Performanceanalyse einer Meltdown-Mitigation

Bachelorarbeit

ausgearbeitet von

Sabrina Heidler
[redacted]

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

vorgelegt an der
Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik IV
Arbeitsgruppe für IT-Sicherheit

im Studiengang
Informatik (B.Sc.)

Erstprüfer:	Dr. Felix Jonathan Boes Universität Bonn
Zweitprüfer:	Prof. Dr. Matthew Smith Universität Bonn
Betreuer:	Dr. Felix Jonathan Boes Universität Bonn

Bonn, 30. April 2021

Danksagung

Mein hauptsächlicher Dank gilt Dr. Felix Jonathan Boes, der sowohl menschlich als auch fachlich ein hervorragender Betreuer war, immer ein offenes Ohr hatte, stets unterstützend zur Seite stand und mein Interesse für dieses Thema erst so tiefgreifend geweckt hat. Des Weiteren gilt mein Dank meiner Familie und meinen Freunden für das Korrekturlesen. Für die moralische und finanzielle Unterstützung durch meine Eltern möchte ich mich ebenfalls bedanken, da sie mir so erst meinen Studienfortschritt und das Erreichen meiner Ziele ermöglicht haben. Auch bei der Familie meines Partners möchte ich mich ganz herzlich dafür bedanken, dass mir die Möglichkeit geboten wurde, in Zeiten der Corona-Pandemie einen Arbeitsplatz außer Haus nutzen zu können, um konzentriert an dieser Bachelorarbeit schreiben zu können. Außerdem möchte ich meinem Partner einen besonderen Dank dafür aussprechen, dass er mich über ein selbstverständliches Maß hinaus unterstützt und motiviert hat.

Kurzfassung

In dieser Bachelorarbeit werden Performanceverluste, die durch die Meltdown-Mitigation *Kernel Page Table Isolation (KPTI)* entstehen, auf verschiedenen Testgeräten mit einem Intel-Prozessor und einem modernen Linux-Betriebssystem analysiert und evaluiert. Bereits bestehende Literatur geht meist nur auf die Gesamtperformance von KPTI ein, sodass unklar ist, welche Faktoren bei dieser Gesamtperformance besonderen Einfluss nehmen oder ob die angegebenen Performanceverluste die relevanten Aspekte der Änderungen an Betriebsabläufen durch KPTI vollständig darstellen. Es bleibt also offen, welche Vorgänge besonders von Performanceverlusten durch KPTI betroffen sind und mit welchen Performanceeinbußen bei diesen zu rechnen ist. Da die Meltdown-Mitigation KPTI insbesondere Veränderungen einführt, die Auswirkungen auf prozessinterne Kontextwechsel hat und dies die Ausführungszeit von Systemcalls und die Verfügbarkeit von Adressübersetzungen im *Translation Lookaside Buffer (TLB)* beeinflusst, werden diese Aspekte hier genauer untersucht. Dazu wird ein Analyseprogramm geschrieben, anhand dessen auf mehreren Geräten die Ausführungszeit von Systemcalls und die Verfügbarkeit von Adressübersetzungen im TLB nach einem Kontextwechsel untersucht wird. Dabei werden die Messwerte aus Vorgängen mit aktivierter Kernel Page Table Isolation verglichen mit den Messwerten derselben Vorgänge unter deaktivierter Kernel Page Table Isolation. Es wird festgestellt, dass die im Rahmen dieser Arbeit gemessenen Performanceverluste die in der Literatur angegebenen Einbußen durch KPTI bei Weitem übersteigen und die Größe der Performanceverluste zudem von der jeweiligen Prozessorarchitektur bedingt werden. Durch diese Arbeit wird somit eine Lücke in der wissenschaftlichen Literatur geschlossen, da genauere Angaben zu Performanceverlusten an den durch KPTI veränderten und besonders relevanten Stellen gemacht werden.

Inhaltsverzeichnis

1 Einleitung

Meltdown (CVE-2017-5754) [Com18c], ein 2018 veröffentlichter Angriff auf eine Hardware-Sicherheitslücke von Prozessoren [Lip18], ermöglicht es einem Angreifer Speicher von anderen Nutzerprozessen oder sogar den gesamten Hauptspeicher auszulesen, ohne Berechtigungen dafür zu haben. Meltdown wird oft auch *Rouge Data Cache Load (RDCL)* genannt. Ein zentraler Aspekt der Sicherheit von Computersystemen im Hinblick auf die Vertraulichkeit ist Memory Isolation. Dabei stellt das Betriebssystem sicher, dass Nutzeranwendungen gegenseitig nicht auf ihren Speicher zugreifen können und hindert diese auch daran den Kernel-Speicher auszulesen. Der Meltdown-Angriff nutzt Optimierungen moderner Prozessoren wie Out-Of-Order Execution aus, um diese Sicherheitsmechanismen zu umgehen und somit in Kombination mit cachebasierten Seitenkanälen unberechtigt Speicherbereiche auszulesen. Dadurch können unter anderem persönliche Daten, wie zum Beispiel Passwörter oder private Schlüssel ausgelesen werden. Primär betroffen sind davon Intel-Prozessoren, die in Millionen von Computern verbaut sind und dieser Thematik somit große Relevanz verleihen.

Damit effiziente Übergänge vom Nutzerprozess zum Kernel stattfinden können, um beispielsweise Interrupts oder Hardware-Zugriffe zu behandeln, wird der Kernel in den Adressraum jedes Prozesses abgebildet. Die Isolierung zwischen dem Kernel und dem Nutzerprozess wird dabei von einem Supervisor-Bit realisiert, welches definiert ob auf eine Page im Speicher zugegriffen werden darf. Dieses Bit wird beim Eintritt in den Kernel gesetzt und beim Wechsel zurück zum Nutzerprozess wieder entfernt, sodass ein Zugriff auf die entsprechenden Kernel-Pages nur während der Kernel aktiv ist möglich sein sollte. Meltdown macht sich bei verwundbaren CPUs Race Conditions bei der Out-Of-Order Execution und der Zugriffsberechtigungs-Überprüfung zunutze. Findet ein unerlaubter Speicherzugriff statt, wird eine Exception und ein Zugriffsverletzungssignal ausgelöst. Bauen auf den Daten dieses Speicherzugriffs jedoch weitere simple Befehle auf, kann es passieren, dass die CPU auch die folgenden Instruktionen bereits vor dem hervorrufen der Exception ausgeführt hat. Dadurch ergeben sich mikroarchitekturelle Veränderungen der Hardware, die auch nach dem Behandeln der Exception noch bestehen. So sind beispielsweise im CPU-nahen Hardware-Cache die Daten der ausgeführten Befehle auch nach der ausgelösten Exception noch vorhanden. Über die geschickte Verwendung eines cachebasierten Seitenkanals können so die geheimen Informationen im Cache codiert und anschließend dennoch ausgelesen werden.

Um die Meltdown-Verwundbarkeit zu eliminieren, sind mehrere hardware- als auch softwarebasierte Mitigations entwickelt worden. Zu den Software-Mitigations zählen *„Kernel Page Table Isolation“ (KPTI)* [Gru18], *„Kernel Address Isolation to have Sidechannels Efficiently Removed“ (KAISER)* [Gru17b] und *„Kernel Address Space Layout Randomization“ (KASLR)* [Can20]. Diese Mitigations haben einerseits zum Ziel zu verhindern, dass die entsprechenden Adressen der Kernel-Pages überhaupt herausgefunden werden können (KASLR). Andererseits besteht das Ziel darin Kernel-Pages

so sinnvoll zu verstecken, dass, selbst wenn Adressen relevanter Kernel-Pages herausgefunden werden können, ein Zugriff auf diese Kernel-Inhalte aus dem Nutzerprozess nicht mehr möglich ist. Zweiteres ist insbesondere deshalb relevant, da KASLR mit einem Meltdown-Angriff gebrochen werden kann und somit keine allumfassende Prävention gegen Meltdown darstellt [Can20].

Im Zuge dieser Bachelorarbeit werden die Funktionsweisen der verschiedenen Meltdown-Mitigations beleuchtet. Darauf wird in Kapitel 5 näher eingegangen. Der Fokus dieser Bachelorarbeit liegt auf der Untersuchung und Evaluation von Performanceverlusten der in Linux Betriebssysteme eingebundenen softwarebasierten Meltdown-Mitigation KPTI. Dies wird in Kapitel 6 behandelt. Um die Konzeption und Ansatzpunkte von Meltdown-Mitigations verständlicher zu machen, wird in Kapitel 4 auf die technische Funktionsweise des Meltdown-Angriffs eingegangen. Damit der Meltdown-Angriff nachvollziehbar erläutert werden kann, werden in Kapitel 3 die Grundlagen cachebasierter Seitenkanäle beschrieben. Zudem werden in Kapitel 2 die allgemein zum Verständnis der Thematik nötigen Grundlagen von Betriebsabläufen, Hardwarestrukturen und Funktionskonzepten behandelt.

2 Grundlagen

In diesem Kapitel werden die zum Verständnis nötigen Grundlagen behandelt. In Abschnitt 2.1 werden relevante Vorgänge der Speicherverwaltung bei der Verwendung von virtuellem Speicher erklärt. Für einen erfolgreichen Meltdown-Angriff werden Mechanismen der Prozessoroptimierung ausgenutzt. Daher wird in Abschnitt 2.2 auf CPU Optimierungen eingegangen. Da cachebasierte Seitenkanäle bei einem Meltdown-Angriff von besonderer Bedeutung sind, behandelt Abschnitt 2.3 die Grundlagen zu CPU-nahen Hardware-Caches.

2.1 Speicherverwaltung

In diesem Abschnitt wird auf die Speicherverwaltung mit besonderem Hinblick auf Virtuellen Speicher eingegangen. Dabei liegt der Fokus auf modernen x86 Linux Betriebssystemen mit Intel-Prozessoren. Dazu wird in Abschnitt 2.1.1 zunächst erklärt, was Virtueller Speicher und Paging ist. Abschnitt 2.1.2 geht auf die Adressübersetzung von virtuellen zu physischen Adressen ein. Es wird die effektive Zugriffszeit durch Paging besprochen (Abschnitt 2.1.3), um die Verwendung von Adressübersetzungs-Caches (Abschnitt 2.1.4) herzuleiten. Zuletzt wird kurz auf Shared Memory bzw. Memory Deduplication eingegangen (Abschnitt 2.1.5).

2.1.1 Virtueller Speicher und Paging

Die Grundidee von Virtual Memory ist, dass jedes in Ausführung befindliche Programm seinen eigenen Adressraum zur Verfügung hat. Der Adressraum eines Prozesses wird *virtueller Adressraum* genannt und stellt eine aufeinanderfolgende Menge von virtuellen Speicheradressen bereit, die der Prozess während der Ausführung verwenden kann. Typische Segmente eines Prozesses sind beispielsweise der Stack für temporäre lokale Daten, der Heap, um dynamisch Speicher zu allokieren oder das Code bzw. Text Segment, in dem die auszuführenden Instruktionen des Prozesses gespeichert sind. Auf einem x86-64-Bit-System können die virtuellen Adressen eines Nutzerprozesses bis zu 48 Bit lang sein. Die restlichen 64-Bit-Adressen sind für das Betriebssystem reserviert. Somit hat ein Nutzerprozess typischerweise einen Adressraum mit virtuellen Adressen von $0x0$ bis $0x7FFF\ FFFF\ FFFF$ zur Verfügung [Lin]. Daten des Betriebssystems, die in Nutzerprozesse abgebildet werden, liegen üblicherweise in einem Adressbereich von $0x8000\ 0000\ 0000$ bis $0xFFFF\ FFFF\ FFFF\ FFFF$ [Lin]. Der virtuelle Adressraum ist für jeden Prozess privat und kann von anderen Prozessen, die gleichzeitig ausgeführt werden, nicht genutzt werden. [Tan15; Mic]

Eine virtuelle Adresse stellt jedoch nicht die tatsächliche physische Position im Hauptspeicher (RAM) dar. Das bedeutet, es muss beim Zugriff auf eine virtuelle Adresse zunächst eine Übersetzung zur physischen Adresse stattfinden. Der gesamte virtuelle und physische Speicher wird in gleich große Blöcke unterteilt, die *Page* genannt werden und eine feste Größe von meist 4 KB haben. Jede Page beinhaltet einen zusammenhängenden Bereich von Adressen. Virtuelle Pages

werden auf zugehörige physische Pages (Pageframe) im physischen Speicher abgebildet. Dabei müssen sich jedoch nicht alle Pages gleichzeitig im Hauptspeicher befinden, um das Programm auszuführen, sondern lediglich die Pages, die zum aktuellen Zeitpunkt in der Ausführung benötigt werden. Zudem liegen die benötigten Speicherinhalte nicht notwendigerweise in der verwendeten Reihenfolge und zusammenhängend im Hauptspeicher. Pages, die Daten eines laufenden Programms enthalten, können also physisch ungeordnet gespeichert werden, während der virtuelle Adressraum unfragmentiert zur Verfügung gestellt wird. [Tan15; Nul14]

Um die Zuordnungen von virtuellen Pages zu physischen Pages zu verwalten, werden *Page Tables* verwendet. Dies ist eine interne Datenstruktur, die zur Übersetzung virtueller Adressen in die entsprechenden physischen Adressen verwendet wird. Neben dem aktuellen Speicherort einer Page, enthalten die Einträge einer Page Table weitere Informationen zu der jeweiligen Page. Diese geben beispielsweise an, ob sich eine Page im physischen Hauptspeicher befindet oder ob sie ausgelagert wurde. Außerdem sind Berechtigungen für die jeweilige Page angegeben, also zum Beispiel ob Lese-, Schreib- oder Ausführungsrechte vorhanden sind oder ob eine Page ausschließlich vom Betriebssystem und nicht vom Nutzerprozess verwendet werden darf. [Pat16]

Das Ein- und Auslagern von Pages in bzw. aus dem Hauptspeicher bezeichnet man als *Swapping*[1]. Dabei werden zwischen RAM und Festplatte stets ganze Pages transferiert. Soll auf eine Page zugegriffen werden, die sich derzeit nicht im Hauptspeicher befindet, wird dies *Page Fault* genannt. Die benötigte Page muss dann zunächst wieder in den Hauptspeicher geladen werden, bevor darauf zugegriffen werden kann. [Tan15]

Motivation und Zweck

Die ursprüngliche Motivation für eine Abstraktion von Speicheradressen durch virtuelle Adressen ist eine Vereinfachung bei der Verwendung verschiedener Speichermedien. Die Idee ist 1956 entstanden, als noch Röhrenschaltungen, Kernspeicher als Hauptspeicher und Trommelspeicher als Sekundärspeicher standardmäßig verwendet werden. Durch virtuelle Adressen ist es nicht mehr nötig, den Zugriff auf verschiedene Speichermedien manuell zu programmieren oder überhaupt von deren spezifischer Existenz zu wissen. Dies ist der Fall, da virtuelle Adressen physischen Speicher vereinheitlichen und verschiedene Speichermedien zusammenfassen, indem mögliche physische Speicherstellen über eine virtuelle Adresse referenziert und automatisch korrekt übersetzt werden. [Jes96]

Heutzutage werden die Vorteile in der Verwendung von virtuellem Speicher eher im Umgang mit geringer Hauptspeicherkapazität und Sicherheitsaspekten der Prozess Isolation gesehen. Derzeit ist der für einen 64-Bit-Prozess verwendbare virtuelle Adressraum auf aktuellen x86-64-Prozessoren üblicherweise 256 TB groß. Der physische Hauptspeicher ist oft deutlich kleiner und bietet meist eine Speicherkapazität von bis zu einigen Gigabyte [Gra17]. Durch Virtual Memory und Swapping[1] ist es möglich, die Festplatte als „Erweiterung des Hauptspeichers“ zu verwenden. Dadurch ist es möglich, dass auch mehrere Programme gleichzeitig ausgeführt werden können, die in ihrer Summe mehr physischen Speicher benötigen, als der Hauptspeicher in Wirklichkeit zur Verfügung hat. Auf die Festplatte ausgelagerte Pages werden bei Bedarf wieder zurück in den RAM geholt,

[1] Ist kein Platz mehr im Hauptspeicher vorhanden, muss entschieden werden, welche aktuell im Hauptspeicher befindliche Page ersetzt und ausgelagert werden kann, um neuen Platz zu schaffen. Da die dafür benötigten Verdrängungsstrategien für den Kontext dieser Arbeit jedoch keine große Relevanz haben, wird hier nicht näher darauf eingegangen.

wenn der Prozess sie tatsächlich benötigt. Da Prozesse nur ihren eigenen Adressraum kennen, kann nicht versehentlich oder unberechtigt auf den Adressraum eines anderen Prozesses zugegriffen werden[2], sodass aktive Programme besser voneinander isoliert sind. Auch das Betriebssystem kann so vor fehlerhaften oder böswilligen Anwenderprogrammen geschützt werden[2], da während der Übersetzung von virtuellen Adressen auch die Zugriffsberechtigungen für die jeweilige Adresse geprüft werden. Zudem erleichtert virtueller Speicher auch heute noch die Organisation von Speicheradressen. Mehrere aktive Programme können gleichzeitig dieselben virtuellen Adressen ihres virtuellen Adressraums verwenden und dort unterschiedliche Daten liegen haben, da diese virtuellen Adressen auf unterschiedliche physische Adressen abgebildet werden. Bei der Erstellung von ausführbaren Programmen kann die Belegung von virtuellen Speicherstellen für einen Prozess somit unabhängig von den verwendeten Adressen anderer Programme durchgeführt werden, da sie nur für den jeweiligen Prozess gelten. [Tan15; Nul14]

2.1.2 Adressübersetzung

Wenn Virtual Memory verwendet wird, korrespondiert eine Adresse aus einem virtuellen Adressraum nicht direkt mit der gleichwertigen physischen Adresse. Die virtuelle Adresse muss zunächst übersetzt werden. Die Zuordnung von virtuellen zu physischen Adressen übernimmt die *Memory Management Unit (MMU)*. Diese verwendet zur Übersetzung einer Adresse die zugehörige Page Table Datenstruktur. In modernen Systemen wird meist jedoch nicht eine einzige Page Table verwendet, sondern mehrstufige Page Tables. Dies ist wesentlich speicherplatzeffizienter und flexibler, wodurch eine performantere Adressübersetzung ermöglicht wird. Dabei wird eine virtuelle Adresse in Blöcke für jede Stufe so aufgeteilt, dass die Bits des jeweiligen Blocks auf einen Eintrag verweisen, der auf die Page Table der nächsten Stufe verweist. So werden die verschiedenen Page Tables traversiert, bis der Vorgang der Adressübersetzung bei der letzten Page Table Stufe angelangt ist. Dort ist die gesuchte Basis Adresse des benötigten Pageframes zu finden. Mehrstufige Page Tables können somit als unidirektionaler Baum gesehen werden. Das Nachschlagen von Adressen in der Page Table Hierarchie wird auch als *Page Walk* bezeichnet. Die niederwertigsten Bits einer virtuellen Adresse, die einen Speicherbereich adressieren, der kleiner als die Größe einer Page ist, wird Offset genannt. Zusammen mit der Page Basis Adresse, die durch den Page Walk herausgefunden wird, ergibt der Offset die finale physische Adresse. Der Offset innerhalb einer physischen Page ist der Gleiche wie bei einer virtuellen Page und muss nicht übersetzt werden. Ein Page Table Walk findet somit auf einer Granularität von Page Einheiten statt. [Pat16; Tan15]

Eine Page ist üblicherweise 4 KB = 4096 Byte = 2^{12} Byte groß, sodass die niedrigsten 12 Bit einer Adresse den Offset innerhalb einer Page beschreiben. Die aktuelle x86-64-Architektur verwendet nur die unteren 48 Bits für die virtuelle Adressierung von Nutzerdaten. Somit verbleiben 36 Bits, die einen eindeutigen Pfad von der Wurzel des Baumes zu dem Blatt, auf dem die physische Basisadresse der gesuchten Page gespeichert ist, definieren. Auf Intel Architekturen werden meist vier Stufen von Page Tables verwendet. Jede Page Table enthält 512 = 2^9 Einträge. Ein Page Table Eintrag ist 8 Byte = 64 Bit groß. Da eine Page Table somit genauso viel Speicherplatz einnimmt wie eine Page (512 * 8 Byte = 4096 Byte), werden die referenzierten Page Tables oft auch Page Table Page genannt [Bar10]. Da jede Page Table 2^9 Einträge enthält, bestimmen in jeder Page Table Ebene

[2] Hiervon ausgenommen sind natürlich Angriffsstrategien wie beim Meltdown-Angriff, die einen unberechtigten Zugriff auf Speicherinhalte über Prozessgrenzen hinweg realisieren (Vgl. Kapitel 4)

9 der oben erwähnten 36 Bits über den Offset des Page Table Eintrags innerhalb der Page Table Page. [Gra17; Int19]

Page Tables werden im Hauptspeicher abgelegt. Die Page Table aus dem hierarchisch obersten Level wird bei Intel *Page Map Level 4 (PML4)* genannt. Der physische Speicherort der PML4 muss bekannt sein, ohne vorher eine Adressübersetzung durchführen zu müssen. Auf Intel-Prozessoren wird dies durch die Verwendung des Kontroll-Registers CR3 erreicht. Im CR3 Register befindet sich ein Zeiger auf die physische Adresse der PML4 des aktuell ausgeführten Prozesses. Die Einträge der Page Tables der verschiedenen Stufen enthalten jeweils einen Pointer auf die Page Table der nächsten Stufe, Zugriffsrechte und Speicherverwaltungsinformationen. Die nach der PML4 hierarchisch folgenden Page Tables werden in Reihenfolge ihrer Verwendung wie folgt genannt: Page Directory Pointer Table (PDPT), Sets of Page Directories (PD), Sets of Page Tables (PT). [Int19]

Der in diesem Abschnitt beschriebene Vorgang der Adressübersetzung ist schematisch in Abbildung 1 zu sehen.

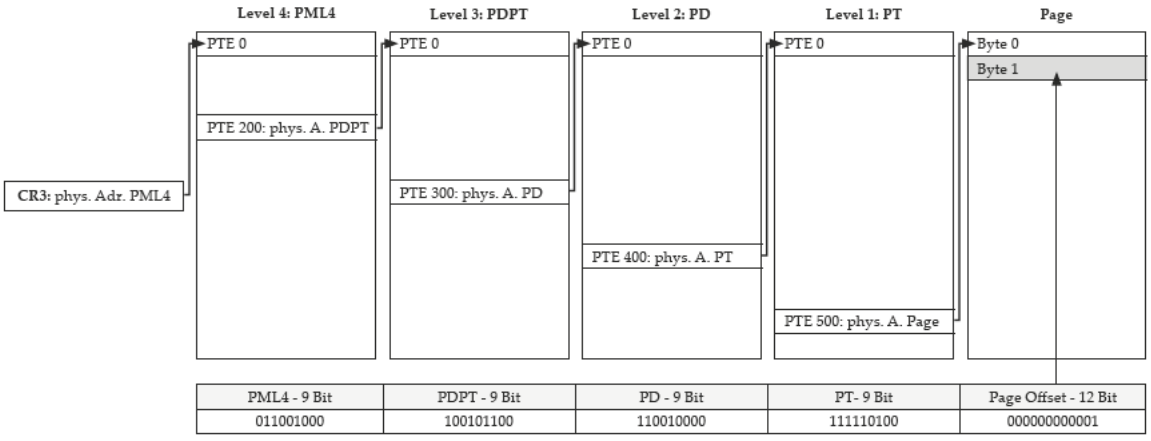


Abbildung 1: *Exemplarischer Page Table Walk, um die Adresse 0x644B321F4001 zur korrespondierenden physischen Adresse zu übersetzen*

2.1.3 Effektive Zugriffszeit durch Paging

Durch die Verwendung von Virtual Memory und Paging entstehen Zeit Einbußen beim Zugriff auf Speicherinhalte des Hauptspeichers. Für jeden Speicherzugriff, der von einem Prozess ausgelöst wird, müssen effektiv mindestens zwei Zugriffe auf den physischen Speicher stattfinden – einer, um die Page Table zu referenzieren und einer, um auf die eigentlichen Daten zuzugreifen. In Systemen mit mehrstufigen Page Tables muss sogar in jeder Stufe ein weiterer Zugriff auf den Hauptspeicher stattfinden, um auf die jeweilige Page Table zuzugreifen. Bei einer vierstufigen Page Table Hierarchie müssen für einen Speicherzugriff aus dem Prozess folglich insgesamt fünf Speicherzugriffe getätigt werden. [Nul14]

Angenommen, es wird nur eine einzige Page Table zur Adressübersetzung verwendet, ein Zugriff auf den Hauptspeicher benötigt 200 ns und es besteht eine Page Fault Rate von 1 % – das bedeutet 99 % der Zeit befinden sich die benötigten Pages im Speicher. Um eine Page in den Hauptspeicher zu laden, werden des weiteren 10 ms (= 10.000.000 ns) angenommen. Diese Zeit beinhaltet die

benötigte Zeit, um die Page in den Speicher zu holen, die Page Table zu aktualisieren und auf diese Daten zuzugreifen. Die effektive Zugriffszeit (EAT), die insgesamt für einen Speicherzugriff benötigt wird, ist:

$$\begin{aligned} EAT = (1 - \textit{Page Fault Rate}) \cdot (\textit{Page Table Zugriffszeit} + \textit{Daten Zugriffszeit}) \\ + \textit{Page Fault Rate} \cdot \textit{Page Fault Handling Zeit} \end{aligned}$$

Somit ergibt sich anhand der Beispielzeiten die folgende effektive Zugriffszeit:

$$EAT = 0,99 \cdot (200\ ns + 200\ ns) + 0,01 \cdot 10.000.000\ ns = 100.396\ ns$$

Sogar wenn 100 % der benötigten Pages im Hauptspeicher vorhanden sind und somit keine Page Faults auftreten, beträgt die Effektive Zugriffszeit 400 ns, was das doppelte an Zugriffszeit eines einzelnen Hauptspeicherzugriffs ist. Dieser Prozess kann jedoch durch die Verwendung von Pufferspeichern, die Adressübersetzungen zwischenspeichern, wie folgt beschleunigt werden. [Nul14]

2.1.4 Verwendung von Adressübersetzungs-Caches

Ein mehrstufiger Übersetzungsvorgang kann, wie oben beschrieben, sehr aufwendig und zeitintensiv sein. Darum wird ein Pufferspeicher verwendet, der vorherige Adressübersetzungen zwischenspeichert. Der *Translation Lookaside Buffer (TLB)* ist ein solcher Adressübersetzungs-Cache. Die meisten Programme neigen dazu, viele Speicherverweise auf eine kleine Anzahl von Pages durchzuführen. Daher wird nur ein kleiner Teil der Page Table Einträge vermehrt gelesen. Die restlichen Einträge werden kaum benutzt. Muss eine Übersetzung einer virtuellen Adresse stattfinden, ist die Wahrscheinlichkeit also relativ hoch, dass die Adresse des entsprechenden Pageframes in naher Zukunft noch einmal benötigt wird. Somit kann durch ein Zwischenspeichern dieser Adressübersetzungen ein erneutes Durchlaufen des Page Walks bei jedem Speicherzugriff vermieden werden. [Tan15]

Wird eine Übersetzung einer virtuellen Adresse benötigt, überprüft der Prozessor zunächst, ob die nötige Übersetzung im TLB gespeichert ist. Wenn sie dort vorhanden ist, wird das als TLB Hit bezeichnet. Ist die benötigte Adressübersetzung nicht im TLB vorhanden, bezeichnet man dies als TLB Miss. Der Page Walk muss dann durchlaufen werden, bevor auf die angefragten Daten zugegriffen werden kann. Zudem wird anschließend die Übersetzung für eine zukünftige Verwendung im TLB gespeichert. [Tan15]

Der Einfluss von TLB Misses auf die Gesamtsystemleistung kann zwischen 5 % - 14 % liegen. Um die Auswirkungen von TLB Misses weiter zu verringern, werden bei Intel mehrere TLBs verwendet, sodass zusätzlich für jede Stufe der Übersetzungstabellen ein eigener Cache verwendet wird. [Bar10]

Jeder TLB Eintrag enthält die Virtual Page Number und die zugehörige Pageframe Number. Die Informationen über die jeweilige Page werden ebenfalls im TLB gespeichert und haben eine Eins-zu-Eins-Korrespondenz zu den in der Page Table gespeicherten Informationen über eine Page. Die Größe eines TLBs variiert meist zwischen 16 bis 512 Einträgen. 0,5 bis 1 Zykel werden bei

einem TLB Hit benötigt. Ein TLB Miss tritt üblicherweise mit einer Rate von 0,01 % - 1 % auf und bringt eine Miss Penalty von 10 bis 100 Zyklen mit sich. Bei einem TLB Miss kann es passieren, dass die Page nicht einmal im Hauptspeicher vorhanden ist. Dann muss der aufgetretene Page Fault zusätzlich behandelt werden, was deutlich mehr Zeit kostet (Vgl. siehe oben), aber auch deutlich seltener als ein einfacher TLB Miss auftritt. [Tan15; Pat16]

2.1.5 Geteilter Speicher

Durch die Abbildung von virtuellen auf physische Adressen kann ein physischer Bereich auch von mehreren Prozessen gleichzeitig verwendet werden. Dies bezeichnet man als *Shared Memory* oder *Memory Deduplication*. Die gemeinsame Nutzung von Speicher zwischen Prozessen kann zwei verschiedenen Zielen dienen: Es kann als Interprozess-Kommunikationsmechanismus zwischen zwei kooperierenden Prozessen verwendet werden und es kann zur Reduzierung des Speicherbedarfs genutzt werden, indem redundante Kopien identischer Inhalte vermieden werden. Um eine Unterscheidung zu erleichtern, wird Ersteres im Folgenden mit „Shared Memory“ und Letzteres mit „Memory Deduplication“ bezeichnet. Von mehreren Prozessen genutzte Pages werden somit nur einmal in den physischen Speicher geladen, statt eine Kopie der Page für jeden Prozess, der sie benötigt, im Speicher zu halten. Mehrere Prozesse können so über eine potenziell verschiedene, in ihrem Adressraum gültige virtuelle Adresse gleichzeitig auf ein und dieselbe physische Page zugreifen. Um insbesondere zwischen nicht kooperierenden Prozessen die Isolation aufrecht zu erhalten, sind die durch Memory Deduplication geteilten Daten nur mit lesendem oder Copy-On-Write Zugriff verfügbar. Das *Context-Aware Sharing* identifiziert identische Pages anhand des Speicherorts. Auf diese Weise können mehrfache Kopien vermieden werden, sodass beispielsweise gemeinsam genutzte Bibliotheken (Shared Libraries) nur einmal in den Speicher geladen und zwischen Prozessen geteilt werden. [Yar14]

2.2 Prozessor und Optimierungen

Alle Computer haben eine *Central Processing Unit (CPU)*, die dafür verantwortlich ist, dass ein Computer ein Programm ausführen und die Daten korrekt verarbeiten kann. Jede CPU enthält eine interne Uhr (Clock), die durch Takte regelt, wie schnell Anweisungen ausgeführt werden können. Diese Uhr synchronisiert alle Komponenten des Systems, sodass auch der Prozessor eine feste Anzahl an Taktzyklen benötigt, um jeden Befehl auszuführen. Daher wird der Durchsatz oft in Taktzyklen und nicht in Sekunden gemessen. [Nul14]

Aufgrund diverser technischer Errungenschaften ist es im Laufe der Zeit möglich geworden, die Verarbeitungsgeschwindigkeit von Befehlen zu erhöhen. Zu den Optimierungen gehören unter anderem eine Erhöhung der Taktfrequenz, Pipelining, die Verwendung von mehreren CPU-Kernen, spekulative Ausführung oder Out-Of-Order Ausführung. Pipelining bezeichnet das Aufteilen von Instruktionen in verschiedene Phasen, sodass bestimmte Berechnungen parallel ausgeführt werden können. Bei der spekulativen Ausführung macht der Prozessor eine Vorhersage über die weitere Programmausführung und führt für diese bereits Berechnungen durch. War diese Vorhersage korrekt, liegen die berechneten Informationen bereits vor, wenn sie benötigt werden. Andernfalls verwirft die CPU die Ergebnisse wieder. Wie der Name schon sagt, werden bei der *Out-Of-Order Execution* Befehle in einer veränderten Reihenfolge abgearbeitet. Programmcode liegt in serieller

Form vor. Instruktionen mit weiteren Datenabhängigkeiten können nicht sofort ausgeführt werden, da diese Datenabhängigkeiten zunächst aufgelöst werden müssen. Eine solche Datenabhängigkeit ist beispielsweise ein noch durchzuführender Speicherzugriff. Das Laden von Inhalten aus dem Speicher benötigt viel Zeit (Vgl. Abschnitt 2.3). Um größere Leerlaufzeiten zu vermeiden, während der Prozessor auf die in Auftrag gegebene Auflösung dieser Datenabhängigkeiten wartet, werden in der Zwischenzeit nachfolgende Instruktionen ausgeführt, bevor die vorherige Instruktion fertig bearbeitet wurde. Instruktionen werden in Mikro-Operationen decodiert und zusammen mit ihren Operanden im Re-Order Buffer (ROB) abgelegt. Während auf weitere Operanden gewartet wird, werden in der Zwischenzeit die Mikro-Operationen eingeplant, deren Operanden bereits verfügbar sind. Dadurch werden Anweisungen außer der Reihe ausgeführt. Die Ergebnisse der so ausgeführten Instruktionen werden zwischengespeichert, bis sie benötigt und abgeholt werden. [Gru17a; Can20]

Wenn festgestellt wird, dass sich durch den weiteren Programmverlauf die Ergebnisse von Instruktionen, die bereits spekulativ oder Our-Of-Order ausgeführt wurden, verändern und nicht mehr korrekt sind, werden diese verworfen und Registerzustände revidiert. Solche Operationen können jedoch auch Einfluss auf diverse mikroarchitekturelle Zustände haben. Beispielsweise hat ein Befehl zum Laden von Daten, der Out-Of-Order ausgeführt wurde, Auswirkungen auf den Zustand des CPU-nahen Hardware-Caches (Vgl. Abschnitt 2.3). Solche mikroarchitekturellen Zustände werden nicht revidiert. [Koc19]

Mit immer performanter werdenden Prozessoren sind die langsamen Zugriffszeiten auf den physischen Hauptspeicher (DRAM) zum neuen Flaschenhals geworden. Aus diesem Grund sind Caches für CPUs eingeführt worden, auf die im Folgenden näher eingegangen wird.

