Funktionsweise und Evaluation von modernen Spectre-Angriffen

Am Beispiel von RIDL, Fallout und ZombieLoad

Bachelorarbeit

ausgearbeitet von

Jan-Niklas Sohn

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. Karl Jonas Hochschule Bonn-Rhein-Sieg
Betreuer:	Dr. Felix Jonathan Boes Universität Bonn

Bonn, 19. April 2021

Kurzfassung

Die im Jahre 2018 vorgestellten Sicherheitslücken Spectre und Meltdown haben die IT-Sicherheit nachhaltig beeinflusst. Das neu begründete Forschungsfeld der Sicherheit spekulativer Ausführung umfasst mittlerweile viele Varianten der ursprünglich entdeckten Sicherheitslücken. Dazu gehören unter anderem RIDL, ZombieLoad, Write Transient Forwarding und Store-to-Leak. Write Transient Forwarding und Store-to-Leak werden zusammen auch als Fallout bezeichnet. Mithilfe dieser Sicherheitslücken können Daten aus verschiedenen Puffern der Mikroarchitektur extrahiert werden. Dies hat weitreichende Konsequenzen für Sicherheitsmodelle mit gemeinsamer Hardware, beispielsweise im Cloud Computing. In der vorliegenden Arbeit wird zunächst die Funktionsweise dieser Sicherheitslücken erläutert. Anschließend werden Angriffe auf diese in verschiedenen Varianten implementiert. Die implementierten Angriffe können von unprivilegierten Nutzerprozessen auf Linux-Systemen eingesetzt werden, um Daten anderer Prozesse oder des Betriebssystem-Kerns zu extrahieren. Schließlich werden die implementierten Angriffe hinsichtlich einheitlicher Metriken evaluiert. Bei dieser Evaluation zeigen drei von den vier betrachteten Angriffen die erwarteten Ergebnisse in den meisten ihrer Varianten. RIDL, ZombieLoad und Store-to-Leak extrahieren erfolgreich anvisierte Daten. Durch Write Transient Forwarding hingegen können in keiner der evaluierten Varianten die anvisierten Daten erfolgreich extrahiert werden.

Inhaltsverzeichnis

1 Einleitung

Moderne Prozessoren verwenden eine Optimierung, die das Ziel bedingter Sprünge vorhersagt. Dabei wird der vorhergesagte Ausführungspfad spekulativ ausgeführt, bis das tatsächliche Ziel des Sprunges feststeht. Im Zusammenhang mit dieser spekulativen Ausführung wurden Anfang des Jahres 2018 die als Spectre und Meltdown bekannten Sicherheitslücken veröffentlicht. Spectre erlaubt einem Angreifer, im Kontext des Opfers auftretende spekulative Ausführung zu beeinflussen und dadurch Daten aus diesem zu extrahieren. Meltdown ermöglicht Nutzerprozessen sämtliche Daten des Betriebssystem-Kerns auszulesen. Durch die Entdeckung dieser Sicherheitslücken ist mit dem Gebiet der Sicherheit spekulativer Ausführung ein neuer, aktiver Forschungszweig entstanden. Mittlerweile existieren viele verschiedene Varianten der ursprünglich entdeckten Sicherheitslücken. Angriffe auf diese werden kollektiv auch als Spectre-Angriffe bezeichnet. Viele Spectre-Angriffe können beispielsweise dazu genutzt werden, auf lokalen Systemen Passwörter anderer Nutzer oder in Cloud-Szenarien Daten einer benachbarten virtuellen Maschine zu extrahieren. Aus diesem Grund sind Spectre-Angriffe von großer Bedeutung für die Sicherheit einer Vielzahl von Systemen. Für Spectre-Angriffe wurden verschiedene Gegenmaßnahmen entwickelt, sowohl in Hardware als auch in Software. Oft sind diese nur gegen einen Teil der Angriffe wirksam und mit Einbußen in der Performance verbunden. Daher sind Gegenmaßnahmen, genauso wie Spectre-Angriffe selber, ein aktiver Gegenstand der Forschung.

Das Ziel dieser Arbeit besteht zunächst darin, die Funktionsweise ausgewählter Spectre-Angriffe zu erklären. Anschließend werden verschiedene Varianten dieser Angriffe implementiert und hinsichtlich einheitlicher Metriken evaluiert. Schließlich werden die Ergebnisse dieser Evaluation mit Ergebnissen der Literatur verglichen und in einen weiteren Kontext gesetzt. Als konkrete Spectre-Angriffe werden RIDL, ZombieLoad, Write Transient Forwarding und Store-to-Leak gewählt. Write Transient Forwarding und Store-to-Leak werden zusammen auch als Fallout bezeichnet.

Zunächst werden in Kapitel 2 einige bekannte Spectre-Angriffe und damit der aktuelle Stand der Forschung dargestellt. In Kapitel 3 werden die Grundlagen der Funktionsweise moderner Prozessoren behandelt, die für ein Verständnis von Spectre-Angriffen benötigt werden. Dies umfasst die Funktionsweise von Prozessor-Caches (Kapitel 3.2) und den Aufbau der Mikroarchitektur moderner Intel-Prozessoren (Kapitel 3.3). Anschließend werden in Kapitel 4.1 Flush+Reload und Flush+Flush als Cache-basierte Seitenkanalangriffe beschrieben. Diese bilden einen wichtigen Teil der Implementierung von Spectre-Angriffen. Danach wird in Kapitel 4.2 die Methodik von Spectre-Angriffen im Allgemeinen dargestellt. Diese Angriffe werden zunächst in zwei Kategorien unterteilt: Spectre-Angriffe basierend auf Branch Prediction (Kapitel 4.3) und Spectre-Angriffe basierend auf Prozessor-Exceptions (Kapitel 4.4). Daraufhin wird in Kapitel 4.7 die Funktionsweise der gewählten konkreten Spectre-Angriffe erklärt und Details ihrer Implementierung beschrieben. Im Zuge der Evaluation in Kapitel 5 werden zunächst einzelne Techniken ausgewertet, die in Spectre-Angriffen Verwendung finden. Dazu gehören die beiden beschriebenen Cache-basierten Seitenkanalangrif-

fe (Kapitel 5.2) sowie eine Auswahl an Techniken zur Unterdrückung von Prozessor-Exceptions (Kapitel 5.3). Schließlich werden in Kapitel 5.4 und Kapitel 5.5 die konkreten Spectre-Angriffe selbst evaluiert. Von jedem Angriff werden dabei verschiedene Varianten nach einheitlichen Metriken ausgewertet und anschließend verglichen. In Kapitel 6 werden die Ergebnisse dieser Arbeit zusammengefasst und ein Ausblick auf weitere Forschungsmöglichkeiten gegeben.

2 Stand der Forschung

Die zuerst entdeckten Sicherheitslücken im Zusammenhang mit spekulativer Ausführung waren Spectre und Meltdown. Spectre wurde im Jahre 2018 von Kocher et al. vorgestellt und erlaubt einem Angreifer, im Kontext des Opfers auftretende spekulative Ausführung zu beeinflussen und dadurch Daten aus diesem zu extrahieren [Koc+19]. Meltdown wurde im Jahre 2018 von Lipp et al. vorgestellt und ermöglicht Nutzerprozessen, sämtliche Daten des Betriebssystem-Kerns auszulesen [Lip+18]. Mittlerweile existieren viele verschiedene Varianten dieser ursprünglich entdeckten Sicherheitslücken.

Horn beschrieb im Jahre 2018 eine Variante von Spectre, die eine inkorrekte Vorhersage von schreibenden Speicherzugriffen ausnutzt [Hor18]. Evtyushkin et al. entdeckten BranchScope im Jahre 2018 als Variante von Spectre. BranchScope beeinflusst die Vorhersage bedingter Sprünge, um den Ausgang von bedingten Sprüngen des Opfers zu beobachten [Evt+18]. Maisuradze und Rossow, sowie Koruyeh et al. stoßen im Jahre 2018 unabhängig voneinander auf eine Variante von Spectre, die spekulative Ausführung nach Rücksprüngen aus Funktionen ausnutzt [MR18] [Kor+18]. Chen et al. stellten im Jahre 2019 SgxPectre vor. Dabei handelt es sich um eine Spectre-Variante, die Daten aus einer Intel SGX Enklave extrahiert [Che+19].

Im Jahre 2018 wurde die Meltdown-Variante Foreshadow von Bulck et al. gefunden. Diese nutzt Prozessor-Exceptions, um Daten aus dem L1 Cache einer Intel SGX Enklave zu extrahieren [Bul+18]. Weisse et al. verallgemeinerten Foreshadow im Jahre 2018 zu Foreshadow-NG, welches es einer virtuellen Maschine erlaubt den gesamten L1 Cache des Wirtsystems zu lesen [Wei+18]. Ebenfalls im Jahre 2018 wurde LazyFP als Variante von Meltdown entdeckt. LazyFP extrahiert den Inhalt von FPU-Prozessorregistern mithilfe von Device-Not-Available Prozessor-Exceptions [SP18]. Van Schaik et al. fanden im Jahre 2019 die Meltdown-Variante RIDL. Diese nutzt Page-Fault Exceptions, um Daten aus dem prozessorinternen Line-Fill Buffer oder aus den Load Ports zu extrahieren [Sch+19a]. Canella et al. stellten im Jahre 2019 Fallout als Meltdown-Variante vor. Fallout extrahiert Daten aus dem Store Buffer und nutzt dafür unter anderem Page-Fault Exceptions oder General Protection Exceptions [Can+19b]. Schwarz et al. entdeckten die Meltdown-Variante ZombieLoad im Jahre 2019. ZombieLoad extrahiert Daten aus dem Line-Fill Buffer mithilfe von Microcode Assists [Sch+19b].

Um diese Varianten von Spectre und Meltdown zu systematisieren, entwickelten Canella et al. im Jahre 2019 einen Klassifizierungsbaum, der alle Varianten nach verschiedenen Kriterien kategorisiert. Außerdem beschrieben und evaluierten sie eine Vielzahl vorgeschlagener Gegenmaßnahmen im Hinblick auf ihre Wirksamkeit im Erschweren oder Unterbinden von Angriffe sowie die mit ihnen verbundenen Performance-Einbußen. [Can+19a]

Bulck et al. entdeckten im Jahre 2020 eine neue Art von Spectre-Angriffen, genannt Load Value Injection. Angriffe dieser Art kombinieren Effekte, die bereits in Varianten von Spectre und Meltdown genutzt wurden, um Daten aus dem Kontext des Opfers zu extrahieren. [Bul+20]

Neben Intel-Prozessoren wurden Spectre-Angriffe beispielsweise auch auf Prozessoren von AMD und ARM demonstriert [Koc+19] [Can+19a].

3 Grundlagen

Das folgende Kapitel behandelt die Grundlagen der Funktionsweise moderner Prozessoren, die für ein Verständnis von Spectre-Angriffen benötigt werden. Zunächst wird in Kapitel 3.1 das Konzept des virtuellen Speichers beschrieben. Virtueller Speicher ermöglicht unter anderem die Isolation von Prozessen, welche Voraussetzung für weitere Zugriffskontrollmechanismen ist. Anschließend wird in Kapitel 3.2 der Aufbau und die Funktionsweise von CPU-Caches dargestellt. CPU-Caches sind wichtig für die Performance moderner Prozessoren, da sie die Latenz des Hauptspeichers verbergen. Außerdem bilden sie die Basis der meisten Seitenkanalangriffe. Schließlich werden in Kapitel 3.3 die wichtigsten Elemente der Prozessororganisation moderner Intel Prozessoren thematisiert. Dies beinhaltet, wie Instruktionen verarbeitet und ausgeführt werden und enthält die Quelle der Spectre-Lücken.

3.1 Physischer und virtueller Speicher

Moderne Prozessoren isolieren Prozesse voneinander dadurch, dass jeder Prozess seinen eigenen virtuellen Adressraum erhält. Einem Teil der virtuellen Adressen wird jeweils eine physische Adresse zugeordnet. Diese Zuordnung wird von der *Memory Management Unit* (MMU) durchgeführt. Sie erfolgt nicht byteweise, sondern nur in größeren Blöcken, genannt *Pages*. Viele Systeme unterstützen verschiedene Page-Größen, wobei kleine Pages von 4 KiB und große Pages von bis zu 1 GiB üblich sind [Int21d, Kap. 4, Tab. 1]. 64-Bit Systeme adressieren den virtuellen Adressraum üblicherweise mit 48 Bit. [Gru17, S. 15]

Eine vollständige Zuordnungstabelle zu speichern, welche jeder virtuellen Page eine physische Page zuordnet, würde bei 64 Bits pro Eintrag 512 GiB an Speicher benötigen. Da so eine Zuordnungstabelle in allen praktischen Fällen nur sehr spärlich besetzt ist, lassen sich diese Platzkosten jedoch wesentlich reduzieren. Auf modernen Prozessoren wird dies mithilfe mehrerer Ebenen an Zuordnungstabellen erreicht. Jeder Eintrag einer Zuordnungstabelle verweist dabei entweder auf eine Tabelle der folgenden Ebene oder zeigt an, dass diese Tabelle vollständig leer ist [Int21d, S. 4-7]. Die Zuordnungstabellen werden im physischen Speicher abgelegt. Ein spezielles Prozessorregister enthält die (physische) Adresse der obersten Tabelle. In x86 Prozessoren ist dies das `CR3`-Register [Int21d, Kap. 4, S. 7].

Die Übersetzung von virtuellen zu physischen Adressen muss bei jedem Speicherzugriff durchgeführt werden. Dass diese Übersetzung schnell erfolgt, ist daher relevant für die Performance des Prozessors. Um eine schnelle Übersetzung zu gewährleisten, werden kürzlich ermittelte Zuordnungen in einem dedizierten Cache (siehe Kapitel 3.2) vorgehalten, der oft *Translation Lookaside Buffer* (TLB) genannt wird [Int21d, Kap. 4, S. 39].

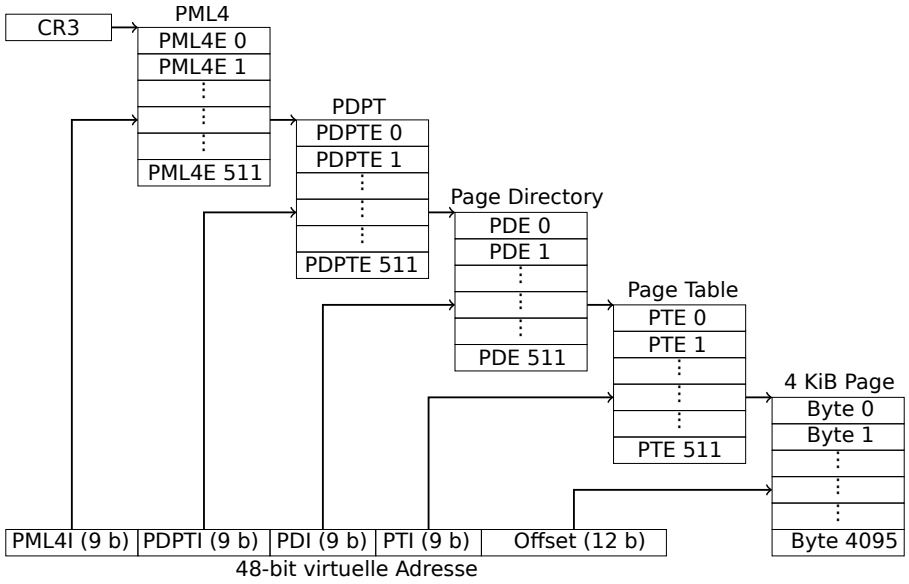

Abbildung 1: *Zuordnungstabellen und Adressübersetzung in modernen x86-64 Prozessoren.* [Gru20, *Abb. 2.4*]

3.1.1 Zuordnungstabellen in modernen Intel Prozessoren

In modernen x86-64 Prozessoren werden die Zuordnungstabellen in vier Ebenen angeordnet. Jede Tabelle hat 512 Einträge. Die obersten 36 Bits der zu übersetzenden virtuellen Adresse sind in vier Abschnitte von je 9 Bit unterteilt. Jeder dieser Abschnitte gibt den Index in eine der Zuordnungstabellen an. Mithilfe dieser Indices wird in der Tabelle der letzten Ebene eine 4 KiB Page ermittelt. Die verbleibenden, niederwertigsten 12 Bit der virtuellen Adresse verweisen auf ein Byte innerhalb dieser Page. Die vier Ebenen der Zuordnungstabellen und die Unterteilung virtueller Adressen sind in Abbildung 1 veranschaulicht. [Int21d, Kap. 4, S. 7]

Anstelle eine folgende Zuordnungstabelle anzugeben, kann ein Tabelleneintrag auch direkt die physische Adresse einer Page enthalten. In diesem Fall verweisen die verbleibenden Bits der virtuellen Adresse auf ein Byte innerhalb der ermittelten großen Page. Auf diese Weise werden unterschiedliche Page-Größen implementiert. [Int21d, Kap. 4, S. 7]

3.1.2 Adressräume in modernen Betriebssystemen

Auf modernen Betriebssystemen in x86-64 Prozessoren ist der virtuelle Adressraum üblicherweise fest in zwei Teile unterteilt. Die höheren Adressen sind dabei für den Kern des Betriebssystems reserviert und die niedrigeren Adressen für Nutzer-Prozesse. Zusätzlich wird auf manchen Betriebssystemen ein Teil des Adressraums des Kerns auf den gesamten physischen Speicher des Systems abgebildet. Diese Speicherzuordnung wird *Direct-Physical Map* genannt. Sie vereinfacht den Zugriff des Betriebssystem-Kerns auf den physischen Speicher. Eine Direct-Physical Map wird beispielsweise in den Kernen von Linux und OS X eingesetzt. [Lip+18, Kap. 2.2]

Um das Ausnutzen von Sicherheitslücken in Programmen zu erschweren, werden die virtuellen Adressen der meisten Zuordnungen randomisiert. Diese Technik wird *Address Space Layout Randomization*, kurz *ASLR* genannt. Sie kommt sowohl bei Nutzer-Prozessen als auch im Betriebssystem-

Kern zum Einsatz. Im Betriebssystem-Kern wird sie auch als *Kernel Address Space Layout Randomization* oder *KASLR* bezeichnet. Aktuelle Linux-Systeme auf x86-64 Prozessoren verwenden ASLR in Nutzer-Prozessen mit einer Entropie von 28 Bits [Lin21c, mmap_rnd_bits] [Lin21a, Z. 265] und KASLR mit einer Entropie von 9 Bits [Kos+20, Tab. 1]. [Can+19b, Kap. 2.5]

3.2 Caches

Die Latenz von Zugriffen auf den Hauptspeicher ist sehr hoch im Vergleich zu der Ausführungszeit von Instruktionen in modernen Prozessoren. Ein Zugriff auf den Hauptspeicher hat üblicherweise eine Latenz von mehreren hundert Prozessorzyklen [Gru20, S. 23]. Um diese Latenz zu verbergen, existieren zusätzliche Speicher, genannt *Caches*. Caches haben eine wesentlich geringere Speicherkapazität, sind aber auch wesentlich schneller als der Hauptspeicher. Üblich sind Unterschiede zwischen Caches und Hauptspeicher von bis zu 6 Größenordnungen in der Kapazität [Int21a, Kap. 2, S. 20] und 2 Größenordnungen in der Latenz [Gru20, S. 23]. Jeder Speicherzugriff des Prozessors erfolgt zunächst auf den Cache. Wenn die angeforderten Daten im Cache vorhanden sind, können sie direkt aus dem Cache geladen werden. In diesem Fall wird von einem *Cache Hit* gesprochen. Sind die angeforderten Daten jedoch nicht im Cache vorhanden, müssen sie aus dem Hauptspeicher geladen werden und, es wird von einem *Cache Miss* gesprochen. [Gru17, S. 17]

Die bei einem Cache Miss aus dem Hauptspeicher geladenen Daten werden anschließend im Cache platziert [Int21d, Kap. 11, S. 5]. Dies basiert auf der Annahme, dass diese Daten zukünftig mit einer höheren Wahrscheinlichkeit angefragt werden als andere Daten. In welchem Cacheeintrag die Daten platziert werden, hängt vom Typ des Caches ab und wird im folgenden Abschnitt beschrieben. Anschließend wird dargestellt, auf welche Art und Weise Daten ersetzt werden, die bereits im Cache vorhanden sind.

