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security'12. USENIX Association, 2012.

[38] J. Aas, "Understanding the Linux 2.6.8.1 CPU Scheduler," http: //joshaas.net/linux/linux_cpu_scheduler.pdf, 2005.

[39] Microsoft, "Description of Performance Options in Windows," http://support.microsoft.com/kb/259025/en-us, 2007.

[40] M. Basseville and I. V. Nikiforov, *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, 1993.

[41] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *International Cryptology Conference (CRYPTO)*, 1996.

[42] M. Weiss, B. Heinz, and F. Stumpf, "A cache timing attack on aes in virtualization environments," in *Financial Cryptography and Data Security (FC)*, 2012.

[43] O. Aciiçmez, "Yet another MicroArchitectural Attack:: exploiting I-Cache," in *ACM Workshop on Computer Security Architecture (CSAW)*, 2007.

[44] O. Aciiçmez, W. Schindler, and Çetin Kaya Koç, "Improving Brumley and Boneh timing attack on unprotected SSL implementations," in *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[45] D. Brumley and D. Boneh, "Remote Timing Attacks are Practical," in *USENIX Security Symposium*, 2003.

[46] D. X. Song, D. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *USENIX Security Symposium*, 2001.

[47] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow," in *IEEE Symposium on Security and Privacy*, 2010.

[48] E. W. Felten and M. A. Schneider, "Timing Attacks on Web Privacy," in *ACM Conference on Computer and Communications Security (CCS)*, 2000.

[49] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson, "Spot Me if You Can: Uncovering Spoken Phrases in Encrypted VoIP Conversations," in *IEEE Symposium on Security and Privacy*, 2008.

[50] A. M. White, A. R. Matthews, K. Z. Snow, and F. Monrose, "Phonotactic Reconstruction of Encrypted VoIP Conversations: Hookt on Fon-iks," in *IEEE Symposium on Security and Privacy*, 2011.

[51] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[52] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *IEEE Symposium on Security and Privacy*, 2011.

[53] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM Side Channels and Their Use to Extract Private Keys," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[54] J. Bonneau and I. Mironov, "Cache-Collision Timing Attacks Against AES," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2006.

[55] C. Percival, "Cache Missing for Fun and Profit," http://www.daemonology.net/hyperthreading-considered-harmful/, 2005.

APPENDIX A.
ADDRESS RESOLUTION

Figure 8 illustrates the address resolution for regular pages (upper part) and large pages (lower part) on PAE systems. Notice that in the first case, the resulting PTE points to one single frame. In the second case, the PDE points to the first one of a set of adjacent frames, that in sum span the same size as a large page.

APPENDIX B.
DOUBLE PAGE FAULT

Figure 9 shows the double page fault measurements on an Intel i7-870 (Bloomfield) processor. It is not possible to use a simple threshold value to tell apart allocated from unallocated pages without introducing a large amount of faulty results. In the zoomed version in Figure 10, one can see that it is still possible to distinguish unallocated from unallocated pages. Note that this figure uses lines instead of dots to stress the focus on transitions from high to low values (or vice versa). We therefore use a *change point detection* (CPD) algorithm [40] in this case.
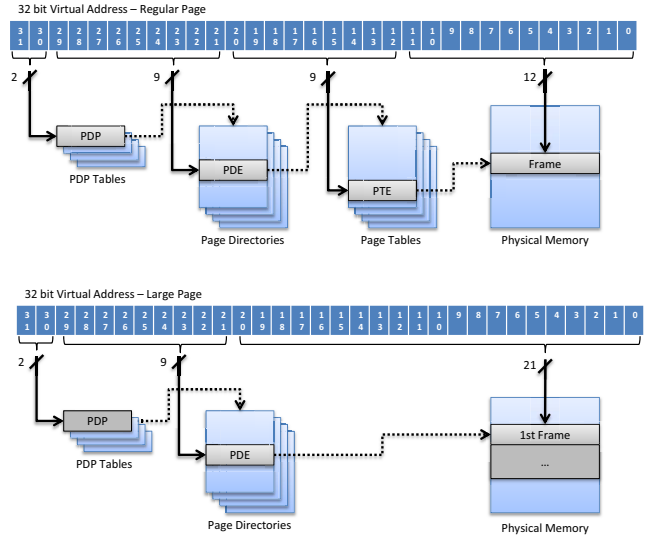

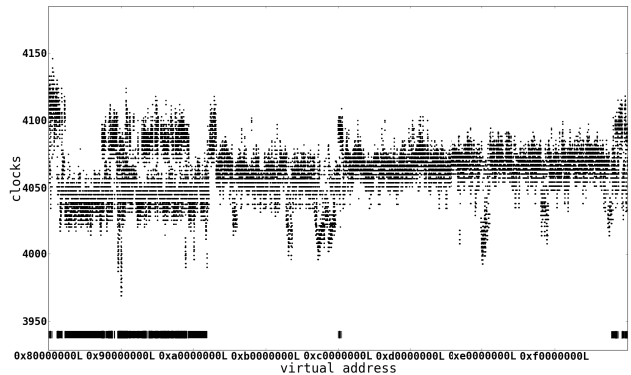Figure 8. Address resolution for regular and large pages on PAE systems

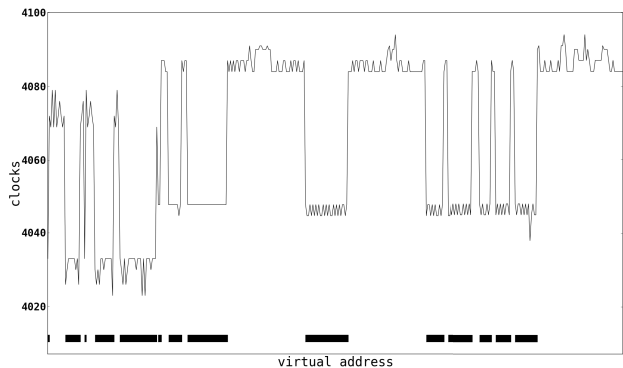Figure 9. Double page fault measurements on Intel i7-870 (Bloomfield) processor

Figure 10. Zoomed-in view of Figure 9