2.3 CPU-nahe Hardware-Caches

In modernen Systemen werden verschiedene Arten von Caches verwendet, wie z. B. der TLB (Vgl. Abschnitt 2.1.4). Wird im Folgenden ausschließlich von „Cache“ gesprochen, ist damit der CPU-nahe Hardware-Cache gemeint. Ein Cache-Speicher ist ein kleiner Hochgeschwindigkeitsspeicher, der als Puffer zwischen dem Prozessor und dem physischen Hauptspeicher verwendet wird, um die Betriebsgeschwindigkeit zu erhöhen. Der Cache-Speicher hat eine Zugriffszeit, die wesentlich kürzer ist als die Zugriffszeit des Hauptspeichers. Typische Zugriffszeiten auf den DRAM betragen meist etwa 60 ns, wohingegen die Zugriffszeit auf den Cache in der Regel etwa 10 ns beträgt. Benötigt die CPU Daten aus einer referenzierten Speicherstelle, so sucht sie zuerst danach im Cache. Wird die zugehörige Adresse im Cache-Speicher gefunden, bezeichnet man dies als Cache Hit. Der Prozessor erhält die im Cache gespeicherten Daten und kann mit seinen weiteren Berechnungen fortfahren. Wird die zugehörige Adresse nicht im Cache-Speicher gefunden, bezeichnet man dies als Cache Miss. In diesem Fall muss auf den vergleichsweise langsamen Hauptspeicher zugegriffen werden, um von dort die benötigten Daten zu erhalten. Diese Daten werden anschließend auch im Cache zur zukünftigen Verwendung abgelegt. Somit können Caches verwendet werden, um das Vorkommen großer Latenzzeiten durch den langsamen Hauptspeicher zu minimieren. [Nul14][Lef91]

2.3.1 Cache Organisation

Da der Cache eine geringere Kapazität hat als der Hauptspeicher, hat er einen kleineren Adressraum. Um diesen kleineren Adressraum bedienen zu können, verwendet die CPU ein Abbildungsschema, welches die Adresse des Hauptspeichers in einen Speicherort des Caches übersetzt. Diese Adressübersetzung erfolgt, indem den Bits der Hauptspeicheradresse besondere Bedeutung zugemessen wird. Je nach Abbildungsschema werden die Bits der Adresse in zwei oder drei Felder unterteilt. Es wird zunächst einmal die Aufteilung der Hauptspeicheradresse betrachtet, bevor im folgenden Verlauf noch näher auf die verschiedenen Abbildungsschemata eingegangen wird. Bei Dircetly-Mapped Caches und Set-Associative Caches findet eine Unterteilung in Tag, Cache-Index und Offset statt, wie auch in Abbildung 2 zu sehen ist. Für die Adressierung von Fully-Associative Caches wird die Adresse nur in Tag und Offset unterteilt. [Nul14]

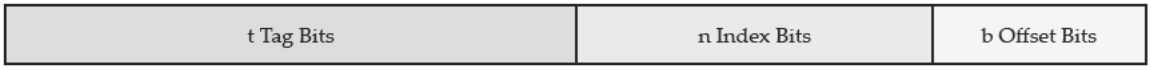


Abbildung 2: *Die Aufteilung der Bits der Hauptspeicheradresse in die drei Felder Tag, Index und Offset*

Bei einer Unterteilung in drei Bereiche werden die mittleren n Bits der Speicheradresse als *Cache-Index* verwendet. Dieser bestimmt, an welcher Stelle im Cache vom Prozessor nach den entsprechenden Daten gesucht werden soll. Verschiedene Adressen mit denselben mittleren n Bits bezeichnet man als kongruent, da sie derselben Speicherstelle zugeordnet werden. Somit konkurrieren die DRAM-Speicherbereiche um ihren Platz im Cache[3]. Die jeweilige Speicherstelle im Cache besteht aus einem Tag und den gepufferten Daten. Hauptspeicher und Cache sind beide in gleich große Blöcke unterteilt. Werden Daten im Cache zwischengespeichert, befindet sich an der jeweiligen Speicherstelle immer der gesamte Datenblock. Die niedrigsten b Bits der Adresse werden als Offset innerhalb dieser im Cache gespeicherten Daten verwendet. Die Anzahl der für den Offset vorgesehen Bits bestimmt mit 2^b die Größe der Daten, die zwischengespeichert werden können. Die meisten modernen Prozessoren haben pro Speicherstelle eine Größe von 64 Byte, sodass mit 6 Offset Bits 2^6 Byte an zugehörigen Daten gespeichert werden können [Gru17a]. Der *Tag* wird dazu verwendet, um festzustellen, ob eine Speicherstelle im Cache derzeit die Daten einer bestimmten Speicheradresse enthält, indem die vordersten t Bits der Hauptspeicheradresse als Tag mit dem im Cache hinterlegten Tag verglichen werden. [Nul14]

Es wird zwischen den folgenden drei Arten von Caches unterschieden:

Directly-Mapped Caches:
Ein Directly-Mapped Cache ist die einfachste Form eines CPU-Caches. Die Speicherstelle im Cache, an der die Daten zwischengespeichert werden, nennt man hier *Cache Line*. Wird nach einer bestimmten Adresse im Cache gesucht, schaut der Prozessor an der durch den Cache-Index vorgegebenen Stelle, ob die Tags der Adresse und der Cache Line übereinstimmen. Je nach Ergebnis wird so ein Cache Miss oder ein Cache Hit festgestellt. [Gru17a]

Ein großes Problem der Directly-Mapped Caches ist, dass sie nur eine einzige Speicherstelle aus allen möglichen kongruenten Speicherstellen für einen bestimmten Index in der jeweiligen Cache Line speichern können. Arbeitet der Prozessor an mehreren kongruenten Speicherstellen,

verdrängen diese sich gegenseitig aus dem Cache, sodass viele Cache Misses entstehen und der Performancevorteil der Verwendung eines Caches verloren geht. [Gru17a]

Fully-Associative Caches:
Anstatt für jeden Hauptspeicherblock einen vordefinierten Speicherort im Cache anzugeben, wie es bei Directly-Mapped Caches der Fall ist, kann bei einem Fully-Associative Cache der jeweilige Inhalt an beliebiger Stelle zwischengespeichert werden[3]. Es werden daher keine Bits für die Indizierung einer Cache Line benötigt, sodass die Bits der Adresse nur in Tag und Offset aufgeteilt werden. Die einzelnen Einträge des Caches werden hier nicht Cache Line, sondern *Cache Way* genannt, da es sich um einen assoziativen Cache-Speicher ohne Reihenfolge handelt. [Gru17a]

Um den gesuchten Cache Way zu finden, werden bei einem Fully-Associative Cache folglich alle sich im Cache befindlichen Tags mit dem gesuchten Tag verglichen. Da für diese Art von Cache assoziative Speicher verwendet werden, ist es möglich, den Cache parallel zu durchsuchen, um möglichst schnell Ergebnisse zu erhalten. Diese Hardware-Komponente ist preislich jedoch recht kostenintensiv. [Nul14]

Set-Associative Caches:
Ein Set-Associative Cache stellt eine Kombination aus den beiden zuvor genannten Varianten dar, der *Cache Sets* anstelle von Cache Lines adressiert. Die Adressierung von Speicherstellen innerhalb des Caches ähnelt der des Directly-Mapped Caches, indem die Hauptspeicheradresse verwendet wird, um den Speicherblock einer bestimmten Position im Cache zuzuordnen[3]. Jedes Cache Set beinhaltet eine feste Anzahl an Ways, also assoziative Speicher ohne Reihenfolge, sodass ein Cache Set mehrere kongruente Adressen beinhalten kann. Hat ein Cache Set eine feste Anzahl m an Ways, so nennt man den Cache-Speicher m-Way Set-Associative Cache. Die Anzahl der im Cache verfügbaren Sets wird in der Regel nicht mit angegeben. In Abbildung 3 ist eine Modellierung eines 3-Way Set-Associative Caches zu sehen. Wie bei einem Fully-Associative Cache kann eine Speicherstelle innerhalb des Cache Sets jedem Way zugeordnet werden. [Nul14]

Um eine bestimmte Speicherstelle im Cache zu finden, wird ein zweistufiger Prozess angewandt. Zuerst wird anhand des Cache-Index das gesuchte Cache Set bestimmt. Anschließend wird der Tag der Hauptspeicheradresse mit all den im Set befindlichen Tags der Ways verglichen, um so, im Falle eines Cache Hits, die gesuchten Daten zu erhalten. Je mehr Ways ein Speicher enthält, desto teurer ist er. Da Set-Associative Caches in der Regel nur wenige Ways in ihren Sets enthalten, ist diese Variante eines Caches ein Kompromiss aus Kosten und Performance. [Gru17a]

Moderne Intel-Prozessoren haben verschiedene Cache-Level, typischerweise drei, die sich in Größe und Schnelligkeit unterscheiden und hierarchisch aufgebaut sind. Der L1 Cache ist der kleinste und schnellste, während der L3 Cache, auch Last-Level Cache genannt, größer und langsamer ist. Die L1 und L2 Caches sind private Caches, sodass sie in einer Architektur mit mehreren CPU-Kernen nicht mit anderen Kernen geteilt werden. Der L3 Cache wird unter allen Kernen des Prozessors geteilt und ist inklusiv. Das heißt, dass alle Daten, die im L1 und L2 Cache vorhanden sind, auch im Last-Level Cache enthalten sind. Im Umkehrschluss bedeutet dies ebenso, dass Cache Inhalte die aus dem Last-Level Cache entfernt werden auch aus dem L1 und L2 Cache entfernt werden. Die verwendeten Caches sind in der Regel Set-Associative Caches[Int19]. [Mau17]

[3] Falls kein freier Speicher mehr im Cache vorhanden ist, muss neuer Platz geschaffen und bestimmt werden, welche Inhalte verdrängt werden. Da diese Verdrängungsstrategien für den Kontext dieser Arbeit jedoch keine große Relevanz haben, wird hier nicht näher darauf eingegangen.

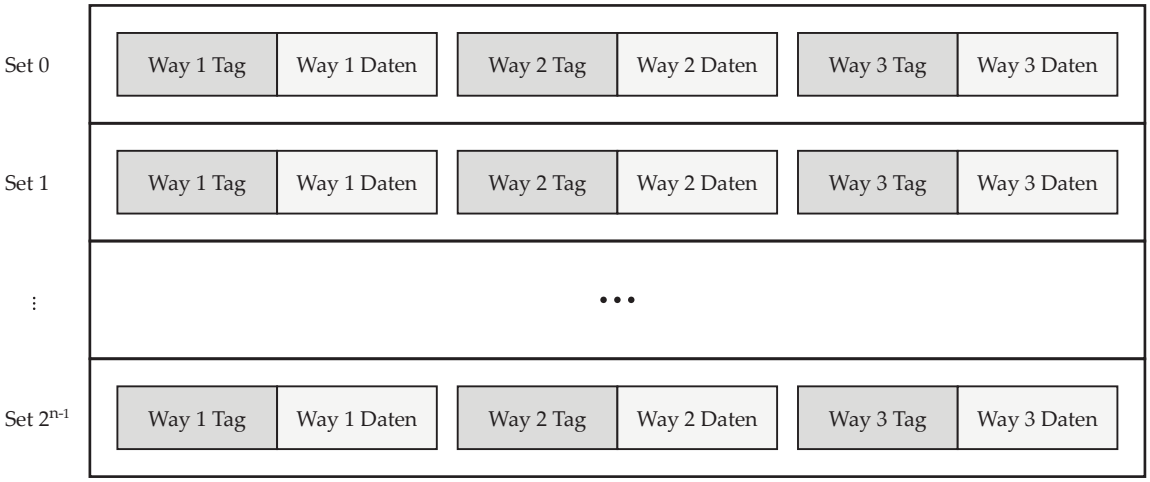


Abbildung 3: *Ein 3-Way Set-Associative Cache. Entsprechend der Anzahl n an Index Bits, gibt es 2^n Cache Sets.*

Um die Performance zu verbessern, ist der Last-Level Cache bei Intel-Prozessoren in Slices aufgeteilt, die jedem Kern seinen eigenen Bereich zuweisen. Diese Bereiche sind durch einen Ringbus miteinander verbunden, sodass allen Kernen der Zugriff auf jeden Bereich ermöglicht wird. Jeder CPU-Kern bekommt gleich viel Speicher mit nicht zwangsläufig zusammenhängenden Bereichen zugewiesen, auf denen der Zugriff des jeweiligen Kerns etwas schneller ist. Wie die Zuordnung der physischen Adressen zu einem bestimmten L3 Cache Speicherbereich geschieht bzw. welche Adressen vom jeweiligen Kern bevorzugt werden, ist vom Hersteller nicht dokumentiert. Jedoch wurde die sogenannte *Complex Addressing Function* für diese Adresszuordnung von Wissenschaftlern Reverse-Engineered [Mau15][Yar15]. [Gru17a]

3 Cachebasierte Seitenkanäle

Die Grundlagen, die zum Verständnis zur Eröffnung eines Seitenkanals über den Cache notwendig sind, wurden bereits in Kapitel 2 behandelt. In diesem Kapitel wird zunächst darauf eingegangen, was überhaupt ein Seitenkanal ist (Abschnitt 3.1). Anschließend wird der historische Verlauf von cachebasierten Seitenkanälen betrachtet (Abschnitt 3.2). Zuletzt werden einige bekannte Strategien zum Aufbau cachebasierter Seitenkanäle vorgestellt (Abschnitt 3.3).

3.1 Was ist ein Seitenkanal

Ein Seitenkanal ist ein Kommunikationskanal, dessen Existenz verborgen oder verdeckt ist. Er eröffnet die Möglichkeit, unbeobachtet geheime Informationen abzuleiten oder weiter zu geben, indem ein bestehender Kanal auf eine, z. B. vom Betriebssystem oder von Hardware Herstellern, unerwünschte Weise genutzt wird. Durch das Klickgeräusch eines mechanischen Zahlenschlosses den Code herzuleiten, zählt vermutlich zu einem der bekanntesten Seitenkanäle in der analogen Welt. So haben auch bei einem Computer bestimmte Vorgänge, beispielsweise eine kryptografische Berechnung, Einflüsse auf Systemressourcen, die beobachtet werden können. Aus diesen Beobachtungen lassen sich geheime Informationen aus Betriebsabläufen, wie Verschlüsselungsvorgängen, herleiten. Dies hat bereits Paul C. Kocher 1996 in seiner Arbeit *„Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems“* beschrieben [Koc96]. In der Vergangenheit wurden Seitenkanalangriffe auf Grundlage praktisch aller messbaren Umweltveränderungen, wie Energieverbrauch, elektromagnetischer Strahlung, Temperatur, akustischen Emissionen und vielen weiteren durchgeführt [Gru17a].

Ein cachebasierter Seitenkanal realisiert einen sogenannten Timing Angriff. Das heißt, es werden insbesondere durch die Beobachtung von Zeitunterschieden Rückschlüsse über interne Veränderungen des Caches möglich, um darüber Informationen zu gewinnen. Das Ausnutzen dieses Seitenkanals ist, wie bei vielen Seitenkanal-Typen, erst aufgrund von Optimierungen von Hardwarestrukturen und Betriebsabläufen, hier die Einführung und Verwendung eines CPU-Caches, möglich geworden. [Gru17a]

3.2 Historischer Verlauf

Der erste cachebasierte Seitenkanal wurde 1992 von Hu [Hu92] theoretisch nachgewiesen. Percival [Per05] war der Erste, der auch praktisch einen verdeckten Kanal auf dem L1 Cache erzeugte. Die ersten Erforschungen von cachebasierten Seitenkanälen wurden zunächst auf kryptografischen Algorithmen durchgeführt, wie die theoretischen Angriffe von Kocher [Koc96] und Kelsey u. a. [Kel98] zeigen. Im weiteren Verlauf führten Page [Pag02], sowie Tsunoo u. a. [Tsu03] praktische Angriffe auf DES anhand eines cachebasierten Seitenkanals durch. 2004 zeigte Bernstein [Ber05],

dass dies auch für den nachfolgenden Verschlüsselungsalgorithmus AES möglich ist. Während zahlreichen Studien zu Seitenkanälen über den CPU-Cache, wurden in Fachkreisen Strategien entwickelt, die sich die verschiedenen Architekturen und damit verbundene Möglichkeiten und Restriktionen besser zunutze machen. So wurde beispielsweise der L1 Cache angegriffen [Gul11], auf Basis von Shared Memory der L3 Cache ausgebeutet [Yar14] oder Seitenkanäle in Cloud Umgebungen aufgebaut. Ristenpart u. a. [Ris09] demonstrierten die Ausbeutung durch cachebasierte Seitenkanäle in Cloud Umgebungen. 2017 wurden von Maurice u. a. [Mau17] Seitenkanäle in Cloud Umgebungen noch verfeinert. Sogar Angriffe auf modernen mobilen Endgeräten konnten auf Basis von cachebasierten Seitenkanälen bereits durchgeführt werden, wobei Bildschirmberührungen, Tastenanschläge, sowie die Länge der auf dem Touchscreen eingegebenen Wörter vom Angreifer erfasst werden konnten [Lip16]. Im Jahr 2018 wurden die Angriffe Meltdown [Lip18] und Spectre [Koc19] vorgestellt, die auf der Ausnutzung von Out-Of-Order Execution und spekulativer Ausführung beruhen. Um durch den Angriff erbeutete Daten so zu übertragen, dass sie lesbar werden, verwenden diese Angriffstechniken zur Übermittlung[4] der ausgelesenen geheimen Daten cachebasierte Seitenkanäle [Lip18].

Die Nutzung cachebasierter Seitenkanäle hat sich in den vergangenen Jahren also immer mehr auch als nützliche Basis zur darauf aufbauenden Entwicklung komplexer und fortschrittlicher Angriffstechniken erwiesen. Somit ist ein aktives neues Forschungsfeld im Bereich der Seitenkanäle entstanden [Can19b]. Bisher sind jedoch keine Publikationen von Malware bekannt, die Techniken cachebasierter Seitenkanäle in allgemeiner Form ausnutzen. Im Folgenden wird auf Strategien zur Eröffnung cachebasierter Seitenkanäle, unter anderem auch die Strategie Flush + Reload, die im weiteren Verlauf dieser Arbeit Anwendung findet, eingegangen.

3.3 Klassische Strategien zu Cache Seitenkanälen

Cachebasierte Seitenkanäle bauen auf Beobachtungen von Zeitunterschieden auf. Das heißt, je nach dem, ob bestimmte Daten gecached sind oder nicht, sind Timing Differenzen messbar. Sind die Daten nicht im Cache vorhanden, müssen sie zunächst aus dem Hauptspeicher geladen werden. Aufgrund größerer Latenzzeiten beim Hauptspeicher-Zugriff (Vgl. Kapitel 2) benötigt das Laden der Daten bei einem Cache Miss deutlich mehr Zeit als bei Daten, die gecached sind. Dadurch lassen sich Rückschlüsse über das Verhalten anderer laufender Prozesse ziehen, bei denen ein potenzieller Zugriff auf diese Daten erwartet werden kann.

Es gibt im Allgemeinen die drei klassischen Strategien Prime + Probe, Evict + Time [Osv06] und Flush + Reload [Yar14] zur Eröffnung eines verdeckten Kanals durch CPU-nahe Hardwarecaches. Zu all diesen Strategien existieren zahlreiche Variationen, die im Folgenden jedoch nicht näher betrachtet werden. In der Regel werden die Schritte der jeweiligen Strategien mehrfach hintereinander ausgeführt, sodass längere Sequenzen des Programmverlaufs beobachtet werden können.

Ein Kommunikationskanal benötigt stets einen Sender und einen Empfänger. Je nach Anwendungsszenario kann hier der Sender als Opfer und der Empfänger als Angreifer gesehen werden. Im Folgenden wird in einer allgemeinen Form von Sender und Empfänger gesprochen. In den folgenden Strategien wird angenommen, dass der Sender die Daten, die zu den beobachteten Cache

[4] Genaueres zu Funktion und Verwendung in diesem Kontext in Kapitel 4

Inhalten gehören, für bestimmte Berechnungen potenziell verwendet, sodass Rückschlüsse über die tatsächlich verwendeten Daten und somit den Programmverlauf gezogen werden können. Diese Rückschlüsse werden anhand von Zeitmessungen gezogen, die eine Klassifizierung zu einem Cache Hit oder Cache Miss erlauben. Um eine solche Klassifizierung zu erleichtern, wird im Voraus ein Schwellwert, auch *Threshold* genannt, bestimmt, der die Grenze zwischen einem Cache Hit und einem Cache Miss darstellt.

3.3.1 Evict + Time

Diese Strategie wird verwendet, um Ausführungszeiten eines Prozesses in Abhängigkeit vom Vorhandensein bestimmter Cache Inhalte zu messen. Um diese Methode anwenden zu können, werden *Eviction Sets* benötigt. Als Eviction Set bezeichnet man eine Menge von kongruenten Adressen, die beim Ablegen im Cache die dort bereits vorhandenen Daten im zugehörigen Cache Set verdrängen [Gru17a].

Annahmen und Voraussetzungen:

- Das OS des Senders verwendet einen Hardware-Cache als CPU-Puffer
- Es sind geeignete Eviction Sets vorhanden
- Die Cache Sets der zu beobachtenden Speicherstellen sind bekannt

1. **Evict:** Der Empfänger verdrängt bestimmte vorhandene Inhalte eines Cache Sets, indem er auf die Adressen des Eviction Sets zugreift oder er entscheidet sich, die Inhalte des Cache Sets nicht zu verdrängen.
2. **Time:** Während der Sender seine Berechnungen durchführt, wird die Ausführungszeit gemessen.

Sind die Daten, die der Sender für seine Berechnungen benötigt, aufgrund der Verdrängung im Evict-Schritt nicht im Cache vorhanden, benötigt die Ausführung des Programms insgesamt mehr Zeit. Dies liegt daran, dass die Daten erst aus dem Hauptspeicher geladen werden müssen. Wurde sich im Evict-Schritt dazu entschieden, die Inhalte des relevanten Cache Sets nicht zu verdrängen, kann davon ausgegangen werden, dass aufgrund von vorherigen Berechnungen des Senders die benötigten Daten im Cache vorhanden sind. In diesem Fall ist die Ausführungszeit im Vergleich kürzer. Anhand dieses Zeitunterschieds kann der Empfänger feststellen, ob Daten aus einem bestimmten Cache Set für die Berechnungen des Senders verwendet werden. [Osv06; Gru17a]

3.3.2 Prime + Probe

Diese Strategie wird verwendet, um Zugriffszeiten auf bestimmte Cache Sets zu beobachten. Um diese Methode anwenden zu können, werden wie bei Evict + Time Eviction Sets benötigt.

Annahmen und Voraussetzungen:

- Das OS des Senders verwendet einen Hardware-Cache als CPU-Puffer
- Es sind geeignete Eviction Sets vorhanden
- Die Cache Sets der zu beobachtenden Speicherstellen sind bekannt

1. **Prime:** Der Empfänger bereitet den Cache vor, indem er auf die Adressen des Eviction Sets zugreift, um das entsprechende Cache Set mit eigenen Daten zu füllen und bereits vorhandene Daten zu verdrängen.
2. Dem Sender wird genug Zeit für einzelne Berechnungen gegeben, um ggf. die gerade aus dem Cache verdrängten Speicherstellen wieder in den Cache zu laden.
3. **Probe:** Der Empfänger überprüft, ob die abgelegten Daten noch vorhanden sind, indem er erneut auf die Adressen des Eviction Sets zugreift und die Zeit, die dafür benötigt wird, misst.

Hat der Sender in der Zwischenzeit nicht auf den relevanten Speicherbereich zugegriffen, befinden sich die im Prime-Schritt zuvor abgelegten Daten in einem idealen Szenario noch im Cache, sodass im Probe-Schritt des Empfängers wenig Zeit benötigt wird, um die eigenen Daten des Eviction Sets zu laden. Wenn der Sender in der Zwischenzeit jedoch auf den Speicherbereich zugegriffen hat, befinden sich Teile der zuvor im Cache Set abgelegten Daten nicht mehr im Cache, weil sie durch die Sender-Daten verdrängt werden. Da die Inhalte dieser Speicherstellen nun erst aus dem Hauptspeicher geladen werden müssen, dauert im Probe-Schritt der Zugriff auf diese Speicherstellen des Cache Sets merklich länger. Durch Messung der Zugriffszeit weiß der Empfänger also nun, ob der Sender auf Daten aus diesem Cache Set zugegriffen hat, oder nicht. [Osv06; Gru17a]

3.3.3 Flush + Reload

Diese Strategie wird verwendet, um Zugriffszeiten auf konkrete Speicherstellen zu beobachten. Da der Last-Level Cache von allen Prozessor-Kernen geteilt wird und diese Strategie auf diesen Cache zielt, müssen der Sender- und der Empfängerprozess nicht zwangsläufig auf demselben Kern ausgeführt werden. Für diese Strategie wird Memory Deduplication, insbesondere Context-Aware Sharing, benötigt. Dies ist notwendig, da die Beobachtung einer konkreten Speicherstelle mit Flush + Reload nur möglich ist, wenn Sender und Empfänger auf dieselbe physische Adresse zugreifen, also keine Kopie der benötigten Pages verwendet wird.

Annahmen und Voraussetzungen:

- Sender und Empfänger befinden sich auf demselben physischen Gerät
- Das OS verwendet einen Hardware-Cache als CPU-Puffer
- Sender und Empfänger benutzen den Last-Level Cache
- Memory Deduplication wird verwendet
- Die `clflush` Instruktion ist auf der Architektur nutzbar
- Die Adressen der zu beobachtenden Speicherstellen sind bekannt

1. **Flush:** Eine für den Empfänger interessante Speicherstelle wird mit der unprivilegierten `clflush` Instruktion aus allen CPU-Caches entfernt.
2. Dem Sender wird genug Zeit für einzelne Berechnungen gegeben, um ggf. die gerade aus dem Cache entfernte Speicherstelle wieder in den Cache zu laden.
3. **Reload:** Der Empfänger lädt die relevante Speicherstelle und misst die Zeit, die dafür benötigt wird.

Hat der Sender in der Zwischenzeit auf den relevanten Speicherbereich zugegriffen, befindet sich der Inhalt im Cache, sodass die Reload-Operation des Empfängers nur wenig Zeit benötigt. Wenn der Sender in der Zwischenzeit nicht auf den Speicherbereich zugegriffen hat, befinden sich die zugehörigen Daten in einem idealen Szenario auch nicht im Cache. Da die Inhalte der Speicherstelle nun erst aus dem Hauptspeicher geladen werden müssen, dauert der Zugriff auf die angefragte Speicherstelle merklich länger. Durch Messung der Zugriffszeit weiß der Empfänger also nun, ob der Sender auf die jeweilige Speicherstelle zugegriffen hat, oder nicht. [Yar14; Gru17a]

3.3.4 Kurzer Vergleich der Strategien

Die Strategien Evict + Time und Prime + Probe ähneln sich stark in Bezug auf die Voraussetzungen und einzelnen Komponenten der Schritte der Strategie. Beide Methoden nutzen die Verdrängung potenziell benötigter Inhalte durch Eviction Sets. Sie bieten beide eine Genauigkeit auf der Ebene von Cache Sets [Gru17a]. Das heißt, es kann zwar die Verwendung eines Cache Sets identifiziert werden, jedoch kann die genaue Speicheradresse nicht bestimmt werden. Sind mehrere zueinander kongruente Speicheradressen in einem Prozessablauf relevant, kann dies nicht erkannt werden und kann somit zu Ungenauigkeiten führen. Der große Unterschied dieser beiden Strategien besteht darin, dass Evict + Time die gesamte Ausführungszeit eines Prozesses während einer Berechnung misst, wohingegen Prime + Probe nur die Zugriffszeit auf bestimmte Cache Inhalte nach einer Berechnung des Senders misst. Folglich ist Evict + Time ungenauer als Prime + Probe, da die Ausführungszeit durch viele Faktoren zusätzlich beeinflusst werden kann und somit mehr Rauschen entsteht [Gru17a].