Caches in der hier beschriebenen Form werden in modernen Prozessoren üblicherweise für zwei Zwecke eingesetzt. Einerseits werden Caches verwendet, um Nutzdaten aus dem Hauptspeicher vorzuhalten. Andererseits werden Caches in TLBs eingesetzt, um die Adressübersetzung zu beschleunigen. Anderen Puffern, wie beispielsweise dem *Load Buffer*, *Line-Fill Buffer* oder *Branch Target Buffer*, liegt eine eigene Funktionsweise zugrunde. Solche Puffer werden daher in diesem Kapitel nicht weiter beschrieben.

Der Inhalt von Caches für Nutzdaten wird immer in Einheiten einer festen Größe verwaltet, sogenannten *Cachezeilen*. Auf modernen x86 Prozessoren beträgt die Größe einer Cachezeile 64 Byte [Int21d, Kap. 11, S. 4]. TLBs speichern eine Zuordnung von virtuellen zu physischen Adressen auf Page-Ebene. In diesem Fall würde eine einzige derartige Zuordnung als Cachezeile bezeichnet, jedoch wird der Begriff der Cachezeile im Kontext von TLBs üblicherweise nicht verwendet.

3.2.1 Typen von Caches

Es gibt verschiedene Möglichkeiten, Speicheradressen auf die Einträge des Caches zu verteilen. Zwei Speicheradressen, denen der gleiche Cacheeintrag zugeordnet wird, werden *kongruent* genannt [Gru17, S. 18]. Die Daten in einem Cacheeintrag können aus jeder der kongruenten Cachezeilen stammen. Um diese Daten einer eindeutigen Cachezeile zuordnen zu können, wird daher im

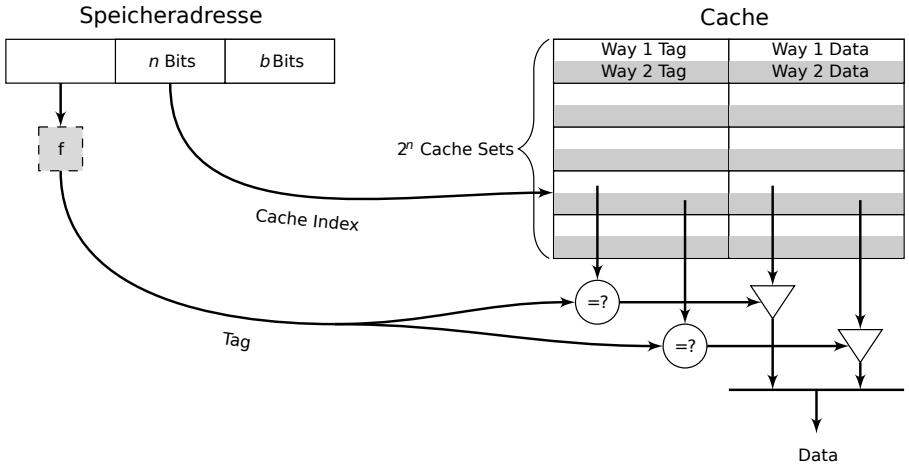

Abbildung 2: *Ein 2-Way Set-Associative Cache.* [Gru17, *Abb. 2.4*]

Cacheeintrag zusätzlich ein sogenannter *Tag* gespeichert [Gru17, S. 18–19]. Der Tag muss bei jedem lesenden Zugriff auf den Cache verglichen werden. Üblicherweise wird der Tag aus den höherwertigsten Bits der Adresse gebildet [Gru17, S. 18–19].

Ein häufig verwendeter Typ von Caches ist der *Set-Associative Cache*. Abbildung 2 zeigt ein Beispiel eines solchen Caches. Set-Associative Caches werden in modernen Intel Prozessoren hauptsächlich verwendet (siehe Kapitel 3.2.5). Sie bestehen aus 2^n Cacheeinträgen, sogenannten *Cache Sets*. Bei einer Cachezeilen-Größe von 2^b Bytes werden die niederwertigsten b Adressbits benutzt, um ein Byte innerhalb einer Cachezeile zu adressieren. Die n folgenden Bits der Adresse werden als Index in den Cache benutzt, um das entsprechende Cache Set zu ermitteln. Die verbleibenden, höherwertigsten Bits der Adresse werden zur Berechnung des Tags verwendet. Außerdem enthält jedes Cache Set mehrere *Cache Ways*. Jeder Cache Way kann eine Cachezeile an Daten speichern. Auf diese Weise kann ein Cache Set die Daten aus mehreren kongruenten Cachezeilen gleichzeitig speichern. Ein Set-Associative Cache mit m Cache Ways wird auch *m-Way Set-Associative Cache* genannt. [Gru17, S. 19]

Außerdem werden zwei weitere Typen von Caches unterschieden. Diese sind jedoch Spezialfälle des Set-Associative Cache. Der *Directly-Mapped Cache* entspricht einem Set-Associative Cache mit nur einem Cache Way pro Cache Set, also einem 1-Way Set-Associative Cache. Folglich kann dieser Cache stets nur eine aus mehreren kongruenten Cachezeilen speichern. Der *Fully-Associative Cache* entspricht einem Set-Associative Cache mit nur einem Cache Set, welches alle Cache Ways enthält. Also sind in diesem Cache sämtliche Cachezeilen kongruent.

3.2.2 Adressierung

Die Speicheradresse fließt ein in die Berechnung des Cache Index und in die Berechnung des Tags. In beiden Fällen kann entweder die physische oder die virtuelle Adresse verwendet werden. In Prozessoren üblich sind die drei Varianten *virtually-indexed virtually-tagged* (VIVT), *physically-indexed physically-tagged* (PIPT) und *virtually-indexed physically-tagged* (VIPT) [Gru17, S. 22]. Diese werden im Folgenden beschrieben.

Virtually-Indexed Virtually-Tagged

VIVT Caches nutzen die virtuelle Adresse für den Cache Index und den Tag. Dies ermöglicht eine geringe Latenz bei Zugriffen auf den Cache, da keine Übersetzung in eine physische Adresse notwendig ist. Ein gemeinsamer Speicherbereich mehrerer Prozesse kann unterschiedliche Cache Indices haben. Dadurch kann ein erhöhter Speicherbedarf auftreten. Außerdem ist der Tag nur für einen virtuellen Adressraum gültig. Daher kann es erforderlich sein, bei einem Context Switch den Cache zu invalidieren. Da TLBs für die Adressübersetzung verwendet werden, werden in ihnen ausschließlich VIVT Caches eingesetzt. [Gru17, S. 22]

Physically-Indexed Physically-Tagged

PIPT Caches nutzen die physische Adresse für den Cache Index und den Tag. Dadurch werden die Nachteile von VIVT Caches vermieden. Gemeinsamer Speicher wird immer auf den gleichen Cache Index abgebildet und alle Tags sind nach einem Context Switch gültig. Jedoch ist dies mit einer höheren Latenz verbunden, da eine Übersetzung der virtuellen in eine physische Adresse notwendig ist. PIPT Caches werden zum Beispiel als Caches für Nutzdaten eingesetzt. [Gru17, S. 22]

Virtually-Indexed Physically-Tagged

VIPT Caches nutzen die virtuelle Adresse für den Cache Index und die physische Adresse für den Tag. Der Cache Index ist unmittelbar verfügbar und wird genutzt um den entsprechenden Cacheeintrag zu laden. Parallel zu diesem Ladevorgang findet die Übersetzung von der virtuellen Adresse in eine physische Adresse statt. Dadurch wird wie bei VIVT Caches eine geringe Latenz bei Zugriffen auf den Cache ermöglicht. Außerdem werden die Tags wie bei PIPT Caches bei einem Context Switch nicht invalidiert. VIPT Adressierung wird typischerweise in den kleinsten und schnellsten Caches eines Systems verwendet. [Gru17, S. 22–23]

3.2.3 Cache Replacement Strategien

Ist im Cache die Kapazität an kongruenten Cachezeilen ausgelastet, so muss beim Zugriff auf eine weitere kongruente Cachezeile einer der vorhandenen Cacheeinträge ersetzt werden. Für optimale Performance müsste der Eintrag ersetzt werden, auf den in Zukunft am wenigsten zugegriffen wird. Um dieses Verhalten zu approximieren verwenden Prozessoren verschiedene Strategien.

Eine verwendete Strategie ist *least-recently used* (LRU) [Gru17, S. 20]. Hierbei wird der Eintrag ersetzt, auf den am längsten nicht mehr zugegriffen wurde. Um dies zu implementieren, beinhaltet jeder Cacheeintrag einen Zeitstempel des letzten Zugriffs [Gru17, S. 20]. Zu Problemen führt diese Strategie, wenn viele kongruente Cachezeilen gleichzeitig benötigt werden. Übersteigt diese Anzahl die Kapazität des Caches, kann im schlimmsten Fall jeder Speicherzugriff ein Cache Miss sein [Gru17, S. 20]. Eine andere Strategie ist die pseudo-zufällige Auswahl des zu ersetzenden Eintrages. Diese ist sehr einfach in Hardware zu implementieren und kann gute Performancewerte erreichen [PS05]. Die *last-in first-out* (LIFO) Strategie ersetzt stets den zuletzt geladenen Cacheeintrag.

Einige moderne Prozessoren wechseln dynamisch zwischen mehreren Strategien, abhängig von der beobachteten Nutzung des Caches. Dies ermöglicht es, die Vorteile verschiedener Strategien zu

kombinieren. In der Praxis können dadurch Performanceverbesserungen erzielt werden. [Gru17, S. 21] [Qur+07]

3.2.4 Cache-Hierarchie

In modernen Systemen werden Caches nicht einzeln verbaut. Stattdessen sind sie Teil einer mehrschichtigen Cache-Hierarchie. Die unteren Schichten dieser Hierarchie bilden schnelle Caches mit geringer Kapazität. Die Caches höherer Schichten sind stets langsamer und größer als die der darunterliegenden Schichten. Manche Caches enthalten immer sämtliche Einträge der darunterliegenden Schichten. In diesem Fall wird der Cache *inklusiv* genannt. Wenn Daten aus dem unteren Cache verdrängt werden, so bleiben sie im inklusiven oberen Cache weiterhin vorhanden. [Gru17, S. 23–24]

Ein Cache kann für Instruktionen oder für Daten reserviert sein oder sowohl Instruktionen als auch Daten beinhalten [Gru17, S. 23–24]. Im letzteren Fall wird der Cache auch *unified* genannt. In Mehrkernsystemen wird ein Cache entweder von allen Kernen gemeinsam verwendet, oder es existiert ein privater Cache für jeden Kern [Gru17, S. 23–24].

3.2.5 Caches in modernen Intel Prozessoren

In modernen Intel x86-64 Prozessoren der Skylake Mikroarchitektur wird eine Cache-Hierarchie bestehend aus drei Schichten verwendet. Die unterste Schicht bilden die L1d und L1i Caches, die jeweils für Daten und für Instruktionen reserviert sind. Die L1 Caches sind virtually-indexed physically-tagged. Darüber liegt der L2 Cache, der sowohl Daten als auch Instruktionen beinhalten kann. Der L2 Cache ist physically-indexed physically-tagged und nicht inklusiv gegenüber den L1 Caches. L1 und L2 existieren pro Kern als private Caches. Der folgende L3 Cache wird auch *Last-Level Cache* (LLC) genannt. Er kann, wie der L2 Cache, sowohl Daten als auch Instruktionen beinhalten und ist physically-indexed physically-tagged. Anders als die L1 und L2 Caches wird derselbe L3 Cache von allen Kernen des Systems verwendet. Außerdem ist der L3 Cache inklusiv gegenüber den L1 und L2 Caches. Die L1 Caches sind 8-Way Set-Associative, der L2 Cache ist 4-Way Set-Associative und der L3 Cache ist bis zu 16-Way Set-Associative. Die Ersetzungsstrategie wechselt dynamisch zwischen LRU und einer Modifikation der LIFO [Gru17, S. 21]. Level 1 und 2 der Cache-Hierarchie enthalten zusätzlich für jeden Nutzdaten-Cache einen TLB. Der L1i TLB ist 8-Way Set-Associative, der L1d TLB ist 4-Way Set-Associative und der L2 TLB ist 12-Way Set-Associative. [Int21a, Kap. 2, S. 20]

Abbildung 3 zeigt den abstrakten Aufbau und die Cache-Hierarchie eines modernen Intel Prozessors. Außerdem zeigt sie individuelle L1 und L2 Caches für jeden Prozessorkern und einen gemeinsamen L3 Cache.

3.3 Prozessororganisation der Intel Skylake Mikroarchitektur

Moderne Prozessoren besitzen nicht nur einen, sondern mehrere identische Ausführungskerne. Diese ermöglichen die Ausführung mehrerer Prozesse gleichzeitig, wodurch die Gesamtperformance

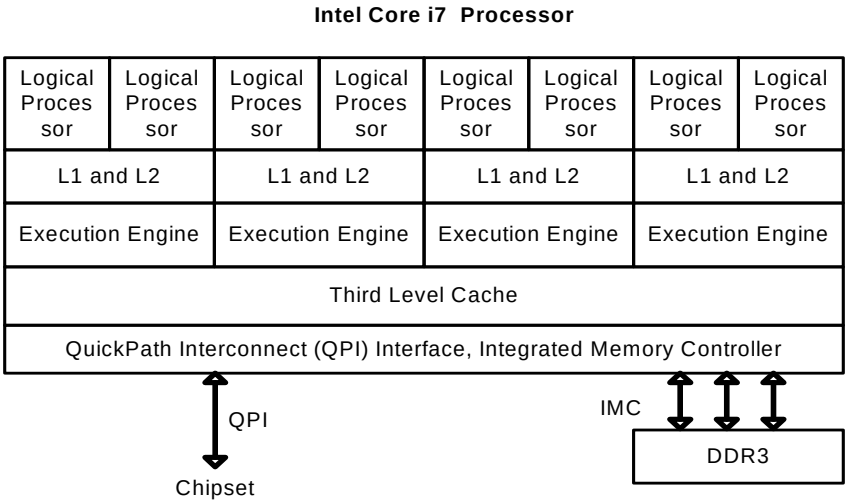

Abbildung 3: *Abstrakter Aufbau und Cache-Hierarchie eines modernen Intel Prozessors.* [*Int21b, Kap. 2, Abb. 8*]

des Systems signifikant erhöht werden kann [Gru17, S. 14]. Jede Resource eines Mehrkernsystems wird entweder von allen Kernen gemeinsam verwendet oder es existiert eine private Instanz der Ressource für jeden Kern [Gru17, S. 14]. Wie in Kapitel 3.2 beschrieben werden beispielsweise der L3 Cache und der Hauptspeicher gemeinsam verwendet. Die L1 und L2 Caches existieren separat für jeden Kern.

Neben mehreren Ausführungskernen besitzen viele moderne Prozessoren die Möglichkeit, mehrere Ausführungsstränge auf einem Kern gleichzeitig zu bearbeiten. Diese Technik wird bezeichnet als *Simultaneous Multithreading* (SMT), in Prozessoren von Intel auch *Hyper-Threading* [Int21d, S. 8-24]. Jeder *physische* Ausführungskern wird dann konzeptuell als mehrere *logische* Kerne betrachtet. Prozessorintern werden die Instruktionen aller logischen Kerne verzahnt ausgeführt [Int21d, Kap. 8, S. 27]. Da die logischen Kerne Teil desselben physischen Kerns sind, verwenden sie auch solche Ressourcen gemeinsam, die separat für jeden physischen Kern existieren.