Der Vorteil der Strategien Evict + Time und Prime + Probe im Vergleich zu Flush + Reload besteht darin, dass kein geteilter Speicher nötig ist und diese Strategien auch auf Systemen, die Memory Deduplication nicht verwenden, angewandt werden können. Auch eine Verwendung, bei der Sender und Empfänger auf verschiedenen physischen Systemen laufen, ist bei Evict + Time und Prime + Probe möglich. Es ist beispielsweise machbar, mit diesen Strategien einen cachebasierten Seitenkanal über SSH in Cloud Umgebungen zu realisieren [Mau17]. Auch mit Flush + Reload ist die Eröffnung eines cachebasierten Seitenkanals über mehrere Virtuelle Maschinen hinweg möglich, solange diese auf derselben physischen Architektur laufen, sodass die Nutzung von geteiltem Speicher ausgebeutet werden kann [Inc16]. In Szenarien, in denen die Nutzung von geteiltem Speicher nicht möglich und die `clflush`-Instruktion nicht nutzbar ist, um den Flush-Schritt durchzuführen, kann Flush + Reload folglich nicht angewendet werden [Gru17a]. Ein Nachteil von Evict + Time und Prime + Probe ist, dass das Finden von geeigneten Eviction Sets aufwendig und schwierig sein kann. Spezifisches Wissen über die verwendeten Verdrängungsstrategien vereinfacht das Finden von Eviction Sets und der Last-Level Cache verwendet Complex Addressing für die Zuweisung von Kernen zu Slices des L3 Cache [Gru16b]. Beides wird von Hardwareherstellern jedoch nicht dokumentiert [Gru16b].

Flush + Reload wird in der Literatur oft als die mächtigste Strategie bezeichnet [Gru17a]. Sie arbeitet auf der Granularität einer Cache Line, sodass eine ganz konkrete Speicherstelle beobachtet werden kann. Zudem gelten cachebasierte Seitenkanal Strategien, die auf der Nutzung von geteiltem Speicher basieren, als die schnellsten und zuverlässigsten [Mau17]. Flush + Reload kann beispielsweise einen Durchsatz von 298 KB/s mit einer Fehlerrate von 0,0 % erreichen, während Prime + Probe einen Durchsatz von 67 KB/s bis 75 KB/s mit Fehlerraten von 0,36 % bis 1 % erreichen kann [Mau17; Gru16b].

4 Meltdown-Angriff

In diesem Kapitel wird die Funktionsweise von Meltdown-Angriffen, auch Rogue Data Cache Load Angriffe genannt [Int18], erklärt. Dazu wird in Abschnitt 4.1 zunächst auf die nötigen Angriffsbausteine eingegangen. In Abschnitt 4.2 wird der Angriff unter Zuhilfenahme dieser Bausteine erläutert. Dort wird auch genauer auf das Angriffsszenario eingegangen, in dem ein unprivilegierter Nutzer Zugang zu vertraulichen Kernel Inhalten erhält.

4.1 Angriffsbausteine

Damit der Meltdown-Angriff funktioniert, muss einerseits die Out-Of-Order Execution sinnvoll ausgenutzt werden, um einen geheimen Wert einer Kernel-Adresse durch Race Conditions bei der Berechtigungsprüfung trotz fehlender Zugriffsberechtigung zu erhalten. Andererseits muss das durch den Angriff erhaltene Geheimnis so übertragen werden, dass es durch Veränderungen der Mikroarchitektur ausgelesen werden kann. Die dafür nötigen Bausteine werden im Folgenden beschrieben. In Abschnitt 4.1.1 wird erläutert, wie durch transiente Befehle die Out-Of-Order Execution für den Angriff ausgenutzt werden kann und wie die unterbrechungsfreie wiederholte Ausführung von Meltdown gewährleistet werden kann. In Abschnitt 4.1.2 wird darauf eingegangen, wie das erhaltene Geheimnis übertragen werden kann, sodass es lesbar wird.

4.1.1 Transiente Befehle

Als *transiente Befehle* werden durch spekulative oder Out-Of-Order Execution ausgeführte Instruktionen bezeichnet, die im normalen Programmfluss nicht ausgeführt würden, da eine vorherige Anweisung eine Exception hervorgerufen hat. Aufgrund der Out-Of-Order Execution findet eine Ausführung dieses Befehls im Hintergrund dennoch statt, wodurch, wie bei regulär ausgeführten Anweisungen, messbare Seiteneffekte entstehen. Transiente Befehle treten grundsätzlich häufiger auf, da die CPU permanent vorausschauend agiert, um Latenzzeiten zu minimieren und die Performance somit zu maximieren. Basiert eine solche transiente Instruktion auf einem geheimen Wert, kann dieser ein Ziel bei der Ausnutzung eines Seitenkanals sein. [Lip18]

Der Zugriff auf eine User-Inaccessible Page, wie beispielsweise Kernel Pages, löst eine Exception aus, die üblicherweise die Anwendung beendet. Zielt ein Angreifer auf ein Geheimnis an einer solchen Adresse, gibt es drei sinnvolle Herangehensweisen, um mit der Exception umzugehen. Entweder wird die Exception vom Angreifer durch einen Exception Handler selbst behandelt, um eine andere Verhaltensweise, als das Programm zu beenden, zu definieren; oder es wird für die Ausführung der transienten Befehle ein Kindprozess gestartet, der problemlos durch die ausgelöste Exception beendet werden kann; oder Exceptions werden von vornherein unterdrückt, sodass sie erst gar nicht auftreten und stattdessen der Programmfluss nach der Ausführung des transienten Befehls umgeleitet wird. [Lip18]

Exception Handling:
Der Angreifer kann einen eigenen Exception Handler definieren, der ausgeführt wird, sobald eine bestimmte Exception auftritt. Beim Zugriff auf eine Page, auf die nicht zugegriffen werden darf, ist die Art der Exception ein Segmentation Fault. Die Verwendung eines eigenen Exception Handlers erlaubt es dem Angreifer, eigene Befehle zur Behandlung der Exception ausführen zu lassen und zu verhindern, dass die Anwendung beendet wird.

Kindprozess durch Exception beenden lassen:
Ein weiterer simpler Ansatz ist es, den angreifenden Prozess zu forken, bevor ein illegaler Speicherzugriff stattfindet und nur den Kindprozess auf die entsprechenden Speicherbereiche zugreifen zu lassen. Der Prozessor führt so die transienten Befehle des Kindprozesses aus, bevor dieser wegen des illegalen Zugriffs beendet wird. Der Elternprozess ist anschließend immer noch aktiv und kann weitere Kindprozesse zur Ausführung der angreifenden transienten Befehlsfolge starten. [Lip18]

Exceptions unterdrücken:
Ein anderer Ansatz ist es zu Verhindern, dass Exceptions überhaupt hervorgerufen werden. Der transaktionale Speicher ermöglicht es, die parallele Programmierung zu vereinfachen, indem Lade- und Speicherbefehle so gruppiert werden, dass sie als scheinbar atomare Anweisung ausgeführt werden können [Har10]. Die Erweiterungen der x86-Architektur um transaktionalen Speicher wird in modernen Intel-Prozessoren durch Intel TSX realisiert. Dadurch können also mehrere Anweisungen zu einer Transaktion zusammengefasst werden, die wie ein atomarer Vorgang aussieht. Das heißt, es werden entweder alle oder keine Anweisung ausgeführt. Wenn eine Anweisung innerhalb der Transaktion fehlschlägt, werden bereits ausgeführte Anweisungen rückgängig gemacht. Es wird jedoch keine Exception ausgelöst, sodass die Programmausführung ohne Unterbrechung fortgeführt wird [Jan16]. Des Weiteren werden aufgrund von spekulativer Ausführung Instruktionen aus abzweigenden Code Pfaden vorbereitend ausgeführt. Wird der illegale Speicherzugriff in einer Verzweigung getätigt, bei der sichergestellt wird, dass die Verzweigungsbedingung im Programmfluss niemals erfüllt ist, wird also auch keine Exception hervorgerufen. Der ungültige Speicherzugriff wird aber dennoch als spekulative Befehlssequenz ausgeführt. Diese Technik erfordert jedoch unter Umständen ein geschicktes Training des Branch-Predictors, sorgt aber auch dafür, dass die aus dem spekulativ ausgeführten illegalen Speicherzugriff resultierenden Daten mikroarchitekturelle Veränderungen der Hardware hinterlassen. [Lip18]

4.1.2 Zustandsüberführung durch Seitenkanäle

Wurden die transienten Befehle, die unerlaubterweise auf einen geheimen Wert zugreifen, der an einer für den Nutzerprozess eigentlich nicht zugreifbaren Adresse liegt, erfolgreich ausgeführt, ergeben sich dadurch mikroarchitekturelle Veränderungen. Diese müssen anschließend in einen architekturellen Zustand überführt werden, sodass dieser Wert ausgelesen und weiter verarbeitet werden kann. Diese Zustandsüberführung lässt sich durch einen Seitenkanal (siehe Kapitel 3) realisieren. [Lip18]

Die transiente Befehlssequenz kann dabei als Sender für den Seitenkanal betrachtet werden. Am empfangenden Ende des Seitenkanals können die Informationen aus dem mikroarchitekturellen Zustand abgeleitet werden. Der Empfänger ist also nicht Teil der transienten Befehlssequenz und kann ein Teil eines anderen Threads oder anderen Prozesses sein. [Lip18]

Der veränderte mikroarchitekturelle Zustand des CPU-nahen Hardware-Caches, der hier betrachtet wird, ist je nach transienter Befehlssequenz in den für Intel üblichen L1, L2 und L3 Caches zu beobachten. Durch Strategien zur Eröffnung von cachebasierten Seitenkanälen, wie z. B. Flush + Reload, kann der mikroarchitekturelle Zustand des L3 Caches in einen architekturellen Zustand überführt werden. Die Eröffnung eines Seitenkanals ist hier jedoch nicht auf diesen Cache begrenzt. Auch andere Seitenkanäle, die Rückschlüsse aufgrund des mikroarchitekturellen Zustands, beispielsweise der ALU (Arithemtic Logic Unit), ziehen lassen, sind denkbar. Der Vorteil einen cachebasierten Seitenkanal durch z. B. Flush + Reload zu verwenden, liegt darin, dass dieser Kanal weniger Rauschen verursacht und eine höhere Übertragungsrate erlaubt. [Lip18]

Wenn in der transienten Befehlssequenz der geheime Wert geschickt verwendet wird, um darauf basierend einen Speicherzugriff durchzuführen, beispielsweise an einer entsprechenden Position in einem Array, werden die Daten der zugehörigen Adresse im Cache für eine später potentiell erneute Verwendung zwischengespeichert. Der Empfänger kann dann beobachten, welche Adresse in den Cache geladen wurde, indem die Zugriffszeit auf diese Adresse gemessen wird. Der Sender kann somit ein „1“-Bit senden, indem auf eine Adresse zugegriffen und damit in den beobachteten Cache geladen wird, oder ein „0“-Bit senden, indem nicht auf eine solche Adresse zugegriffen wird. [Lip18]

Da in einer Cache Line 64 Byte an zugehörigen Daten gespeichert werden können, würde beispielsweise ein um 1 erhöhter geheimer Wert als simpler Positionsindex des Arrays beim Zugriff vermutlich zum Laden derselben Cache Line führen. Um dies zu verhindern, sollte der Zugriff auf das Array so gestaltet sein, dass ein veränderter geheimer Wert als Array-Index auch zum Ablegen der Daten in einer anderen Cache Line führt. Dies lässt sich realisieren, indem der geheime Wert mit der Page Size multipliziert wird, da in diesem Fall mit einem veränderten Wert stets auf eine andere Page zugegriffen wird und die Adressen somit weit genug auseinanderliegen, um an einer anderen Cacheposition geladen zu werden. Da im Programmfluss oft aufeinanderfolgende Speicherbereiche benötigt werden, lädt die CPU präventiv auch angrenzende Speicherbereiche in den Cache, was hier das Ergebnis verfälschen würde und dazu führt, dass ein Abstand zwischen den genutzten Array-Stellen in der Größe der Cache Line Size nicht ausreicht. Zudem wird durch eine Multiplikation in der Bestimmung des Array-Index Prefetching verhindert. Ein Byte wird durch 8 Bits dargestellt und kann somit 2^8=256 verschiedene Werte annehmen. Wird das Array so deklariert, dass damit 256 verschiedene Pages adressiert werden können, repräsentiert der Positions-Index einer in den Cache geladenen Page des Arrays den übertragenen Byte-Wert. Es lässt sich also durch die Übertragung eines einzelnen Bits an der entsprechenden Position ein ganzes Byte rekonstruieren. Eine solche Verwendung eines Arrays ist exemplarisch in Listing 4.1 zu sehen.

```
1 //Variable addr bekommt Adresse x zugewiesen
2 addr=x;
3 //Deklaration array, üblicherweise page_size = 4096 Byte
4 array[256*page_size];
5 //Zugriff auf Adresse addr und speichern der dortigen Daten in Variable data = 1 Byte
6 data=access_illegal_address(addr)      //ruft eine Exception hervor
7 //transienter Befehl legt durch Zugriff Daten in Cache ab
8 access(array[data*page_size]);        //data: Werte zwischen 0-255
```

Listing 4.1: *Zugriff auf eine Arrayspeicherstelle anhand von geheimem Wert aus illegalem Zugriff*

Durch anschließende Messung der Zugriffszeit auf die relevanten Array Speicherstellen i* Page_size, mit i=0-255, kann durch einen cachebasierten Seitenkanal das durch die transiente Befehlsfolge gesendete „1“-Bit und somit der geheime Wert identifiziert werden. Dies ist möglich, da die Zugriffszeit auf diese eine in den Cache geladene Array Speicherstelle im Vergleich deutlich geringer ist als bei den Speicherstellen, die nicht in den Cache geladen wurden.

4.2 Angriffsbeschreibung

In dem hier betrachteten Angriffsszenario hat der Angreifer Zugang zum System als unprivilegierter Nutzer und kann somit Programme auf dem Computer platzieren und im User-Mode ausführen. Das Ziel des Angreifers ist es, den Kernel anzugreifen und dadurch geheime Daten zu erhalten, die er als unprivilegierter Nutzer nicht auslesen können sollte. Die virtuellen Speicherstellen zu solchen geheimen Kernel Inhalten sind dem Angreifer bekannt. Der Angreifer hat die Absicht, Mechanismen, die die Vertraulichkeit von Kernel Inhalten gewährleisten sollen, zu umgehen. Des Weiteren wird hier ein Anwendungsszenario betrachtet, in dem eine verwundbare 64-Bit-x86-Intel-CPU verwendet wird und eine Linux-Distribution als Betriebssystem dient. Es sind insbesondere keine Meltdown-Mitigations aktiv. Als Methode zum Übertragen des Geheimnisses wird ein cachebasierter Seitenkanal mit der Strategie Flush + Reload verwendet. Damit dieser genutzt werden kann, wird zudem angenommen, dass die für Flush + Reload nötigen Annahmen und Voraussetzungen (Kapitel 3) erfüllt sind.

Meltdown kombiniert die in Abschnitt 4.1 vorgestellten Bausteine. Zuerst sorgt der Angreifer dafür, dass der Prozessor die transiente Befehlssequenz ausführt, um einen geheimen Wert, auf den ein normaler Nutzerprozess eigentlich nicht zugreifen darf, zu erhalten und weiter zu verwenden. Im Anschluss wird über einen cachebasierten Seitenkanal das von der transienten Befehlssequenz übermittelte Geheimnis rekonstruiert. Indem die drei folgenden Schritte für verschiedene Speicherstellen wiederholt werden, lässt sich somit der gesamte Kernel-Speicher, inklusive des gesamten physischen Hauptspeichers, auslesen. [Lip18]

Meltdown besteht aus 3 Schritten:

1. Der Inhalt einer vom Angreifer bestimmten Speicherstelle (s.u.), welche für den Angreifer unzugänglich ist, wird geladen.
2. Transiente Befehle sorgen für einen vom geheimen Inhalt abhängigen Zugriff und führen zum Laden der Daten in die entsprechende Cache Line.
3. Der Angreifer verwendetet einen cachebasierten Seitenkanal, um die entsprechende Cache Line zu identifizieren und das Geheimnis somit zu rekonstruieren.

Schritt 1: Geheimnis laden

Um Daten aus dem Hauptspeicher in ein Register zu laden, werden diese über eine virtuelle Adresse des zugreifenden Prozesses referenziert. Diese virtuelle Adresse wird anschließend in eine physische Adresse des Hauptspeichers übersetzt, um von dort die nötigen Daten zu holen. Da der gesamte Kernel-Adressraum in den virtuellen Adressraum jedes Prozesses abgebildet wird, resultieren alle Übersetzungsvorgänge dieser virtuellen Adressen auch in validen physischen Adressen, sodass die CPU auf den Inhalt an diesen Adressen zugreifen kann. Parallel zum Laden

der Daten prüft die CPU auch die Permission-Bits der virtuellen Adresse, also ob diese Adresse für den Benutzer oder nur für den Kernel zugänglich ist. Der einzige Unterschied zum Zugriff auf eine legale Adresse des Nutzerprozesses besteht darin, dass die CPU eine Exception auslöst, da die aktuelle Berechtigungsstufe nicht ausreicht, um auf Kernel-Adressen zuzugreifen. Folglich kann aus dem Nutzerprozess heraus nicht einfach der Inhalt einer solchen Adresse direkt gelesen werden. [Lip18]

Schritt 2: Geheimnis übersenden
Meltdown nutzt die Out-Of-Order Execution moderner CPUs aus, wodurch Befehle in dem kleinen Zeitfenster zwischen dem illegalen Speicherzugriff und der wegen fehlgeschlagener Berechtigungsprüfung ausgelösten Exception dennoch ausführt werden. Die so ausgeführte transiente Befehlsfolge wird durch Berechnungen, die auf dem geheimen Wert basieren, dazu verwendet, das Geheimnis an den Angreifer zu übertragen. Dazu wird der CPU-Cache verwendet, um dieses Geheimnis durch einen cachebasierten Seitenkanal zu übermitteln. Hierfür werden die relevanten Cache Lines zunächst geleert, um das Geheimnis anschließend von der transienten Befehlssequenz in einen mikroarchitekturellen Cache Zustand zu bringen, wie in Abschnitt 4.1.2 beschrieben. Um eine fortlaufende Ausführung des Angriffs zu gewährleisten, wird eine der in Abschnitt 4.1.1 beschriebenen Methoden verwendet, um unerwünschtes Beenden der Anwendung durch die Exception zu verhindern. Wird eine Exception wegen illegalem Speicherzugriff ausgelöst, werden, neben dem Beenden des Prozesses, auch die zugehörigen Daten invalidiert, indem das Register mit den geheimen Daten auf „0“ gesetzt wird. Da eine Null als Wert der geheimen Daten also mit großer Wahrscheinlichkeit auf einen Misserfolg beim Ausnutzen der Race Conditions hinweist, muss der Zugriff auf den Kernel-Speicher und die anschließende transiente Befehlsfolge wiederholt werden, um Meltdown dennoch zum Erfolg zu führen. Da diese Befehlsfolge mit dem Auslösen der Exception zeitlich konkurriert, kann die Reduzierung der Laufzeit dieser Befehlssequenz die Performance des Angriffs erheblich verbessern. Wenn beispielsweise dafür gesorgt wird, dass die Adressübersetzung für das Probe-Array im TLB zwischengespeichert ist, erhöht dies die Angriffsleistung auf einigen Systemen. [Lip18]

Schritt 3: Geheimnis auslesen
Durch das Flushen aller relevanten Cache Lines und der anschließenden Ausführung der transienten Befehlssequenz aus Schritt 2, wird genau eine Cache Line des Probe-Arrays im Cache zwischengespeichert sein. Die Position dieser Cache Line innerhalb des Arrays hängt von dem Geheimnis ab, das in Schritt 1 gelesen wurde. Der Angreifer iteriert also über alle 256 Pages des Arrays und misst die Zugriffszeit. Die Nummer der Page, bei der die Zugriffszeit deutlich kürzer ist, da sie sich im Cache befindet, entspricht dem geheimen Wert, wie in Abschnitt 4.1.2 beschrieben. [Lip18]

In Listing 4.2 ist die in der Arbeit *Meltdown* [Lip18] vorgestellte grundlegende Implementierung der transienten Befehle in x86 Assembler Instruktionen in Intel Syntax zu sehen. In Register rcx befindet sich die Adresse des Kernel-Speichers, auf den zugegriffen werden soll. Der Pointer auf die Basisadresse des Arrays befindet sich in Register rbx. In Zeile 3 wird ein Anweisungsblock definiert, der wiederholt werden soll, falls das Auslesen des geheimen Werts nicht funktioniert hat. In Zeile 6 findet eine Überprüfung dessen statt und es wird, falls der ausgelesene Wert „0“ ist und somit auf einen Misserfolg hinweist, zum Anfang des Blocks zurückgesprungen. In Zeile 4 werden die Daten an der Kernel-Adresse geladen und in das Least-Significant Byte des Registers rax, repräsentiert

durch al, gespeichert. Anschließend wird dieser Wert mit der Page Size multipliziert. War dieser Vorgang erfolgreich, wird in Zeile 7 auf die Daten am entsprechenden Array-Index zugegriffen und somit diese Arrayspeicherstelle in den Cache geladen.

```
1 # rcx: Kernel-Adresse
2 # rbx: Arraypointer
3 retry:
4 mov al, byte [rcx]   # Wert an Kernel-Adresse in unteren Teil des Registers rax legen
5 shl rax, 0xc         # Links-Shift um 12-Bit, Multiplikation mit 4096
6 jz retry             # Wiederholen, falls ausgelesener Wert 0 ist
7 mov rbx, qword [rbx + rax]    # Wert an Arrayposition in rbx legen
```

Listing 4.2: *Assembler Meltdown transiente Befehlssequenz [Lip18]*

Durch Wiederholung dieser drei Schritte kann der Angreifer den gesamten Speicher auslesen, indem er über alle möglichen Adressen iteriert. Die drei populären Betriebssysteme Windows, MAC OS und Linux bilden typischerweise den gesamten physischen Speicher in den Kernel-Adressraum ab. Somit ist Meltdown nicht nur auf das Auslesen des Kernel-Speichers beschränkt, sondern ist in der Lage, den gesamten physischen Speicher des Computers auszulesen. [Lip18]

4.3 Meltdown verwandte Angriffe

Meltdown-Angriffe gehören zur Klasse der Seitenkanalangriffe. Zeitgleich zur Veröffentlichung der Meltdown-Verwundbarkeit wurden weitere ähnliche Angriffsstrategien veröffentlicht. Diese nutzen die spekulative Ausführung durch CPUs und sich dadurch verändernde mikroarchitekturelle Zustände aus. Daher wird im Zusammenhang von Meltdown auch oft von Spectre Angriffen gesprochen. Spectre wird klassischerweise in Variante 1 und Variante 2 unterschieden. Meltdown wird oft als Variante 3 betitelt und als Unterklasse von Spectre Angriffen gesehen. [Goo18]

Viele weitere Variationen von Meltdown und Spectre sind bereits erforscht worden [Can19c]. Auch darüber hinaus gehende ähnliche Angriffsstrategien, die einer Angriffsklasse durch transiente Befehlsausführung oder spekulativer Ausführung zugehören, sind seitdem veröffentlicht worden. Es folgt eine Übersicht einiger solcher Angriffsstrategien:

- Spectre, Variante 1: Bounds Check Bypass (CVE-2017-5753) [Com18b; Koc19]
- Spectre, Variante 2: Branch Target Injection (CVE-2017-5715) [Com18a; Koc19]
- Meltdown, Variante 3: Rogue Data Cache Load (CVE-2017-5754) [Com18c; Lip18]
- Foreshadow (CVE-2018-3615) [Com18f; Van18]
- ZombieLoad (CVE-2019-11091) [Com18h; Sch19b]
- Fallout (CVE-2018-12126) [Com18d; Can19a]
- RIDL (CVE-2018-12127) [Com18e; Van19]
- SPOILER (CVE-2019-0162) [Com18g; Isl19]

5 Mitigations gegen die Meltdown-Verwundbarkeit

Da Hardware-Patches meist kompliziert und zeitintensiv in der Entwicklung sind, bestand nach Bekanntwerden der Meltdown-Verwundbarkeit ein Bedarf an Software Lösungen um Meltdown zu verhindern, bis neue Hardware entwickelt wurde und eingesetzt werden kann. Softwarebasierte Mitigations nehmen Änderungen an Betriebsabläufen des Betriebssystems vor und sind somit Veränderungen am Kernel-Code. Da Microcode Updates von Prozessoren Teil der Firmware und Prozessor spezifisch sind, werden diese zu den Hardware Mitigations gezählt. Die zur Meltdown-Prävention relevanten softwarebasierten und hardwarebasierten Mitigations werden mit Fokus auf die softwarebasierten Mitigations im Folgenden vorgestellt.

5.1 Softwarebasierte Mitigations

Durch Meltdown wird keine Software-Schwachstelle, sondern Schwachstellen in der Funktionalität von CPUs ausgenutzt. Folglich kann ein Software-Patch das Problem einer Hardware-Schwachstelle nicht in voller Gänze lösen, ist jedoch eine gute Abhilfe, bis sichere Hardware verfügbar ist. [Lip18]

In den drei folgenden Unterkapiteln werden die zugrunde liegenden Konzepte zu Software Mitigations erläutert, die bei der Prävention von Meltdown-Angriffen Anwendung finden. Da diese Arbeit sich auf Linux Betriebssysteme konzentriert, werden nur die für Linux relevanten Meltdown Software Mitigations vorgestellt. Diese sind KASLR (Abschnitt 5.1.1), KAISER (Abschnitt 5.1.2) und KPTI (Abschnitt 5.1.3). Einige andere Ansätze zur Meltdown-Prävention sind naheliegend, jedoch insgesamt nicht zielführend. Da diese dennoch ein besseres Verständnis zur Auswahl der angewendeten Vorgehensweisen erleichtern, werden zwei unpraktikable Ansätze im Folgenden kurz beschrieben.

Da Meltdown die Out-of-Order-Execution von Prozessoren ausnutzt, ist eine triviale Gegenmaßnahme diese vollständig zu deaktivieren. Allerdings wären die Auswirkungen auf die Leistung verheerend, da die parallele Ausführung von Mikro-Operationen in modernen Prozessoren so nicht mehr wie gewohnt verwendet werden kann. Eine parallele Ausführung spielt jedoch eine entscheidende Rolle für eine möglichst schnelle Abarbeitung von Befehlen. Daher ist dies keine praktikable Lösung. [Lip18]

Während das Betriebssystem die Berechtigungen für einen Speicherzugriff prüft, werden die angefragten Daten bereits aus dem Speicher geladen. Aufgrund der Out-Of-Order Execution wird mit diesem Wert bereits spekulativ weiter gearbeitet, bis das Ergebnis der Berechtigungsprüfung vorliegt. Besteht keine Zugriffsberechtigung für die Adresse, wird ein Segmentation Fault hervorge-

rufen. Meltdown nutzt Race Conditions zwischen diesen beiden Vorgängen aus. Es ist also möglich, Meltdown zu verhindern, indem dieser Vorgang serialisiert wird, sodass erst auf die Daten an der entsprechenden Stelle zugegriffen wird, wenn die Prüfung der Berechtigung abgeschlossen und erfolgreich ist. Dies bedeutet jedoch einen erheblichen Performanceoverhead bei jedem Speicherabruf, da der Zugriff auf den tatsächlichen Inhalt der Speicheradresse pausiert werden muss, bis die Berechtigungsprüfung abgeschlossen ist. Diese Herangehensweise führt folglich auch zu keiner akzeptablen und performanten Lösung. [Lip18]

Eine zielführendere Lösung stellt die Einführung einer strikten Trennung von Benutzer- und Kernel-Bereich dar, sodass der Kernel-Adressraum nicht in den Adressraum jedes Prozesses abgebildet wird. Eine strikte Trennung von User-Space und Kernel-Space soll verhindern, dass ein Zugriff auf den Kernel-Adressraum aus dem Nutzerprozess heraus überhaupt möglich ist. Dadurch können nicht nur Meltdown-Angriffe auf den Kernel verhindert werden, sondern auch andere Angriffe auf den Kernel, wie zum Beispiel Angriffe auf KASLR. Die Idee einer stärkeren Trennung ist ein wichtiger Ansatz, der auch in den umgesetzten softwarebasierten Mitigations Anwendung findet, auf die im Folgenden nun eingegangen wird. [Lip18]

5.1.1 KASLR – Kernel Address Space Layout Randomization

Seit der Einführung von Non-Executable Bits (NX) müssen Angriffe auf den Speicher auf bestehenden Code im Opferprozess, anstelle von Code-Injections, zurückgreifen. Diese Code-Reuse Angriffe, heutzutage eher bekannt unter dem Namen Return-Oriented-Programming, erfordern spezifisches Wissen über die Adressen der benötigten Code-Elemente. Auch Angriffe auf bestimmte Datenstrukturen erfordern Kenntnis der Größe und Position dieser Daten, um diese auszulesen oder sogar zu verändern. Im Laufe der Jahre wurden dagegen viele verschiedene Abwehrtechniken, wie z. B. NX Stacks, Canaries und ASLR entwickelt [Pro20]. [Tan15]

Um zu verhindern, dass Angreifer bei Vorhandensein einer Schwachstelle geeignete Ziele finden, um beispielsweise den Kontrollfluss eines Programms umzuleiten, unterstützen alle modernen Betriebssysteme heutzutage die Address Space Layout Randomization (ASLR). ASLR randomisiert bei jedem Programmstart die Positionen von Prozess-Segmenten, wie Code, Heap, Stack und Shared Libraries im Speicher. Bei der Linux Distribution Debian besteht auf einem 64-Bit System für die Platzierung der Startadressen einzelner Segmente eine Entropie[5] von 30 Bits für den Stack und 29 Bits je für das Heap- und Code-Segment [Det15]. Somit bildet ASLR eine effektive, erste Verteidigungslinie gegen Speicherangriffe, da es bedeutend schwerer ist, Fehler in Anwendungen auszunutzen, wenn die Positionen von für Code-Reuse Angriffe verwendbaren Code-Elementen unbekannt sind. [Tan15; Can20]

Die Kernel-Variante von ASLR ist bekannt als KASLR (Kernel Address Space Layout Randomization). KASLR randomisiert den Speicherort der einzelnen Kernel-Segmente, Daten und Treiber, die sich im Kernel-Bereich befinden. Wenn KASLR aktiviert ist, wird es einem Angreifer folglich erschwert relevante und gültige Adressen für einen Angriff auf den Kernel herauszufinden. Die Sicherheit des Kernels im Bezug auf die Vertraulichkeit hängt grundlegend davon ab, den Zugriff auf Kernel-Adressinformationen zu verhindern, was KASLR somit grundsätzlich erfüllt, sofern es nicht gebrochen wird. KASLR ist jedoch keine klassische Meltdown-Mitigation. Einerseits ist es durch Angriffstechniken wie EchoLoad [Can20], welche Meltdown-Mechanismen verwendet, möglich KASLR überhaupt erst zu brechen. Andererseits wird KASLR in gängigen Betriebssyste-

men bereits lange vor dem Bekanntwerden der Meltdown-Verwundbarkeit im Jahr 2018 eingesetzt. Dies ist im Vergleich zu Abbildung 4 erkennbar, welche die Einbindung von ASLR und KASLR in gängigen Betriebssystemen zeigt. Dennoch wird ein unautorisierter Zugriff auf Kernel-Adressen und somit die erfolgreiche Durchführung eines Meltdown-Angriffs durch KASLR erschwert. Für einen erfolgreichen Meltdown-Angriff benötigt der Angreifer entsprechende Kernel-Adressen, um auf diesen den unerlaubten Speicherzugriff durchzuführen und somit an vertrauliche Informationen zu gelangen. Es kann also kein Meltdown-Angriff auf den Kernel durchgeführt werden, wenn dem Angreifer die benötigten virtuellen Speicherstellen zu vertraulichen Daten nicht bekannt sind. Folglich muss zunächst ein weiterer Angriff zum Brechen von KASLR durchgeführt werden, bevor der eigentliche Meltdown-Angriff durchgeführt werden kann. KASLR stellt somit eine zusätzliche Hürde dar, sodass auch wenn KASLR nicht als Mitigation gegen Meltdown entwickelt worden ist, KASLR als Meltdown-Mitigation gesehen werden kann. [Sta13; LWN13; Can20]

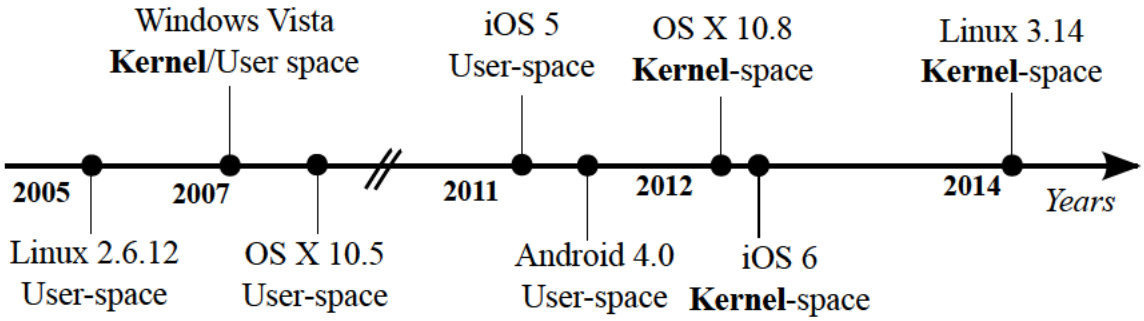


Abbildung 4: *Einbindung von ASLR und KASLR in gängigen Betriebssystemen, geordnet nach Jahr. Siehe [Jan16]*

Die KASLR Implementierungen der drei populärsten Betriebssystem-Typen (Linux, Windows, macOS) verwenden nur eine grobkörnige Randomisierung der Basisadresse. Feingranularere KASLR Implementierungen mit Code-Diversifikation [Tel15] werden in der Praxis nicht verwendet. Eine weitere Eigenschaft von KASLR Implementierungen ist, dass der Kernel entweder auf Small Pages (4 KB) oder Large Pages (2 MB) abgebildet wird. Das Mapping ist 2 MB aligned [Sch19a], wodurch die Anzahl der möglichen Offsets reduziert wird. Pro 1 GB an physischem Hauptspeicher gibt es 512 mögliche Plätze, an denen der Kernel mit einem Page Alignment von 2 MB positioniert werden kann [LWN13; Gru16a]. Somit bietet Linux' KASLR eine Entropie[5] von 9 Bits (da $2^9 = 512$). In *„Breaking Kernel Address Space Layout Randomization with Intel TSX"* wurden jedoch nur 6 Bit Entropie bei Linux-Systemen gemessen [Jan16]. Außerdem wird die Reihenfolge der randomisierten Segmente nicht verändert. Beispielsweise hat das Text-Segment stets eine niedrigere Adresse als die Module. Folglich bietet KASLR im Vergleich eine geringere Sicherheit durch die Adressrandomisierung als typische User-Space ASLR Implementierungen. Bei genauer Beobachtung durch den Angreifer wird demnach kein ausreichender Schutz gegen die Meltdown-Verwundbarkeit durch KASLR geboten, da mit etwas erhöhtem Angriffsaufwand dennoch die Adressen relevanter Kernel-Bereiche gefunden werden können. [Can20]

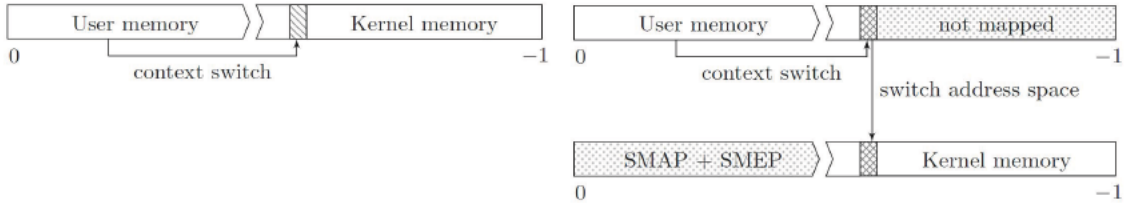
[5] Als Entropie wird hier der Grad an Zufälligkeit bezeichnet.

5.1.2 KAISER – Kernel Address Isolation to have Sidechannels Efficiently Removed

Der Kernel wird ohne Mitigations vollständig in den Adressraum jedes Nutzerprozesses abgebildet und ist nur durch die entsprechenden Permission-Bits vor dem Zugriff aus dem Nutzerprozess heraus geschützt. Somit kann ein Angreifer die Meltdown-Verwundbarkeit trotz KASLR ausnutzen, wenn KASLR gebrochen wurde und relevante Adressen herausgefunden werden konnten. Dass das Brechen von KASLR möglich ist, wird unter anderem in der Arbeit *„KASLR: Break It, Fix It, Repeat“* gezeigt [Can20]. Permission-Bits stellen eine kompakte Art der Realisierung von zwei getrennten Adressräumen dar – einen für den Benutzer-Modus und einen für den Kernel-Modus. Dennoch reicht ein Sicherheitsmechanismus von alleinig auf Permission-Bits basierenden Zugriffsberechtigungen nicht aus, da die Möglichkeit zur Ausbeutung der Out-Of-Order Execution besteht. KAISER wurde als Software-Mitigation entworfen, um den Kernel trotz Out-Of-Order Execution und üblichen Vorgängen der Berechtigungsprüfung vor unautorisierten Zugriffen zu schützen. Der Name KAISER ist ein Akronym für *„Kernel Address Isolation to have Sidechannels Efficiently Removed“*. Er ist jedoch auch eine Anspielung auf den Kaiserpinguin – den größten Pinguin der Erde. Linux verwendet als Markenzeichen einen Pinguin. KAISER wurde als Patch für Linux entwickelt, um es stärker und sicherer zu machen. Die Mitigation wurde im Juli 2017 als Methode, um Angriffe gegen KASLR zu verhindern, veröffentlicht. [Gru18; Gru17b]

Trennung der Adressräume

KAISER erzeugt eine stärkere Trennung zwischen dem Nutzer- und dem Kernel-Adressraum. Im Kernel-Modus sind alle Kernel-Adressen zugänglich und die Adressen des Nutzerprozesses werden durch SMAP (*„Supervisor-Mode Access Prevention“*) und SMEP (*„Supervisor-Mode Execution Prevention“*) und NX geschützt. Der Adressraum des laufenden Anwenderprogramms wird *Shadow Address Space* genannt. Dieser enthält neben den Pages des Nutzerprozesses durch KAISER nur noch einen sehr kleinen, notwendigen Teil des Kernels. Auf die dazugehörigen Teile wird im folgenden eingegangen. Interrupts werden für einen Kontextwechsel benötigt, weshalb die Interrupt Descriptor Tabellen und die Interrupt-Entry und -Exit .text Sections gemappt werden müssen. Aufgrund von Multi-Threading stehen einem Prozess mehrere Kerne zur Verfügung. Daher müssen die gesamten Informationen pro CPU, einschließlich Interrupt Request Stack und Vektor, dem Task State Segment und die Globale Deskriptor Tabelle in beiden Adressräumen abgebildet werden. Außerdem legt die CPU beim Wechsel in den privilegierten Modus implizit einige Register auf dem aktuellen Kernel-Stack ab. Dies können Informationen einer der pro-CPU-Stacks sein, die bereits gemappt wurden oder die eines Thread-Stacks. Folglich müssen auch die Thread-Stacks abgebildet werden. Diese kleinen Teile des Kernels sind im Adressraum des Nutzerprozesses erforderlich aufgrund der Art und Weise, wie Context Switches auf der x86-Architektur definiert sind, sodass ein reibungsfreier Ablauf zum Wechsel in den Kernel-Modus ermöglicht wird. Durch KAISER wird im User-Mode und im Kernel-Mode also jeweils ein anderer virtueller Adressraum verwendet. In Abbildung 5 ist dieser Mechanismus einer stärkeren Trennung der Adressräume konzeptionell zu sehen. [Gru18; Gru17b]



(a) Mit Permission-Bits (b) Mit KAISER

Abbildung 5: *Zwei Adressräume: durch Permission-Bits (a) und durch KAISER (b) realisiert. Siehe [Gru17b]*

Schutz des verbleibenden Kernel-Speichers im User-Space

Unmittelbar nach dem Wechsel in den Kernel-Modus wird vom Benutzer-Adressraum in den Kernel-Adressraum gewechselt. Es muss also nur sichergestellt werden, dass der Read-Only Zugriff auf den kleinen Teil des Kernels, der im Nutzerprozess noch abgebildet wird, kein Sicherheitsproblem darstellt. Da einige privilegierte Speicherbereiche in den User-Space gemappt werden müssen, verbleibt jedoch eine kleine Angriffsfläche für Meltdown, weil diese Speicherstellen immer noch aus dem Nutzer-Adressraum gelesen werden können. Obwohl an diesen Adressen keine geheimen Werte enthalten sind, wie z. B. Anmeldeinformationen, können sie dennoch Pointer enthalten. Einen Pointer zu leaken kann ausreichen, um wiederum KASLR zu brechen, da die Randomisierung der Offsets aus dem Wert des Pointers berechnet werden kann. Durch Einführung sogenannter Trampolin-Funktionen, die als Zwischenstufe zwischen Benutzer- und Kernel-Adressraum fungieren, kann dies jedoch auch verhindert werden. Statt also direkt den Kernel-Code aufzurufen, wird über einen Trampolin-Pointer auf die Trampolin-Funktion zugegriffen, um indirekt zum Kernel-Code zu springen. Wenn diese Trampolin-Funktion nur im Kernel und mit einer anderen Randomisierung gemappt wird, kann ein Angreifer nur den Zeiger auf den Trampolin-Code, nicht jedoch den randomisierten Offset des restlichen Kernels ausspähen. Ein solcher Trampolin-Code wird also für jede Kernel-Speicheradresse, die noch im Nutzerprozess abgebildet wird, benötigt. [Lip18]

Diese Trampolin-Funktionen sind Teil der sogenannten `cpu_entry_area`. Diese Datenstruktur bildet alle Daten und Funktionen ab, die benötigt werden, damit die CPU den Kernel-Modus betreten kann. Genauso wird eine `cpu_exit_area` zum Verlassen des Kernels eingeführt. Beide Datenstrukturen befinden sich an einer konstanten virtuellen Adresse, wodurch sie in der eingeschränkten Umgebung beim Betreten oder Verlassen des Kernels einfach zu verwenden sind. Die `cpu_entry_area` und `cpu_exit_area` sind lediglich ein Synonym für Speicher, der vom Kernel an anderer Stelle abgebildet wird. Dies erlaubt es, feste Berechtigungen für Strukturen wie das Task State Segment zu definieren, indem es schreibgeschützt in die `cpu_entry_area` abgebildet wird, während in der `cpu_exit_area` beim Verlassen des Kernels weiterhin Änderungen vorgenommen werden dürfen. Da Interrupts und Systemcalls kurze Zeit nach dem Eintritt in den Kernel prozess-spezifische Kernel-Stacks verwenden, werden diese, wie oben bereits erwähnt, ebenfalls im User-Space abgebildet. Das Abbilden des Kernel-Stacks kann jedoch Stack Inhalte preisgeben und für einen Performanceoverhead beim Erzeugen oder Beenden von Prozessen sorgen. Daher werden spezielle *Entry Stacks* in der `cpu_entry_area` eingeführt. Diese Stacks werden nur für eine kurze Zeit nach dem Betreten oder Verlassen des Kernels verwendet und enthalten weniger Daten als

der vollständige Prozess Stack, was die Wahrscheinlichkeit mindert, dass geheime Daten enthalten sind. [Gru18]

In *„KASLR is Dead: Long Live KASLR“* [Gru17b] wird die Methode der *Stronger Kernel Isolation* vorgeschlagen. Dabei wird im Nutzeradressraum an Kernel-Adressen ausschließlich der Code für Interrupt Handling zugänglich gemacht und die Adressen des Nutzerprozesses werden im Kernel-Modus vollständig entfernt. Dies bietet eine noch stärkere Isolation der beiden Adressräume. Jedoch ist dies ohne Weiteres nicht umsetzbar, da es ein Umschreiben von großen Teilen des Linux Kernels erfordert. [Gru17b]

Umgang mit Address Translation Tables

Da ohne Meltdown-Mitigations der gesamte Kernel vollständig in den Adressraum des Nutzerprozesses abgebildet wird, sind sowohl die User-Pages, als auch die Kernel-Pages über dieselbe Top-Level Page Table PML4 adressierbar. Bei Intel-Prozessoren enthält das CR3 Register einen Pointer auf die Top-Level Übersetzungstabelle PML4. Da jeder Adressraum seine eigenen Übersetzungstabellen besitzt, sind durch die Verwendung von zwei getrennten Adressräumen durch die KAISER Mitigation auch zwei Adressübersetzungstabellen notwendig. Somit wird durch diese Mitigation eine Aktualisierung des CR3 Registers auf die Adresse der entsprechenden PML4 bei einem Context Switch zwischen Nutzerprozess und Kernel-Modus erforderlich. Dies kann jedoch in Zusammenhang mit bestehenden Betriebsabläufen problematisch sein. Bei einem Kontextwechsel bleibt das CR3 Register zunächst auf dem alten Wert und zeigt auf die Page Table des bisherigen Adressraums. Zu diesem Zeitpunkt kann durch KAISER nur noch auf Teile des Kernels zugegriffen werden, die sowohl im Kernel- als auch im Benutzer-Adressraum abgebildet sind. Es kann nur eine sehr begrenzte Anzahl von Berechnungen durchgeführt werden. Interrupts können sowohl aus dem Benutzer- als auch aus dem Kernel-Bereich ausgelöst werden. Die Interrupt-Quelle innerhalb der begrenzten Menge von Berechnungen zu bestimmen ist daher an dieser Stelle nicht möglich. Folglich muss das Wechseln des CR3 Registers eine kurze, statische Berechnung sein. [Gru17b]

Auch dieses Problem wird von der KAISER Mitigation behandelt, indem ein fester Offset zwischen der Kernel PML4 und der Shadow PML4 festgelegt wird. Dies ermöglicht das Wechseln auf die Kernel-PML4 bzw. die Shadow-PML4 unabhängig von der Interrupt-Quelle. Die PML4 hat ein Alignment von 8 KB und Page Tables benötigen üblicherweise 4 KB an Speicher [Pat16, Kapitel 5]. Die Shadow-PML4 wird an ein Offset von 4 KB zur Kernel-PML4 gelegt. Vier Kilobyte entsprechen 4096 Byte = 2^{12} Byte, sodass das bei einem Offset von 4 KB das 12. Bit der Adresse gesetzt ist. Somit kann Bit 12 der physischen Adresse verwendet werden, um diesen Offset zu adressieren und einfach zwischen dem Kernel- und dem Shadow-Space umzuschalten, wie auch in Abbildung 6 zu erkennen ist. Wird das 12. Bit der Adresse im CR3 Register auf „0“ gesetzt, wird die Kernel PML4 adressiert. Ist dieses Bit auf „1“ gesetzt, wird die Shadow PML4 referenziert. Durch diesen „Trick“ müssen keine weiteren Speicheradressen nachgeschlagen werden und es wird nur ein Register benötigt, um den Adressraum zu wechseln. Der dadurch entstehende Speicheroverhead ist gering – er beläuft sich auf wenige Megabyte. Benötigt werden pro User Thread für Kernel Page Directories und Page Tables 8 KB an Speicher, für die Shadow PML4 12 KB und um 256 globale Kernel Page Directory Pointer Tabellen zu allokieren 1 MB an zusätzlichem Speicher. [Gru17b]

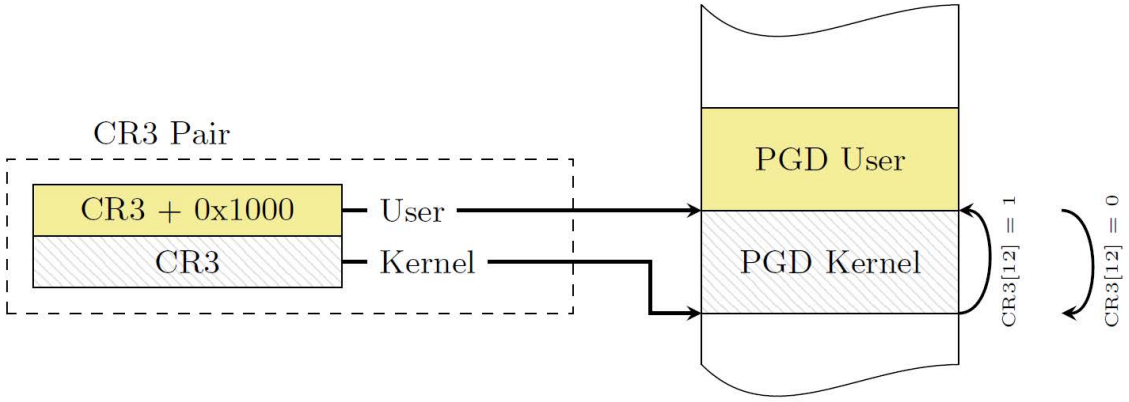


Abbildung 6: *Die Shadow PML4 und die Kernel PML4 liegen direkt hintereinander im Hauptspeicher und erlauben einen Wechsel zwischen den Adressräumen, indem eine Bitmaske auf das CR3 Register angewendet wird. Siehe [Gru17b]*

Auswirkungen von Aktualisierungen des CR3 Registers

Da Page Table Lookups viel Zeit in Anspruch nehmen können (Vgl. Kapitel 2), wird eine mehrstufige Cache-Hierarchie zum Zwischenspeichern von Adressübersetzungen verwendet. Der Translation Lookaside Buffer (TLB) (Vgl. Kapitel 2) ist ein solcher Cache und wird genutzt, um die Performance bei der Adressierung zu verbessern. Werden virtuelle Adressräume gewechselt, so muss der Wert im CR3-Register, der auf die aktuelle Page Table verweist, aktualisiert werden. Der Wechsel von virtuellen Adressräumen tritt beispielsweise wegen einem Wechsel zwischen Prozessen auf. Das definierte Verhalten von aktuellen Intel x86 Prozessoren schreibt vor den TLB bei jeder Aktualisierung des CR3 Registers zu leeren. Somit muss der TLB beim Wechsel zu einem anderen Prozess geleert werden. Dadurch gehen zwischengespeicherte Adressübersetzungen verloren. Beim Wechsel zu einem anderen Prozess ist dies völlig legitim, da Übersetzungen von virtuellen Adressen des vorherigen Prozesses dort nicht mehr gültig sind. Da die durch KAISER eingeführten separaten Adressräume für Nutzer- und Kernel-Modus unterschiedliche Page Tables verwenden, ist durch KAISER ein CR3-Update bei jedem Kontextwechsel, auch innerhalb eines Prozesses, nötig. Somit erhöht sich die Anzahl von TLB Flushes nun auch zwischen einem üblichen Wechsel zwischen Nutzer- und Kernel-Modus. Da im Anschluss an einen TLB Flush vermehrt TLB Misses auftreten, muss der Page Walk erneut durchlaufen werden, um virtuelle Adressen zu den benötigten physischen Adressen zu übersetzen. Im Vergleich zu einem TLB Hit kostet dies pro Adresse etwa 10 - 200 Zykel mehr [Pat16, Kapitel 5]. Daher führt das Flushen des TLBs zu Performance Verlusten, die durch eine Trennung der Adressräume verstärkt zum tragen kommen. [Gru17b; Gru18]

Fortbestehen von Adressübersetzungen über TLB Flushes hinweg

Die meisten Betriebssysteme optimieren die Performance von Kontextwechseln, indem ein globales Bit für TLB Einträge verwendet wird, das Adressen markiert, die von jedem Prozess verwendet werden können. So markiert das Global-Bit eines Page Table Entries Pages, die von impliziten TLB Flushes ausgeschlossen werden sollen und somit im TLB erhalten bleiben. Dies erhöht die Performance, da Adressen von Pages mit denselben Inhalten nicht erneut übersetzt werden müssen.

Insbesondere für Kernel-Adressen ist dies eine gängige Praxis. Folglich muss dieses Global-Bit bei der Umsetzung des KAISER Konzepts mit Vorsicht verwendet werden, da insbesondere das als global Markieren von Kernel Pages die Sicherheit, die der KAISER Mechanismus bietet, untergräbt. Das Deaktivieren des Global-Bits beseitigt zwar dieses Problem, kann aufgrund von vermehrten TLB Misses jedoch zu Performance Verlusten führen. [Gru17b]

Aktuelle x86 Prozessoren haben jedoch bereits Optimierungen implementiert, die für Kontextwechsel zwischen Nutzerprozess und Kernel verwendet werden können, um diese Performance Verluste zu minimieren. Eine solche Optimierung sind Process Context Identifiers (PCIDs), die auf Intels Haswell CPUs bzw. Prozessoren der 4. Generation und neueren Prozessoren verfügbar sind [Dav17]. Durch PCIDs ist es möglich, dass TLB Einträge auch über eine Aktualisierung des CR3 Registers hinweg bestehen können. Im TLB gespeicherte Page Table Entries werden mit einem Kontext-Identifikator für den zugehörigen Prozess getagged. Suchvorgänge im TLB sind nur dann erfolgreich, wenn die Kennung im TLB Eintrag mit der Kennung des aktuellen Prozesses übereinstimmt. Somit können bei Systemcalls und Interrupts durch PCID häufige TLB Flushes aufgrund von Kontextwechseln vermieden werden. Das heißt, die Auswirkungen von CR3-Updates auf den TLB können dadurch erheblich reduziert werden, während die Sicherheit gleichzeitig aufrecht erhalten werden kann. Wenn mehrere Adressräume gleichzeitig im TLB präsent sein dürfen, erfordert das jedoch zusätzliche Arbeit, um dies zu überwachen und entsprechende Einträge zu invalidieren. [Gru17b; Gru18]

KAISER bietet also einen grundlegenden Schutz gegen Meltdown-Angriffe auf den Kernel, da, selbst bei der Ausführung von transienten Befehlen, durch die Trennung der Adressräume ein Zugriff aus dem Nutzerprozess auf vertrauliche Kernel Inhalte nicht mehr möglich ist. Durch die KAISER Mechanismen kann Meltdown also effektiv verhindert werden, sodass keine solchen unautorisierten Kenntnisnahmen mehr erfolgen können.

5.1.3 KPTI – Kernel Page Table Isolation

KAISER ist letztendlich keine vollständige Implementierung eines Kernel-Patches, sondern nur ein Proof-Of-Concept von Forschern, das an die Linux Entwickler herangetragen wurde. KAISER liefert jedoch das zu Grunde liegende Konzept der Kernel Page Table Isolation, die von Linux kurz vor der Einbindung in den Kernel umbenannt wurde, da „KPTI“ besser als „KAISER“ in das typische Benennungsschema von Linux passt. [Gru18; Dav17]

Die Funktionalität von KPTI ist konzeptionell dieselbe wie bei KAISER (Vgl. Abschnitt 5.1.2). Da offizielle Kernel Versionen jedoch auf allen gängigen Hardware Architekturen reibungslos funktionieren müssen, wurden einzelne kleinere Änderungen vorgenommen. Weil ältere Prozessoren beispielsweise das CPU-Feature PCID noch nicht besitzen, wurde für diese non-PCID Systeme die Verwendung von Global-Bits wieder aufgenommen. Systeme mit neuerer Hardware bleiben davon unberührt und Verwenden den durch KAISER vorgeschlagenen Mechanismus des Markierens von TLB Einträgen durch PCID. [Dav18; Dav17]

KPTI wurde in den Linux Kernel 4.15 integriert [LWN17a] und auf die Linux Kernel Versionen 4.4.110 [Lin18c], 4.9.75 [Lin18d] und 4.14.11 [Lin18b] zurückportiert und fortan in allen neueren Kernel Versionen verwendet. Auch Windows und Apple haben ähnliche Konzepte zur

Vorbeugung gegen die Meltdown-Verwundbarkeit entwickelt. Windows nennt die Mitigation „KVA Shadow“ und Apple nennt sie „XNU Double Map“ [Gru18]. Konzeptionell unterscheiden sich die Implementierungen nicht stark von Linux's KPTI, jedoch beinhalten sie natürlich Anpassungen für das entsprechende Betriebssystem. Wie eingangs bereits erwähnt, werden die Umsetzungen für Nicht-Linux-Betriebssysteme hier jedoch nicht näher im Detail erläutert. Mehr Informationen zu Windows' KVA Shadow sind beispielsweise hier [Joh18] zu finden und weitere Informationen zu Apples XNU Double Map können z. B. hier [Ion18; Lev12] nachgelesen werden. [Gru18]

Meltdown-Angriffe auf den Kernel werden durch KPTI verhindert, da keine Daten aus dem Kernel geleaked werden können, weil es keine gültige Abbildung zum Kernel oder physischen Hauptspeicher mehr im Nutzerprozess gibt. Als Konsequenz daraus gibt es keinen Kernel-Speicher, der, durch transiente Befehle bei der Out-of-Order Execution, zwischengespeichert und verwendet werden kann. Somit können auch keine mikroarchitekturellen Veränderungen stattfinden, sodass durch einen Seitenkanal vertrauliche Daten des Kernels erbeuten werden können.

5.2 Hardwarebasierte Mitigations

Diese Arbeit konzentriert sich stärker auf die softwarebasierten Mitigations. Hardwarebasierte Mitigations versprechen jedoch im Vergleich zu softwarebasierten Mitigations weniger Performance Einbußen, weshalb sie auf lange Sicht vermutlich die bessere Wahl sind. Leider halten Prozessorhersteller den genauen Aufbau und die Funktionsweise ihrer Produkte meist geheim, weshalb es äußerst schwierig ist, zuverlässige und detaillierte Daten über die Hardware Mitigations von Intel zu finden. Daher werden Funktionalitäten oft von Forschern Reverse-Engineered oder aufgestellte Hypothesen über die Funktionsweisen durch Tests verifiziert.

Im White Paper zu *„Intel Analysis of Speculative Execution Side Channels“* [Int18] von Januar 2018 schreibt Intel, dass zukünftige Intel-Prozessoren Hardware-Unterstützung für die Software Mitigations von Rogue Data Cache Load Angriffen haben werden, ohne näher auf die geplanten Hardware-Features und Anpassungen einzugehen [Int18]. In einem Blogpost von 2018 kündigt Intel CEO Brian Krzanich Microcode Updates, welche einen Schutz gegen Meltdown bieten sollen, für 100 Prozent der in den letzten fünf Jahren auf dem Markt erschienenen Intel Produkte an [Bri18]. Während zudem die Spectre Variante 1 weiterhin über Software-Mitigations adressiert wird, plant Intel Änderungen am Hardware-Design, um vor den Spectre Varianten 2 & 3 zu schützen [Bri18]. Laut Krzanich wurden Teile des Prozessors umgestaltet, um neue Schutzebenen durch Partitionierung einzuführen, die sowohl vor Variante 2 als auch vor Variante 3 (Meltdown) schützen sollen. Diese Partitionierung wird als zusätzliche „Schutzmauer“ zwischen Anwendungen und Benutzerprivilegstufen, um ein Hindernis für böswillige Akteure zu schaffen, beschrieben [Bri18]. Wie diese Partitionierung genau aussieht, bleibt unerwähnt.

Ist noch kein BIOS/UEFI Update vorhanden, das die neuesten CPU Microcode Updates enthält, ist es möglich über das Betriebssystem solche Microcode Updates nachzuladen, ohne ein BIOS/UEFI Update durchführen zu müssen [Kre19]. Intel hat seit 2018 solche Microcode Updates für Linux Betriebssysteme veröffentlicht, die Mitigations gegen die Spectre und Meltdown-Verwundbarkeit enthalten [Ble18]. Sie sind für 40 verschiedene Linux Versionen und 2.371 Intel-Prozessoren

Tabelle 1: *Intel's Core Architecture Meltdown und Spectre v2 Mitigations, Stand: März 2018 [Rya18]*

Microarchitecture	Core Generation	Status
Penryn	45nm Core 2	Microcode Planning
Nehalem/Westmere	1st	Planning/Pre-Beta
Sandy Bridge	2nd	Microcode Released
Ivy Bridge	3rd	Microcode Released
Haswell	4th	Microcode Released
Broadwell	5th	Microcode Released
Skylake	6th	Microcode Released
Kaby Lake	7th	Microcode Released
Coffee Lake	8th	Microcode Released
H2'2018 Core	8th	Hardware Immune
Cascade Lake	X	Hardware Immune

verfügbar (Stand: Januar 2018) [Ble18]. Eine Übersicht über die Prozessortyp bezogene Art der Mitigation ist in Tabelle 1 zu sehen [Rya18].

Die Entwicklung bei Intel verlief jedoch nicht ohne einige signifikante Rückschläge wie z.B. fehlerhafte Microcode Updates [Rya18]. Auch der Initiator der Entwicklung des Linux Kernels Linus Torvalds war mit der Qualität der Arbeit von Intel nicht zufrieden: „As it is, the patches are COMPLETE AND UTTER GARBAGE. They do literally insane things. They do things that do not make sense. [...] The patches do things that are not sane. WHAT THE F*CK IS GOING ON?" [Lin18a].