Bei der Beschreibung eines Prozessors sind zwei Begriffe zu unterscheiden. Die *Befehlssatzarchitektur* bezeichnet das Verhalten des Prozessors gegenüber der Software. Dies beinhaltet unter anderem Prozessormodi, Instruktionen und Register der Architektur. Die *Mikroarchitektur* bezeichnet eine konkrete Implementierung der Befehlssatzarchitektur. Dies beinhaltet unter anderem, wie die Instruktionen der Architektur prozessorintern umgesetzt und ausgeführt werden. Als *architektureller Zustand* wird der gemeinsame Zustand aller Elemente der Befehlssatzarchitektur bezeichnet. Dies umfasst beispielsweise die Prozessorregister und den Inhalt des Hauptspeichers. [Gru20, S. 13–14, 43]

In diesem Kapitel werden speziell die Ausführungskerne moderner x86-64 Prozessoren der Intel Skylake Mikroarchitektur beschrieben. Viele andere Prozessortypen folgen jedoch einem ähnlichen Aufbau. Abbildung 4 zeigt den abstrakten Aufbau einer dieser Ausführungskerne. Jeder Kern ist in drei verschiedene Elemente aufgeteilt, die im Folgenden detailliert beschrieben werden. Das *Front-End* lädt und dekodiert Instruktionen. Die *Execution Engine* bereitet die dekodierten Instruktionen auf ihre Ausführung vor und führt sie anschließend aus. Das *Memory Subsystem* enthält die Prozessorcaches und bildet die Schnittstelle zum Rest des Systems. [Lip+18, Kap. 2.1]

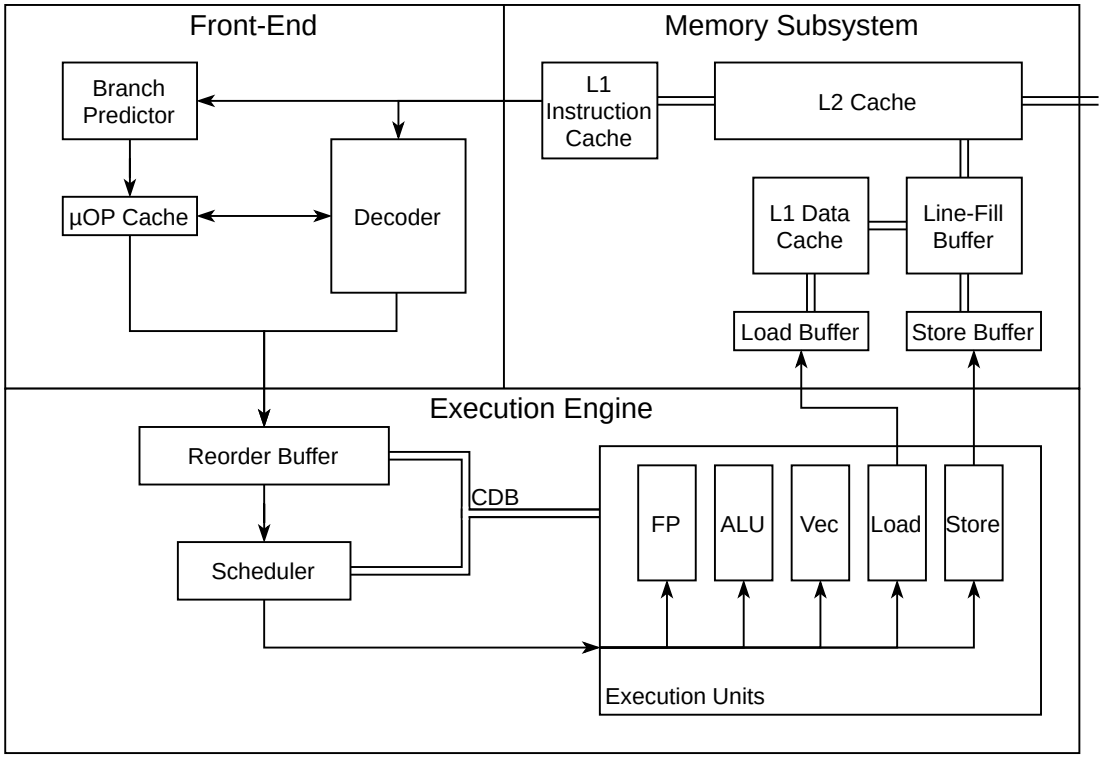

Abbildung 4: *Vereinfachte Darstellung eines einzelnen Ausführungskerns der Intel Skylake Mikroarchitektur.* [Lip+18, *Abb. 1*] [Sch+19a, *Abb. 9*]

3.3.1 Front-End

Das Front-End lädt die Instruktionen aus dem Speicher, die der Kern anschließend ausführen wird. Diese Instruktionen werden vom *Decoder* zunächst in eine oder mehrere kleinere Operationen dekodiert. Die resultierenden Operationen werden auch als Mikrooperationen oder *µOPs* bezeichnet. Um die erwartete Latenz des Dekodierens zu verkürzen, wird die Zuordnung zwischen Instruktionen und Mikrooperationen in dem *µOP Cache* zwischengespeichert. Wenn anschließend eine bereits dekodierte Instruktion nochmal dekodiert werden soll, können die entsprechenden Mikrooperationen direkt aus dem µOP Cache geladen werden. Auf diese Weise wird ein erneutes Dekodieren der Instruktion vermieden. Ein weiterer Teil des Front-End ist der *Branch Predictor*. Dieser trifft Vorhersagen über bedingte Sprünge, deren Bedingungen noch nicht ausgewertet sind. Der Branch Predictor wird in Kapitel 3.3.4 detailliert beschrieben. Abschließend gibt das Front-End die Mikrooperationen an die Execution Engine weiter. [Lip+18, Kap. 2.1]

3.3.2 Execution Engine

Die Execution Engine nimmt die im Front-End dekodierten Mikrooperationen entgegen. Diese werden zunächst in den *Reorder Buffer* eingereiht. Der Reorder Buffer allokiert die von der Mikrooperation benötigten Register. Wenn die Ausführung einer Mikrooperation abgeschlossen ist, überträgt der Reorder Buffer außerdem die Ergebnisse der Ausführung in den Registerspeicher. Die Mikrooperationen werden vom Reorder Buffer weitergeleitet an die *Unified Reservation Station*, auch *Scheduler* genannt. Sind alle Operanden einer Mikrooperation verfügbar, so übergibt der Scheduler

die Mikrooperation der nächsten freien *Execution Unit*. Dabei können diese Mikrooperation in beliebiger Reihenfolge ausgeführt werden, unabhängig von der ursprünglichen Reihenfolge der Instruktionen. Diese Art der Ausführung wird *Out-of-Order Execution* genannt. Die Ergebnisse der Ausführung werden jedoch streng in der ursprünglichen Reihenfolge in den architekturellen Zustand übertragen. Mit dieser, *Retirement* genannten, Übertragung ist die Bearbeitung der Mikrooperationen abgeschlossen. Die Execution Units führen die eigentlichen Berechnungen durch. Jede Execution Unit ist auf bestimme Berechnungen spezialisiert, z.B. arithmetische Operationen, Gleitkommaoperationen, Vektoroperationen oder Speicherzugriffe. Execution Units für lesende Speicherzugriffe werden auch *Load Ports* genannt. Der Reorder Buffer, der Scheduler und die Execution Units sind außerdem über die *Common Data Buses* (CDB) miteinander verbunden. Über diese werden unter anderem die Ergebnisse der Mikrooperationen ausgetauscht. [Lip+18, Kap. 2.1] [Sch+19a, Kap. II.B]

3.3.3 Memory Subsystem

Das Memory Subsystem beinhaltet die privaten Caches des Kerns, also die L1 und L2 Caches. Es ist mit dem L3 Cache und über diesen mit dem restlichen Speicher des Systems verbunden. Außerdem enthält es drei Pufferspeicher[1]. Schreibende Speicherzugriffe werden von der jeweiligen Execution Unit in den *Store Buffer* gegeben. Mithilfe dieses Puffers wird der eigentliche Speicherzugriff durchgeführt. Dadurch wird die Latenz des Speicherzugriffs verborgen und die Execution Unit kann bereits die nächste Mikrooperation ausführen. Lesende Speicherzugriffe werden ähnlich zu schreibenden, aber über den *Load Buffer* verwaltet. Löst ein lesender Speicherzugriff einen Cache Miss im L1d Cache aus, so wird zusätzlich ein Eintrag im *Line-Fill Buffer* allokiert. Dieser überwacht den Ladevorgang aus dem L2 oder L3 Cache. Wie bei schreibenden Speicherzugriffen können auf diese Weise auch lesende Speicherzugriffe ohne die Latenz des Speichers bearbeitet werden. [Sch+19a, Kap. II.D]

3.3.4 Branch Prediction und Transient Execution

Der *Branch Predictor* ist Teil des Front-Ends. Wenn die Bedingung eines bedingten Sprunges nicht unmittelbar bekannt ist, sagt der Branch Predictor den weiteren Kontrollfluss voraus. Diese Vorhersage kann basieren auf vorherigen Auswertungen ähnlicher Bedingungen oder auf dem bedingten Sprung selber [Lip+18, Kap. 2.1]. Auf diese Weise können folgende Instruktionen bereits bearbeitet werden, bevor die Bedingung tatsächlich ausgewertet wurde. Dies wird spekulative Ausführung oder *Speculative Execution* genannt [Sch+19a, Kap. II.C]. War die Vorhersage korrekt, so können die Ergebnisse der spekulativ ausgeführten Instruktionen in den architekturellen Zustand übernommen werden [Lip+18, Kap. 2.1]. War die Vorhersage inkorrekt, werden die spekulativ ausgeführten Instruktionen und ihre Ergebnisse verworfen. Der Reorder Buffer wird geleert, der Scheduler neu initialisiert und das Front-End beginnt, die Instruktionen auf dem korrekten Pfad zu dekodieren [Lip+18, Kap. 2.1]. Die Instruktionen, die nach einer falschen Vorhersage ausgeführt und anschließend verworfen werden, werden *Transient Instructions* genannt [Can+19b, Kap. 2.2]. Die spekulative Ausführung heißt im Falle einer falschen Vorhersage auch *Transient Execution* [Can+19b, Kap. 2.2].

[1] Wie bereits in Kapitel 3.2 dargestellt funktionieren diese Puffer anders als die in Kapitel 3.2 beschriebenen Caches und sind daher hier unter einem eigenen Begriff gefasst.

Eine andere Form der Transient Execution tritt auch in linearem Kontrollfluss auf. Löst eine Instruktion eine Prozessor-Exception aus, so beginnt eine Transient Execution, die folgende Instruktionen weiterhin ausführen kann. Grund dafür ist, dass der Prozessor die Exception erst bei Retirement der auslösenden Instruktion behandelt. Sind die Operanden folgender Instruktionen bereits verfügbar, können diese also noch ausgeführt werden, bevor die Prozessor-Exception behandelt wird. Siehe auch Kapitel 4.4 für Angriffe, die diese Form der Transient Execution ausnutzen. [Lip+18, Kap. 3]

Die Transient Instructions und ihre Ergebnisse werden bei Abschluss der Transient Execution verworfen. Der Zustand der Caches wird jedoch nicht in den Zustand vor Beginn der Transient Execution zurückversetzt. Folglich kann ein Speicherzugriff während einer Transient Execution eine Eintragung im Cache verursachen. Diese ist sogar nach Abschluss der Transient Execution über einen Cache-basierten Seitenkanal (siehe Kapitel 4.1) beobachtbar. [Lip+18, Kap. 3]

4 Spectre-Angriffe und Details ihrer Implementierung

Spectre-Angriffe nutzen Fehler in der Mikroarchitektur aus, um während einer Transient Execution unberechtigt auf Daten zuzugreifen. Diese Daten übertragen sie anschließend aus dem Kontext der Transient Execution in den architekturellen Zustand. Bei allen in dieser Arbeit betrachteten Angriffen werden für diese Übertragung *Cache-basierte Seitenkanalangriffe* benutzt. Zwei wichtige Seitenkanalangriffe dieser Art werden zunächst in Kapitel 4.1 erklärt. Diese ermöglichen einem Angreifer, Zeitpunkt und Adresse von Speicherzugriffen zu ermitteln. Spectre-Angriffe werden auch als *Transient Execution Attacks* bezeichnet. [Can+19a, Kap. 1]

Neben Cache-basierten Seitenkanalangriffen behandelt dieses Kapitel Spectre-Angriffe allgemein sowie ausgewählte konkrete Spectre-Angriffe. In Kapitel 4.2 werden die Eigenschaften und die Funktionsweise von Spectre-Angriffen im Allgemeinen beschrieben. Spectre-Angriffe werden grundlegend unterteilt in Angriffe basierend auf Branch Prediction und Angriffe basierend auf Prozessor-Exceptions [Can+19a, Kap. 2]. Die Unterschiede zwischen diesen Arten werden in Kapitel 4.3 und Kapitel 4.4 detailliert dargestellt. In Kapitel 4.5 werden Angriffsszenarien für Spectre-Angriffe allgemein sowie die in dieser Arbeit betrachteten Szenarien beschrieben. Anschließend werden in Kapitel 4.6 wiederverwendbare Elemente der Implementierung von Spectre-Angriffen thematisiert. Diese Elemente werden in den konkreten Spectre-Angriffen verwendet, die schließlich in Kapitel 4.7 erklärt werden. Dabei handelt es sich um vier Angriffe basierend auf Prozessor-Exceptions. Es werden sowohl die Funktionsweise als auch Details der Implementierung dieser Angriffe beschrieben.

4.1 Cache-basierte Seitenkanalangriffe

Cache-basierte Seitenkanalangriffe nutzen durch einen Cache bedingte Unterschiede in der Zugriffszeit, um vertrauliche Informationen abzuleiten [YF14, Kap. 3]. Grundlage dafür ist, dass die Latenz des Zugriffs kürzer ist, wenn die angefragten Daten bereits im Cache vorhanden sind. Auf diese Weise kann ein Angreifer herausfinden, ob von ihm angefragte Daten bereits im Cache vorhanden sind oder nicht. Da die Unterschiede in der Zugriffszeit direkt aus der allgemeinen Funktionsweise von Caches folgen (siehe Kapitel 3.2), weisen alle Caches einen solchen Seitenkanal auf. Diese Angriffe können folglich sowohl auf Caches der Adressübersetzung [HWH13] [Gra+18] als auch auf Caches für Nutzdaten angewandt werden. Im Folgenden werden zwei Cache-basierte Seitenkanalangriffe auf Caches für Nutzdaten beschrieben.

Alle konkreten Angriffe, die in dieser Arbeit betrachtet werden (siehe Kapitel 4.7), führen Cache-basierte Seitenkanalangriffe ausschließlich auf den L1d Cache aus. Siehe auch Kapitel 4.6.3 für eine genaue Betrachtung dieses Umstandes.

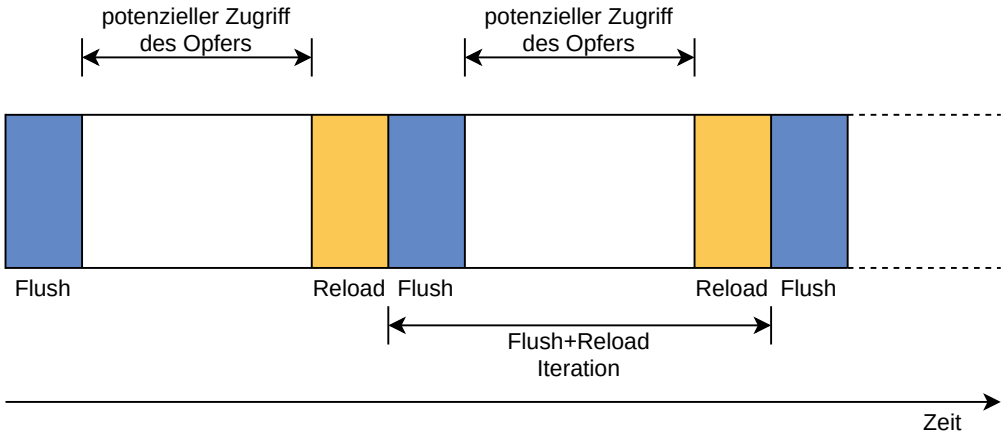

Abbildung 5: *Zeitlicher Verlauf von Flush+Reload.*

4.1.1 Flush+Reload

Flush+Reload ist ein Cache-basierter Seitenkanalangriff. Um diesen Angriff auf ein bestimmtes Opfer und eine bestimmte Cachezeile durchführen zu können, muss der Angreifer sowohl den Speicherbereich der Cachezeile als auch einen zugehörigen Cache mit dem Opfer gemeinsam haben. Dies ermöglicht dem Angreifer, einen Zugriff des Opfers auf die Cachezeile zu detektieren. [YF14, Kap. 3]

Der Angriff besteht aus drei Phasen, die in Abbildung 5 illustriert werden. In der ersten, sogenannten *Flush* Phase invalidiert der Angreifer die gewählte Cachezeile. In der zweiten Phase wartet der Angreifer. Das Opfer wird in dieser Zeit entweder auf die Cachezeile zugreifen oder nicht. In der letzten Phase, genannt *Reload*, lädt der Angreifer die Cachezeile ein und misst die Latenz dieses Zugriffs. Misst der Angreifer eine kurze Latenz, war die Cachezeile bereits im Cache vorhanden. In diesem Fall hat das Opfer in der Zwischenzeit auf die Cachezeile zugegriffen. Misst der Angreifer jedoch eine große Latenz, war die Cachezeile nicht im Cache vorhanden und das Opfer hat in der Zwischenzeit nicht auf diese zugegriffen. Welche konkreten Latenzen der Angreifer als Cache Hit oder Cache Miss klassifiziert, hängt von dem angegriffenen System ab. Wird die Cachezeile nach Zugriff des Opfers von anderen Cachezeilen aus dem Cache verdrängt, so kommt es zu falsch-negativen Ergebnissen. Falsch-positive Ergebnisse treten auf, wenn der Prozessor unabhängig von tatsächlichem Zugriff die Cachezeile einlädt (sogenanntes *Prefetching*). [YF14, Kap. 3]