Mit der Whiskey Lake CPU hat Intel Schwachstellen in der Hardware behoben, ohne weitere Details über die Funktionsweise der Mitigation anzugeben. Diese CPUs zeigen an, dass sie nicht Meltdown verwundbar sind, indem sie das RDCL_NO-Bit im modellspezifischen Register IA32_ARCH_CAPABILITIES gesetzt haben [Int19]. In *„KASLR: Break It, Fix It, Repeat"* [Can20] wurde die Hypothese aufgestellt und verifiziert, dass die Mitigation der Whiskey Lake Mikroarchitektur das Ergebnis eines illegalen Speicherzugriffs nur auf null setzt, anstatt diese Speicherstellen erst gar nicht zu laden. Folglich können mit diesem CPU-Modell, auch mit Hardware-Mitigation, immer noch unerlaubt Inhalte von Speicherstellen ohne die nötige Berechtigungsstufe geladen werden. Da also durch unautorisierte Zugriffe auf Kernel-Adressen der tatsächliche Wert abgerufen wird und erst später auf null gesetzt wird, ist anzunehmen, dass die Race Conditions, die Meltdown sich zunutze macht, immer noch zum tragen kommen und Meltdown durch diese Hardware Mitigation effektiv nicht verhindert wird. Diese Hypothese wird dadurch gestützt, dass KASLR auch mit dieser Mitigation immer noch problemlos gebrochen werden kann. Die Verwendung von Software Mitigations wie KPTI ist somit immer noch unerlässlich, um ein gutes Maß an Sicherheit zu gewährleisten. [Can20]

Auf ARM Prozessoren haben klassische Meltdown-Angriffe nicht zuverlässig funktioniert. Es wird vermutet, dass ein Grund dafür der unterschiedliche Aufbau für die Referenzierung von Adressübersetzungstabellen sein kann. Während bei Intel-Prozessoren das CR3 Register verwendet wird, um die aktuelle Übersetzungstabelle zu referenzieren, verwenden ARM CPUs zwei Register (TTB0 und TTB1) [ARM20], um die physischen Adressen der Translation Tables für den User-Space und Kernel-Space getrennt voneinander zu speichern. Auch ARM selbst behauptet, dass die eigenen Prozessoren, ausgenommen Cortex-A75, nicht von der klassischen Meltdown-Variante betroffen

sind [ARM21]. Aus diesem Grund sind ARM Prozessoren auch von der Linux KPTI Mitigation ausgenommen worden [Tor18]. Ein Umdenken im grundlegenden Aufbau von CPUs kann Angriffe dieser Art zukünftig also möglicherweise verhindern. [Gru17b]

6 Performance von KPTI

Die durch KAISER vorgestellten Mechanismen, welche in den Linux Kernel unter der Bezeichnung KPTI eingebunden werden, verändern Betriebsabläufe beim Zugriff und beim Mapping von Kernel Inhalten (Vgl. Kapitel 5). Es ist anzunehmen, dass die Anpassungen durch diese Software-Mitigation Performanceverluste verursachen. In diesem Kapitel werden die Auswirkungen von KPTI auf die Performance untersucht. Anhand dessen wird die zentrale Forschungsfrage dieser Bachelorarbeit, wie sich die Meltdown-Mitigation KPTI auf die Gesamtperformance des Systems auswirkt, beantwortet. Dazu wird in Abschnitt 6.1 zunächst auf vorhandene Forschungsergebnisse zur Performance eingegangen, die von den Entwicklern des KAISER Konzepts bei der Veröffentlichung präsentiert worden sind. Da KPTI auf KAISER beruht, sind Evaluationsergebnisse der Performance von KAISER auch auf KPTI übertragbar. In Abschnitt 6.2 wird beschrieben, welche Test für eine Analyse der Performance im Rahmen dieser Arbeit durchgeführt werden. Abschließend werden in Abschnitt 6.3 die dadurch erhaltenen Ergebnisse zur Performance von KPTI vorgestellt und evaluiert.

6.1 Initiale Forschungsergebnisse zur Performance von KAISER und KPTI

Von den Wissenschaftlern, die das KAISER Proof-Of-Concept entwickelt haben, wird ein Gesamtperformanceoverhead durch KAISER von 0,08 % bis 0,68 %, mit einem durchschnittlichen Overhead von 0,28 % angegeben. Aus verwendeten Grafiken zur Darstellung der Evaluationsergebnisse in *„Kaslr is dead: long live kaslr“* [Gru17b] können Werte zu weiteren Tests wie folgt abgelesen werden. Die minimale Zugriffszeit nach dem Prefetching einer physisch abgebildeten Page beträgt ohne KAISER meist schwankende Werte um ca. 250 Zykel. Durch eine deutlich niedrigere Zugriffszeit von ca. 120 Zyklen am entsprechenden Page Offset des Kernel-Mappings ist erkennbar, auf welche physische Adresse eine virtuelle Adresse abgebildet wird. Wenn KAISER aktiv ist, wird letzteres nicht mehr offenbart und es besteht eine meist konstante Zugriffszeit von ca. 280 Zyklen. Der Performance Verlust durch KAISER beträgt bei der Zugriffszeit auf vorgeladene Adressen also etwa 30 Zykel, wie in Abbildung 7 erkennbar ist. Wie sich die Laufzeit durch KAISER erhöht, wird anhand von drei verschiedenen Benchmarking-Tools evaluiert. Bei keiner der verwendeten Werkzeuge wird ein Performanceoverhead bei der Laufzeit von über ca. 0,7 % gemessen. Dies ist in Abbildung 8 zu sehen. [Gru17b]

Leider werden die von den KAISER-Entwicklern durchgeführten Tests zur Performanceanalyse in der Veröffentlichung des KAISER Konzepts nicht näher im Detail beschrieben. Dadurch bleiben einige Fragen zu der konkreten Umsetzung der Test offen, wie z. B. was genau bei den durchgeführten Laufzeittests evaluiert wird. Daher ist unklar, ob weitere relevante Auswirkungen, die durch KAISER entstehen können, in den vorliegenden Ergebnissen betrachtet werden.

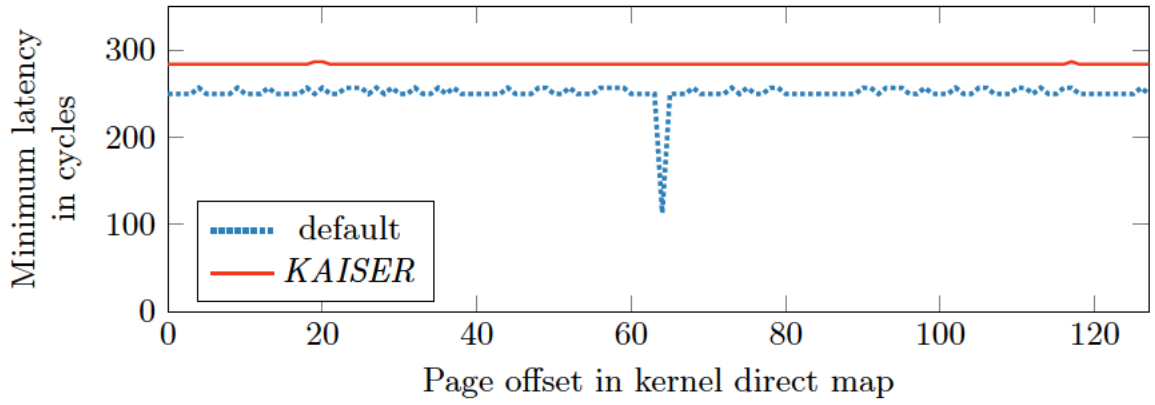


Abbildung 7: *Minimale Zugriffszeit nach einem Prefetch der jeweiligen physischen Adressen. Unter Verwendung des Standard Kernels zeigt die tiefe Spitze der gestrichelten Linie die Zuordnung, auf welche physische Adresse eine virtuelle Adresse abgebildet wird. Die durchgezogene Linie zeigt den gleichen Test unter Verwendung von KAISER. Siehe [Gru17b]*

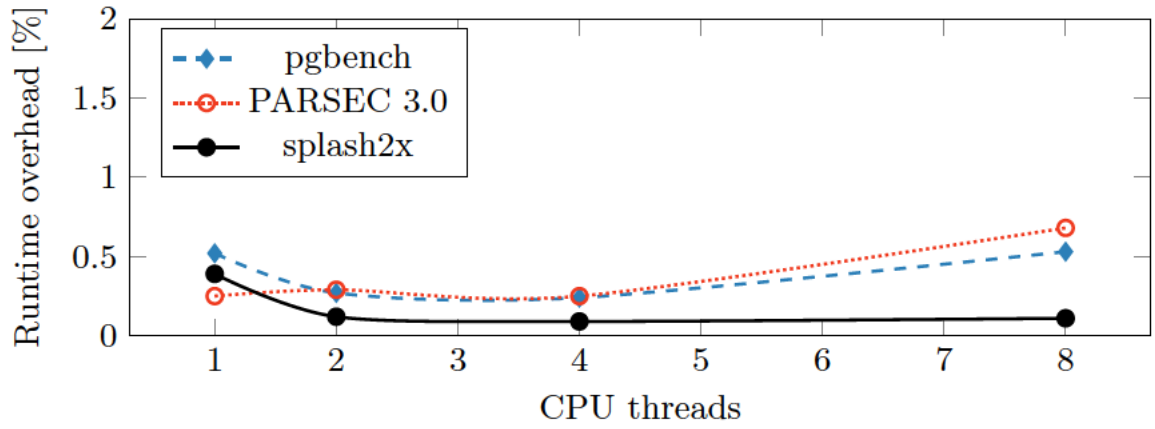


Abbildung 8: *Vergleich der Laufzeiten unterschiedlicher Benchmarking-Programme auf einem durch KAISER geschützten Kernel. Der Standard Kernel dient als Vergleichsbasis (= 100%). Siehe [Gru17b]*

Möglicherweise könnte die Performance unter der Nutzung von KPTI folglich auch zu höheren Leistungsverlusten führen. Auch in anderen Quellen, die auf die Gesamtperformance von KPTI eingehen, wird teilweise von deutlich höheren Performanceeinbußen gesprochen. So wird beispielsweise im Onlinemagazin LWN, welches insbesondere Neuigkeiten zu Linux veröffentlicht, ein Gesamtlaufzeitoverhead von etwa 5 % genannt [LWN17b]. Red Hat nennt ebenfalls höhere Verluste der Gesamtperformance. Im Januar 2018 werden Performanceverluste von 1 % - 20% gemessen und nach weiteren Kernel Updates für Red Hat Enterprise Linux 6 und 7 werden im März 2019 1 % - 8% an Performanceverlusten angegeben [Red].

Um die zu erwartenden Performanceverluste in einem eigenen konkreten Anwendungsszenario abschätzen zu können, reichen Angaben über einen Gesamtperformanceverlust nicht aus. Dies ist insbesondere der Fall, wenn unklar ist, bei welchen Vorgängen primär mit Performanceverlusten zu rechnen ist. Brendan Gregg deutet in seinem Blog eine etwas detailliertere Analyse der Performance von KPTI an, indem Faktoren wie z .B. die Systemcall Rate, Kontextwechsel oder die Page Fault

Rate als beeinflussende Ereignisse bei der Performance von KPTI genannt werden [Gre]. Auch hier werden deutlich höhere Performanceverluste, als in der Veröffentlichung des KAISER Konzepts, geschildert. Bei einer Rate von 50.000 Systemcalls pro Sekunde pro CPU wird beispielsweise ein Overhead durch KPTI von 2 % erwähnt [Gre]. Auch auf das Flushen des TLBs wird in diesem Blogpost Bezug genommen. Bei einer Größe des Arbeitssets von mehr als 10 MB entstehen bis zu 7 % Performanceoverhead durch KPTI aufgrund von TLB Flushes [Gre]. Die Performanceverluste werden in diesem Blogeintrag mit 1 % bis über 800 % zusammengefasst [Gre]. Auch diese Angaben von KPTI-Performanceverlusten lassen jedoch keine ausreichend detaillierte Prognose zum Einfluss verschiedener Faktoren und Vorgänge unter KPTI zu. Es werden im Zuge dieser Bachelorarbeit im Folgenden eingehendere Evaluationen der Performance von KAISER bzw. KPTI durchgeführt.

6.2 Evaluations Parameter und methodisches Vorgehen

Für eine Performanceanalyse der Meltdown-Mitigation KPTI (Vgl. Kapitel 5) werden auf mehreren Geräten Vorgänge während KPTI aktiviert ist verglichen mit denselben Vorgängen während KPTI deaktiviert ist. Alle Testsysteme verwenden einen Intel-Prozessor und laufen unter derselben Linux Distribution (Ubuntu 20.04). Im Folgenden wird auf die Tests zur Performanceanalyse eingegangen, die im Rahmen dieser Bachelorarbeit durchgeführt werden. In Abschnitt 6.2.1 wird dazu auf die Ausführungszeit von Systemcalls und in Abschnitt 6.2.2 auf Auswirkungen von KPTI auf den TLB eingegangen.

Die Analyse wird in einer realitätsnahen Umgebung durchgeführt. Das heißt, es finden keine Veränderungen der Laufzeitumgebung, die vom Zustand einer 64-Bit Ubuntu 20.04 Standard Installation abweichen, statt. Es wird lediglich der CPU Governor auf den Performance Modus gesetzt, um eine optimale Nutzung der CPU Kapazitäten zu gewährleisten. Zudem nehmen die KPTI Mechanismen insbesondere Änderungen an Betriebsabläufen zum Schutz vor unprivilegierten Nutzerprogrammen vor, sodass auch das im Rahmen dieser Bachelorarbeit entwickelte Analyseprogramm ohne höhere Berechtigungsstufen auskommt. Das entwickelte Analyseprogramm ist in der Programmiersprache C geschrieben. Damit das Scheduling des Analyseprozesses die Messungen nicht zu stark beeinflusst, wird dieser Prozess fest an einen CPU Kern gebunden.

6.2.1 Ausführungszeit von Systemcalls

Um einen Systemcall aus einem Nutzerprozess heraus auszuführen, wechselt der Prozessor ohne KPTI in den Kernel-Modus und es wird aus dem aktiven Adressraum direkt zum entsprechenden Kernel-Code gesprungen. Wenn KPTI aktiv ist, sind im Nutzerprozess ausschließlich die `cpu_entry_area` und `cpu_exit_area` mit Entry Stacks und Zeigern auf Trampolin-Funktionen zu den verbleibenden Kernel-Elementen enthalten (Vgl. Kapitel 5). Um zu dem benötigten Kernel-Code für einen Systemcall zu gelangen, muss also zuerst auf die in diesen Bereichen liegenden Daten und Trampolin-Funktionen zugegriffen werden. Um vollständig in den Kernel-Modus zu wechseln, muss der aktive Adressraum vom Shadow Address Space zum Kernel Address Space gewechselt werden, indem das CR3 Register aktualisiert wird.

Da unter Verwendung von KPTI also einige zusätzliche Operationen durchgeführt werden müssen, ist anzunehmen, dass Leistungseinbußen bei der Nutzung von Systemcalls entstehen. Diese Performance Verluste werden analysiert, indem für einige grundlegende Systemcalls die

Ausführungszeit mit und ohne KPTI verglichen wird. Dazu werden typische Systemcalls aus der Kategorie File Management und Process Control untersucht.

File Management: Durch den zusätzlichen Verwaltungsaufwand, inklusive zusätzlichen Speicherzugriffen, beispielsweise in der `cpu_entry_area`, und dem Wechseln des Adressraums, ist es möglich, dass mit KPTI mehr Zeit für einen Systemcall benötigt wird. Das Arbeiten mit Dateien ist ein typisches Anwendungsszenario, in dem Systemcalls verwendet werden. Daher werden die benötigten Zeiten für die Systemcalls `open`, `read` und `close` gemessen.

Process Control: Durch die Verwendung von KPTI wird für die Verwaltung eines Prozesses mehr Speicher benötigt als ohne KPTI. Für jeden Prozess müssen unter anderem zwei Adressräume, inklusive benötigter Page Tables, verwaltet werden und die benötigten Strukturen zur Absicherung der im User-Space verbleibenden Kernel-Elemente erzeugt werden. Das Abfragen der eigenen Prozess ID, das Kopieren eines Prozesses, das allokieren von Speicher oder das Freigeben von Speicher kann durch KPTI möglicherweise länger dauern. Daher wird die benötigte Zeit der Systemcalls `getpid`, `fork`, `mmap` und `munmap` gemessen. Die Größe des zu erzeugenden oder freizugebenden Speichers kann möglicherweise auch Auswirkungen auf die Ausführungszeit haben. Daher wird das Allokieren und Freigeben von Speicher für Speichergrößen von einem Byte, einem Kilobyte, einem Megabyte und einem Gigabyte untersucht.

Umsetzung:

Die Ausführungszeit der gewählten Systemcalls wird in Zyklen und Nanosekunden gemessen. Für die Messung der benötigten Zeit in Zyklen wird die `rdtsc` Instruktion verwendet [Int98; Int10]. Diese liefert einen Zeitstempel mit einer genauen Zählung jedes CPU Zykles, der vom Prozessor ausgeführt wird. Der Time Stamp Counter befindet sich in einem Modellspezifischen 64-Bit Register. Da die `rdtsc` Instruktion keine Serialisierung beinhaltet, wird durch die Barrieren `mfence` und `lfence` gewährleistet, dass die Instruktionen auch in der Reihenfolge ausgeführt werden, wie sie im Code vorkommen. Ohne diese Barrieren kann aufgrund der Out-Of-Order Execution nicht garantiert werden, dass die `rdtsc` Instruktion auch tatsächlich zur gewünschten Zeit ausgeführt wird, sodass irreführende Zykel Zählungen entstehen können. Die Instruktionen `mfence` und `lfence` sind Serialisierungsinstruktionen, die sicherstellen, dass alle vorherigen Lade- und Speicheroperationen vollständig durchgeführt worden sind, bevor mit dem Code nach der Barriere fortgefahren wird [Int19, Vol.1, Abschnitt 5.6.4]. Die Implementierung der Verwendung der `rdtsc` Instruktion ist in Listing 6.1 zu sehen. Damit der Funktionsaufruf an sich weniger Overhead verursacht, wird die Funktion als `inline` markiert. Um die Ausführungszeit eines Systemcalls in Zyklen zu messen, wird die Differenz zwischen dem Zeitstempel direkt vor und nach der Ausführung berechnet. Dies wird in einer Schleife mehrfach wiederholt. Compiler Optimierungen der Schleife können hier eine genaue Messung potenziell negativ beeinflussen. Um dies zu verhindern und zu garantieren, dass alle vorherigen Instruktionen vollständig abgeschlossen sind, wird die zur Serialisierung verwendbare Instruktion `cpuid` genutzt [Int10]. Diese wird innerhalb der Schleife vorm Beginn einer neuen Messung aufgerufen. Die zur Messung verwendeten Instruktionen benötigen in ihrer Ausführung ebenso eine gewisse Menge an Zeit, die einen geringen Overhead bei jeder Messung verursachen. Da dieser Overhead bei einem prozentualen Vergleich der Messergebnisse ins Gewicht

fallen kann, wird dieser Overhead vor dem Beginn der eigentlichen Messungen bestimmt und im Anschluss von jeder Messung abgezogen. Das Vorgehen zur Messung ist exemplarisch in Listing 6.2 zu sehen. Das methodische Vorgehen zur Messung der Ausführungszeit in Nanosekunden ist analog. Der Unterschied besteht lediglich darin, dass anstelle der `rdtsc` Instruktion die C-Funktion `clock_gettime()` aus der C-Bibliothek `time.h` verwendet wird [GNU]. Da sich die Messungen in Zyklen und Nanosekunden bei gleichzeitiger Ausführung gegenseitig beeinflussen würden, finden diese Messungen nacheinander statt.

```
1  static inline uint64_t rdtsc() {
2    uint64_t eax, edx;  // Variablen sind nach repräsentierten Registern benannt
3    __asm__ volatile ("lfence"); // Barriere lfence
4    __asm__ volatile ("mfence"); // Barriere mfence
5    // Liest Zeitstempel. Verwendet Register EDX:EAX. Untere Bits in eax, obere in edx
6    __asm__ volatile ("rdtsc" : "=a" (eax), "=d" (edx));
7    // Verschiebt in edx gespeicherte Bits in oberen Bit-Bereich von rdx und hängt eax
       durch OR an
8    uint64_t rdx = (edx<<32) | eax;
9    return rdx;   // Rückgabe Zeitstempel
10 }
```

Listing 6.1: *Verwendung der rdtsc Instruktion in C*

```
1  // Variablen zur Speicherung des Anfangszeitstempels und der am Ende gemessenen Zeit
2  uint64_t measured_time_start, measured_time;
3  // REP = Anzahl Messwiederholungen
4  uint64_t time_syscall[REP];       // Array zur Speicherung der Messwerte
5
6  for(int i = 0 ; i < REP ; i++){   // Schleife wiederholt Messung REP mal
7    __asm__ volatile ("cpuid");     // Serialisierung
8    measured_time_start = rdtsc();  // Zeitstempel zu Anfang speichern
9    systemcall();                   // Jeweiligen Systemcall ausführen (Platzhalter)
10   // Gemessene Ausführungszeit in Zyklen ist Differenz zwischen Anfang und Ende
11   measured_time = rdtsc() - measured_time_start;
12   // Zieht zuvor gemessenen Overhead von jeder Messung ab
13   measured_time = measured_time - overhead;
14   time_syscall[i] = measured_time;  // Speichert jeden Messwert
15 }
```

Listing 6.2: *Schema zur Messung der Ausführungszeit eines Systemcalls in Zyklen in C*

In einem Analysedurchlauf werden pro Test-Parameter 200.000 wiederholte Messungen durchgeführt. Um das Rauschen durch Faktoren, welche die Ausführung des Analyseprozesses beeinflussen können, zu minimieren, wird das Analyseprogramm fünf mal unabhängig voneinander ausgeführt. Aus den Durchschnittswerten der einzelnen Analysedurchläufe wird ein gemeinsamer Durchschnitt berechnet. Somit bestehen die finalen Durchschnittsergebnisse aus einer Millionen einzelnen Messungen pro Test-Parameter.

Alle Messungen werden in einem Array, welches dynamisch an die Anzahl der Messungen angepasst ist, im Analyseprozess zwischengespeichert (Vgl. Listing 6.2), um die Messergebnisse im Anschluss an die Messung gesammelt in eine Datei zu schreiben. Der Analyseprozess verwendet also eine dynamische und große Menge von Speicher, welches Auswirkungen auf das Kopieren des Prozesses durch forken haben kann. Um möglichst verlässliche Ergebnisse bei der Messung des Systemcalls `fork` zu erhalten, wird diese folglich in einen anderen Prozess ausgelagert, welcher

vom Analyseprogramm aufgerufen wird. Dieses Programm enthält eine minimale Menge an lokalen Variablen und Instruktionen, die für die Messung der Ausführungszeit des Systemcalls `fork` benötigt werden, sodass beim forken nur eine konstant kleine Menge an Speicher in den Kindprozess kopiert werden muss. Auch dieser Prozess wird fest an einen (anderen) Kern gebunden. Die Messwerte werden direkt in eine Datei geschrieben und im Anschluss vom Analyseprogramm ausgelesen und mit den restlichen Messungen in einer Datei zusammengeführt.

Um die Performance mit und ohne KPTI zu vergleichen, wird in einem Evaluationsprogramm der Performanceverlust der Systemcalls in Zyklen bzw. Nanosekunden und Prozent berechnet. Der Performanceverlust in Zyklen oder Nanosekunden wird durch die Differenz der beiden Durchschnittswerte berechnet, indem der ohne KPTI gemessene Wert vom Messergebnis mit KPTI subtrahiert wird. Zur Bestimmung des Performanceverlusts in Prozent wird der Messwert ohne KPTI auf dem jeweiligen Testgerät als Vergleichsbasis gewählt. Dabei entspricht ein dazu gleichwertiges Messergebnis aus den Messungen mit aktivierter Kernel Page Table Isolation 0 % Performanceverlust. Ein negativer Wert würde hier also bedeuten, dass KPTI sogar zu einem Performancegewinn führt und entsprechende Systemcalls weniger Ausführungszeit mit KPTI benötigen als ohne KPTI. Zudem werden automatisiert Tabellen und Grafiken erstellt, welche die Performance mit und ohne KPTI visuell anschaulich darstellen.

6.2.2 Verlust von Adressübersetzungen im TLB

Durch KPTI erfordert der Wechsel zwischen Kernel- und Nutzer-Modus ebenso einen Wechsel des virtuellen Adressraums. Dadurch wird das CR3 Register, welches auf die entsprechende Page Table zeigt, aktualisiert. Um die Anzahl an dadurch entstehenden impliziten TLB Flushes zu minimieren, werden auf modernen Intel-Prozessoren PCIDs verwendet (Vgl. Kapitel 5). Die Nutzung von PCIDs soll dafür sorgen, dass zum Prozess gehörige TLB Einträge markiert werden können, sodass die gespeicherten Adressübersetzungen beim Wechsel zwischen Kernel- und Nutzer-Modus nicht aus dem TLB entfernt werden. Dadurch sollen Performanceverluste durch KPTI minimiert werden. Die Durchführung einer Analyse der TLB Belegung nach einem prozessinternen Kontextwechsel soll zeigen, welche Auswirkungen auf den TLB KPTI im Vergleich zu deaktivierter Kernel Page Table Isolation tatsächlich hat.

Leider halten Hardwarehersteller genaue Details über ihre Produkte oft geheim. Daher ist der TLB ein Hardwarecache mit einer hier unbekannten Struktur. Das bedeutet insbesondere, dass unklar ist, wie genau Adressübersetzungen im TLB verdrängt oder überhaupt abgebildet werden und wie der konkrete Zusammenhang zwischen den verschiedenen TLB Leveln besteht. Adressen gezielt zu verdrängen ist somit nicht offensichtlich. In *„Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks“* werden diese Zusammenhänge in Teilen Reverse-Engineered [Gra+18]. Dabei wird festgestellt, dass Unterschiede zwischen den untersuchten Intel-Prozessoren bestehen. Insbesondere variieren Details zum TLB-Mapping, der Inklusivität von TLBs aus verschiedenen Leveln und der verwendeten Verdrängungsstrategien je nach verwendeter Architektur [Gra+18]. Die Analyse der Auswirkungen von KPTI wird auf mehreren Geräten mit verschiedenen Intel-Prozessoren durchgeführt. Daher ist hier ein individueller Ansatz, der sämtliche Feinheiten der jeweiligen Architekturen berücksichtigt, im Umfang einer Bachelorarbeit, gerade da viele Details nicht bekannt sind, nicht möglich. Es wird jedoch die Annahme getroffen, dass der TLB in seiner Struktur wie ein Set-Associative Cache (Vgl. Kapitel 2) aufgebaut ist. Zudem wird an-

genommen, dass Adressen anhand einer Random Replacment Policy verdrängt werden, sodass keine klare Vorhersage möglich ist, welche Adressübersetzung als nächstes verdrängt wird. Da Informationen über den Zusammenhang der verschiedenen TLB Level fehlen, die möglicherweise eine feingranularere Unterscheidung von Messergebnissen ermöglichen würden, wird der TLB in der Umsetzung der Performanceanalyse als eine einzelne Hardwarestruktur betrachtet.

Umsetzung:

Ist eine Adressübersetzung im TLB zwischengespeichert, benötigt ein Zugriff auf diese Übersetzung durchschnittlich etwa 0,5 - 1 Zykel [Pat16, Kapitel 5]. Ist eine Übersetzung der virtuellen Adresse zur physischen Adresse nicht im TLB vorhanden, muss der Page Walk erneut durchlaufen werden (Vgl. Kapitel 2). Dies benötigt etwa 10 - 100 zusätzliche Zykel [Pat16, Kapitel 5]. Durch die Messung von Zugriffszeiten können also Rückschlüsse darüber gezogen werden, ob eine Adressübersetzung im TLB vorhanden ist oder nicht. Die benötigten Zugriffszeiten werden in CPU Zyklen anhand der `rdtsc` Instruktion gemessen, wie bereits im vorangegangenen Unterkapitel beschrieben. Der Zugriff auf Adressen erfolgt durch einen inline Assemblerbefehl, welcher durch die inline Funktion `access_memory()` aufgerufen wird, wie in Listing 6.3 zu sehen ist. Systemcalls erfordern einen Wechsel in den Kernel-Modus und veranlassen somit eine Aktualisierung des CR3 Registers. Somit lassen sich die Auswirkungen von KPTI auf den TLB beobachten, indem die Zugriffszeit auf eine im TLB gespeicherte Adresse nach einem durch einen Systemcall hervorgerufenen Kontextwechsel überprüft wird. Dieses Vorgehen ähnelt dem Aufbau eines Prime + Probe Seitenkanals (Vgl. Kapitel 3). Um die Verfügbarkeit von Adressübersetzungen im TLB nach einem prozessinternen Kontextwechsel zu analysieren, muss eine Klassifikation der gemessenen Zugriffszeit als TLB Hit oder Miss stattfinden. Dazu werden zuerst die benötigten Zugriffszeiten bei einem TLB Hit und einem TLB Miss auf dem jeweiligen Testgerät gemessen.

```
1 static inline void access_memory(void* address){
2  // Zugriff: legt Adresse (0. Parameter address) in Register rax
3   __asm__ volatile ("movq (%0), %%rax\n"
4   :                                       // Keine Output-Operanden
5   : "c" (address)                         // Input-Operand address in register rcx
6   : "rax");                               // rax zur Zwischenspeicherung
7 }
```

Listing 6.3: *Zugriff auf Speicherstellen*

Messung TLB Hit Zur Bestimmung der benötigten Zugriffszeit bei TLB Hits wird zunächst ein Zugriff auf eine bestimmte virtuelle Testadresse durchgeführt, sodass die Übersetzung zur physischen Adresse im TLB zwischengespeichert wird. Auch im CPU Cache werden dadurch die Inhalte dieser Adresse abgelegt, sodass Latenzzeiten durch einen Cache Miss bei der Messung vermieden werden. Es wird erneut auf die Testadresse zugegriffen, um die benötigte Zugriffszeit bei einem TLB Hit zu erhalten.

Messung TLB Miss Um die Zugriffszeit bei TLB Misses zu bestimmen, wird durch den Aufruf genügend vieler anderer Adressen eine durch einen vorherigen Zugriff gespeicherte Testadresse aus dem TLB verdrängt. Dabei ist es wichtig, dass die Verdrängungsadressen verschiedene Pages referenzieren, da verschiedene Adressen innerhalb einer Page nur einen TLB Eintrag benötigen.

Nachdem eine ausreichende Anzahl an Zugriffen auf Verdrängungsadressen stattgefunden hat, wird die Zeit bei einem erneuten Zugriff auf die Testadresse gemessen. Diese Zeit repräsentiert die Zugriffszeit bei einem TLB Miss.

Die richtige Anzahl und Art an Test- und Verdrängungsadressen zu finden ist jedoch nicht trivial. Um ein größeres Maß an Verlässlichkeit der Messungen zu erhalten, ist es sinnvoll, mehr als eine Testadresse zu verwenden. Damit die Verdrängung von Testadressen wie gewünscht zur Messung von Zugriffszeiten bei TLB Misses funktioniert, muss zudem die richtige Anzahl und Art von Verdrängungsadressen gefunden werden.