Um den Flush+Reload Angriff durchzuführen, muss der Angreifer die gewählte Cachezeile invalidieren können. Auf x86 Prozessoren ist dies mit der `clflush` Instruktion möglich, die keine Privilegien benötigt [Int21c, S. 3-145]. Außerdem muss der Angreifer in der Lage sein die Zugriffslatenz zu bestimmen. Auf x86 Prozessoren lässt sich dies mithilfe der `rdtsc` Instruktion umsetzen [Int21c, Kap. 4, S. 547]. Diese erfordert ebenfalls keine Privilegien. [YF14, Kap. 3]

Der Flush+Reload Angriff kann wiederholt ausgeführt werden, um Zugriffe des Opfers über einen längeren Zeitraum zu beobachten. Außerdem kann der Angriff gleichzeitig für unterschiedliche Cachezeilen ausgeführt werden, um Zugriffe auf einem größeren Speicherbereich zu beobachten. Insgesamt kann so das Zugriffsverhalten des Opfers über den gesamten geteilten Speicherbereich und über einen beliebigen Zeitraum ermittelt werden. Die Speicherzugriffe lassen sich dabei räum-

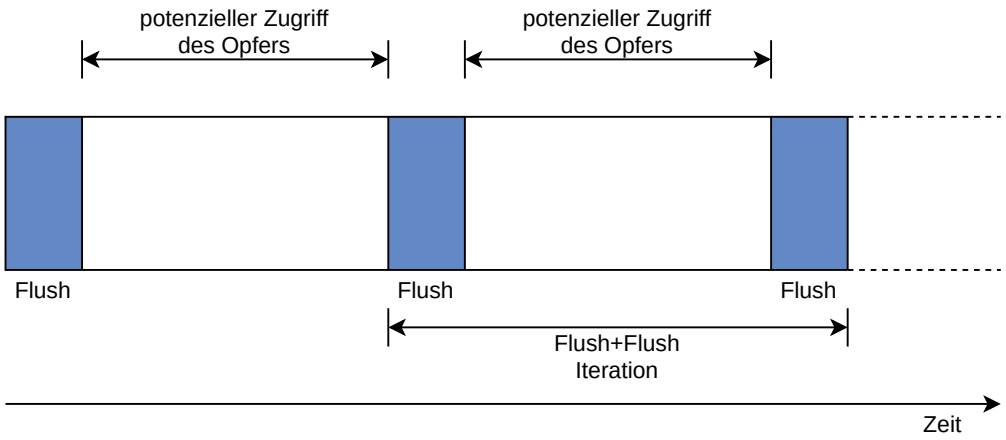

Abbildung 6: *Zeitlicher Verlauf von Flush+Flush.*

lich auf die Größe einer Cachezeile und zeitlich auf die Dauer eines einzelnen Flush+Reload Angriffs auflösen. [YF14, Kap. 3]

In der Praxis wurde Flush+Reload beispielsweise eingesetzt, um die RSA-Implementierung von GnuPG anzugreifen. Dabei wurde mittels Flush+Reload die Aktivität eines `gpg` Prozesses beobachtet, während dieser eine Signatur berechnet. Aus den dabei erhobenen Messungen wurde anschließend eine Komponente des verwendeten privaten Schlüssels rekonstruiert. [YF14, Kap. 4]

4.1.2 Flush+Flush

Der *Flush+Flush* Angriff ist eine Variante von Flush+Reload. Er basiert nicht auf Unterschieden in der Latenz von Speicherzugriffen, sondern auf Zeitunterschieden bei der Invalidierung einer Cachezeile. Befindet sich eine Cachezeile im Cache, so muss die Invalidierung des Cacheeintrages tatsächlich erfolgen. Befindet sich die Cachezeile hingegen nicht im Cache, so kann die Invalidierung vorzeitig beendet werden. Aufgrund dieses Unterschiedes weist die Invalidierung einer Cachezeile eine größere Latenz auf, wenn diese im Cache vorhanden ist. Die Einordnung der Messergebnisse ist damit umgekehrt zu der im Flush+Reload Angriff: Eine lange Latenz signalisiert einen Zugriff des Opfers. Der Flush+Flush Angriff besteht aus den gleichen drei Phasen wie der Flush+Reload Angriff. Der einzige Unterschied liegt entsprechend in der dritten Phase, in der der Angreifer die anvisierte Cachezeile invalidiert, anstatt sie einzuladen. [Gru+16, Kap. 3]

Die Invalidierung einer Cachezeile löst kein Prefetching aus [Gru+16, Kap. 3]. Daher kann Prefetching beim Flush+Flush Angriff nicht mehr durch den Angreifer, sondern nur noch durch das Opfer ausgelöst werden. Dies reduziert die Zahl von falsch-positiven Ergebnissen. Bei wiederholter Ausführung des Flush+Flush Angriffs kann außerdem die erste Phase übersprungen werden, da die dritte Phase bereits für eine Invalidierung der Cachezeile sorgt. Dies wird in Abbildung 6 illustriert.

In der Praxis wurde Flush+Flush beispielsweise eingesetzt, um eine AES-Implementierung von OpenSSL anzugreifen. Dabei wurde mittels Flush+Flush die Aktivität eines Prozesses beobachtet, während dieser wiederholt einen bekannten Klartext verschlüsselt. Aus den dabei erhobenen Messungen wurde anschließend ein Teil des verwendeten Schlüssels rekonstruiert. [Gru+16, Kap. 7]

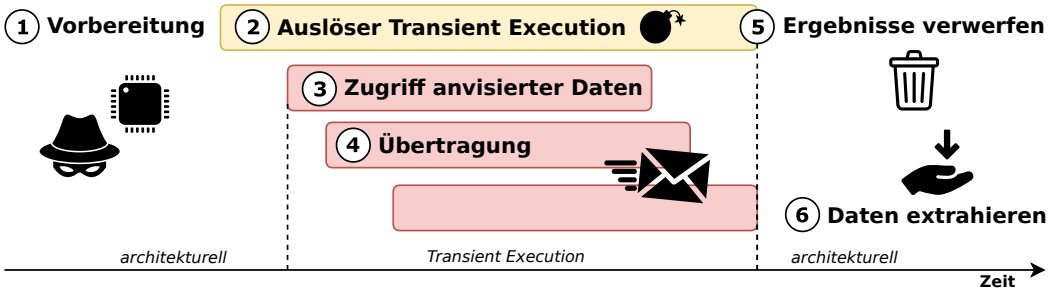

Abbildung 7: *Methodik von Spectre-Angriffen, bestehend aus 6 Phasen.* [Gru20, Abb. 3.1]

4.2 Allgemeine Funktionsweise von Spectre-Angriffen

Die Methodik von Spectre-Angriffen stimmt bei allen Angriffen überein. Abbildung 7 illustriert die 6 Phasen, in die sich ein solcher Angriff unterteilen lässt:

- In **Phase 1** werden die Mikroarchitektur und der Übertragungskanal vorbereitet. Es wird dafür gesorgt, dass die später ausgelöste Transient Execution auf die anvisierten Daten zugreifen kann. Außerdem wird sichergestellt, dass die Transient Execution lange genug anhält, um diese Daten in den Übertragungskanal zu kodieren. Diese Phase wird immer in dem Kontext des Angreifers ausgeführt. [Gru20, S. 43–45]
- **Phase 2** bezeichnet den Eintritt in die Transient Execution. Wie in Kapitel 3.3.4 erläutert, kann diese durch eine falsche Branch Prediction oder durch eine Prozessor-Exception ausgelöst werden. Des Weiteren kann die Transient Execution sowohl im Kontext des Angreifers als auch im Kontext des Opfers erfolgen. Beides ist abhängig von der Art des Spectre-Angriffs und wird in Kapitel 4.3 und Kapitel 4.4 differenziert. [Gru20, S. 43–45]
- In **Phase 3** werden Transient Instructions ausgeführt, die auf die anvisierten Daten zugreifen. Anschließend werden die Daten auf Kodierung in den Übertragungskanal vorbereitet. Dies findet in dem gleichen Kontext wie Phase 2 statt. [Gru20, S. 43–45]
- **Phase 4** findet weiterhin innerhalb der Transient Execution statt. In dieser Phase werden die Daten in den Übertragungskanal kodiert, sodass diese nach Abschluss der Transient Execution extrahiert werden können. Dies findet in dem gleichen Kontext wie die Phasen 2 und 3 statt. [Gru20, S. 43–45]
- In **Phase 5** wird die Transient Execution beendet und damit die Ergebnisse aller Transient Instructions verworfen. Der Übertragungskanal wurde jedoch so gewählt, dass die in ihm kodierten Daten erhalten bleiben und anschließend dekodiert werden können. [Gru20, S. 43–45]
- In **Phase 6** wird diese Dekodierung durchgeführt: Die im Übertragungskanal enthaltenen Daten werden extrahiert. Dadurch werden diese Daten von der Domäne der Transient Execution in den architekturellen Zustand übertragen. Diese Phase wird immer in dem Kontext des Angreifers ausgeführt. [Gru20, S. 43–45]

Üblicherweise wird ein Cache-basierter Übertragungskanal verwendet. Dies ist insbesondere bei allen in Kapitel 4.7 betrachteten Angriffen der Fall. Bei diesem Übertragungskanal werden die Daten in den Zustand des Caches kodiert. Dabei bezeichnet der Zustand des Caches, welche Cachezeilen im Cache enthalten sind. Dieser Zustand kann mit einem Cache-basierten Seitenkanalangriff (siehe Kapitel 4.1) beobachtet werden. Auf diese Weise werden die kodierten Daten extrahiert. Die Vorbe-

reitung eines Cache-basierten Übertragungskanals in Phase 1 des Spectre-Angriffs besteht darin, alle beobachteten Cachezeilen zu invalidieren. Dies entspricht der ersten Phase des Cache-basierten Seitenkanalangriffs. In Phase 6 des Spectre-Angriffs wird auf die beobachteten Cachezeilen zugegriffen und die Latenz dieses Zugriffs gemessen. Dies entspricht der dritten Phase des Cache-basierten Seitenkanalangriffs. [Gru20, S. 43–45]

Spectre-Angriffe werden unterteilt in zwei verschiedene Arten: Angriffe basierend auf Branch Prediction und Angriffe basierend auf Prozessor-Exceptions [Can+19a, Kap. 2]. Diese beiden Arten werden im Folgenden detailliert beschrieben.

4.3 Spectre-Angriffe basierend auf Branch Prediction

Spectre-Angriffe basierend auf Branch Prediction werden auch *Spectre-type Attacks* genannt nach dem Paper, das die ersten Angriffe dieser Art beschrieben hat [Koc+19]. Bei Angriffen basierend auf Branch Prediction wird die Transient Execution ab Phase 2 durch eine falsche Branch Prediction ausgelöst (siehe Kapitel 3.3.4). Außerdem werden die Phasen 2 bis 5 im Kontext des Opfers ausgeführt. [Can+19a, Kap. 2]

Um in Phase 2 eine falsche Branch Prediction zu induzieren, wird in Phase 1 ein *Training* des Branch Predictors durchgeführt. Im Allgemeinen beeinflusst dieses Training den mikroarchitekturellen Zustand des Branch Predictors, um für einen anvisierten Sprung in bestimmter Weise eine falsche Vorhersage zu verursachen. Ein konkretes Training kann beispielsweise darin bestehen, einen anvisierten bedingten Sprung wiederholt mit dem gewünschten Ausgang auszuführen. In diesem Fall wird der Sprung in Phase 2 mit einem anderen Ausgang ausgeführt, während der Branch Predictor weiterhin den trainierten Ausgang voraussagt. Es kommt also zu einer falschen Branch Prediction. [Can+19a, Kap. 3]

Spectre-Angriffe basierend auf Branch Prediction werden weiter unterteilt nach der Art und Weise, auf die der Branch Predictor trainiert wird:

- Dies umfasst einerseits die Art der verwendeten Sprünge. Eine Branch Prediction kann ausgelöst werden durch **bedingte Sprünge**, **indirekte Sprünge** oder **Rücksprünge aus Funktionen**. Jede Art der Branch Prediction kann damit Ziel des Trainings werden. [Can+19a, Kap. 3]

- Da der Branch Predictor virtuelle Adressen verwendet [Koc+19, Kap. V], kann das Training andererseits auch in unterschiedlichen Adressräumen stattfinden, sofern die virtuelle Adresse der zum Training und zum Angriff verwendeten Sprünge übereinstimmt. Erfolgt das Training in dem **gleichen Adressraum** wie die inkorrekte Branch Prediction während des Angriffs, so spricht man von einem *Same-Address-Space* Training. In diesem Fall geschieht das Training typischerweise in dem Kontext des Opfers. Erfolgt das Training in einem **anderen Adressraum** als die inkorrekte Branch Prediction während des Angriffs, so spricht man von einem *Cross-Address-Space* Training. In diesem Fall geschieht das Training typischerweise in dem Kontext des Angreifers. [Can+19a, Kap. 3]

- Der Branch Predictor verwendet außerdem nicht die gesamte virtuelle Adresse eines Sprunges, sondern nur eine feste Anzahl der niederwertigsten Bits [Can+19a, Kap. 3]. Dies ermöglicht

das Training mit einem Sprung an einer **anderen virtuellen Adresse**, die kongruent[1] zu der Adresse des anvisierten Sprunges ist. In diesem Fall spricht man von einem *Out-Of-Place* Training. Erfolgt das Training hingegen an der **gleichen virtuellen Adresse** wie der Angriff, spricht man von einem *In-Place* Training. [Can+19a, Kap. 3]

Zu den Spectre-Angriffen basierend auf Branch Prediction gehören beispielsweise *Spectre Variant 1* [Koc+19, Kap. IV], der bedingte Sprünge verwendet, und *Spectre Variant 2* [Koc+19, Kap. V], der indirekte Sprünge verwendet. Weitere Beispiele sind durch *SpectreRSB* [Kor+18] und *ret2spec* [MR18] gegeben, die Rücksprünge aus Funktionen verwenden. Alle in Kapitel 4.7 beschriebenen Spectre-Angriffe basieren auf Prozessor-Exceptions. Die Untersuchung von zusätzlichen Angriffen basierend auf Branch Prediction würde den Rahmen dieser Bachelorarbeit überschreiten. Aufgrund dessen werden in dieser Arbeit keine konkreten Spectre-Angriffe basierend auf Branch Prediction betrachtet.

4.4 Spectre-Angriffe basierend auf Prozessor-Exceptions

Spectre-Angriffe basierend auf Prozessor-Exceptions werden auch *Meltdown-type Attacks* genannt nach dem Paper, das den ersten Angriff dieser Art beschrieben hat [Lip+18]. Bei Angriffen basierend auf Prozessor-Exceptions wird die Transient Execution ab Phase 2 durch eine Prozessor-Exception ausgelöst (siehe Kapitel 3.3.4). Anders als bei Spectre-Angriffen, die auf Branch Prediction basieren, werden die Phasen 2 bis 5 im Kontext des Angreifers ausgeführt. [Can+19a, Kap. 2]

Phasen 2 und 3 bestehen bei diesen Angriffen aus einem lesenden Speicherzugriff, der eine Prozessor-Exception und damit Transient Execution auslöst. Dieser initiale Speicherzugriff wird dabei so gewählt, dass potentiell Daten des Opfers geladen werden. Auf diese können folgende Instruktionen innerhalb der Transient Execution anschließend zugreifen. Die Details des initialen Speicherzugriffs sind abhängig von dem konkreten betrachteten Angriff. [Can+19a, Kap. 2]

Spectre-Angriffe basierend auf Prozessor-Exceptions werden nach zwei Gesichtspunkten weiter unterteilt:

- Ein Aspekt ist die **Art der ausgelösten Prozessor-Exception**. Dazu zählen alle architekturellen Exceptions, die im Prozessor ausgelöst werden können [Can+19a, Kap. 4]. *Page-Fault* Exceptions treten beispielsweise auf, wenn ein Zugriff auf eine virtuelle Adresse erfolgt, der kein physischer Speicher zugeordnet ist [Int21d, S. 6-44]. *General Protection* Exceptions treten unter anderem auf, wenn auf eine virtuelle Adresse zugegriffen wird, die nicht Teil des 48-Bit Adressraumes ist [Int21d, S. 6-42]. Eine solche virtuelle Adresse wird *nicht-kanonisch* genannt [Int21d, Kap. 6, S. 42]. Neben den architekturellen Exceptions können auch *Microcode Assists* einen Spectre-Angriff ermöglichen [Sch+19b, Kap. 3.2]. Diese werden auch *mikroarchitekturelle Exceptions* genannt [Sch+19b, Kap. 5.1]. Microcode Assists werden durch bestimme Ausnahmezustände in der Mikroarchitektur ausgelöst. Ihr Effekt auf die Mikroarchitektur ist ähnlich zu dem von Prozessor-Exceptions und sie können ebenfalls Transient Execution auslösen. Anders als Prozessor-Exceptions sind Assists jedoch nicht architekturell sichtbar, d.h. sie ändern den architekturellen Zustand nicht. [Can+19b, Kap. 5.1]

[1] Das Konzept kongruenter Adressen aus Kapitel 3.2 lässt sich hier analog anwenden: Zwei Adressen sind kongruent, wenn sie in den vom Branch Predictor verwendeten Bits übereinstimmen.

- Der andere Aspekt ist das **Element der Architektur oder Mikroarchitektur**, aus dem unberechtigt Daten extrahiert werden. Übliche Ziele sind hierbei der L1 Data Cache, der Line-Fill Buffer, der Store Buffer und die Load Ports (siehe Kapitel 3.3.3). [Gru20, S. 73–74] [Can+19a, Erweiterter Klassifizierungsbaum]

Sofern der Angriff Prozessor-Exceptions und nicht Microcode Assists verwendet, müssen die ausgelösten Exceptions behandelt oder unterdrückt werden. Dafür finden verschiedene Techniken Verwendung, die im Folgenden skizziert werden.

Wird eine Prozessor-Exception ausgelöst, senden alle gängigen, modernen Betriebssysteme ein Signal an den aktuellen Prozess. Dieses Signal kann von dem Prozess abgefangen und behandelt werden. Diese Möglichkeit, mit Exceptions umzugehen, wird als *Exception Handling* bezeichnet. Da in diesem Fall zunächst das Betriebssystem die Prozessor-Exception bearbeitet, ist diese Methode langsamer als ihre Alternativen. Dies führt zu zusätzlicher Interferenz durch das Betriebssystem oder dritte Prozesse. [Lip+18, Kap. 4.1]

Eine Alternative zur Behandlung von Prozessor-Exceptions ist die *Exception Suppression*, die entstehende Exceptions unterdrückt. Dadurch erreichen diese nie das Betriebssystem und mögliche Interferenzquellen werden reduziert. Wird die Exception innerhalb einer Intel TSX[2] Transaktion ausgelöst, so wird die Exception unterdrückt und stattdessen die Transaktion abgebrochen. Diese Art der Exception Suppression benötigt jedoch die Intel TSX Prozessorerweiterung. Eine andere Art der Exception Suppression nutzt Transient Execution durch eine falsche Branch Prediction. Wird eine Prozessor-Exception von einer Transient Instruction ausgelöst, so wird diese Exception ebenfalls unterdrückt. Für Exception Suppression durch Transient Execution muss der Branch Predictor entsprechend trainiert werden. Dies erfolgt auf die gleiche Art und Weise wie bei Spectre-Angriffen basierend auf Branch-Prediction, die in Kapitel 4.3 beschrieben wurde. [Lip+18, Kap. 4.1]

In Kapitel 4.7 werden vier konkrete Spectre-Angriffe basierend auf Prozessor-Exceptions detailliert beschrieben. Details der Implementierung von Techniken zur Vermeidung von Prozessor-Exceptions werden in Kapitel 4.6.4 erläutert.

4.5 Angriffsszenarien

Im Kontext von Spectre-Angriffen werden verschiedene Angriffsszenarien unterschieden. Dies umfasst einerseits die vorhandenen Privilegien, die betrachteten Angreifern zugeschrieben werden, sowie andererseits die von diesen Angreifern anvisierten Strukturen.

In der Literatur wurden unter anderem konkrete Angriffe demonstriert, die von Angreifern in den folgenden Kontexten ausgehen:

- Unprivilegierte Nutzerprozesse [Lip+18, Kap. 6.1]
- Betriebssysteme [Sch+19b, Kap. 4]
- JavaScript-Programme in Browsern [Sch+19a, Kap. VI]

[2] *Intel Transactional Synchronization Extensions*, kurz *TSX*, sind Erweiterungen für Intel Prozessoren. Diese erlauben es, mehrere Speicheroperationen innerhalb einer *Transaktion* auszuführen, sodass diese anderen Ausführungskernen als eine einzige atomare Operation erscheinen. [Int21b, Kap. 16, S. 1]

- eBPF-Programme im Linux-Kern [Koc+19, Kap. IV.D]

Die betrachteten Angreifer visierten dabei folgende Ziele an:

- Andere Nutzerprozesse [Lip+18, Kap. 6.1]
- Den Betriebssystem-Kern [Sch+19b, Kap. 4]
- Andere Container [Lip+18, Kap. 6.1]
- Andere Virtuelle Maschinen [Sch+19a, Kap. VI]
- Hypervisors [Sch+19b, Kap. 4]
- Trusted Execution Environments wie Intel SGX [Sch+19a, Kap. VI]

Außerdem wird die Relation der Ausführungskerne, auf denen Angreifer und Opfer ausgeführt werden, betrachtet. Dabei wird zwischen drei Konstellationen unterschieden:

- *Same-Thread*: Angreifer und Opfer werden auf dem gleichen logischen Kern ausgeführt. [Sch+19a, Kap. 1]
- *Cross-Thread*: Angreifer und Opfer werden auf dem gleichen physischen Kern, aber auf unterschiedlichen logischen Kernen ausgeführt. [Sch+19a, Kap. 1]
- *Cross-Core*: Angreifer und Opfer werden auf unterschiedlichen physischen Kernen ausgeführt. [Sch+19a, Kap. 1]

Alle in dieser Arbeit implementierten Angriffe setzen voraus, dass Angreifer und Opfer bestimmte Elemente physischer Kerne gemeinsam verwenden. Damit handelt es sich bei diesen um Cross-Thread Angriffe. Um die hier implementierten Angriffe möglichst allgemein anwendbar zu halten, werden außerdem nur Angriffe betrachtet, die von unprivilegierten Nutzerprozessen auf Linux-Systemen durchgeführt werden können. Anvisiert werden dabei andere Nutzerprozesse oder der Betriebssystem-Kern. Zusätzlich werden aus demselben Grund keine Angriffe betrachtet, die Unterstützung für Prozessorerweiterungen wie Intel TSX voraussetzen.

4.6 Wiederverwendbare Elemente der Implementierung von Spectre-Angriffen

Dieses Kapitel beschreibt Elemente der Implementierung von Spectre-Angriffen, die unabhängig von den konkreten Angriffen sind und in mehreren der in Kapitel 4.7 betrachteten Angriffen wiederverwendet werden. Zunächst wird in Kapitel 4.6.1 beschrieben, wie eine häufige Quelle der Interferenz vermieden wird. Danach wird in Kapitel 4.6.2 eine Technik zur Messung der Latenz einzelner Instruktionen erläutert. Anschließend wird in Kapitel 4.6.3 dargestellt, wie Daten in den Cache kodiert und aus diesem dekodiert werden. Schließlich werden in Kapitel 4.6.4 zwei Methoden zum Umgang mit auftretenden Prozessor-Exceptions erklärt.

Alle hier beschriebenen Elemente wurden im Rahmen dieser Arbeit implementiert. Der Programmcode der Implementierung ist in den Dateien `common.c` und `common.h` beigelegt.

4.6.1 Reduzierung der Interferenz durch Wechsel zwischen Ausführungskernen

Werden während eines Spectre-Angriffs der Prozess des Angreifers oder der Prozess des Opfers von dem Betriebssystem auf einen anderen Ausführungskern migriert, so kann dies mit dem laufenden Angriff interferieren. Viele Betriebssysteme erlauben auch unprivilegierten Nutzern, ihre Prozesse an einzelne logische Ausführungskerne zu binden. Dadurch werden diese Prozesse nur auf den gewählten logischen Kernen ausgeführt. Um Interferenzen durch den Wechsel zwischen Ausführungskernen zu vermeiden, werden daher sowohl der angreifende Prozess als auch der Prozess des Opfers an einzelne logische Kerne gebunden.

Den Prozess des Angreifers und den Prozess des Opfers an den gleichen logischen Ausführungskern zu binden, ermöglicht es Same-Thread Angriffen zuverlässig zu funktionieren. Bei Cross-Thread Angriffen hingegen sollten der Prozess des Angreifers und der des Opfers an unterschiedliche logische Kerne des gleichen physischen Kerns gebunden werden, um zuverlässige Funktion zu ermöglichen. Außerdem kann die Stabilität von Cross-Core Angriffen erhöht werden, indem der Prozess des Angreifers und der Prozess des Opfers an einzelne logische Kerne von unterschiedlichen physischen Kernen gebunden werden.

4.6.2 Messung der Latenz einzelner Instruktionen

Um Cache-basierte Seitenkanalangriffe durchzuführen, muss ein Angreifer die Ausführungslatenz einzelner Instruktionen messen können. Wie in Kapitel 4.1 erwähnt, geht dies auf x86 Prozessoren mithilfe der Instruktion `rdtsc` (Read Time-Stamp Counter). Diese Instruktion lädt ein prozessorinternes Register, das mit jedem Prozessorzyklus hochgezählt wird [Int21c, Kap. 4, S. 547].

Um die Messung der Latenz vorzunehmen, genügt es jedoch nicht, die einzelne Instruktion mit `rdtsc` Instruktionen zu umgeben, da die Ausführungsreihenfolge dieser Instruktionen nicht festgelegt ist. Durch Out-Of-Order Execution kann die zu messende Instruktion beispielsweise vor oder hinter beide `rdtsc` Instruktionen geordnet werden, wodurch das Messergebnis bedeutungslos würde. Um dieses Problem zu umgehen, unterstützen x86 Prozessoren besondere Instruktionen. Diese ordnen die Ausführungsreihenfolge von Instruktionen oder von Speicherzugriffen auf festgelegte Weise. Im Folgenden wird zunächst die Messung der Latenz von lesenden Speicherzugriffen betrachtet, wie sie für den Flush+Reload Angriff benötigt wird. Die `clflush` Instruktion, die Cacheeinträge invalidiert, wird anschließend untersucht. Eine Latenzmessung dieser wird für den Flush+Flush Angriff benötigt.

x86 Prozessoren besitzen unter anderem folgende Instruktionen, die die Ausführung anderer Instruktionen ordnen:

- `cpuid` (CPU Identification) ruft Informationen über den Prozessor des Systems ab. Außerdem serialisiert `cpuid` den Instruktionsfluss: Alle ihr vorhergehenden Instruktionen werden ausgeführt und alle Speicherzugriffe abgeschlossen, bevor ihr nachfolgende Instruktionen geladen werden. [Int21c, Kap. 3, S. 198]
- `lfence` (Load Fence) ordnet lesende Speicherzugriffe sowie teilweise auch den gesamten Instruktionsfluss. Bevor `lfence` ausgeführt wird, werden alle vorhergehenden Instruktionen

```
1 cpuid();
2 start = _rdtsc();
3 cpuid();
4 mem_access(target);
5 cpuid();
6 end = _rdtsc();
7 cpuid();
8 latency = end - start;
```

Listing 1: *Latenzmessung eines lesenden Speicherzugriffs, durch cpuid geordnet.*

ausgeführt. Außerdem wird keine nachfolgende Instruktion ausgeführt, bevor `lfence` abgeschlossen wurde. Dadurch unterbricht `lfence` auch spekulative Ausführung. Diese Eigenschaft besitzt nur `lfence`; andere Fence-Instruktionen ordnen den Instruktionsfluss nicht. Da die Ausführung schreibender Instruktionen beendet wird, bevor der von ihnen ausgelöste Speicherzugriff abgeschlossen ist, ordnet `lfence` keine schreibenden Speicherzugriffe. [Int21c, Kap. 3, S. 559]

- `sfence` (Store Fence) ordnet schreibende Speicherzugriffe. Alle vorhergehenden schreibenden Speicherzugriffe werden abgeschlossen, bevor ein nachfolgender schreibender Speicherzugriff abgeschlossen wird. [Int21c, Kap. 4, S. 611]
- `mfence` (Memory Fence) ist ähnlich zu `sfence`, ordnet aber zusätzlich auch lesende Speicherzugriffe. Alle vorhergehenden Speicherzugriffe werden abgeschlossen, bevor ein nachfolgender Speicherzugriff abgeschlossen wird. [Int21c, Kap. 4, S. 22]
- `rdtscp` (Read Time-Stamp Counter and Processor ID) ist in ihrer Funktionsweise ähnlich zu der `rdtsc` Instruktion. Zusätzlich wird `rdtscp` erst ausgeführt, wenn alle vorhergehenden Instruktionen ausgeführt wurden. Damit ist `rdtscp` ähnlich zu `lfence` gefolgt von `rdtsc`. [Int21c, Kap. 4, S. 549]

Eine einfache Möglichkeit, einen lesenden Speicherzugriff gegenüber `rdtsc` Instruktionen zu ordnen, besteht also darin, sämtliche Instruktionen mit `cpuid` zu umgeben. Dies zeigt der Beispielcode in Listing 1. Dabei handelt es sich um C Code, der die von Intel angegebenen Funktionen zur Ausführung der entsprechenden Prozessorinstruktionen verwendet [Int21e]. Für `cpuid` gibt Intel keine Funktion an, daher wird für diese Instruktion eine fiktive Funktion `void cpuid(void)` verwendet. Außerdem wird eine Funktion `void mem_access(const void *)` verwendet, die auf die übergebene Speicheradresse lesend zugreift.

Da Fence-Instruktionen effizienter sind als `cpuid` [Int21d, Kap. 8, S. 16], kann diese Technik der Latenzmessung optimiert werden. Alle Instruktionen und Speicherzugriffe vor dem ersten `rdtsc` sollen vor diesem abgeschlossen werden. Daher wird ein `mfence` gefolgt von einem `lfence` vor dem ersten `rdtsc` platziert. Der zu messende Speicherzugriff soll nach dem ersten `rdtsc` ausgeführt werden, daher wird zwischen diese ein `lfence` platziert. Andererseits soll der zu messende Speicherzugriff vor dem zweiten `rdtsc` ausgeführt werden, daher wird zwischen diese ein weiterer `lfence` platziert. Schließlich soll die zweite `rdtsc` Instruktion vor allen ihr folgenden Instruktionen ausgeführt werden. Aus diesem Grund wird hinter ihr erneut ein `lfence` platziert. Da beide `rdtsc` Instruktionen unmittelbar auf `lfence` Instruktionen folgen, können diese Paare jeweils zu einer `rdtscp` Instruktion kombiniert werden. Insgesamt ergibt sich dadurch der Code in Listing 2.

```
1 _mm_mfence();
2 start = __rdtscp(&unused);
3 _mm_lfence();
4 mem_access(target);
5 end = __rdtscp(&unused);
6 _mm_lfence();
7 latency = end - start;
```

Listing 2: *Latenzmessung eines lesenden Speicherzugriffs, durch Fence-Instruktionen geordnet.*

```
1 _mm_mfence();
2 start = __rdtscp(&unused);
3 _mm_mfence();
4 _mm_clflush(target);
5 _mm_mfence();
6 end = _rdtsc();
7 _mm_lfence();
8 latency = end - start;
```

Listing 3: *Latenzmessung einer clflush Instruktion, durch Fence-Instruktionen geordnet.*

Ohne den Aufruf von `mem_access` benötigt die durch `cpuid` geordnete Latenzmessung (Listing 1) im Schnitt 135 Prozessorzyklen, während die durch Fence-Instruktionen geordnete Latenzmessung (Listing 2) im Schnitt nur 24 Zyklen benötigt. Die optimierte Variante der Latenzmessung führt also zu einer wesentlichen Reduktion der Zeit, die für die Messung selber benötigt wird. Dadurch wird das Zeitfenster für potentielle Interferenz durch dritte Prozesse oder durch das Betriebssystem verringert. Außerdem wird Angriffen auf diese Weise eine höhere Übertragungsrate ermöglicht.

Um die Latenz von `clflush` Instruktionen zu messen, müssen an diesem Code zwei Änderungen vorgenommen werden. `clflush` Instruktionen werden nur durch `mfence` Instruktionen geordnet und nicht durch die vorhandenen `lfence` oder `rdtscp` Instruktionen [Gru+16, Kap. 3]. Aus diesem Grund wird die zweite `rdtscp` Instruktion wieder durch eine `lfence` Instruktion, gefolgt von einer `rdtsc` Instruktion, ersetzt. Anschließend werden die den Speicherzugriff umgebenden `lfence` Instruktionen durch `mfence` Instruktionen ersetzt. Mit diesen Änderungen ergibt sich der in Listing 3 abgedruckte Code.

4.6.3 Cache als Übertragungskanal

Spectre-Angriffe erfordern einen Übertragungskanal aus dem Kontext der Transient Execution in den architekturellen Zustand. Wie in Kapitel 4.2 beschrieben, wird in den hier implementierten Angriffen dafür der Cache verwendet. Um Daten in diesen Kanal zu kodieren, werden ausgewählte Cachezeilen eingeladen. Um die Daten anschließend aus dem Kanal zu extrahieren, wird der Zustand des Caches über einen Cache-basierten Seitenkanalangriff beobachtet.

Bei allen konkreten Angriffen, die in Kapitel 4.7 betrachtet werden, wird ausschließlich der L1d Cache als Übertragungskanal verwendet. Die von diesen Angriffen anvisierten Puffer der Mikroarchitektur werden nicht zwischen mehreren physischen Prozessorkernen geteilt. Folglich können die Angriffe in einem Cross-Core Szenario nicht funktionieren. Da innerhalb eines physischen Kerns der L1d

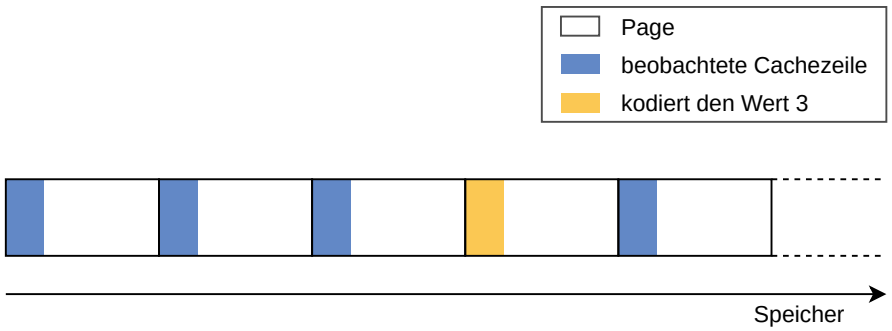

Abbildung 8: *Beobachtete Cachezeilen, jede auf einer eigenen Page.*

Cache der schnellste von logischen Kernen gemeinsam verwendete Cache ist, wird ausschließlich dieser als Übertragungskanal verwendet.

Zwei Aspekte der Implementierung eines solchen Cache-basierten Übertragungskanals werden im Folgenden dargestellt. In Kapitel 5.2 wird dieser Übertragungskanal unter Verwendung von Flush+Reload und Flush+Flush evaluiert.