Testadressen Damit sich bei der Messung von Zugriffszeiten bei TLB Hits die TLB Einträge nicht gegenseitig verdrängen und somit die Messung verfälschen, ist es wichtig, dass die Anzahl an Testadressen in jedem Fall kleiner als die Anzahl von möglichen TLB Einträgen ist. Wie eingangs erwähnt, ist nicht klar, nach welchem Schema Einträge im TLB verdrängt werden bzw. eine Verdrängung nach Random Replacement Policy ist nicht einfach vorhersagbar. Insofern ist es auch nicht trivial zu prognostizieren, welche Adressen nicht verdrängt werden. In der Praxis hat sich jedoch eine Anzahl von 4 Testadressen bewährt. Nach den zuvor getroffenen Annahmen und laut *„Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks“* ähneln heutige TLBs konzeptionell der Funktionsweise von Set-Associative Caches (Vgl. Kapitel 2), indem Sets und Ways zur Organisation der Einträge verwendet werden [Gra+18]. TLBs verwenden heutzutage meist mindestens 4-Way Sets [Gra+18]. Dies könnte also eine mögliche Erklärung dafür sein, weshalb 4 Testadressen sich als besonders geeignet bewährt haben, da somit voraussichtlich keine gegenseitige Verdrängung der TLB Einträge stattfindet, selbst wenn die verwendeten Testadressen kongruent zueinander sein sollten.

Verdrängungsadressen Der TLB ist üblicherweise deutlich kleiner als der CPU Cache, wie beispielsweise in den Systemspezifikationen der zum Testen verwendeten Geräte im folgenden Kapitel erkennbar ist. Eine Anzahl von Zugriffen auf Verdrängungsadressen, die größer als die Anzahl der Einträge des TLBs aber kleiner als die Anzahl an Einträgen des Caches ist, führt in der Theorie dazu, dass lediglich die Adressübersetzungen im TLB, nicht jedoch die gespeicherten Daten im Cache verdrängt werden. In der Praxis kann eine Verdrängung auch aus dem CPU Cache jedoch nicht vollständig ausgeschlossen werden, beispielsweise aufgrund von kongruenten Adressen oder der entsprechenden Verdrängungsstrategie des Caches. Dadurch entstehende Latenzzeiten beim Zugriff auf Testadressen nach der Verdrängung verfälschen die Messergebnisse zu TLB Misses. Die gemessenen Zeiten beinhalten so nicht nur die Zugriffszeit bei einem TLB Miss, sondern teilweise ebenso die benötigte Zeit für einen Cache Miss. Da so entstehende Cache Misses ungeplant und somit unvorhersehbar auftreten, macht dies Messungen zu TLB Misses undurchsichtig und unbrauchbar. Abhilfe zu dieser Problematik wird dadurch geschaffen, dass mehrere virtuelle Speicherstellen auf dieselbe physische Speicherstelle verweisen. Dadurch wird Rauschen durch den CPU Cache vermieden [Gra+18]. Realisiert wird dies, indem die physische Adresse eines Shared Memory Segments mehrmals mittels Aufruf von `mmap` an verschiedene virtuelle Adressen gebunden wird. Die Größe des Shared Memory Segments beträgt 4096 Byte, also genau eine Page. Eine Anzahl von 5000 virtuellen Verdrängungsadressen hat sich in der Praxis bei der Durchführung der Messungen bewährt.

Messung nach Kontextwechsel Für die eigentliche Messung der benötigten Zugriffszeit nach einem Kontextwechsel wird zunächst wieder auf die Testadressen zugegriffen, sodass die Übersetzungen im TLB zwischengespeichert sind. Anschließend werden die Systemcalls `getpid`, `mmap` und `munmap` ausgeführt. Dadurch werden Kontextwechsel durchgeführt, die eine Aktualisierung des CR3 Registers erfordern. Dies kann potenziell Auswirkungen auf die anschließend im TLB noch verfügbaren Adressübersetzungen, aufgrund von impliziten TLB Flushes, haben. Abschließend wird im Nutzerprozess die Zugriffszeit auf die Testadressen erneut gemessen, um nach einer Klassifikation der benötigten Zeit identifizieren zu können, ob ein TLB Hit oder ein TLB Miss vorliegt.

Um typische Zugriffszeiten für alle Messungen zuverlässig zu bestimmen, werden die oben beschriebenen Vorgänge mehrfach wiederholt. Da 4 Testadressen verwendet werden und genauso wie bei der Analyse der Ausführungszeit von Systemcalls 200.000 Wiederholungen pro Durchlauf stattfinden und 5 Analysedurchläufe getätigt werden, finden insgesamt 4 Millionen einzelne Messungen statt. Die Ergebnisse der Messungen werden in einem als Histogramm aufgebauten Array gespeichert. Dabei repräsentiert der jeweilige Positionsindex die benötigte Anzahl von Zyklen und der an dieser Position gespeicherte Wert gibt an, wie häufig diese Zeit bei den Messungen vorkommt. Da übliche Zugriffszeiten auf im CPU Cache vorhandene Daten deutlich unterhalb einer Zeit von 300 Zyklen liegen, ist das Histogramm-Array auf diese Zeit beschränkt. Benötigt eine Messung mehr Zeit als diese 300 Zykel, wird sie der letzten Position im Array zugeordnet. Zudem sind so hohe Zugriffszeiten nicht eindeutig mit einem TLB Hit oder Miss in Verbindung zu bringen, sondern sind auf andere Einflüsse während der Messung zurückzuführen. Daher werden diese Messungen für die anschließende Evaluation nicht weiter betrachtet. Ein solcher beeinflussender Faktor kann beispielsweise die Unterbrechung des Analyseprozesses durch das Scheduling anderer Prozesse sein. Um Unterbrechungen innerhalb einer Messung minimieren zu können, wird an geeigneten Stellen die C-Funktion `sched_yield()` aus der C-Bibliothek `sched.h` aufgerufen [die]. Diese signalisiert dem Betriebssystem die Bereitschaft für einen Kontextwechsel, um anderen Prozessen frühzeitig ihre Ausführungszeit zu gewähren. Somit wird eine Unterbrechung zu einer ungeeigneten Stelle in der Ausführung der Analyse möglichst vermieden.

Um die Messergebnisse der TLB Belegung nach einem Kontextwechsel als TLB Hit oder TLB Miss klassifizieren zu können, werden zwei verschiedene Strategien angewendet. Die erste Strategie verwendet einen Grenzwert, auch *Threshold* genannt. Die zweite Strategie führt eine Klassifizierung der Messergebnisse nach Ähnlichkeit zu den TLB Hit und Miss Messungen durch.

Klassifikation nach Threshold: Aus den Messungen der benötigten Zeit für TLB Hits und TLB Misses wird jeweils das Maximum der am häufigsten benötigten Zeit in CPU Zyklen bestimmt. Der Durchschnitt dieser beiden Maxima ist der Threshold. Ein TLB Hit benötigt weniger CPU Zykel als ein TLB Miss. Folglich werden Messergebnisse nach einem Kontextwechsel, die unterhalb des Thresholds liegen als TLB Hit und Werte, die sich oberhalb des Grenzwerts befinden als TLB Miss klassifiziert.

Klassifikation nach Ähnlichkeit: Die zweite Strategie klassifiziert die Messwerte nach einem Kontextwechsel als TLB Hit oder Miss je nach Ähnlichkeit zu den zuvor durchgeführten Messungen für TLB Hits und Misses. Dafür wird zu allen durchgeführten Messungen (TLB Hits, TLB

Misses, Messung nach Kontextwechsel) pro Positionsindex des jeweiligen Histogramm Arrays die Häufigkeit dieser benötigten Zykel Zeit betrachtet. Durch Berechnung und Vergleich der Differenz zwischen der eigentlichen Messung und der jeweiligen Messung für TLB Hits bzw. TLB Misses findet die Klassifikation der Messung statt. Dabei werden die Messergebnisse als TLB Hit oder Miss eingestuft, je nach dem welche Differenz der Häufigkeit für eine bestimmte Zykel Zahl geringer und somit ähnlicher ist.

Die Messungen finden im Analyseprogramm statt. Die finale Klassifikation und eine tabellarische und grafische Aufbereitung der Ergebnisse wird im Evaluationsprogramm durchgeführt. Der vollständige Code für die Performanceanalyse ist dieser Arbeit beigelegt und auch auf Github zu finden [Hei21].

6.3 Evaluation der Ergebnisse

Anhand der zuvor beschriebenen Methodik zur Messung der Performance mit und ohne KPTI (Vgl. Abschnitt 6.2) wird in diesem Unterkapitel evaluiert, welche Performanceeinbußen durch die Meltdown-Mitigation auf den Testgeräten entstehen und welche allgemeinen Schlüsse sich daraus ziehen lassen. Dazu werden zunächst die Ergebnisse zur Performance von Systemcalls besprochen und anschließend auf die Ergebnisse zum Verlust von Adressübersetzungen im TLB durch KPTI eingegangen. Am Ende dieses Unterkapitels befindet sich eine Zusammenfassung, welche die wichtigsten Erkenntnisse durch diese Arbeit kurz beschreibt. Die Messungen werden auf den in Tabelle 2 genannten Geräten durchgeführt. Die Testgeräte sind nach der Art des Geräts (Laptops mit OS auf externer Festplatte, Laptop mit OS auf eingebauter Festplatte und Desktop PCs) und absteigend nach der Prozessor-Serie der verwendeten CPUs sortiert. Die technischen Daten der Geräte werden anhand von Kommandozeilen Befehlen ermittelt, welche dem Makefile des beigefügten Codes entnommen werden können [Hei21]. Wie im vorherigen Unterkapitel bereits erwähnt, verwenden alle Testgeräte Ubuntu 20.04 als Betriebssystem.

Tabelle 2: *Zur Analyse verwendete Geräte*

	Intel-Prozessor	**CPU Cache**	**TLB für 4 KB Pages**	**PCID**	**Kernel Version**
Gerät 1: Laptop[6] *HP Pavilion x360*	i7-8550U, Kaby Lake, 14 nm	L1d: 128 KiB L1i: 128 KiB L2: 1 MiB L3: 8 MiB	dTLB: 4-Way, 64 Einträge iTLB: 8-Way, 64 Einträge L2: 6-Way, 1536 Einträge	✓	5.4.0-26-generic
Gerät 2: Laptop[6] *Acer Aspire V3*	i5-3230M, Ivy Bridge, 22 nm	L1d: 64 KiB L1i: 64 KiB L2: 512 KiB L3: 3 MiB	dTLB: 4-Way, 64 Einträge iTLB: 4-Way, 64 Einträge L2: 4-Way, 512 Einträge	✓	5.4.0-26-generic
Gerät 3: Laptop *Lenovo ThinkPad*	i7-4800MQ, Haswell, 22 nm	L1d: 128 KiB L1i: 128 KiB L2: 1 MiB L3: 6 MiB	dTLB: 4-Way, 64 Einträge iTLB: 8-Way, 64 Einträge L2: 8-Way, 1024 Einträge	✓	5.8.0-49-generic
Gerät 4: Desktop PC	i7-4790K, Haswell, 22 nm	L1d: 128 KiB L1i: 128 KiB L2: 1 MiB L3: 8 MiB	dTLB: 4-Way, 64 Einträge iTLB: 8-Way, 64 Einträge L2: 8-Way, 1024 Einträge	✓	5.8.0-48-generic
Gerät 5: Desktop PC	i5-4440, Haswell, 22 nm	L1d: 128 KiB L1i: 128 KiB L2: 1 MiB L3: 6 MiB	dTLB: 4-Way, 64 Einträge iTLB: 8-Way, 128 Einträge L2: 8-Way, 1024 Einträge	✓	5.8.0-49-generic
Gerät 6: Desktop PC	Core 2 Duo (CPU 6320), Conroe, 65 nm	L1d: 64 KiB L1i: 64 KiB L2: 4 MiB	dTLB: 4-Way, 16 Einträge iTLB: 4-Way, 128 Einträge L2: 4-Way, 256 Einträge	✗	5.8.0-49-generic

Evaluation der Messergebnisse zur Performance von Systemcalls

Auf allen getesteten Geräten ist ein deutlicher Unterschied in der benötigten Ausführungszeit von Systemcalls messbar, je nach dem ob KPTI aktiviert oder deaktiviert ist. Der gemessene Performanceverlust durch KPTI in Zyklen beträgt für die ausgewählten File Management Systemcalls auf den Testgeräten 10,94 % - 35,86 % für den Systemcall `open`, 24,43 % - 69,45 % für den Systemcall `read`

[6] Das Betriebssystem befindet sich auf einer über USB angeschlossenen Festplatte (Samsung MZ-76E500B/EU 860 EVO)

und 28,71 % - 66,73 % für den Systemcall close. Der gemessene Performanceverlust in Nanosekunden verhält sich sehr ähnlich, wie in der Tabelle in Abbildung 10 zu sehen ist. Eine Übersicht der gemessenen Ausführungszeiten von File Management Systemcalls in Zyklen und Nanosekunden ist anhand von Balkendiagrammen für alle Testgeräte in Abbildung 9 dargestellt. Die Messungen der Performance in Zyklen und in Nanosekunden werden nicht gleichzeitig, sondern nacheinander durchgeführt, sodass die jeweiligen Messungen nicht vollständig korrespondieren. Eine Tendenz des Performanceverlusts ist jedoch durch beide Messungen im Vergleich erkennbar, wie auch in Abbildung 11 zu sehen ist.

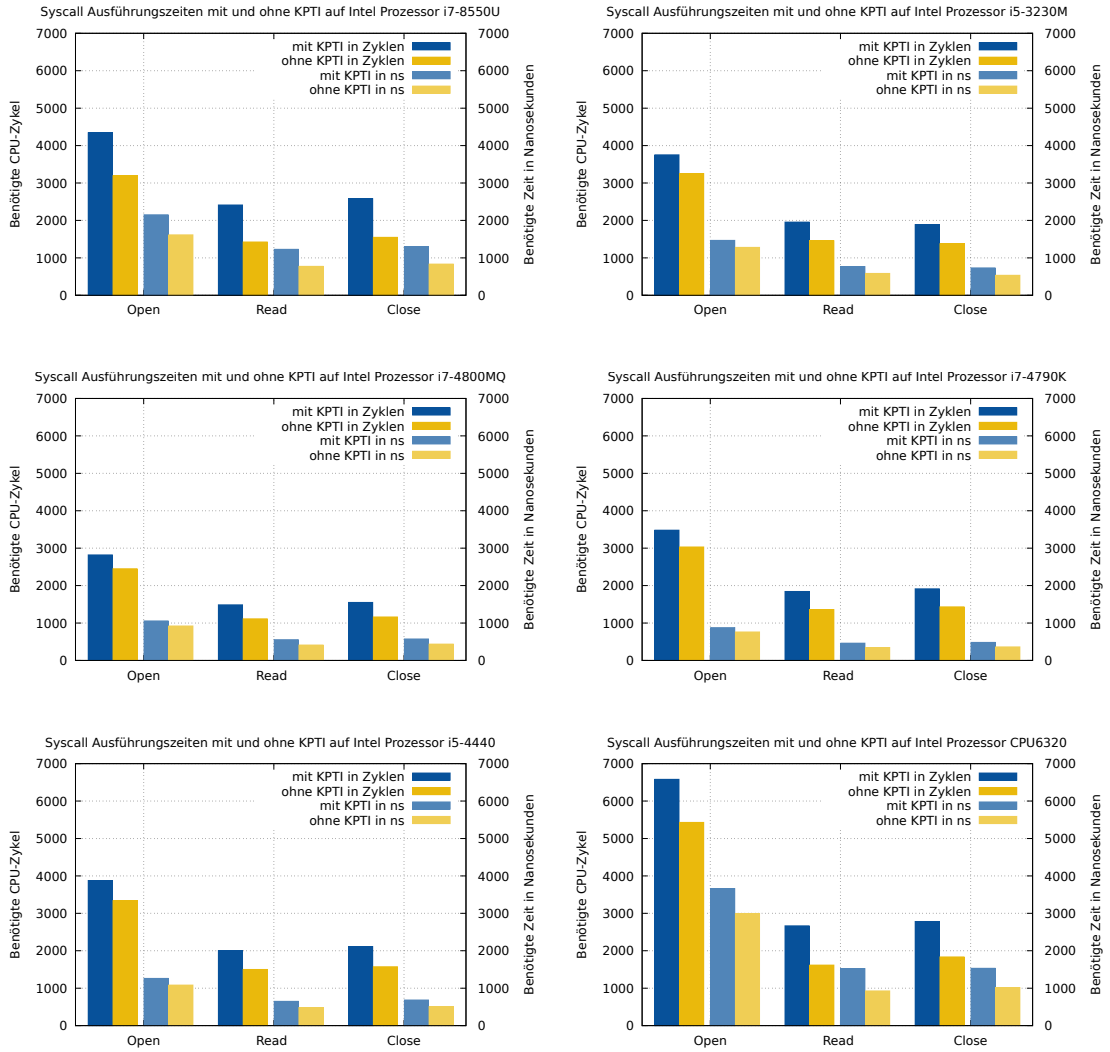


Abbildung 9: *Durchschnittliche Ausführungszeiten von File Management Systemcalls mit und ohne KPTI von allen Testgeräten bei fünf Analysedurchläufen (5 · 200.000 Messungen)*

Die Durchschnittswerte der Analysedurchläufe in Nanosekunden weisen meist eine geringere Standardabweichung auf als die Durchschnittswerte der Messungen in Zyklen. Beispielsweise haben die 5 Analysedurchläufe auf Gerät 4 mit aktivierter Kernel Page Table Isolation eine Standardabweichung bei der Messung in Zyklen von 12,08 für den Systemcall open, 11,41 für den Systemcall read und 7,17 für den Systemcall close. In Nanosekunden betragen diese nur 5,04 für

	Messungs Einheit	Systemcall	KPTI	NOPTI	Performanceverlust
Gerät 1 *i7-8550U*	Zykel	Open	4.342	3.196	1.146 Zykel / 35,86 %
		Read	2.413	1.424	989 Zykel / 69,45 %
		Close	2.581	1.548	1.033 Zykel / 66,73 %
	Nanosekunden	Open	2.151	1.615	536 ns / 33,19 %
		Read	1.227	773	454 ns / 58,73 %
		Close	1.300	834	466 ns / 55,88 %
Gerät 2 *i5-3230M*	Zykel	Open	3.753	3.252	501 Zykel / 15,41 %
		Read	1.955	1.457	498 Zykel / 34,18 %
		Close	1.888	1.377	511 Zykel / 37,11 %
	Nanosekunden	Open	1.467	1.280	187 ns / 14,61 %
		Read	766	584	182 ns / 31,16 %
		Close	732	535	197 ns / 36,82 %
Gerät 3 *i7-4800MQ*	Zykel	Open	2.820	2.450	370 Zykel / 15,10 %
		Read	1.486	1.111	375 Zykel / 33,75 %
		Close	1.551	1.162	389 Zykel / 33,48 %
	Nanosekunden	Open	1.055	921	134 ns / 14,55 %
		Read	555	412	143 ns / 34,71 %
		Close	573	436	137 ns / 31,42 %
Gerät 4 *i7-4790K*	Zykel	Open	3.480	3.031	449 Zykel / 14,81 %
		Read	1.842	1.359	483 Zykel / 35,54 %
		Close	1.913	1.431	482 Zykel / 33,68 %
	Nanosekunden	Open	876	761	115 ns / 15,11 %
		Read	462	345	117 ns / 33,91 %
		Close	483	362	121 ns / 33,43 %
Gerät 5 *i5-4440*	Zykel	Open	3.877	3.338	539 Zykel / 16,15 %
		Read	2.007	1.500	507 Zykel / 33,80 %
		Close	2.113	1.573	540 Zykel / 34,33 %
	Nanosekunden	Open	1.262	1.083	179 ns / 16,53 %
		Read	649	483	166 ns / 34,37 %
		Close	684	507	177 ns / 34,91 %
Gerät 6 *Core 2 Duo*	Zykel	Open	6.580	5.429	1.151 Zykel / 21,20 %
		Read	2.665	1.616	1.049 Zykel / 64,91 %
		Close	2.783	1.835	948 Zykel / 51,66 %
	Nanosekunden	Open	3.661	2.997	664 ns / 22,16 %
		Read	1.523	927	596 ns / 64,29 %
		Close	1.531	1.019	512 ns / 50,25 %

Abbildung 10: *Durchschnittswerte aller File Management Systemcalls Performance Messungen*

den Systemcall open, 2,80 für den Systemcall read und 3,50 für den Systemcall close. Auch in der Betrachtung der Standardabweichung der 200.000 Messungen eines einzelnen Analysedurchlaufs verhält sich dies ähnlich. Da diese Messwerte jedoch noch nicht durch den Durchschnitt normalisiert worden sind, weisen sie selbstverständlich insgesamt höhere Standardabweichungen auf. Beispielsweise betragen die Standardabweichungen der einzelnen Messungen in Zyklen eines Analysedurchlaufs mit KPTI auf Gerät 4 315,82 für den Systemcall open, 35,99 für den Systemcall read und 39,89 für den Systemcall close. In Nanosekunden betragen diese nur 32,34 für den Systemcall open, 7,43 für den Systemcall read und 7,61 für den Systemcall close. Diese Auffälligkeit besteht bei allen Messungen zur Performance von Systemcalls. Daher ist der Performanceverlust bei der Messung in Nanosekunden vermutlich akkurater. Eine Aussage darüber, dass Messergebnisse mit oder ohne KPTI stärker variieren, lässt sich nicht treffen, da hier keine auffälligen Regelmäßigkeiten dazu bestehen. Eine Auflistung aller Varianzen und Standardabweichungen liegt zusammen mit den Messergebnissen der einzelnen Testgeräte dieser Arbeit bei [Hei21].

Wird der Performanceverlust des Systemcalls fork betrachtet, fällt auf, dass dieser Systemcall im Vergleich einen deutlich geringeren Performanceverlust durch KPTI erleidet als andere Systemcalls. Dies ist in Abbildung 11 erkennbar. Der Systemcall fork benötigt grundsätzlich deutlich mehr Zeit als die anderen getesteten Systemcalls. Während alle anderen getesteten Systemcalls Ausführungszeiten von einigen Hundert bis Tausend Zyklen bzw. Nanosekunden benötigen, beläuft sich die Ausführungsdauer des Systemcalls fork auf mehrere Zehntausende Zykel bzw. Nanosekunden. Dies ist exemplarisch anhand der Auflistung der auf Gerät 1 benötigten Zeiten in der Tabelle in Abbildung 13 und Abbildung 14 erkennbar. Auch wenn der absolute Performanceverlust in Zyklen bzw. Nanosekunden im Vergleich deutlich höher ist als bei den anderen Systemcalls, fällt dieser Overhead aufgrund der ohnehin größeren Ausführungszeit weniger stark ins Gewicht. Dies gilt für alle Messungen. Beispielsweise benötigen die Systemcalls getpid, mmap und munmap auf Gerät 1 alle etwas mehr als 500 ns zusätzlich, wenn KPTI aktiviert ist. Der Systemcall fork benötigt hier mit aktivierter Kernel Page Table Isolation etwas mehr als 8400 ns zusätzlich. Der Performanceverlust beim Systemcall fork beläuft sich auf Gerät 1 jedoch auf nur etwa 29 %, während die anderen Systemcalls aus der Kategorie Process Control auf diesem Gerät Performanceverluste von über 100 % erleiden (Vgl. Abbildung 11). Dabei wird auf Gerät 1 bereits insgesamt der größte prozentuale Performanceverlust gemessen. Der Performanceverlust beim Systemcall fork beläuft sich auf den anderen Testgeräten auf 0,52 % - 6,29 % bei der Messung in Zyklen und auf 2,32 % - 10, 52 % in Nanosekunden. Der absolute Performanceverlust in Zyklen bzw. Nanosekunden ist auf allen Testgeräten für den Systemcall fork etwa 4 bis 15 mal so hoch wie der größte absolute Performanceverlust der anderen Systemcalls. Im Schnitt benötigen alle Systemcalls ausgenommen fork auf Gerät 1 1067,5 Zykel mit einer Varianz des durchschnittlichen Overheads von 6568,42 und einer Standardabweichung von 81,05. In Nanosekunden beträgt dieser durchschnittliche Overhead 524,58 ns mit einer Varianz von 1540,91 und einer Standardabweichung von 39,25. Alle anderen Systemcalls haben mit KPTI also einen nahezu konstant ähnlichen Performanceoverhead. Bei Systemcalls, die grundsätzlich weniger Ausführungszeit benötigen, fällt dieser Overhead stärker ins Gewicht. Beispielsweise benötigt auf Gerät 1 der Systemcall getpid mit KPTI durchschnittlich 820 ns und hat damit im Vergleich zur Messung ohne KPTI einen absoluten Performanceverlust von 509 ns und einen prozentualen Performanceverlust von 163,67 %. Der Systemcall mmap, der 1 Byte Speicher allokiert, hat mit einer durchschnittlichen Ausführungszeit von 1117 ns unter KPTI zwar einen höheren absoluten Performanceverlust von 569 ns, jedoch im Vergleich zum Systemcall

Abbildung 11: *Durchschnittlicher Systemcall Performanceverlust durch KPTI auf allen Testgeräten bei fünf Analysedurchläufen (5 · 200.000 Messungen). Die Geräte 3 - 5 mit einer Haswell Prozessorarchitektur weisen besonders hohe Ähnlichkeiten auf.*

getpid einen geringeren prozentualen Performanceverlust von 103,83 %. Die Beobachtung, dass Systemcalls mit grundsätzlich höheren Ausführungszeiten zwar auch einen leicht höheren absoluten Performanceoverhead haben, jedoch einen geringeren prozentualen Performanceoverhead aufweisen, lässt sich auf alle Messergebnisse übertragen. Der hohe absolute Performanceverlust beim forken ist somit im Vergleich zur grundsätzlich hohen Ausführungszeit des Systemcalls auf allen Testgeräten plausibel. Während der absolute Performanceverlust bei allen anderen Systemcalls auf Gerät 1 prozentual zwischen 24,92 % und 60,71 % der mit KPTI benötigten Zeit aufweist, beträgt der absolute Performanceverlust des Systemcalls fork prozentual 18,74 % der mit KPTI benötigten Ausführungszeit in Zyklen und 22,50 % in Nanosekunden. Daher lassen sich leider keine eindeutigen Schlüsse darüber ziehen, ob das Anlegen von mehr Speicherstrukturen beim Kreieren eines Kindprozesses die Performance unter KPTI stark beeinflusst. Die gemessenen Performanceverluste können, genau wie bei den anderen Systemcalls, auch auf den grundsätzlich veränderten Ablauf bei Kontextwechseln zurückzuführen sein.

Werden die absoluten Performanceverluste der jeweiligen Testgeräte betrachtet, ist zu beobachten, dass diese pro Gerät stets sehr ähnlich sind. Dies weist darauf hin, dass es einen konstanten mindest Basis-Verlust durch KPTI zu geben scheint. In Abhängigkeit einer grundsätzlich erhöhten Basis-Laufzeit eines Systemcalls ist der absolute Verlust durch KPTI jedoch ebenso leicht erhöht, wie exemplarisch an den Messwerten von Gerät 1 erkennbar ist (Vgl. Abbildung 13). Eine Aufbereitung der gemessenen absoluten Performanceverluste ist in der Tabelle in Abbildung 12 zu sehen. Dort ist erkennbar, dass die Geräte 2 - 5 eine verhältnismäßig geringe Standardabweichung beim absoluten Verlust aufweisen. Dies zeigt, dass alle gemessenen durchschnittlichen absoluten Performanceverluste in etwa gleich sind (ausgenommen fork). Gerät 1 und Gerät 6 weisen hier die höchsten Standardabweichungen auf. Auch der minimale und durchschnittliche absolute Verlust ist bei diesen beiden Testgeräten deutlich höher als bei den anderen Geräten. Da die von Gerät 6 verwendete Conroe Prozessorarchitektur die älteste unter den Testgeräten ist, sind hier deutlich höhere durchschnittliche absolute Performanceverluste zu erwarten. Gerät 1 verwendet die neueste Prozessorarchitektur unter den Testgeräten, weshalb es überraschend ist, dass dieses Gerät die größten Performanceeinbußen durch KPTI bei der Ausführung von Systemcalls zeigt.

Im Durchschnitt benötigen die ausgewählten Process Control Systemcalls sowohl mit als auch ohne KPTI weniger Ausführungszeit als die untersuchen File Management Systemcalls. (Der Systemcall fork wird aufgrund seiner Besonderheiten in der folgenden Betrachtung ausgelassen.) Beispielsweise beträgt die Laufzeit von Process Control Systemcalls auf Gerät 1 ohne KPTI durchschnittlich 428,67 ns, während die File Management Systemcalls hier durchschnittlich 1074 ns benötigen (Vgl. Abbildung 10 und Abbildung 13). Wie bereits erwähnt, wird pro Gerät ein konstant ähnlicher mindest Performanceverlust beobachtet und dieser Overhead fällt bei geringeren Basis-Laufzeiten prozentual stärker ins Gewicht. Beispielsweise weist Gerät 1 bei File Management Systemcalls im Durchschnitt einen absoluten Performanceverlust von 485,33 ns mit einem prozentualen Overhead von 49,27 % auf. Bei Process Control Systemcalls betragen diese 537,67 ns und 131,63 %. Somit sind die hier betrachteten Systemcalls aus der Kategorie Process Control prozentual stärker von Performanceeinbußen durch KPTI betroffen, als Systemcalls aus der Kategorie File Management. Dies ist jedoch nicht grundsätzlich auf die Art oder Funktion des Systemcalls, sondern lediglich auf die höhere Basis-Laufzeit des jeweiligen Systemcalls, zurückzuführen.

	Minimum absoluter Verlust	Durchschnitt absoluter Verlust	Standardabweichung absoluter Verlust
Gerät 1 *(Kaby Lake)*	914 Zykel	1.067,5 Zykel	84,65
	454 ns	524,58 ns	41
Gerät 2 *(Ivy Bridge)*	492 Zykel	511 Zykel	15,83
	182 ns	193,33 ns	7,29
Gerät 3 *(Haswell)*	364 Zykel	370 Zykel	7,24
	134 ns	136,5 ns	2,54
Gerät 4 *(Haswell)*	440 Zykel	454,08 Zykel	14,59
	114 ns	128,25 ns	9,52
Gerät 5 *(Haswell)*	502 Zykel	511,83 Zykel	13,42
	161 ns	165,83 ns	6,04
Gerät 6 *(Conroe)*	731 Zykel	882,42 Zykel	143,31
	412 ns	496,25 ns	83,13

Abbildung 12: *Performanceverlust durch KPTI bei Systemcalls auf Basis der Durchschnittswerte von fünf Analysedurchläufen. Der Systemcall fork wird in dieser Betrachtung aufgrund seiner Besonderheiten ausgelassen.*

Wie in Abbildung 11 erkennbar ist, wirkt sich die Größe des zu allozierenden oder freizugebenden Speichers nicht auf die Ausführungszeit aus. Auch in Abbildung 14 ist dies exemplarisch für das Testgerät 1 sichtbar. Unabhängig von der Speichergröße benötigen die Systemcalls mmap und munmap jeweils stets in etwa dieselbe Zeit zur Ausführung des Befehls.