Klassifizierung von Latenzen

Ein Cache-basierter Seitenkanalangriff muss die gemessenen Latenzen der jeweiligen Instruktion danach klassifizieren, ob sie auf einen Cache Hit oder einen Cache Miss hindeuten (siehe Kapitel 4.1). Um diese Klassifikation vorzunehmen, wird üblicherweise ein fester Grenzwert verwendet. Dieser wird auch *Threshold* genannt. [YF14, Kap. 3]

Der Threshold kann in Vorbereitung auf einen Angriff wie folgt bestimmt werden. Zunächst werden möglichst viele Latenzmessungen für beide möglichen Zustände des Caches durchgeführt, um beschriftete Messergebnisse zu erhalten. Aus diesen Ergebnissen werden anschließend Ausreißer gefiltert. Diese können beispielsweise auftreten, wenn der Prozess während einer Latenzmessung von dem Betriebssystem unterbrochen wird. Nun wird ein geeigneter Threshold gewählt, der den Anteil der korrekt klassifizierten Messungen maximiert. Die Auswahl dieses Thresholds findet üblicherweise manuell statt [YF14, Kap. 3].

In Kapitel 5.2 wird eine solche Bestimmung des Thresholds anhand konkreter Messungen durchgeführt.

Kodierung von Daten im Cache

Üblicherweise kodieren Spectre-Angriffe 8 Bit in den Zustand des Caches, indem eine von 256 Cachezeilen in den Cache geladen wird [Lip+18, Kap. 4.2] [Sch+19a, Kap. IV] [Can+19b, Kap. 3.1]. Um Interferenz durch Optimierungen wie Prefetching zu reduzieren, wird jede der Cachezeilen auf einer eigenen Page platziert [Int21a, Kap. 3, S. 61–62]. Der verwendete Cache-basierte Seitenkanalangriff beobachtet diese 256 Cachezeilen. Wird eine Cachezeile als im Cache vorhanden erkannt, so gibt ihr Index die übertragenen 8 Bit an. Abbildung 8 illustriert die beobachteten Cachezeilen und zeigt beispielhaft die Cachezeile, die bei Kodierung des Wertes 3 eingeladen wird.

Auf analoge Art und Weise kann eine beliebige Anzahl an Möglichkeiten in diesem Übertragungskanal kodiert und aus ihm extrahiert werden.

4.6.4 Behandlung und Unterdrückung von Prozessor-Exceptions

Wie in Kapitel 4.4 beschrieben, müssen Spectre-Angriffe ausgelöste Prozessor-Exceptions entweder behandeln oder unterdrücken.

Um Prozessor-Exceptions zu behandeln, registriert der angreifende Prozess einen *Signal Handler* mittels der Funktion `sigaction` [Lin21e]. Vor Auslösen der Exception speichert der Prozess seinen aktuellen Ausführungszustand mithilfe der Funktion `setjmp` [Lin21d]. Nach Auslösen der Exception wird dieser Zustand durch die Funktion `longjmp` wiederhergestellt [Lin21d].

Um Prozessor-Exceptions hingegen zu unterdrücken, führt der angreifende Prozess die auslösende Instruktion während einer Transient Execution aus. Diese Transient Execution folgt dabei auf die falsche Vorhersage eines Sprunges durch den Branch Predictor. Das dafür erforderliche Training des Branch Predictors erfolgt wie bei Spectre-Angriffen basierend auf Branch Prediction. Die Art und Weise dieses Trainings wurde bereits in Kapitel 4.3 behandelt. Der einzige Unterschied des hier vorliegenden Anwendungsfalles zu dem aus Kapitel 4.3 besteht darin, dass hier die falsche Vorhersage im Kontext des Angreifers ausgelöst wird. Da der Angreifer in diesem Kontext die Kontrolle über jeglichen ausgeführten Code hat, kann er stets ein Same-Address-Space In-Place Training durchführen. Die Art der verwendeten Sprünge kann weiterhin variiert werden. In Kapitel 5.3 wird diese Technik zur Unterdrückung von Prozessor-Exceptions evaluiert unter der Verwendung von bedingten Sprüngen, indirekten Sprüngen und Rücksprüngen aus Funktionen.

Die andere in Kapitel 4.4 beschriebene Technik zur Unterdrückung von Prozessor-Exceptions setzt voraus, dass der Prozessor die Intel TSX Prozessorerweiterung unterstützt. Wie in Kapitel 4.5 erwähnt, werden in dieser Arbeit keine Angriffe betrachtet, die eine Unterstützung für Intel TSX voraussetzen. Daher wird eine Implementierung dieser Technik hier nicht behandelt.

4.7 Funktionsweise und Details der Implementierung ausgewählter Spectre-Angriffe

In diesem Kapitel werden ausgewählte konkrete Spectre-Angriffe beschrieben. Dabei wird sowohl auf ihre Funktionsweise als auch auf Details ihrer Implementierung eingegangen. Alle behandelten Angriffe sind Spectre-Angriffe basierend auf Prozessor-Exceptions (siehe Kapitel 4.4).

Wie in Kapitel 4.5 bereits erläutert, werden alle hier behandelten Angriffe von unprivilegierten Nutzerprozessen auf einem aktuellen Linux-System durchgeführt. Anvisiert werden dabei entweder andere Nutzerprozesse oder der Betriebssystem-Kern. Außerdem werden keine Angriffe betrachtet, die Unterstützung für Prozessorerweiterungen wie Intel TSX voraussetzen.

Alle hier beschriebenen Angriffe wurden im Rahmen dieser Arbeit implementiert. Der Programmcode der Implementierungen ist in den Dateien `ridl.c`, `wtf.c`, `storetoleak.c` und `zombieload.c` beigelegt.

```
// Vorbereitung
uint8_t *target = alloc_page();
evict_page(target);

// Angriff
uint8_t leak = *target;
encode_in_cache(leak);

// Rekonstruktion
uint8_t data = decode_from_cache();
```

Listing 4: *Vereinfachte Implementierung von RIDL.*

4.7.1 RIDL: Rogue In-Flight Data Load

Rogue In-Flight Data Load, kurz *RIDL*, ist ein Spectre-Angriff basierend auf Prozessor-Exceptions (siehe Kapitel 4.4). RIDL verwendet Page-Fault Exceptions, um unberechtigt auf Daten aus dem Line-Fill Buffer oder den Load Ports zuzugreifen. [Sch+19a, Kap. IV, VI]

Phase 2 dieses Spectre-Angriffs besteht aus einem lesenden Speicherzugriff, durch den eine Page-Fault Exception ausgelöst wird. Wie in Kapitel 3.3.4 beschrieben, löst dies eine Transient Execution aus, bevor die Exception bei Retirement des Speicherzugriffs behandelt wird. Trotz der Exception können folgende Transient Instructions die Daten des initialen Speicherzugriffs verwenden. An dieser Stelle wird zwischen zwei Varianten von RIDL unterschieden, die in der Art des initialen Speicherzugriffs voneinander abweichen. Lädt dieser Speicherzugriff eine einzelne Cachezeile ein, so werden die resultierenden Daten aus dem Line-Fill Buffer bereitgestellt. Lädt der Speicherzugriff jedoch von mehreren Cachezeilen, so werden die resultierenden Daten aus einem der Load Ports bereitgestellt. In beiden Fällen werden diese Daten ohne jegliche Prüfung bereitgestellt und unabhängig davon, ob sie aus dem angreifenden Prozess selber, aus anderen Prozessen des Systems oder sogar aus dem Betriebssystem-Kern stammen. [Sch+19a, Kap. VI]

Insgesamt erlaubt dies einem Angreifer, den gesamten Inhalt des Line-Fill Buffers und der Load Ports zu lesen. Da der Line-Fill Buffer und die Load Ports Teil eines physischen Kerns sind und von logischen Kernen gemeinsam verwendet werden, kann ein Angreifer folglich die Daten von Speicherzugriffen anderer Prozesse auf dem gleichen physischen Kern beobachten. [Sch+19a, Kap. VI]

Die im Rahmen dieser Arbeit erstellte Implementierung von RIDL nutzt den L1d Cache als Übertragungskanal, wie in Kapitel 4.6.3 beschrieben. Der initiale Speicherzugriff greift auf eine ausgelagerte Page zu. Auf diese Weise wird eine Page-Fault Exception ausgelöst, ohne dass das Betriebssystem dem angreifenden Prozess anschließend ein Signal schickt. Alternativ kann der initiale Speicherzugriff auf eine virtuelle Adresse ohne Zuordnung zu physischem Speicher erfolgen. In diesem Fall muss die auftretende Exception behandelt oder unterdrückt werden, wie in Kapitel 4.6.4 beschrieben.

Listing 4 zeigt eine stark vereinfachte Implementierung von RIDL. Diese bereitet den Angriff vor, extrahiert ein Byte aus dem Line-Fill Buffer und überträgt es durch den Cache in den architekturellen Zustand.

In Kapitel 5.4.1 werden beide Varianten von RIDL evaluiert vor dem Hintergrund Cross-Thread Daten aus einem anderen Prozess zu laden.

4.7.2 Fallout

Unter dem Begriff *Fallout* werden zwei unterschiedliche Angriffe zusammengefasst. Der erste dieser Angriffe ist *Write Transient Forwarding*, kurz *WTF*, der es ermöglicht Daten aus dem Store Buffer zu lesen [Can+19b, Kap. 3.1]. Bei dem zweiten Angriff handelt es sich um *Store-to-Leak*, der es ermöglicht, die Anwesenheit von Speicherzuordnungen an gewählten virtuellen Adressen festzustellen [Can+19b, Kap. 3.2].

Beide Angriffe basieren auf einer Optimierung, die *Store-To-Load Forwarding* genannt wird. Diese optimiert lesende Speicherzugriffe auf Adressen, in die kürzlich geschrieben wurde: Der lesende Speicherzugriff lädt die Daten direkt aus dem Store Buffer (siehe Kapitel 3.3.3) Eintrag des vorhergehenden schreibenden Speicherzugriffs. Auf diese Weise muss der lesende Speicherzugriff nicht tatsächlich auf den Speicher zugreifen. [Can+19b, Kap. 2.3]

Write Transient Forwarding

Write Transient Forwarding ist ein Spectre-Angriff basierend auf Prozessor-Exceptions (siehe Kapitel 4.4). Er verwendet unter anderem General Protection oder Page-Fault Exceptions, um unberechtigt auf Daten aus dem Store Buffer zuzugreifen [Can+19b, Kap. 3.1, 5.2].

Write Transient Forwarding basiert darauf, dass Store-to-Load Forwarding fälschlicherweise ausgelöst wird, wenn die Adressübersetzung des lesenden Speicherzugriffs fehlschlägt. Dies passiert beispielsweise, wenn der Speicherzugriff eine General Protection Exception durch Verwendung einer nicht-kanonischen Adresse oder eine Page-Fault Exception durch Verwendung einer Adresse des Betriebssystem-Kerns verursacht. Wie in Kapitel 3.3.4 beschrieben, löst diese Exception außerdem eine Transient Execution aus. Aufgrund des Store-to-Load Forwarding nimmt der lesende Speicherzugriff Daten aus dem Store Buffer auf. Diese Daten werden folgenden Transient Instructions zur Verfügung gestellt. [Can+19b, Kap. 5.2]

Insgesamt erlaubt dies einem Angreifer, den gesamten Inhalt des Store Buffers zu lesen. Da der Store Buffer Teil eines physischen Kerns ist und von logischen Kernen gemeinsam verwendet wird, kann ein Angreifer folglich die Daten von schreibenden Speicherzugriffen anderer Prozesse auf dem gleichen physischen Kern beobachten. [Can+19b, Kap. 5.2]

Die im Rahmen dieser Arbeit erstellte Implementierung des Write Transient Forwarding nutzt den L1d Cache als Übertragungskanal, wie in Kapitel 4.6.3 beschrieben. Der initiale lesende Speicherzugriff dereferenziert eine nicht-kanonische Adresse. Alternativ kann eine Adresse des Betriebssystem-Kerns verwendet werden. Die dabei auftretende Prozessor-Exception muss behandelt oder unterdrückt werden. Dies erfolgt mit einer der in Kapitel 4.6.4 beschriebenen Techniken.

Listing 5 zeigt eine stark vereinfachte Implementierung von Write Transient Forwarding. Diese extrahiert ein Byte aus dem Store Buffer und überträgt es durch den Cache in den architekturellen Zustand.

```
1 // Angriff
2 uint8_t leak = *(uint8_t *)0x4141414141410000;
3 encode_in_cache(leak);
4
5 // Rekonstruktion
6 uint8_t data = decode_from_cache();
```

Listing 5: *Vereinfachte Implementierung von Write Transient Forwarding.*

In Kapitel 5.4.2 wird Write Transient Forwarding evaluiert und benutzt, um Cross-Thread Daten aus einem anderen Prozess zu laden.

Store-to-Leak

Store-to-Leak unterscheidet sich von den bisher beschriebenen Angriffen, da er keine Daten aus einem der üblichen Puffer der Mikroarchitektur extrahiert. Stattdessen benutzt Store-to-Leak eine Eigenschaft des Store-to-Load Forwarding, um die Anwesenheit von Einträgen im TLB zu beobachten. [Can+19b, Kap. 3.2]

Store-to-Load Forwarding erfolgt nur, wenn zu der virtuellen Adresse des lesenden Speicherzugriffs ein valider Eintrag im TLB existiert. Zusätzlich erfolgt Store-to-Load Forwarding auch bei Zugriffen auf den Speicherbereich den Betriebssystem-Kerns. In diesem Fall löst ein schreibender Speicherzugriff eine Prozessor-Exception aus. In der folgenden Transient Execution kann ein lesender Speicherzugriff durchgeführt und anschließend erkannt werden, ob zwischen den beiden Zugriffen Store-to-Load Forwarding stattgefunden hat. Ein Angreifer kann folglich herausfinden, ob für eine gewählte virtuelle Adresse des Betriebssystem-Kerns ein TLB Eintrag existiert. Durch einen vorhergehenden Speicherzugriff kann außerdem sichergestellt werden, dass ein solcher TLB Eintrag existiert, falls der gewählten virtuellen Adresse physischer Speicher zugeordnet ist. [Can+19b, Kap. 3.2, 5.3]

Die im Rahmen dieser Arbeit erstellte Implementierung von Store-to-Leak nutzt den L1d Cache als Übertragungskanal, wie in Kapitel 4.6.3 beschrieben. Um zu die Anwesenheit einer Speicherzuordnung an einer gewählten virtuellen Adresse zu detektieren, wird wie folgt vorgegangen. Zunächst wird ein fest gewählter Wert an diese Adresse geschrieben. Da es sich um eine Adresse aus dem Bereich des Betriebssystem-Kerns handelt, löst dies eine Prozessor-Exception aus. Innerhalb der folgenden Transient Execution wird ein Wert von der gewählten Adresse geladen und in den Cache-basierten Übertragungskanal kodiert. Existiert ein valider TLB Eintrag für die gewählte Adresse, wird Store-to-Load Forwarding ausgelöst. In diesem Fall wird der zuvor geschriebene Wert in den Cache kodiert. Existiert hingegen kein valider TLB Eintrag, so geschieht kein Store-to-Load Forwarding und es wird kein Wert in den Cache kodiert. Die auftretende Prozessor-Exception wird mit einer der Techniken aus Kapitel 4.6.4 behandelt oder unterdrückt. Auf diese Weise lässt sich die Anwesenheit eines validen TLB Eintrages für die gewählte virtuelle Adresse detektieren. Um die Anwesenheit einer Speicherzuordnung zu beobachten, kann diese Technik zweimal hintereinander ausgeführt werden. [Can+19b, Kap. 3.2, 5.3]

Listing 6 zeigt eine stark vereinfachte Implementierung von Store-to-Leak. Diese schreibt ein festes Byte an die anvisierte Adresse, liest ein Byte von dieser Adresse und überträgt das gelesene Byte

```
1 // Angriff
2 volatile uint8_t *target = TARGET_ADDR;
3 *target = 0x42;
4 uint8_t leak = *target;
5 encode_in_cache(leak);
6
7 // Rekonstruktion
8 uint8_t data = decode_from_cache();
```

Listing 6: *Vereinfachte Implementierung von Store-to-Leak.*

durch den Cache in den architekturellen Zustand. Stimmt das gelesene mit dem geschriebenen Byte überein, so deutet dies auf die Existenz eines TLB Eintrages für die anvisierte Adresse hin.

In Kapitel 5.5 wird Store-to-Leak evaluiert und verwendet, um die virtuelle Adresse des Betriebssystem-Kerns zu bestimmen und dadurch KASLR zu brechen.