Der Performanceverlust durch KPTI ist auf den Geräten 3 - 5 sehr ähnlich, wie in der Tabelle in Abbildung 10 und in der Abbildung 11 erkennbar ist. All diese Geräte verwenden eine Haswell Prozessorarchitektur. Auch die Messungen auf Gerät 2 mit einer Ivy Bridge Prozessorarchitektur ähneln den Messungen auf den Geräten mit einer Haswell Architektur, wenn auch teilweise nicht so deutlich. Die Haswell Architektur ist der Nachfolger der Ivy Bridge Prozessorarchitektur und behält einen Großteil des Architekturdesigns des Vorgängers bei [PC]. Die Testgeräte 1 und 6 zeigen zwar ähnliche Tendenzen zum Performanceverlust auf, jedoch weichen einzelne Performancemessungen im Vergleich stärker ab, als dies der Fall bei den Geräten 2 bis 5 untereinander ist. Demnach wird angenommen, dass der Prozessortyp große Auswirkungen darauf hat, wie stark das System von Leistungseinbußen durch KPTI betroffen ist. Zudem bestehen für dieselben oder eng verwandten Prozessorarchitekturen rekonstruierbar ähnliche Performanceverluste durch KPTI.

Das Testgerät 1 mit einem Intel i7-8550U Prozessor weist insgesamt die größten Leistungseinbußen bei der Nutzung von Systemcalls auf. Der durchschnittliche Performanceverlust bei der Ausführung der gewählten Systemcalls beträgt hier 105,02 %. Damit ist dieses Testgerät sogar stärker von Performanceeinbußen durch KPTI betroffen, als das Gerät 6 mit dem ältesten Intel-Prozessor unter den Testgeräten. Auf Gerät 6 beträgt der durchschnittliche Performanceverlust gerade einmal 78,29 %. Ein durch KPTI verursachter Performanceoverhead ist auf den restlichen Testgeräten noch einmal geringer. Auf dem Testgerät 2 mit einer Ivy Bridge Prozessorarchitektur beträgt der durchschnittliche Performanceverlust bei der Ausführung von Systemcalls 55,12 %. Die durchschnittlichen Performanceverluste auf den Testgeräten mit einer Haswell Architektur sind alle sehr ähnlich: Auf Gerät 3 wird ein durchschnittlicher Performanceverlust von 39,87 %, auf Gerät 4 40,93 % und auf Gerät 5 40,49 % gemessen. Bemerkenswert ist insbesondere, dass

Messungs Einheit	Systemcall	KPTI	NOPTI	Performanceverlust
Zykel	Fork	76.997	62.567	14.430 Zykel / 23,06 %
	Getpid	1.623	709	914 Zykel / 128,91 %
	Mmap (1 Byte)	2.230	1.075	1.155 Zykel / 107,44 %
	Munmap (1 Byte)	1.687	665	1.022 Zykel / 153,68 %
	Mmap (1 KiB)	2.229	1.073	1.156 Zykel / 107,74 %
	Munmap (1 KiB)	1.687	663	1.024 Zykel / 154,45 %
	Mmap (1 MiB)	2.265	1.101	1.164 Zykel / 105,72 %
	Munmap (1 MiB)	1.686	664	1.022 Zykel / 153,92 %
	Mmap (1 GiB)	2.242	1.080	1.162 Zykel / 107,59 %
	Munmap (1 GiB)	1.685	662	1.023 Zykel / 154,53 %
Nanosekunden	Fork	37.470	29.041	8.429 ns / 29,02 %
	Getpid	820	311	509 ns / 163,67 %
	Mmap (1 Byte)	1.117	548	569 ns / 103,83 %
	Munmap (1 Byte)	849	340	509 ns / 149,71 %
	Mmap (1 KiB)	1.109	540	569 ns / 105,37 %
	Munmap (1 KiB)	848	337	511 ns / 151,63 %
	Mmap (1 MiB)	1.134	558	576 ns / 103,23 %
	Munmap (1 MiB)	850	338	512 ns / 151,48 %
	Mmap (1 GiB)	1.121	548	573 ns / 104,56 %
	Munmap (1 GiB)	849	338	511 ns / 151,18 %

Abbildung 13: *Tabelle von Ausführungszeiten der Process Control Systemcalls mit und ohne KPTI auf Testgerät 1*

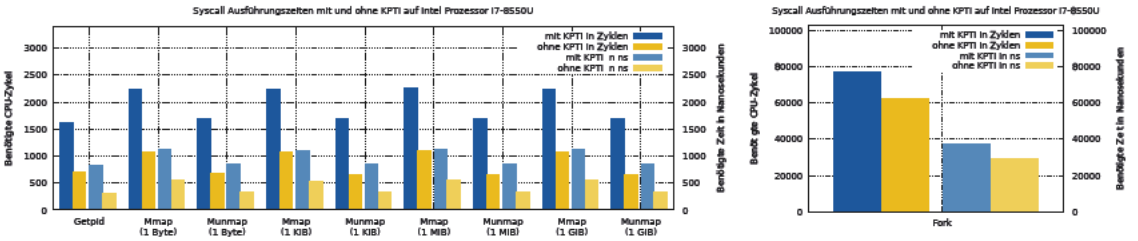


Abbildung 14: *Diagramm von Ausführungszeiten der Process Control Systemcalls mit und ohne KPTI auf Testgerät 1*

die hier detailreicher gemessenen Performanceverluste die in anderen Quellen prognostizierten Gesamtperformanceverluste weit übersteigen (Vgl. Abschnitt 6.1).

Evaluation der Messergebnisse zur TLB Belegung

Auch bei der Analyse der TLB Belegung ist ein Unterschied zwischen aktivierter und deaktivierter Kernel Page Table Isolation messbar. Im Durchschnitt benötigen alle Messungen mit KPTI nach

einem durch Aufruf von Systemcalls herbeigeführten Kontextwechsel 1 bis 7 Zykel zusätzlich. Die Werte zu Varianz und Standardabweichung können der Tabelle 15 entnommen werden. Da die Belegung des TLBs von außen nicht einsehbar ist, werden die Messwerte zur Klassifikation als TLB Hit oder Miss verwendet. Alle als TLB Miss eingestuften Messergebnisse weisen darauf hin, dass Adressübersetzungen nach einem Kontextwechsel verloren gegangen sind. Nach der Klassifikation nach Threshold wird so auf den Testgeräten ein Performanceverlust von 11,97 % bis 290,86 % durch KPTI gemessen. Nach Ähnlichkeitsklassifikation werden -14,07 % bis 324,09 % an zusätzlich verlorenen Adressübersetzungen mit KPTI verzeichnet.

		KPTI		NOPTI	
	Messung	Varianz	Standardabweichung	Varianz	Standardabweichung
Gerät 1 *i7-8550U*	TLB Hit	27.94	5.29	5624.64	75.00
	TLB Miss	31.48	5.61	4543.78	67.41
	Nach Kontextwechsel	30.81	5.55	5393.28	73.44
Gerät 2 *i5-3230M*	TLB Hit	17.76	4.21	19.93	4.46
	TLB Miss	36.69	6.06	10.04	3.17
	Nach Kontextwechsel	14.87	3.86	26.32	5.13
Gerät 3 *i7-4800MQ*	TLB Hit	7063.47	84.04	7103.38	84.28
	TLB Miss	4341.54	65.89	3782.36	61.50
	Nach Kontextwechsel	3136.69	56.01	4237.92	65.10
Gerät 4 *i7-4790K*	TLB Hit	88.96	9.43	309.36	17.59
	TLB Miss	140.85	11.87	96.12	9.80
	Nach Kontextwechsel	48.03	6.93	111.30	10.55
Gerät 5 *i5-4440*	TLB Hit	5.37	2.32	4.99	2.23
	TLB Miss	31.74	5.63	26.83	5.18
	Nach Kontextwechsel	21.96	4.69	23.95	4.89
Gerät 6 *Core 2 Duo*	TLB Hit	6.93	2.63	11.80	3.44
	TLB Miss	35.07	5.92	25.50	5.05
	Nach Kontextwechsel	145.59	12.07	34.42	5.87

Abbildung 15: *Varianz und Standardabweichung aller 4.000.000 Messungen zur TLB Belegung mit und ohne KPTI auf allen Testgeräten*

Auf dem Testgerät 5 wird nach Ähnlichkeitsklassifikation also sogar ein Performancegewinn festgestellt. Durch eine Klassifikation nach Threshold wird auf demselben Gerät hingegen ein Performanceverlust von 97,21 % verzeichnet. Zu sehen ist dies in der tabellarischen Auflistung der Messergebnisse in Abbildung 16. Dies zeigt, dass eine numerische Klassifikation der Messwerte als TLB Hit oder Miss schwierig sein kann und die zu den Messergebnissen passende Methode der Klassifizierung eine entscheidende Rolle spielt. Werden die Messungen mit und ohne KPTI im Vergleich betrachtet (Vgl. Abbildung 17), ist erkennbar, dass mit aktivierter Kernel Page Table Isolation auf Gerät 5 eine Verschiebung der gemessenen Zeiten nach rechts, in die Richtung höherer benötigter Zykel Zeiten, stattfindet. Die benötigten Durchschnittszeiten für TLB Hits, Misses und den Threshold sind auf diesem Gerät in etwa gleich, wie in der Tabelle in Abbildung 16 nachvollzogen werden kann. Somit ist die in Abbildung 17 sichtbare Verschiebung der Messungen nicht auf grundsätzlich höhere Zugriffszeiten unter KPTI zurückzuführen, sondern tatsächlich auf einen erhöhten Verlust von Adressübersetzungen im TLB. Für dieses Testgerät ist somit eine Klassifikation nach Threshold die geeignetere Methode. Diese Methode liefert genau dann zuverlässige Ergebnisse, wenn eine klare Trennung von TLB Hits und Misses möglich ist, was bei allen im Rahmen dieser Bachelorarbeit durchgeführten Messungen größtenteils der Fall ist.

		KPTI	NOPTI	Verlust durch KPTI
Gerät 1 *i7-8550U*	Gesamt gewertete Messungen nach Kontextwechsel	3.999.481 (4.000.000)	3.199.757 (4.000.000)	-
	Durchschnitt TLB Hit	13 Zykel	11 Zykel	2 Zykel / 18,18 %
	Durchschnitt TLB Miss	34 Zykel	31 Zykel	3 Zykel / 9,68 %
	Durchschnitt Messung nach Kontextwechsel	23 Zykel	16 Zykel	7 Zykel / 43,75 %
	Threshold	25 Zykel	24 Zykel	1 Zykel / 4,17 %
	Nach Threshold als TLB Miss klassifiziert	1.529.453 / 38,24 %	398.933 / 12,47 %	1.130.520 Hits / 283,39 %
	TLB Misses nach Ähnlichkeitsklassifikation	3.355.504 / 83,90 %	791.223 / 24,73 %	2.564.281 Hits / 324,09 %
Gerät 2 *i5-3230M*	Gesamt gewertete Messungen nach Kontextwechsel	3.999.707 (4.000.000)	3.999.397 (4.000.000)	-
	Durchschnitt TLB Hit	10 Zykel	13 Zykel	-3 Zykel / -23,08 %
	Durchschnitt TLB Miss	22 Zykel	25 Zykel	-3 Zykel / -12,00 %
	Durchschnitt Messung nach Kontextwechsel	16 Zykel	15 Zykel	1 Zykel / 6,67 %
	Threshold	17 Zykel	19 Zykel	-2 Zykel / -10,53 %
	Nach Threshold als TLB Miss klassifiziert	1.173.524 / 29,34 %	709.464 / 17,74 %	464.060 Hits / 65,41 %
	TLB Misses nach Ähnlichkeitsklassifikation	1.153.058 / 28,83 %	675.514 / 16,89 %	477.544 Hits / 70,69 %
Gerät 3 *i7-4800MQ*	Gesamt gewertete Messungen nach Kontextwechsel	3.543.703 (4.000.000)	3.358.331 (4.000.000)	-
	Durchschnitt TLB Hit	17 Zykel	18 Zykel	-1 Zykel / -5,56 %
	Durchschnitt TLB Miss	26 Zykel	27 Zykel	-1 Zykel / -3,70 %
	Durchschnitt Messung nach Kontextwechsel	22 Zykel	21 Zykel	1 Zykel / 4,76 %
	Threshold	22 Zykel	22 Zykel	0 Zykel / 0,00 %
	Nach Threshold als TLB Miss klassifiziert	1.569.880 / 44,30 %	1.151.722 / 34,29 %	418.158 Hits / 36,31 %
	TLB Misses nach Ähnlichkeitsklassifikation	2.797.214 / 78,93 %	2.765.820 / 82,36 %	31.394 Hits / 1,14 %
Gerät 4 *i7-4790K*	Gesamt gewertete Messungen nach Kontextwechsel	3.999.406 (4.000.000)	3.992.567 (4.000.000)	-
	Durchschnitt TLB Hit	23 Zykel	17 Zykel	6 Zykel / 35,29 %
	Durchschnitt TLB Miss	36 Zykel	34 Zykel	2 Zykel / 5,88 %
	Durchschnitt Messung nach Kontextwechsel	22 Zykel	20 Zykel	2 Zykel / 10,00 %
	Threshold	23 Zykel	21 Zykel	2 Zykel / 9,52 %
	Nach Threshold als TLB Miss klassifiziert	1.797.650 / 44,95 %	1.605.505 / 40,21 %	192.145 Hits / 11,97 %
	TLB Misses nach Ähnlichkeitsklassifikation	1.410.661 / 35,27 %	613.885 / 15,38 %	796.776 Hits / 129,79 %
Gerät 5 *i5-4440*	Gesamt gewertete Messungen nach Kontextwechsel	3.999.945 (4.000.000)	3.999.939 (4.000.000)	-
	Durchschnitt TLB Hit	13 Zykel	13 Zykel	0 Zykel / 0,00 %
	Durchschnitt TLB Miss	29 Zykel	31 Zykel	-2 Zykel / -6,45 %
	Durchschnitt Messung nach Kontextwechsel	23 Zykel	18 Zykel	5 Zykel / 27,78 %
	Threshold	20 Zykel	20 Zykel	0 Zykel / 0,00 %
	Nach Threshold als TLB Miss klassifiziert	1.919.931 / 48,00 %	973.561 / 24,34 %	946.370 Hits / 97,21 %
	TLB Misses nach Ähnlichkeitsklassifikation	1.282.901 / 32,07 %	1.492.951 / 37,32 %	-210.050 Hits / -14,07 %
Gerät 6 *Core 2 Duo (ohne PCIDs)*	Gesamt gewertete Messungen nach Kontextwechsel	3.999.913 (4.000.000)	3.999.923 (4.000.000)	-
	Durchschnitt TLB Hit	15 Zykel	13 Zykel	2 Zykel / 15,38 %
	Durchschnitt TLB Miss	24 Zykel	23 Zykel	1 Zykel / 4,35 %
	Durchschnitt Messung nach Kontextwechsel	28 Zykel	17 Zykel	11 Zykel / 64,71 %
	Threshold	19 Zykel	19 Zykel	0 Zykel / 0,00 %
	Nach Threshold als TLB Miss klassifiziert	3.616.524 / 90,42 %	925.280 / 23,13 %	2.691.244 Hits / 290,86 %
	TLB Misses nach Ähnlichkeitsklassifikation	3.356.203 / 83,91 %	1.001.333 / 25,03 %	2.354.870 Hits / 235,17 %

Abbildung 16: *Tabellarische Auflistung der Messergebnisse zur TLB Belegung auf allen Testgeräten*

Eine Klassifikation nach Ähnlichkeit ist dann besonders geeignet, wenn starke Korrelationen zwischen den eigentlichen Messwerten und den Messungen für TLB Hits bzw. Misses besteht. Dies ist beispielsweise bei Testgerät 6 teilweise der Fall, wie in Abbildung 18 erkennbar ist. Wird die am häufigsten benötigte Zeit bei der Messung der Zugriffszeit nach einem Kontextwechsel betrachtet, besteht bei der Messung ohne KPTI nahezu Deckungsgleichheit zur häufigsten benötigten Zeit bei einem TLB Hit. Mit aktivierter Kernel Page Table Isolation hingegen besteht große Ähnlichkeit zwischen der häufigsten benötigten Zeit der Messung nach einem Kontextwechsel und der häufigsten benötigten Zeit bei einem TLB Miss. Die gemessenen Performanceverluste sind nach beiden Klassifikationsmethoden auf diesem Gerät ähnlich. Nach Klassifikation nach Threshold besteht mit KPTI ein Performanceverlust von 290,86 % mit einer Verlustrate der Adressübersetzungen von 23,13 % ohne KPTI und von 90,42 % mit KPTI. Nach Ähnlichkeitsklassifikation besteht ein Performanceverlust von 235,17 % mit einer Verlustrate der Adressübersetzungen von 25,03 % ohne KPTI und von 83,91 % mit KPTI. Auch auf Gerät 2 liefern beide Klassifikationsmethoden ähnliche Ergebnisse, wie in Abbildung 16 erkennbar ist. Da die Klassifikation nach Threshold jedoch über alle Messungen hinweg zuverlässiger funktioniert, wird im Folgenden lediglich nur noch die Klassifikation nach Threshold betrachtet.

Mit einer als TLB Miss klassifizierten Rate der Messungen von über 90 % besteht auf Gerät 6 zudem die größte Verlustrate unter Verwendung von KPTI. Da der Prozessor dieses Geräts keine Process Context Identifier zur Verfügung hat, können diese hier folglich nicht verwendet werden, um implizite TLB Flushes zu minimieren. Die auf den anderen Testgeräten prozentuale Anzahl von als TLB Miss klassifizierten Messungen mit KPTI liegt zwischen 29,34 % und 48,00 %. Das heißt, weniger als die Hälfte aller Messungen werden auf Geräten, die PCIDs verwenden, mit KPTI als TLB Miss eingestuft. Auf dem Testgerät 6 ohne PCIDs hingegen werden die meisten Messungen als TLB Miss eingestuft, sodass die Verlustrate etwa doppelt bis dreimal so hoch ist wie auf den anderen Testgeräten. Es kann daraus folglich der Schluss gezogen werden, dass PCIDs auch in der Praxis tatsächlich den Verlust von Adressübersetzungen minimieren.

Ähnlich wie bei der Performanceanalyse von Systemcalls können auch bei der Analyse der TLB Belegung im Vergleich der Testgeräte architekturelle Unterschiede und Gemeinsamkeiten festgestellt werden. Die Messungen zur TLB Belegung auf Testgerät 1 mit einer Kaby Lake Architektur tendieren zu einer Gaußschen Normalverteilung (Vgl. Abbildung 17 und Abbildung 18). Auf dem Testgerät 6 mit einer Conroe Architektur gibt es bei der Messung mit und ohne KPTI jeweils eine Zugriffszeit, die bei der Messung besonders häufig benötigt wird. Die absolute Häufigkeit dieser benötigten Zykel Zeit liegt bei etwa 2.600.000. Alle anderen benötigten Zeiten für den Zugriff auf eine Adressübersetzung nach einem prozessinternen Kontextwechsel kommen auf diesem Gerät weniger als 600.000 Mal vor. Die Geräte 2 - 5 mit einer Ivy Bridge und Haswell Architektur weisen hingegen wieder Ähnlichkeiten untereinander auf. Die absolute Häufigkeit der gemessenen Zugriffszeiten teilt sich auf mehr verschiedene benötigte CPU-Zykel auf, sodass eine größere Streuung in y-Achsen Richtung entsteht (Vgl. Abbildung 17). Auf dem Testgerät 4 liegt beispielsweise die größte Häufigkeit einer benötigten Zugriffszeit bei gerade einmal etwa 900.000. Da das Gerät 5, welches eine Haswell Architektur verwendet, jedoch sowohl Ähnlichkeiten zu den Ergebnissen der Geräte 2 - 4 als auch Ähnlichkeiten zu Gerät 6 aufweist, ist eine architekturelle Abhängigkeit in Bezug zu den Messergebnissen hier nicht so deutlich.

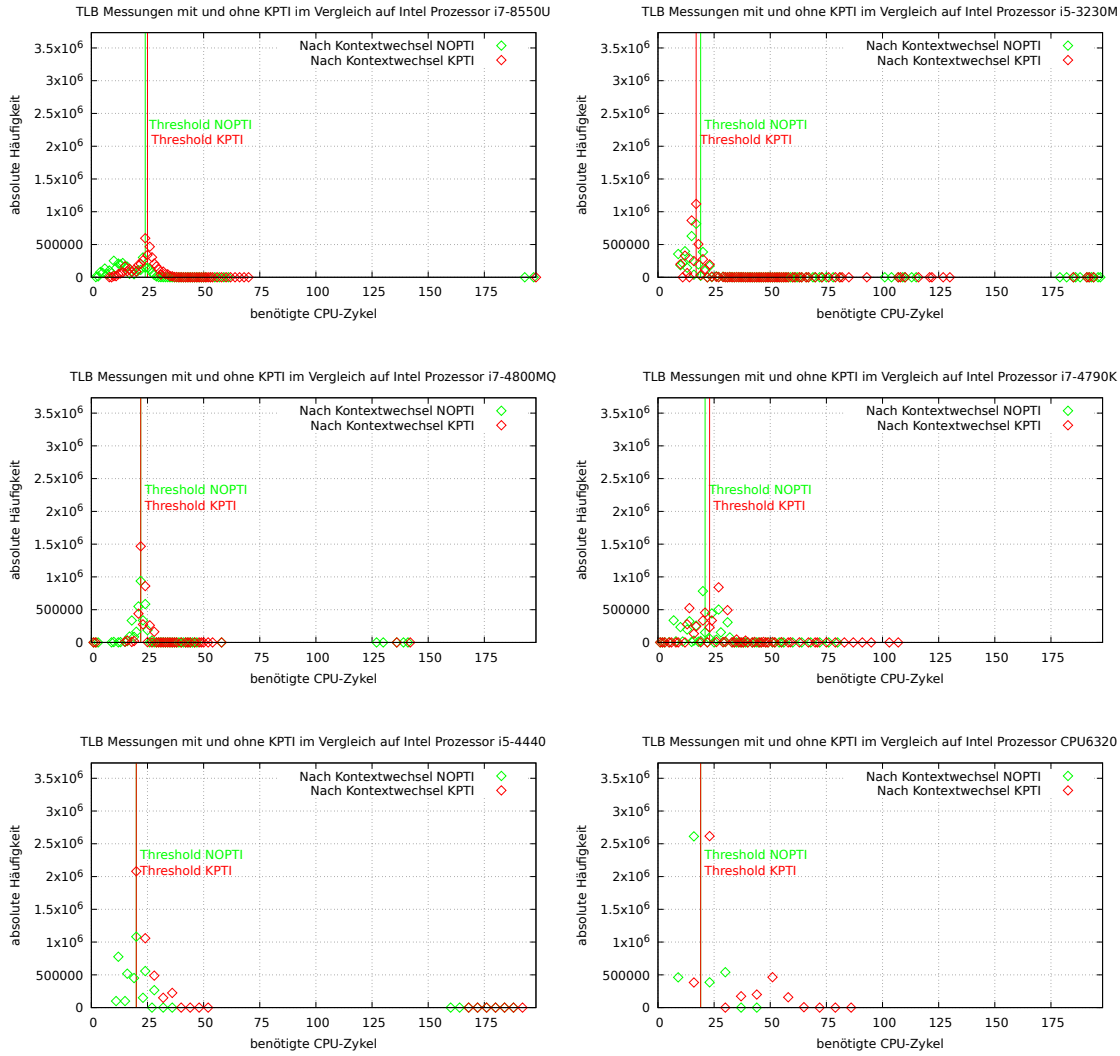


Abbildung 17: *Messungen der Zugriffszeit auf ursprünglich im TLB vorhandene Adressübersetzungen nach einem Kontextwechsel mit und ohne KPTI von allen Testgeräten*

Tabelle 3: *Performanceverluste der absoluten Anzahl von verlorenen Adressübersetzungen im Vergleich zur Gesamtanzahl der Messungen*

Gerät	Durch implizite TLB Flushes verlorene Adressübersetzungen
Gerät 1	28,26 %
Gerät 2	11,60 %
Gerät 3	10,45 %
Gerät 4	19,92 %
Gerät 5	23,66 %
Gerät 6	67,28 %

Wie bereits im Kapitel zur Performance von Systemcalls beschrieben, fallen auch bei diesen Messungen Performanceverluste bei geringeren Messwerten prozentual stärker ins Gewicht. Zur Berechnung des Performanceverlusts wird der ohne KPTI gemessene Wert als Vergleichsbasis genutzt. Beispielsweise werden auf Testgerät 2 mit KPTI 29,34 % aller Messungen und ohne KPTI

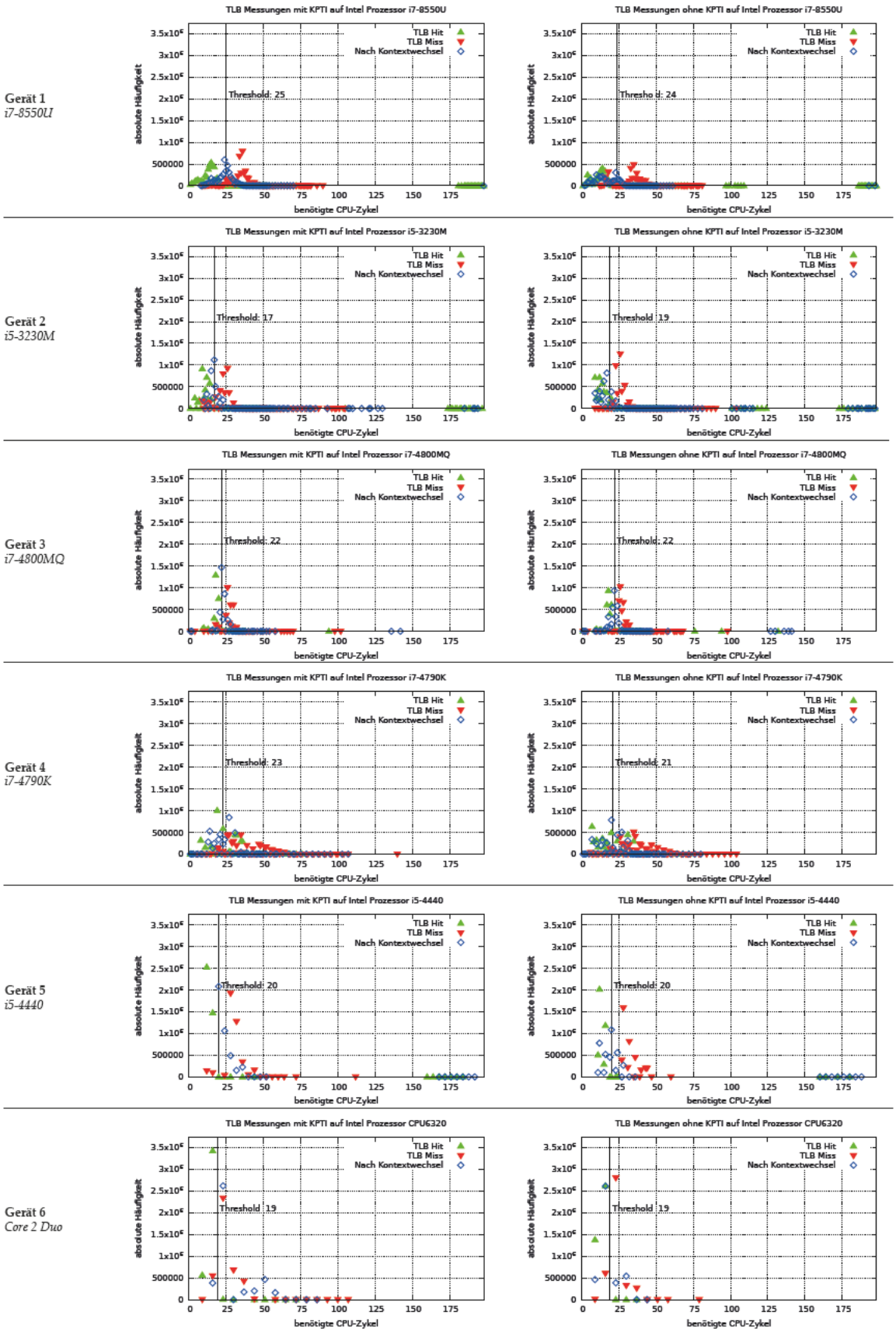


Abbildung 18: *Messungen zur TLB Belegung im Vergleich mit und ohne KPTI auf allen Testgeräten*

17,74 % aller Messungen als TLB Miss eingestuft. Dies führt im Vergleich zu einem prozentualen Performanceverlust durch KPTI von 65,41 % bei einem absoluten Verlust von 464.060 zusätzlichen TLB Misses, wenn KPTI aktiviert ist. Der absolute Overhead der als TLB Miss eingestuften Messungen ist auf Testgerät 3 nicht sehr verschieden. Dieser liegt hier bei 418.158 durch KPTI verlorenen TLB Hits. Da mit KPTI 44,30 % aller Messungen und ohne KPTI 34,29 % der Messungen als TLB Miss eingestuft werden, besteht hier lediglich ein Performanceverlust von 36,31 %. Obwohl die Differenz der absoluten Performanceverluste hier im Vergleich nur 45.902 beträgt und bei Gerät 3 somit etwa 90 % des absoluten Performanceverlusts auf Gerät 2 abgedeckt werden, besteht eine prozentuale Performanceverlustdifferenz von 29,1 %. Die prozentualen Performanceverluste durch KPTI sind relativ zu den Messwerten ohne KPTI, die als Grundwert verwendet werden. In Bezug zur Anzahl der Gesamtmessungen sind diese Performanceverluste folglich nur schwer vergleichbar. Zudem ist anzumerken, dass in einem idealen Szenario ohne KPTI nahezu keine Messungen nach einem Kontextwechsel als TLB Miss eingestuft werden sollten. Da ohne KPTI kein prozessinterner Kontextwechsel stattfindet, entstehen keine impliziten TLB Flushes. Jedoch ist auch eine Verdrängung der Testadressen durch die Adressen der aufgerufenen Systemcalls potenziell möglich. Da jedoch genaue Details über die Struktur von TLBs und die verwendeten Verdrängungsstrategien unbekannt sind, können bei der Umsetzung der Messung der TLB Belegung einige möglicherweise relevante Details nicht berücksichtigt werden. Weil die Anzahl an TLB Misses bei Messungen ohne KPTI also nicht auf implizite TLB Flushes zurückzuführen ist, kann angenommen werden, dass diese Anzahl auch für Messungen mit KPTI auf andere Ursachen als implizite TLB Flushes zurückzuführen ist. Folglich repräsentiert die Differenz der Anzahl an TLB Misses mit und ohne KPTI die Anzahl von TLB Misses, die sich durch verlorene Adressübersetzungen durch implizite TLB Flushes ergeben. Wird der absolute Verlust von Adressübersetzungen in Bezug zu der Gesamtmenge der Messungen gesetzt, ergeben sich vergleichbarere Ergebnisse. Diese können der Tabelle 3 entnommen werden. Nach den in Tabelle 3 berechneten Performanceverlusten wird im Durchschnitt auf allen Geräten, die PCIDs verwenden, ein Performanceverlust von 18,78 % gemessen.