4.7.3 ZombieLoad

ZombieLoad ist ein Spectre-Angriff basierend auf Prozessor-Exceptions (siehe Kapitel 4.4). ZombieLoad verwendet Microcode Assists, um unberechtigt auf Daten aus dem Line-Fill Buffer zuzugreifen. [Sch+19b, Kap. 3.1]

Wenn ein lesender Speicherzugriff einen Microcode Assist erfordert, lädt dieser zunächst fälschlicherweise veraltete Daten aus dem Line-Fill Buffer. Nachdem die durch den Microcode Assist ausgelöste Transient Execution beendet ist, wird der Speicherzugriff erneut getätigt. Dabei löst er keinen Microcode Assist mehr aus und lädt die korrekten Daten ein. Wie in Kapitel 4.4 dargestellt, kann die Transient Execution genutzt werden, um die fälschlicherweise geladenen Daten in einen Cache-basierten Übertragungskanal zu kodieren. [Sch+19b, Kap. 3.2]

Insgesamt erlaubt dies einem Angreifer, den gesamten Inhalt des Line-Fill Buffers zu lesen. Da der Line-Fill Buffer Teil eines physischen Kerns ist und von logischen Kernen gemeinsam verwendet wird, kann ein Angreifer folglich die Daten von Speicherzugriffen anderer Prozesse auf dem gleichen physischen Kern beobachten. [Sch+19b, Kap. 3.2]

Varianten von ZombieLoad

Es werden drei Varianten von ZombieLoad unterschieden. Um einen Angriff der ersten Variante durchzuführen, benötigt ein Angreifer eine Page, die in den Adressbereich des Nutzer-Prozesses und in den Adressbereich des Betriebssystem-Kerns eingebunden ist. Diese Situation ist in Abbildung 9 dargestellt. Da Linux über eine Direct-Physical Map verfügt, ist diese Voraussetzung auf Linux für jede Page des Nutzer-Prozesses erfüllt (siehe Kapitel 3.1.2). Der Angreifer invalidiert nun eine Cachezeile der gewählten Page über die zugehörige virtuelle Adresse im Bereich des Nutzer-Prozesses. Unmittelbar danach greift der Angreifer lesend auf diese Cachezeile zu. Der Zugriff erfolgt dabei über die zugehörige virtuelle Adresse im Bereich des Betriebssystem-Kerns. Auf diese Weise durchgeführt, löst der lesende Zugriff einen Microcode Assist aus und lädt wie bereits beschrieben zunächst Daten aus dem Line-Fill Buffer. Diese Daten können über einen Cache-basierten Übertragungskanal extrahiert werden. [Sch+19b, Kap. 5.1]

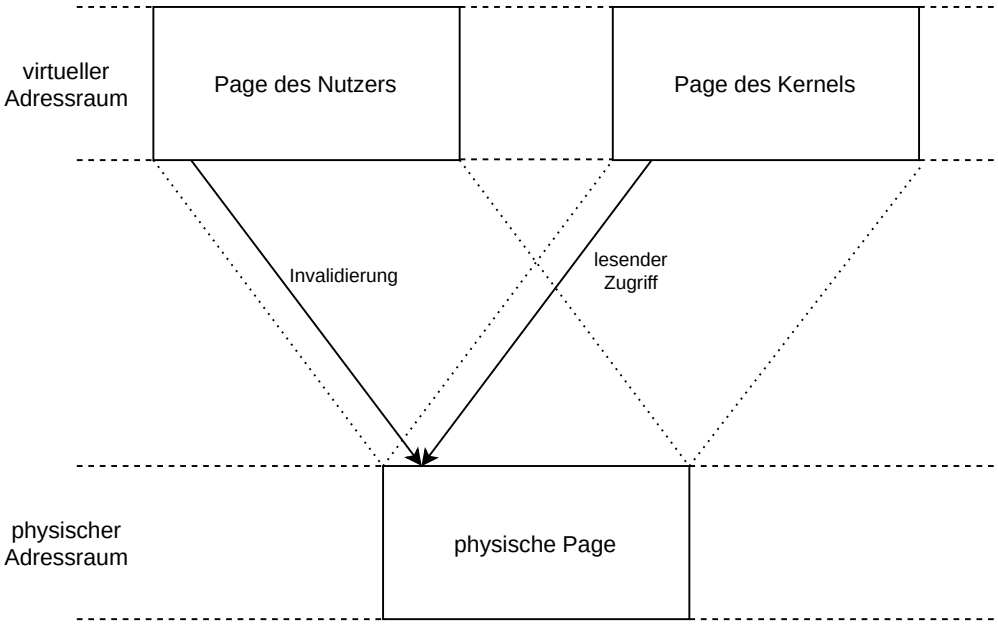

Abbildung 9: *Situation und Vorgehen eines Angriffs auf die erste ZombieLoad-Variante.*

Bei einem Angriff der zweiten Variante führt ein Angreifer einen ladenden Speicherzugriff innerhalb einer Intel TSX Transaktion durch. Gleichzeitig schreibt der Angreifer von einem anderen logischen Kern aus an die gleiche Adresse. Dies führt zu einem Konflikt in dem *Read Set* der Transaktion und damit zu einem Abbruch der Transaktion. Der Effekt dieses Abbruches ist ähnlich zu dem eines Microcode Assists. Insbesondere lädt der Speicherzugriff innerhalb der abbrechenden Transaktion fälschlicherweise aus dem Line-Fill Buffer. Die geladenen Daten können über einen Cache-basierten Übertragungskanal extrahiert werden. [Sch+19b, Kap. 5.1]

Ein Angriff der dritten Variante löst eine Adressübersetzung aus, die bestimmte Bits in einer der Zuordnungstabellen (siehe Kapitel 3.1) ändert. Diese Änderung wird durch einen Microcode Assist durchgeführt. Löst also ein lesender Speicherzugriff eine solche Adressübersetzung aus, so lädt dieser wie bereits beschrieben zunächst Daten aus dem Line-Fill Buffer. Diese Daten können über einen Cache-basierten Übertragungskanal extrahiert werden. [Sch+19b, Kap. 5.1]

Zur Implementierung von ZombieLoad

Die im Rahmen dieser Arbeit erstellte Implementierung der ersten ZombieLoad Variante nutzt den L1d Cache als Übertragungskanal, wie in Kapitel 4.6.3 beschrieben. Zunächst wird eine Page im Adressbereich des Nutzer-Prozesses allokiert. Anschließend wird die Adresse dieser Page in der Direct-Physical Map des Betriebssystem-Kerns ermittelt. Diese setzt sich aus der Basisadresse der Direct-Physical Map und der physischen Adresse der allokierten Page zusammen. In dieser Implementierung wird eine feste Adresse der Direct-Physical Map vorausgesetzt. Die physische Adresse der allokierten Page wird über die Datei `/proc/self/pagemap` ermittelt, auf die standardmäßig nur privilegierte Nutzer zugreifen können. Alternativ können die benötigten Adressen durch verschiedene Seitenkanäle ermittelt werden [Can+19b, Kap. 6.1] [Kos+20]. Damit ist eine virtuelle Adresse im Bereich des Nutzer-Prozesses und eine virtuelle Adresse im Bereich des Betriebssystem-Kerns bekannt, denen die gleiche physische Adresse zugeordnet ist. Die Voraussetzungen für die erste

```
1  // Vorbereitung
2  uint8_t *target = alloc_page();
3  uint8_t *target_kernel = get_kernel_mapping(target);
4
5  // Angriff
6  _mm_clflush(target);
7  uint8_t leak = *target_kernel;
8  encode_in_cache(leak);
9
10 // Rekonstruktion
11 uint8_t data = decode_from_cache();
```

Listing 7: *Vereinfachte Implementierung von ZombieLoad.*

ZombieLoad Variante sind also gegeben und der Angriff kann wie bereits beschrieben durchgeführt werden. Da dabei ein unberechtigter Zugriff auf eine Adresse des Betriebssystem-Kerns erfolgt, tritt eine Prozessor-Exception auf. Diese wird wie in Kapitel 4.6.4 beschrieben behandelt oder unterdrückt.

Listing 7 zeigt eine stark vereinfachte Implementierung von ZombieLoad. Diese bereitet den Angriff vor, extrahiert ein Byte aus dem Line-Fill Buffer und überträgt es durch den Cache in den architekturellen Zustand.

Eine Implementierung der zweiten ZombieLoad Variante setzt Unterstützung für die Intel TSX Prozessorerweiterungen voraus. Ein Angriff der dritten Variante lässt sich nicht von unprivilegierten Nutzerprozessen auf Linux-Systemen durchführen. Aus diesen Gründen werden beide Varianten in dieser Arbeit nicht implementiert (siehe Kapitel 4.5).

In Kapitel 5.4.3 wird die erste ZombieLoad Variante verwendet, um Cross-Thread Daten aus einem anderen Prozess zu laden, und daran evaluiert.

4.7.4 Zusammenfassung

In diesem Kapitel wurden die Funktionsweise und Details der Implementierung ausgewählter konkreter Spectre-Angriffe beschrieben. Bei den Angriffen handelt es sich um RIDL, Write Transient Forwarding, Store-to-Leak und ZombieLoad. RIDL verwendet Page-Fault Exceptions, um unberechtigt auf Daten aus dem Line-Fill Buffer oder den Load Ports zuzugreifen. Write Transient Forwarding verwendet unter anderem General Protection oder Page-Fault Exceptions, um unberechtigt auf Daten aus dem Store Buffer zuzugreifen. Store-to-Leak benutzt eine Eigenschaft des Store-to-Load Forwarding, um die Anwesenheit von Einträgen im TLB zu beobachten. ZombieLoad verwendet Microcode Assists, um unberechtigt auf Daten aus dem Line-Fill Buffer zuzugreifen. Diese Spectre-Angriffe werden in Kapitel 5.4 und Kapitel 5.5 hinsichtlich einheitlicher Metriken evaluiert.

5 Evaluation

In diesem Kapitel werden die in Kapitel 4 beschriebenen Techniken und Angriffe in verschiedenen Varianten evaluiert. Zunächst wird in Kapitel 5.1 das System beschrieben, auf dem diese Evaluation durchgeführt wird. Dabei handelt es sich um ein aktuelles Linux-System, auf dem Maßnahmen gegen Spectre-Angriffe deaktiviert wurden. Anschließend werden in Kapitel 5.2 die Cache-basierten Seitenkanalangriffe betrachtet, die in Kapitel 4.1 beschrieben wurden. Diese Seitenkanalangriffe werden evaluiert hinsichtlich ihrer Fähigkeit, den Zustand von Cachezeilen korrekt zu klassifizieren. Danach werden in Kapitel 5.3 die Techniken, die Transient Execution durch eine falsche Vorhersage des Branch Predictors auslösen, hinsichtlich ihrer Zuverlässigkeit evaluiert. Diese wurden in Kapitel 4.6.4 beschrieben. Schließlich evaluiert Kapitel 5.4 verschiedene Varianten von drei der konkreten Spectre-Angriffe, die in Kapitel 4.7 beschrieben wurden. Dabei handelt es sich um RIDL, Write Transient Forwarding und ZombieLoad. Dies geschieht hinsichtlich der erreichten Datenrate mit zugehöriger Erfolgsrate, sowie der Dauer des Angriffs. Innerhalb jedes Unterkapitels wird zunächst die Vorgehensweise der Evaluation beschrieben. Anschließend werden die ermittelten Ergebnisse dargestellt, um schließlich einen Vergleich mit Ergebnissen anderer Arbeiten zu ziehen. Der verbleibende Angriff aus Kapitel 4.7, Store-to-Leak, unterscheidet sich in seiner Funktionsweise von den anderen drei Angriffen. Deshalb wird Store-to-Leak separat in Kapitel 5.5 evaluiert.

Der Programmcode aller erstellten Implementierungen ist dieser Bachelorarbeit beigelegt. Diese und weitere beigelegte Dateien, sowie benötigte externe Abhängigkeiten sind in der Datei `README.md` aufgelistet. Außerdem enthält `run.sh` ein Programm, das alle zur Evaluation verwendeten Daten erhebt. `eval_and_plot.py` enthält ein Programm, das die erhobenen Daten auswertet, sowie alle in diesem Kapitel verwendeten Diagramme und Tabellen erstellt.

5.1 Hard- und Softwareumgebung

Das System, auf dem die Evaluation durchgeführt wird, basiert auf einem Thinkpad E580. Als Prozessor kommt ein Intel Core i5-8250U der Kaby Lake R Mikroarchitektur zum Einsatz. Dieser besitzt 4 physische und 8 logische Prozessorkerne. Die L1-Caches für Instruktionen und für Daten sind jeweils 8-Way Set-Associative mit einer Kapazität von 32 KiB pro physischem Prozessorkern. Die L2-Caches sind 4-Way Set-Associative mit einer Kapazität von 256 KiB pro physischem Prozessorkern. Der L3-Cache ist 12-Way Set-Associative mit einer Kapazität von 6 MiB und wird von allen physischen Prozessorkernen gemeinsam verwendet[1]. Das System verfügt über 16 GiB Hauptspeicher. Als Betriebssystem wird Manjaro Linux mit einem Kernel der Version `5.10.18-1` eingesetzt. Dieser wurde mit den zusätzlichen Parametern `nokaslr` und `mitigations=off` gestartet [Lin21f], um KASLR (siehe Kapitel 3.1.2) und Maßnahmen gegen Spectre-Angriffe zu deaktivieren.

[1] Die Größe und Assoziativität der Caches wurde über das Kommando `lscpu` und die Dateien `/sys/devices/system/cpu/cpu0/cache/index*/ways_of_associativity` ermittelt.

Insbesondere ist Hyper-Threading aktiviert (siehe Kapitel 3.3). Das neueste Mikrocode-Update der Version `0xE0` ist installiert. Da viele Maßnahmen gegen Spectre-Angriffe durch das Betriebssystem deaktiviert wurden, ist das System dennoch angreifbar. Als *Governor* der Prozessorfrequenz wird der `performance` Governor eingesetzt [Lin21b]. Die relevanten installierten Softwarepakete sind `clang` in der Version `11.1.0` und die GNU C Library in der Version `2.33`. Die Evaluation wird mit minimaler Systemlast durchgeführt; neben den notwendigen Systemprogrammen werden keine weiteren Programme gestartet.

5.2 Cache-basierte Seitenkanalangriffe

In diesem Kapitel werden Flush+Reload und Flush+Flush als Cache-basierte Seitenkanalangriffe evaluiert. Diese wurden in Kapitel 4.1 und Kapitel 4.6.3 beschrieben.

Zum Zweck der Evaluation wird ein fester Wert wiederholt in den Cache kodiert und anschließend mithilfe des zu evaluierenden Seitenkanalangriffs dekodiert. Anstelle wie in Kapitel 4.6.3 beschrieben jede der 256 beobachteten Cachezeilen als im Cache oder nicht im Cache zu klassifizieren, wird dabei direkt die gemessene Zugriffslatenz erfasst. Von zwei ausgewählten Cachezeilen wird diese Latenz protokolliert: Von der Cachezeile, die sich tatsächlich im Cache befindet, sowie von einer beliebigen derer, die sich nicht im Cache befinden. Auf diese Art und Weise werden jeweils zehn Millionen Zugriffslatenzen erfasst. Programmcode, der die Messungen wie beschrieben durchführt, ist dieser Arbeit in der Datei `sidechannel.c` beigelegt.

Ergebnisse

Abbildung 10 zeigt ein Histogramm der gemessenen Zugriffslatenzen für beide Seitenkanalangriffe (Flush+Reload und Flush+Flush) und jeweils beide Kategorien von Cachezeilen (im Cache und nicht im Cache). Die gemessenen Zugriffslatenzen beinhalten dabei die Zeit, die für die Messung selber benötigt wird (siehe Kapitel 4.6.2). Auf Grundlage dieses Histogramms wird manuell ein geeigneter Threshold gewählt, wie in Kapitel 4.6.3 beschrieben. Dieser Threshold ist in Abbildung 10 als vertikale Linie eingezeichnet.

Tabelle 1: *Der gewählte Threshold und die ermittelten Metriken für jeden Seitenkanalangriff, angegeben auf 4 gültige Ziffern.*

Seitenkanalangriff	Threshold (Zyklen)	Sensitivität (%)	Spezifität (%)
Flush+Reload	100	100,0	99,99
Flush+Flush	124	49,68	93,33

Die ermittelten Thresholds sind außerdem in Tabelle 1 angegeben. Diese zeigt auch die *Sensitivität* und *Spezifität*, mit der der jeweilige Seitenkanalangriff den Zustand einer Cachezeile klassifiziert. Die Sensitivität (auch Richtig-positiv-Rate) bezeichnet den Anteil der Cachezeilen im Cache, die korrekt als solche erkannt werden [Tha20, Kap. 2.3]. Alternativ kann die Sensitivität auch interpretiert werden als die Wahrscheinlichkeit, dass eine Cachezeile, die sich im Cache befindet, richtig erkannt

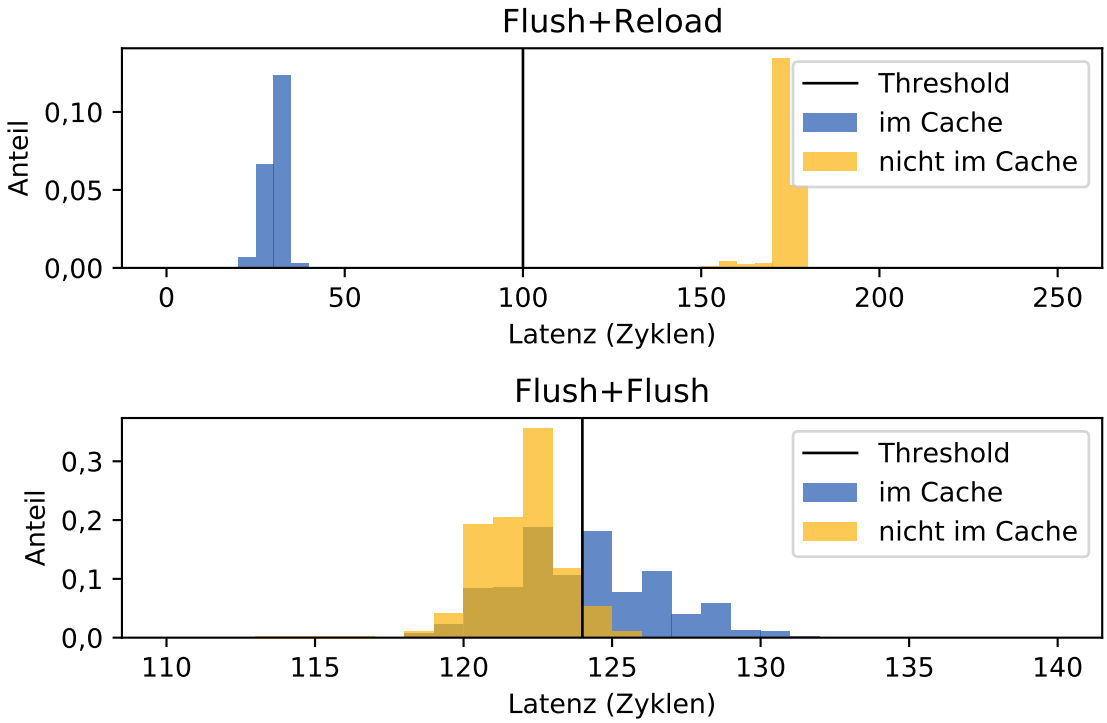

Abbildung 10: *Histogramm der gemessenen Latenzen für die ausgewählten Seitenkanalangriffe.*

wird. Die Spezifität (auch Richtig-negativ-Rate) bezeichnet den Anteil der Cachezeilen nicht im Cache, die korrekt als solche erkannt werden [Tha20, Kap. 2.3]. Alternativ kann die Spezifität auch interpretiert werden als die Wahrscheinlichkeit, dass eine Cachezeile, die sich nicht im Cache befindet, richtig erkannt wird.