Zusammenfassung und allgemeine Schlüsse

Alle Messungen zur Performanceanalyse zeigen deutliche Performanceverluste mit aktivierter Kernel Page Table Isolation. Diese übersteigen die in anderen Quellen angegebenen Gesamtperformanceverluste von unter 1 % bzw. etwa 5 % bei weitem (Vgl. Abschnitt 6.1). Da diese Gesamtperformanceverluste inhaltlich meist nicht genauer spezifiziert werden und somit beispielsweise unklar ist, welche Rolle Systemcalls oder Auswirkungen von Kontextwechseln einnehmen, sind die Resultate dieser Arbeit mit solchen Angaben zur Gesamtperformance jedoch nur schwer vergleichbar und eher als eine Ergänzung zu sehen.

Durch KPTI verursachte Performanceeinbußen bei der Ausführung von Systemcalls belaufen sich im Durchschnitt über alle Testgeräte auf 59,98 %. Das bedeutet, dass unter KPTI die Ausführungszeit von Systemcalls im Schnitt mehr als die Hälfte der Zeit, die bei diesem Systemcall im Durchschnitt ohne KPTI benötigt wird, länger dauert. Am erstaunlichsten ist, dass bei dem Testgerät mit dem neusten Intel-Prozessor (Kaby Lake, 8. Generation) die größten Performanceeinbußen gemessen werden. Diese übersteigen sogar die Performanceverluste durch KPTI auf dem Gerät mit dem ältesten Prozessor, einer Intel Core 2 Duo CPU, bei der die höchsten Performanceverluste

erwartet worden sind. Auch die Performanceverluste von ca. 40 - 55 %, die auf den restlichen Testgeräten gemessen werden, übersteigen die Erwartungen leicht.

Außerdem wird beobachtet, dass es pro Gerät einen Basis-Performanceverlust gibt, welcher bei jedem Systemcall mindestens an zusätzlicher Laufzeit benötigt wird. Die durch KPTI veränderte Struktur des Kernel-Mappings wirkt sich in ihrer Grundfunktionalität auf alle Systemcalls aus, sodass diese Beobachtung zu den Erwartungen passt. Die Anzahl und Art von Kernel-Inhalten, auf die beim Aufruf von Systemcalls zugegriffen wird, unterscheiden sich selbstverständlich von Systemcall zu Systemcall. Demnach sind in Abhängigkeit des Systemcalls durch KPTI auch verschieden starke Auswirkungen auf die Performance erwartet worden. Diese Erwartungen haben sich bestätigt. Aufgrund des Basis-Performanceverlusts sind zudem Systemcalls mit geringeren Laufzeiten prozentual stärker von Performanceverlusten durch KPTI betroffen und umgekehrt.

Es wird festgestellt, dass sich die Performance innerhalb eng verwandter Prozessorarchitekturen ähnlich verhält und zwischen verschiedenen Architekturen größere Unterschiede bestehen. Es ist lediglich erwartet worden, dass im Bezug zur Prozessorarchitektur die Verfügbarkeit von PCIDs auffällige Unterschiede in der Performance von KPTI verursacht. Da aktuelle Literatur kaum bis gar nicht auf architekturelle Unterschiede verschiedener Intel-Prozessoren im Bezug zur Performance von KPTI eingeht, ist diese Beobachtung somit eine überraschende Neuheit.

Die Größe des Speichers bei der Nutzung der Systemcalls mmap und munmap wirkt sich nicht auf die Performance aus. Dies passt nicht zu den Erwartungen, dass mit steigender Größe des zu allozierenden Speichers auch der Performanceverlust steigt. Die Verwaltung von Speicherbereichen in zwei durch KPTI erzeugten virtuellen Adressräumen, ist also unabhängig von der Speichergröße.

Auch bei der Analyse der TLB Belegung werden deutliche Performanceeinbußen durch KPTI beobachtet. Im Durchschnitt gehen auf Geräten mit PCIDs 18,78 % an Adressübersetzungen durch implizite TLB Flushes bei prozessinternen Kontextwechseln verloren. Auf dem ältesten Testgerät, auf dem keine Process Context Identifier zur Verfügung stehen, gehen 67,28 % an Adressübersetzungen durch einen prozessinternen Kontextwechsel verloren. Dass PCIDs die Performance von KPTI verbessern und implizite TLB Flushes minimieren, kann wie erwartet also bestätigt werden.

Wie stark Nutzeranwendungen von Performanceverlusten durch KPTI betroffen sind, hängt schlussendlich vom Gebrauch von Systemcalls und der Häufigkeit prozessinterner Kontextwechsel ab. Anwendungen, die besonders viele Systemcalls verwenden, sind also stärker von Performanceeinbußen durch KPTI betroffen, als Anwendungen die weniger Systemcalls verwenden. Auch die Art bzw. Laufzeit des jeweiligen Systemcalls ist entscheidend. Systemcalls, die weniger Laufzeit benötigen, sind prozentual stärker durch KPTI-Performanceeinbußen belastet. Auch bei Anwendungen, die vermehrt prozessinterne Kontextwechsel durchführen, werden durch KPTI aufgrund von verlorenen Adressübersetzungen im TLB größere Performanceeinbußen erwartet. Verwendet eine Anwendung also beispielsweise viele Systemcalls, die zu prozessinternen Kontextwechseln führen, muss bei aktivierter Kernel Page Table Isolation auch vermehrt der zeitintensive Page Walk erneut durchlaufen werden, um die durch implizite TLB Flushes verlorenen Übersetzungen zu rekonstruieren. Dass mit höheren Systemcall Raten und vermehrten Kontextwechseln auch höhere Performanceoverheads durch KPTI gemessen werden, passt auch zu den auf Brendan Greggs Blog vorgestellten Ergebnissen [Gre]. Genaue zu erwartende Performanceverluste durch KPTI lassen sich folglich nur in Zusammenhang einer vorliegenden Anwendung qualifiziert prognostizieren.

7 Ausblick

In dieser Arbeit wird nur ein kleiner Teil der Auswirkungen durch KPTI untersucht. Beispielsweise werden nur einige wenige Systemcalls ausgewählt, anhand derer die Performance von KPTI analysiert wird. Zudem werden die Performanceverluste durch KPTI nur auf sechs Testgeräten mit nur 4 verschiedennamigen Intel-Prozessorarchitekturen untersucht. Auch auf Geräten mit Prozessoren von anderen Herstellern können Verluste durch KPTI möglich sein. Folglich kann diese Arbeit noch erweitert und fortgeführt werden.

Es wird beobachtet, dass Performanceeinbußen durch KPTI in Abhängigkeit zur Prozessorarchitektur korrelieren bzw. variieren. Drei der sechs Testgeräte verwenden eine Haswell Architektur und eins der Geräte verwendet eine Ivy Bridge Prozessorarchitektur, den Vorgänger der Haswell Architektur. Zwischen diesen Geräten werden große Gemeinsamkeiten in der Performance von KPTI festgestellt. Eine weitere Analyse der Auswirkungen von KPTI mit besonderem Hinblick auf Gemeinsamkeiten eng verwandter Prozessorarchitekturen und Unterschieden zu verschiedenen Prozessorarchitekturen kann daher weitere spannende Erkenntnisse liefern. Insbesondere ist durch eine solche weitere Analyse eine differenziertere Aussage über die zu erwartenden Performanceverluste je nach Prozessorarchitektur möglich.

Die Auswirkung der Verwendung von Process Context Identifiern wird nur auf einem einzigen und zudem sehr alten Intel-Prozessor durchgeführt. Die Nutzung von PCIDs kann auch auf modernen Prozessoren, die PCIDs zur Verfügung haben, deaktiviert werden. Somit kann im Zusammenhang der Verfügbarkeit von PCIDs die Performance von KPTI in weiterführenden Arbeiten ebenfalls näher beleuchtet werden.

Alle durchgeführten Tests werden auf einem Ubuntu 20.04 Betriebssystem durchgeführt. Möglicherweise verhalten sich Performanceverluste durch KPTI auf anderen Linux Distributionen verschieden. Auch haben andere Betriebssystementwickler Teile des KAISER Konzepts zur Verhinderung von Meltdown-Angriffen aufgenommen. Daher ist auch eine nähergehende Untersuchung der Performance von softwarebasierten Meltdown-Mitigations auf verschiedenen Betriebssystemen sicherlich erkenntnisreich. Genauso kann eine genauere Untersuchung verschiedener Linux-Kernel-Versionen spannend sein.

Die Entwicklung eines Analysetools zur Messung der TLB Belegung ist nicht trivial, da relevante Informationen zu den Details der Struktur von TLBs fehlen. Es ist nicht bekannt, wie Adressübersetzungen im TLB abgebildet werden, nach welchem Schema diese verdrängt werden und welche Informationen in den verschiedenen TLB Leveln enthalten sind, also z. B. ob der Last-Level TLB inklusiv ist. Eine dahingehende weitere Analyse kann die Entwicklung von Analysewerkzeugen zur Performance des TLBs folglich erheblich erleichtern und weitere interessante Einblicke liefern. Zudem ist Messergebnissen, die sämtliche Feinheiten der internen Struktur von TLBs berücksichtigen

und somit eine genauere Analyse erlauben, mehr Bedeutung zuzumessen. Es ist also möglich, dass mit mehr Informationen über TLBs auch akkuratere Messungen der TLB Belegung durchgeführt werden können. Auch eine qualitative Ergänzung dieser Arbeit ist somit denkbar.

Die Analyse der TLB Belegung wird anhand eines seitenkanalähnlichen Aufbaus durchgeführt, da somit keine höheren Berechtigungsstufen erforderlich sind. Eine Analyse des Verlusts von Adressübersetzungen durch KPTI bei prozessinternen Kontextwechseln kann auch anhand von Hardwareperformancecountern durchgeführt werden. Diese benötigen jedoch höhere Berechtigungsstufen und passen daher nicht in das hier gewählte realitätsnahe Szenario eines unprivilegierten Nutzers. Dennoch können auch Messungen anhand dieser Methode weitere interessante Erkenntnisse liefern.

8 Fazit

In dieser Arbeit wird der Performanceverlust durch die Meltdown-Mitigation Kernel Page Table Isolation unter realitätsnahen Bedingungen auf Intel-Prozessoren analysiert und evaluiert. Insbesondere werden die Hauptursachen für Performanceverluste durch KPTI identifiziert und Auswirkungen dieser auf die Performance im Detail betrachtet. In der Literatur werden bislang meist nur Angaben zur Gesamtperformance von KPTI gemacht, wodurch eine genaue Prognose der durch KPTI zu erwartenden Performanceverluste für spezifische Anwendungsszenarien nahezu unmöglich ist.

Diese Arbeit zeigt unter anderem zu erwartende KPTI-Performanceverluste bei der Nutzung verschiedener Systemcalls auf und zeigt, welcher Einfluss auf die Verfügbarkeit von Adressübersetzungen im TLB von prozessinternen Kontextwechseln ausgeübt wird. Dabei wird festgestellt, dass in den relevanten Kernpunkten deutlich höhere Performanceverluste durch KPTI auf Intel-Prozessoren entstehen, als sie durch Angaben zur Gesamtperformance in der Literatur genannt werden. Zudem wird beobachtet, dass Performanceeinbußen durch KPTI je nach Prozessorarchitektur variieren und innerhalb eng verwandter CPUs große Gemeinsamkeiten bestehen. Es ist also ein detaillierter Einblick in die Performance der Meltdown-Mitigation KPTI hervorgegangen.

Eine weitere Errungenschaft dieser Arbeit ist das mitgelieferte Analyseprogramm, anhand dessen weitere Analysen der Performance von KPTI auf Geräten mit Intel-Prozessoren komfortabel durchgeführt werden können und automatisiert anschaulich aufbereitet werden. Die hier vorgestellten Forschungsergebnisse können folglich reproduziert werden und durch weitere Analysen in eigenen Anwendungsszenarien fortgeführt werden.

Das Ziel dieser Bachelorarbeit, Performanceeinbußen durch KPTI in den relevanten Kernpunkten im Detail zu analysieren und zu evaluieren, wird somit vollständig umgesetzt. KPTI bietet als Software-Mitigation auch heutzutage noch einen unverzichtbaren Bestandteil zum Schutz vor Meltdown-Angriffen und ist in jedem aktuellen Linux-Kernel enthalten. Das Analysetool und die hier vorgestellten Forschungsergebnisse können es folglich erleichtern, eine fundierte Entscheidung darüber zu treffen, ob in performancekritischen Infrastrukturen die Verwendung von KPTI deaktiviert werden sollte (sofern das dadurch entstehende Sicherheitsrisiko in Kauf genommen werden kann) oder ob die Performanceverluste vertretbar sind. Auch eine Optimierung bestehender Software, dahingehend, dass diese weniger von KPTI-Performanceverlusten betroffen sind, wird durch die vorliegenden Forschungsergebnisse erleichtert. Somit leistet diese Arbeit einen wichtigen Beitrag zur Einschätzung von zu erwartenden Performanceeinbußen durch diese Meltdown-Mitigation und ermöglicht zudem eine an das spezifische Anwendungsszenario angepasste Prognose der Performanceverluste. Dadurch wird eine bestehende Lücke in der Literatur geschlossen und es werden weitere Wege zur Ergänzung der Literatur aufgezeigt.

Literaturverzeichnis

[ARM20] ARM: *Arm System Memory Management Unit Architecture Specification, SMMU architecture version 3*. https://documentation-service.arm.com/static/5f901081f86e16515cdc0919. Zugriff am 26.02.2021. 2020.

[ARM21] ARM: *Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism*. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability. Zugriff am 26.02.2021. 2021.

[Bar10] Barr, Thomas W. und Cox, Alan L. und Rixner, Scott: „Translation caching: skip, don't walk (the page table)". In: *ACM SIGARCH Computer Architecture News* 38.3 (2010), S. 48–59.

[Ber05] Bernstein, Daniel J.: *Cache-timing attacks on AES*. Department of Mathematics, Statistics, und Computer Science, University of Illinois at Chicago, 2005.

[Ble18] Bleepingcomputer: Lawrence Abrams: *Intel Releases Linux CPU Microcodes To fix Meltdown & Spectre Bugs*. https://www.bleepingcomputer.com/news/security/intel-releases-linux-cpu-microcodes-to-fix-meltdown-and-spectre-bugs. Zugriff am 26.02.2021. 2018.

[Bri18] Brian Krzanich, CEO Intel Corporation: *Advancing Security at the Silicon Level*. https://newsroom.intel.com/editorials/advancing-security-silicon-level/. Zugriff am 30.12.2020. 2018.

[Can19a] Canella, Claudio und Genkin, Daniel und Giner, Lukas und Gruss, Daniel und Lipp, Moritz und Minkin, Marina und Moghimi, Daniel und Piessens, Frank und Schwarz, Michael und Sunar, Berk und andere: „Fallout: Leaking data on meltdown-resistant cpus". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, S. 769–784.

[Can19b] Canella, Claudio und Van Bulck, Jo und Schwarz, Michael und Lipp, Moritz und Von Berg, Benjamin und Ortner, Philipp und Piessens, Frank und Evtyushkin, Dmitry und Gruss, Daniel: „A systematic evaluation of transient execution attacks and defenses". In: *28th USENIX Security Symposium*. 2019, S. 249–266.

[Can19c] Canella, Claudio und Van Bulck, Jo und Schwarz, Michael und Lipp, Moritz und Von Berg, Benjamin und Ortner, Philipp und Piessens, Frank und Evtyushkin, Dmitry und Gruss, Daniel: „A systematic evaluation of transient execution attacks and defenses". In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, S. 249–266.

[Can20] Canella, Claudio und Schwarz, Michael und Haubenwallner, Martin und Schwarzl, Martin und Gruss, Daniel: „KASLR: Break It, Fix It, Repeat“. In: *ASIA CCS 2020-Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ACM/IEEE. 2020.

[Com18a] Common Vulnerabilities and Exposures: *CVE-2017-5715*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5715. Zugriff am 26.02.2021. 2018.

[Com18b] Common Vulnerabilities and Exposures: *CVE-2017-5753*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5753. Zugriff am 26.02.2021. 2018.

[Com18c] Common Vulnerabilities and Exposures: *CVE-2017-5754*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754. Zugriff am 26.02.2021. 2018.

[Com18d] Common Vulnerabilities and Exposures: *CVE-2018-12126*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12126. Zugriff am 26.02.2021. 2018.

[Com18e] Common Vulnerabilities and Exposures: *CVE-2018-12127*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12127. Zugriff am 26.02.2021. 2018.

[Com18f] Common Vulnerabilities and Exposures: *CVE-2018-3615*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3615. Zugriff am 26.02.2021. 2018.

[Com18g] Common Vulnerabilities and Exposures: *CVE-2019-0162*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-0162. Zugriff am 26.02.2021. 2018.

[Com18h] Common Vulnerabilities and Exposures: *CVE-2019-11091*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11091. Zugriff am 26.02.2021. 2018.

[Dav17] Dave Hansen: *[PATCH 00/30] [v3] KAISER: unmap most of the kernel from userspace page tables*. https://lwn.net/Articles/738997/. Zugriff am 29.12.2020. 2017.

[Dav18] Dave Hansen: *Use global pages with PTI*. https://lwn.net/Articles/750049/. Zugriff am 28.12.2020. 2018.

[Det15] Detter, John und Mutschlechner, Riccardo: *Performance and Entropy of Various ASLR Implementations*. 2015.

[die] die.net: *sched_yield(2) - Linux man page*. https://linux.die.net/man/2/sched_yield. Zugriff am 15.04.2021.

[GNU] GNU: *Getting the Time (The GNU C Library)*. https://www.gnu.org/software/libc/manual/html_node/Getting-the-Time.html. Zugriff am 08.04.2021.

[Goo18] Google Project Zero: Jann Horn: *Reading privileged memory with a side-channel*. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html. Zugriff am 26.02.2021. 2018.

[Gra+18] Gras, Ben u. a.: „Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks“. In: *27th USENIX Security Symposium USENIX Security 18*. 2018, S. 955–972.

[Gra17] Gras, Ben und Razavi, Kaveh und Bosman, Erik und Bos, Herbert und Giuffrida, Cristiano: „ASLR on the Line: Practical Cache Attacks on the MMU.“ In: *NDSS*. Bd. 17. 2017, S. 26.

[Gre] Gregg, Brendan: *KPTI/KAISER Meltdown Initial Performance Regressions*. http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html. Zugriff am 26.04.2021.

[Gru16a] Gruss, Daniel und Maurice, Clémentine und Fogh, Anders und Lipp, Moritz und Mangard, Stefan: „Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, S. 368–379.

[Gru16b] Gruss, Daniel und Maurice, Clémentine und Wagner, Klaus und Mangard, Stefan: „Flush+ Flush: a fast and stealthy cache attack". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, S. 279–299.

[Gru17a] Gruss, Daniel: „Software-based microarchitectural attacks". Diss. Graz University of Technology, 2017.

[Gru17b] Gruss, Daniel und Lipp, Moritz und Schwarz, Michael und Fellner, Richard und Maurice, Clémentine und Mangard, Stefan: „Kaslr is dead: long live kaslr". In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2017, S. 161–176.

[Gru18] Gruss, Daniel und Hansen, Dave und Gregg, Brendan: „Kernel isolation: From an academic idea to an efficient patch for every computer". In: *; login: the USENIX Magazine* 43.4 (2018). USENIX Association, S. 10–14.

[Gul11] Gullasch, David und Bangerter, Endre und Krenn, Stephan: „Cache games – bringing access-based cache attacks on AES to practice". In: *2011 IEEE Symposium on Security and Privacy*. IEEE. 2011, S. 490–505.

[Har10] Harris, Tim und Larus, James und Rajwar, Ravi: „Transactional memory". In: *Synthesis Lectures on Computer Architecture* 5.1 (2010). Morgan & Claypool Publishers, S. 1–263.

[Hei21] Heidler, Sabrina: *GitHub-Repository zu dieser Arbeit*. https://github.com/SabrinaHei/Bachelorarbeit-Performanceanalyse_der_Meltdown_Mitigation_KPTI.git. Hochgeladen am 30.04.2021. 2021.

[Hu92] Hu, W-M: „Lattice scheduling and covert channels". In: *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE. 1992, S. 52–61.

[Inc16] Inci, Mehmet Sinan und Gulmezoglu, Berk und Irazoqui, Gorka und Eisenbarth, Thomas und Sunar, Berk: „Cache attacks enable bulk key recovery on the cloud". In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2016, S. 368–388.

[Int10] Intel Corporation: *How to Benchmark Code Execution Times on Intel®IA-32 and IA-64 Instruction Set Architectures*. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf. Zugriff am 06.04.2021. 2010.

[Int18] Intel Corporation: *Intel Analysis of Speculative Execution Side Channels*. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf. Zugriff am 30.12.2020. 2018.

[Int19] Intel Corporation: *Intel® 64 and IA-32 Architectures Software Developer's Manual*. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf. Zugriff am 26.02.2021. 2019.

[Int98] Intel Corporation: *Using the RDTSC Instruction for Performance Monitoring*. https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf. Zugriff am 06.04.2021. 1998.

[Ion18] Ionescu, A.: *Twitter: Apple Double Map*. https://twitter.com/aionescu/status/948609809540046849. Zugriff am 26.02.2021. 2018.

[Isl19] Islam, Saad und Moghimi, Ahmad und Bruhns, Ida und Krebbel, Moritz und Gulmezoglu, Berk und Eisenbarth, Thomas und Sunar, Berk: „SPOILER: Speculative load hazards boost rowhammer and cache attacks". In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, S. 621–637.

[Jan16] Jang, Yeongjin und Lee, Sangho und Kim, Taesoo: „Breaking kernel address space layout randomization with intel tsx". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, S. 380–392.

[Jes96] Jessen, Eike: „Die Entwicklung des virtuellen Speichers". In: *Informatik-Spektrum* 19.4 (1996). Springer, S. 216–219.

[Joh18] Johnson, K: *KVA Shadow: Mitigating Meltdown on Windows*. https://msrc-blog.microsoft.com/2018/03/23/kva-shadow-mitigating-meltdown-on-windows. Zugriff am 25.02.2021. 2018.

[Kel98] Kelsey, John und Schneier, Bruce und Wagner, David und Hall, Chris: „Side channel cryptanalysis of product ciphers". In: *European Symposium on Research in Computer Security*. Springer. 1998, S. 97–110.

[Koc19] Kocher, Paul und Horn, Jann und Fogh, Anders und Genkin, Daniel und Gruss, Daniel und Haas, Werner und Hamburg, Mike und Lipp, Moritz und Mangard, Stefan und Prescher, Thomas und andere: „Spectre attacks: Exploiting speculative execution". In: *2019 IEEE Symposium on Security and Privacy*. IEEE. 2019, S. 1–19.

[Koc96] Kocher, Paul C.: „Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems". In: *Annual International Cryptology Conference*. Springer. 1996, S. 104–113.

[Kre19] Krenn, Thomas: *Intel Microcode*. https://www.thomas-krenn.com/de/wiki/Intel_Microcode. Zugriff am 26.02.2021. 2019.

[Lef91] Lefsky, Brian und Natusch, Mary E: *High availability cache organization*. Google Patents, US Patent 5,019,971. 1991.

[Lev12] Levin, Jonathan: *Mac OS X and IOS Internals: To the Apple's Core*. John Wiley & Sons, 2012.

[Lin] Linux Kernel Documentation: *Memory Management*. https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html. Zugriff am 09.03.2021.

[Lin18a] Linus Torvalds: *E-Mail: Re: [RFC 09/10] x86/enter: Create macros to restrict/unrestrict Indirect Branch Speculation*. https://lkml.org/lkml/2018/1/21/192. Zugriff am 30.12.2020. 2018.

[Lin18b] Linux Entwickler: *Linux 4.14.11 Changelog*. https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.14.11. Zugriff am 29.12.2020. 2018.

[Lin18c] Linux Entwickler: *Linux 4.4.110 Changelog*. https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.4.110. Zugriff am 29.12.2020. 2018.

[Lin18d] Linux Entwickler: *Linux 4.9.75 Changelog*. https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.9.75. Zugriff am 29.12.2020. 2018.

[Lip16] Lipp, Moritz und Gruss, Daniel und Spreitzer, Raphael und Maurice, Clémentine und Mangard, Stefan: „Armageddon: Cache attacks on mobile devices“. In: *25th USENIX Security Symposium*. 2016, S. 549–564.

[Lip18] Lipp, Moritz und Schwarz, Michael und Gruss, Daniel und Prescher, Thomas und Haas, Werner und Mangard, Stefan und Kocher, Paul und Genkin, Daniel und Yarom, Yuval und Hamburg, Mike: „Meltdown“. In: *arXiv preprint arXiv:1801.01207* (2018).

[LWN13] LWN.net: *Kernel address space layout randomization*. https://lwn.net/Articles/569635/. Zugriff am 23.02.2021. 2013.

[LWN17a] LWN.net: *Kernel page-table isolation merged*. https://lwn.net/Articles/742404/. Zugriff am 28.12.2020. 2017.

[LWN17b] LWN.net: *The current state of kernel page-table isolation*. https://lwn.net/Articles/741878/. Zugriff am 15.04.2021. 2017.

[Mau15] Maurice, Clémentine und Le Scouarnec, Nicolas und Neumann, Christoph und Heen, Olivier und Francillon, Aurélien: „Reverse engineering Intel last-level cache complex addressing using performance counters“. In: *International Symposium on Recent Advances in Intrusion Detection*. Springer. 2015, S. 48–65.

[Mau17] Maurice, Clémentine und Weber, Manuel und Schwarz, Michael und Giner, Lukas und Gruss, Daniel und Boano, Carlo Alberto und Mangard, Stefan und Römer, Kay: „Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud.“ In: *NDSS*. Bd. 17. 2017, S. 8–11.

[Mic] Microsoft: *Virtual Address Space (Memory Management)*. https://docs.microsoft.com/de-de/windows/win32/memory/virtual-address-space. Zugriff am 03.03.2021.

[Nul14] Null, Linda und Lobur, Julia: *The Essentials of Computer Organization and Architecture*. 4th. USA: Jones und Bartlett Publishers, Inc., 2014. Kap. 6.

[Osv06] Osvik, Dag Arne und Shamir, Adi und Tromer, Eran: „Cache attacks and countermeasures: the case of AES“. In: *Cryptographers' track at the RSA conference*. Springer. 2006, S. 1–20.

[Pag02] Page, Dan: „Theoretical use of cache memory as a cryptanalytic side-channel.“ In: *IACR Cryptology ePrint Archive* 2002.169 (2002). Citeseer.

[Pat16] Patterson David A und Hennessy, John L: *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan kaufmann, 2016.

[PC] PC Perspective: *IDF 2012: Intel Haswell Architecture Revealed*. https://pcper.com/2012/09/idf-2012-intel-haswell-architecture-revealed/. Zugriff am 23.04.2021.

[Per05] Percival, Colin: *Cache missing for fun and profit*. BSDCan, 2005.

[Pro20] Prof. Dr. Michael Meier und Dr. rer. nat. Felix Jonathan Boes: *Vorlesung Reaktive Sicherheit der Universität Bonn*. https://net.cs.uni-bonn.de/fileadmin/ag/itsec/lehre/20ss/Reaktive_Sicherheit/Folien/ReSi20-3-ProgrammUndWebVerwundbarkeiten-Teil2.pdf. Zugriff am 14.12.2020. Rheinische Friedrich-Wilhelms-Universität Bonn, Institute of Computer Science 4, Security und Networked Systems, Arbeitsgruppe IT-Sicherheit, 2020.

[Red] Red Hat: *Speculative Execution Exploit Performance Impacts - Describing the performance impacts to security patches for CVE-2017-5754 CVE-2017-5753 and CVE-2017-5715*. https://access.redhat.com/articles/3307751. Zugriff am 25.04.2021.

[Ris09] Ristenpart, Thomas und Tromer, Eran und Shacham, Hovav und Savage, Stefan: „Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds“. In: *Proceedings of the 16th ACM conference on Computer and communications security*. 2009, S. 199–212.

[Rya18] Ryan Smith on AnandTech.com: *Intel Publishes Spectre and Meltdown Hardware Plans: Fixed Gear Later This Year*. https://www.anandtech.com/show/12533/intel-spectre-meltdown. Zugriff am 30.12.2020. 2018.

[Sch19a] Schwarz, Michael und Canella, Claudio und Giner, Lukas und Gruss, Daniel: „Store-to-leak forwarding: Leaking data on meltdown-resistant cpus“. In: *arXiv preprint arXiv:1905.05725* (2019).

[Sch19b] Schwarz, Michael und Lipp, Moritz und Moghimi, Daniel und Van Bulck, Jo und Stecklina, Julian und Prescher, Thomas und Gruss, Daniel: „ZombieLoad: Cross-privilege-boundary data sampling“. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, S. 753–768.

[Sta13] Stanley, Dannie M und Xu, Dongyan und Spafford, Eugene H: „Improved kernel security through memory layout randomization“. In: *2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC)*. IEEE. 2013, S. 1–10.

[Tan15] Tanenbaum, Andrew S und Bos, Herbert: *Modern operating systems*. Pearson, 2015.

[Tel15] Tello, Brady und Winterrose, Michael L und Baah, George K und Zhivich, Michael: „Simulation based Evaluation of a Code Diversification Strategy.“ In: *SIMULTECH*. 2015, S. 36–43.

[Tor18] Torvalds, Linus: *A couple of urgent fixes for PTI*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=00a5ae218d57741088068799b810416ac249a9ce&utm_source=anz. Zugriff am 26.02.2021. 2018.

[Tsu03] Tsunoo, Yukiyasu und Saito, Teruo und Suzaki, Tomoyasu und Shigeri, Maki und Miyauchi, Hiroshi: „Cryptanalysis of DES implemented on computers with cache“. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2003, S. 62–76.

[Van18] Van Bulck, Jo und Minkin, Marina und Weisse, Ofir und Genkin, Daniel und Kasikci, Baris und Piessens, Frank und Silberstein, Mark und Wenisch, Thomas F und Yarom, Yuval und Strackx, Raoul: „Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, S. 991–1008.

[Van19] Van Schaik, Stephan und Milburn, Alyssa und Österlund, Sebastian und Frigo, Pietro und Maisuradze, Giorgi und Razavi, Kaveh und Bos, Herbert und Giuffrida, Cristiano: „RIDL: Rogue in-flight data load". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, S. 88–105.

[Yar14] Yarom, Yuval und Falkner, Katrina: „FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack". In: *23rd USENIX Security Symposium*. 2014, S. 719–732.

[Yar15] Yarom, Yuval und Ge, Qian und Liu, Fangfei und Lee, Ruby B und Heiser, Gernot: „Mapping the Intel Last-Level Cache." In: *IACR Cryptology ePrint Archive* 2015 (2015), S. 905.