Anstelle der Sensitivität und Spezifität werden üblicherweise die *Treffergenauigkeit* (engl. *accuracy*) und die *Genauigkeit* (engl. *precision*) verwendet, um binäre Klassifikatoren zu beurteilen [Tha20, Kap. 2.1, 2.2]. Diese Metriken können hier jedoch nicht sinnvoll angewandt werden, da sie davon beeinflusst werden, wie balanciert die klassifizierten Datensätze sind [Tha20, Kap. 2.1]. Ein Datensatz wird *balanciert* genannt, wenn dieser ähnlich viele tatsächlich positive wie tatsächlich negative Messpunkte beinhaltet [Tha20, Kap. 2.1]. Dem gegenüber wird ein Datensatz, der wesentlich mehr tatsächlich positive oder tatsächlich negative Messpunkte beinhaltet, *unbalanciert* genannt [Tha20, Kap. 2.1]. Im Fall der hier betrachteten Seitenkanalangriffe hängt die Balance der gemessenen Daten von dem konkreten Anwendungsfall ab. Um die berechneten Metriken für verschiedene Anwendungsfälle verwenden zu können, werden hier die Sensitivität und die Spezifität erfasst. Diese Metriken sind unabhängig von der Balance der gemessenen Daten.

Auftretende Phänomene

Wie in Abbildung 10 zu erkennen ist, haben bei Verwendung von Flush+Reload die beiden Verteilungen für Cachezeilen, die im Cache sind, und Cachezeilen, die nicht im Cache sind, einen Abstand von ungefähr 150 Zyklen. Ein Latenzmessung lässt sich also in den meisten Fällen eindeutig einer der beiden Verteilungen zuordnen. Dies deckt sich mit der hohen Sensitivität und Spezifität, die für

Flush+Reload berechnet wurde. Außerdem hat dieser Abstand Auswirkungen auf die Ausführungszeit von Flush+Reload. Ein höheres Verhältnis von beobachteten Cache Misses zu beobachteten Cache Hits erhöht die Ausführungszeit des Angriffs.

Bei Verwendung von Flush+Flush hingegen liegen die Verteilungen für Cachezeilen, die im Cache sind, und Cachezeilen, die nicht im Cache sind, mit einem Abstand von ungefähr 2 Zyklen nah beieinander. Eine Latenzmessung lässt sich also weniger zuverlässig einer der beiden Verteilungen zuordnen. Dies deckt sich mit der reduzierten Sensitivität und Spezifität von Flush+Flush im Vergleich zu Flush+Reload. Insbesondere die niedrige Sensitivität von Flush+Flush stimmt damit überein, dass die Verteilung der Cachezeilen im Cache zu einem großen Teil unter dem Threshold liegt. Da im Fall von Flush+Flush Cache Hits und Cache Misses ähnliche Latenzen haben, wird die Ausführungszeit von Flush+Flush kaum von dem Verhältnis von beobachteten Cache Misses zu beobachteten Cache Hits beeinflusst. Werden wesentlich weniger Cache Hits als Cache Misses erwartet, so ist Flush+Flush folglich schneller als Flush+Reload. Dies ist bei den in Kapitel 5.4 evaluierten Angriffen der Fall, da bei diesen normalerweise genau eine der 256 beobachteten Cachezeilen eingeladen ist (siehe Kapitel 4.6.3).

Vergleich mit anderen Arbeiten

In der Literatur zu Flush+Reload wurde für Cache Hits eine Latenz von 44 Zyklen angegeben, sowie für Cache Misses eine Latenz von 270 bis 290 Zyklen mit wenigen Ausreißern um 1 000 Zyklen [YF14, Kap. 3]. In dieser Arbeit wurde für 99,97 % der Cache Hits eine Latenz von unter 50 Zyklen gemessen. Außerdem lagen 97,00 % der Cache Misses bei einer Latenz von 150 bis 200 Zyklen und 2,070 % der Cache Misses zwischen 600 und 800 Zyklen. Die Form der Verteilungen aus der Literatur stimmt also mit der Form der hier beobachteten Verteilungen überein. Die unterschiedliche konkrete Anzahl von Zyklen kann durch Unterschiede in der Leistungsfähigkeit der verwendeten Systeme erklärt werden.

In der Literatur zu Flush+Flush wurde eine Distanz von ungefähr 10 Zyklen zwischen den jeweils am häufigsten gemessenen Latenzen von Cache Hits und Cache Misses angegeben [Gru+16, Kap. 3]. Außerdem wurde für alle Messungen eine Latenz zwischen 100 und 200 Zyklen angegeben [Gru+16, Abb. 1]. In dieser Arbeit wurde eine Distanz von 2 Zyklen zwischen den am häufigsten gemessenen Latenzen ermittelt. Zusätzlich lagen 99,99 % der Cache Hits und 99,98 % der Cache Misses bei einer Latenz zwischen 100 und 200 Zyklen. Die Form der Verteilungen aus der Literatur stimmt also mit der Form der hier beobachteten Verteilungen überein. Die unterschiedliche Distanz zwischen den am häufigsten gemessenen Latenzen lässt sich durch Unterschiede in der Mikroarchitektur der verwendeten Prozessoren erklären.

Insgesamt können also die Ergebnisse der Literatur für beide Cache-basierten Seitenkanalangriffe reproduziert werden.

5.3 Unterdrückung von Prozessor-Exceptions

In diesem Kapitel werden verschiedene Techniken, die Transient Execution durch eine falsche Vorhersage des Branch Predictors auslösen, hinsichtlich ihrer Zuverlässigkeit evaluiert. Diese Techniken ermöglichen die Unterdrückung von Prozessor-Exceptions (siehe Kapitel 4.6.4). Wie bereits

in Kapitel 4.3 erwähnt, kann eine falsche Vorhersage des Branch Predictors durch verschiedene Arten von Sprüngen ausgelöst werden: Bedingte Sprünge, indirekte Sprünge und Rücksprünge aus Funktionen. Diese drei Arten werden im Folgenden evaluiert und verglichen.

Wie in Kapitel 4.6.4 beschrieben, wird bei den betrachteten Techniken zunächst der Branch Predictor trainiert[2], um anschließend zuverlässig eine falsche Vorhersage auslösen zu können. Zum Zweck der Evaluation wird der Branch Predictor trainiert, indem der anvisierte Sprung wiederholt in die gewünschte Richtung ausgeführt wird. Anschließend wird dieser Sprung in die andere Richtung ausgeführt. Dies geschieht dabei so, dass das Ziel des Sprunges von Daten abhängt, die aus dem Hauptspeicher geladen werden müssen. Auf diese Weise wird der Branch Predictor zu einer Vorhersage gezwungen. Die gewünschte Richtung des Sprunges wird also in einer Transient Execution ausgeführt. Innerhalb dieser Transient Execution wird anschließend ein fester Wert in den Cache kodiert. Nach Ende der Transient Execution wird mittels Flush+Reload festgestellt und protokolliert, ob dieser feste Wert korrekt kodiert wurde. Außerdem wird die Anzahl der benötigten Zyklen für das Training des Branch Predictors und für den eigentlichen Angriff ermittelt und protokolliert.

Ergebnisse

Tabelle 2: *Die ermittelten Metriken für jede Art verwendeter Sprünge, angegeben als $\mu \pm \sigma$ auf 4 gültige Ziffern.*

Art verwendeter Sprünge	Erfolgsrate (%)	Training (Zyklen)	Angriff (Zyklen)
Direkte Sprünge	$99{,}93 \pm 0{,}1006$	$3{,}446 \pm 0{,}03336$	$224{,}0 \pm 10{,}17$
Indirekte Sprünge	$99{,}99 \pm 0{,}05006$	$3{,}208 \pm 0{,}01991$	$254{,}7 \pm 8{,}253$
Rücksprünge	$99{,}99 \pm 0{,}05985$	—	$232{,}2 \pm 9{,}705$

Das Training besteht aus 4 096 ausgeführten Sprüngen. Für jede betrachtete Art von Sprung wird das Training und der eigentliche Angriff jeweils 10 000 mal durchgeführt. Die dabei erfassten Metriken werden zu einem einzelnen Messpunkt gemittelt. Insgesamt werden auf diese Weise 1 000 Messpunkte ermittelt, aus denen anschließend die Mittelwerte und Standardabweichungen der Metriken berechnet werden. Programmcode, der die beschriebenen Techniken ausführt und die benötigten Daten ermittelt, ist dieser Arbeit in der Datei `misprediction.c` beigelegt. Die daraus berechneten Ergebnisse, unterteilt nach der Art der verwendeten Sprünge, sind in Tabelle 2 dargestellt. Die Spalte *Erfolgsrate* zeigt dabei den Anteil der Angriffe, die den festen Wert erfolgreich in den Cache kodiert haben. *Training* bezeichnet die Zahl benötigter Zyklen für einen einzelnen Sprung während des Trainings und *Angriff* bezeichnet die Zahl benötigter Zyklen für den Sprung während des Angriffs.

Auftretende Phänomene

Wie in Tabelle 2 zu sehen ist, funktionieren alle Techniken sehr zuverlässig mit einer durchschnittlichen Erfolgsrate über 99,9 %. Die Dauer eines einzelnen Sprunges während des Trainings ist mit weniger als 4 Zyklen in allen Fällen sehr gering. Dies ist darauf zurückzuführen, dass der Branch

[2] Die Technik, die Rücksprünge aus Funktionen verwendet, erfordert kein Training des Branch Predictors. Anders als bei bedingten oder indirekten Sprüngen basiert die Vorhersage des Branch Predictors bei Rücksprüngen nicht auf vorherigen Sprüngen, sondern auf vorhergehenden Funktionsaufrufen [Kor+18, Kap. 3.1].

Predictor während des Trainings fast alle Sprünge korrekt vorhersagt, wodurch diese sehr schnell ausgeführt werden können. Dem gegenüber benötigen die Angriffe mit 220 bis 250 Zyklen wesentlich mehr Zeit. In diesem Fall trifft der Branch Predictor in fast allen Fällen eine falsche Vorhersage. Wie in Kapitel 3.3.4 erklärt, ist dies mit einem starken Performanceverlust verbunden. Außerdem braucht ein Angriff unter Verwendung direkter Sprünge im Schnitt ungefähr 10 Zyklen weniger als ein Angriff, der Rücksprünge verwendet. Dieser wiederum braucht im Schnitt ungefähr 20 Zyklen weniger als ein Angriff durch indirekte Sprünge. Dieser Umstand legt die Vermutung nahe, dass der Prozessor umso länger für die Korrektur einer falschen Vorhersage braucht, je komplexer die Art des vorhergesagten Sprunges ist. Direkte Sprünge kodieren die möglichen Sprungziele in der Instruktion selber, Rücksprünge laden das Sprungziel von einer bekannten Speicheradresse, und indirekte Sprünge können das Sprungziel von einer Speicheradresse laden, die aus mehreren Registern zusammengesetzt ist.

Wie durch die genannten Ergebnisse bestätigt, kann bei Verwendung von Rücksprüngen eine Transient Execution zuverlässig ausgelöst werden. Dabei ist kein Training des Branch Predictors nötig. Dies vereinfacht eine Implementierung der Technik und erhöht außerdem ihre Geschwindigkeit. Aus diesen Gründen wird in Kapitel 5.4, in dem konkrete Angriffe evaluiert werden, Transient Execution zur Unterdrückung von Exceptions stets mithilfe von Rücksprüngen ausgelöst.

5.4 Angriffe auf Daten anderer Prozesse

In diesem Kapitel werden RIDL, Write Transient Forwarding und ZombieLoad, die in Kapitel 4.7 beschrieben wurden, in verschiedenen Varianten evaluiert. Dies geschieht hinsichtlich der erreichten Datenrate mit zugehöriger Erfolgsrate, sowie der Dauer des Angriffs. Wie schon in Kapitel 4.7.2 beschrieben, unterscheidet sich Store-to-Leak in seiner Funktionsweise von den anderen drei Angriffen. Deshalb wird Store-to-Leak separat in Kapitel 5.5 evaluiert.

Alle hier betrachteten Varianten werden auf die gleiche Art und Weise evaluiert. Diese wird im Folgenden beschrieben. Neben dem angreifenden Prozess wird stets ein Opfer-Prozess ausgeführt. Dieser liest oder schreibt wiederholt einen festen Wert, sodass dieser von dem angreifenden Prozess aus einem Puffer der Mikroarchitektur extrahiert werden kann. Um ein einzelnes Byte zu extrahieren, führt der angreifende Prozess den jeweiligen Angriff 200 mal aus. Dabei wird das am häufigsten aus dem Cache dekodierte Byte übernommen. Dies wird wiederholt, um 100 Bytes zu extrahieren. Über diese Wiederholungen werden alle erfassten Metriken gemittelt, wodurch ein einziger Messpunkt erhalten wird. Auf diese Art und Weise werden nacheinander 1 000 Messpunkte ermittelt, um die Messungen zeitlich zu separieren. Aus diesen Messpunkten werden anschließend die Mittelwerte und Standardabweichungen der Metriken berechnet.

Die Vorgehensweise dieser Evaluation stellt das Szenario eines angreifenden Nutzerprozesses nach, der Daten eines anderen (nicht kooperierenden) Prozesses extrahiert. Beispielsweise kann ein solcher Nutzerprozess wiederholt das `passwd`-Programm aufrufen und angreifen, um das root-Passwort des Systems zu extrahieren [Sch+19a, Kap. VI.A]. Andere konkrete Angriffe aus diesem Szenario ermöglichen das Extrahieren von Seitenverläufen aus Webbrowsern [Sch+19b, Kap. 6.4] oder geheimen Schlüsseln aus kryptographischen Anwendungen [Sch+19b, Kap. 6.1].

Es werden die folgenden Metriken erfasst:

- Die **Erfolgsrate** des Angriffs. Dies ist der Anteil der korrekt ermittelten Bytes.
- Die erreichte **Datenrate**. Diese wird berechnet aus der Dauer des Angriffs. Die Erfolgsrate fließt nicht in die Berechnung ein; für sinnvolle Vergleiche muss die Datenrate also stets mit der Erfolgsrate gemeinsam betrachtet werden.
- Die Dauer des unberechtigten Datenzugriffs und des **Kodieren** in den Cache.
- Die Dauer des **Dekodierens** der übertragenen Daten aus dem Cache.

Wie bereits in Kapitel 4.7 erwähnt, ist der Programmcode aller Angriffe und ihrer Varianten dieser Arbeit beigelegt in den Dateien `ridl.c`, `wtf.c` und `zombieload.c`.

Von allen Angriffen werden verschiedene Varianten untersucht. Dabei werden folgende Aspekte der Angriffe verändert:

- Der **Opfer-Prozess** führt wiederholt einen Speicherzugriff durch. Dieser ist entweder lesend oder schreibend.
- Als **Cache-basierter Seitenkanalangriff** wird entweder Flush+Reload oder Flush+Flush verwendet (siehe Kapitel 4.1).
- Auftretende **Exceptions** werden durch eine der Techniken aus Kapitel 4.6.4 vermieden. Entweder wird die Exception über einen Signal Handler behandelt oder durch Transient Execution unterdrückt. Bei RIDL kann die Exception alternativ vermieden werden, indem der auslösende Speicherzugriff auf eine ausgelagerte Page erfolgt (siehe Kapitel 4.7.1).
- Zusätzlich gibt es für RIDL und Write Transient Forwarding **spezifische Variationen**: Bei RIDL werden Daten entweder aus dem Line-Fill Buffer oder aus den Load Ports extrahiert (siehe Kapitel 4.7.1), bei Write Transient Forwarding erfolgt der initiale Speicherzugriff entweder auf eine nicht-kanonische Adresse oder auf eine Adresse des Betriebssystem-Kerns (siehe Kapitel 4.7.2).

Alle möglichen Kombinationen der verschiedenen Aspekte zu untersuchen würde den Rahmen dieser Bachelorarbeit sprengen. Daher werden für jeden Angriff eine Basiskonfiguration gewählt und ausgehend von dieser nur einzelne Aspekte variiert. Der dieser Arbeit beigelegte Programmcode ist jedoch so aufgebaut, dass mit geringem Aufwand weitere Konfigurationen evaluiert werden können.

In den folgenden Unterkapiteln werden die verschiedenen Varianten der drei betrachteten Angriffe untersucht. Anschließend werden die Beobachtungen der einzelnen Angriffe verglichen und übergreifende Phänomene erläutert.

5.4.1 RIDL: Rogue In-Flight Data Load

In der Basiskonfiguration von RIDL (siehe auch Kapitel 4.7.1) führt der Opfer-Prozess schreibende Speicherzugriffe durch. Als Cache-basierter Seitenkanalangriff wird Flush+Reload verwendet. Das Auftreten einer Exception wird vermieden, indem auf eine ausgelagerte Page zugegriffen wird. Daten werden aus dem Line-Fill Buffer extrahiert.

Zusätzlich zu dieser Basiskonfiguration werden verschiedene Varianten betrachtet, die jeweils in einzelnen Aspekten abweichen:

- **Flush+Flush**: Als Cache-basierter Seitenkanalangriff wird Flush+Flush verwendet.
- **Load**: Der Opfer-Prozess führt lesende Speicherzugriffe durch.
- **Load Port**: Daten werden aus den Load Ports extrahiert und der Opfer-Prozess führt lesende Speicherzugriffe durch[3].
- **Signal**: Auftretende Exceptions werden mittels eines Signal Handlers behandelt (siehe Kapitel 4.6.4).
- **Transient**: Auftretende Exceptions werden durch Transient Execution unterdrückt (siehe Kapitel 4.6.4).

Tabelle 3: *Die ermittelten Metriken für jede RIDL-Variante, angegeben als $\mu \pm \sigma$ auf 4 gültige Ziffern.*

Variante	Erfolgsrate (%)	Datenrate (B/s)	Kodieren (Zyklen)	Dekodieren (Zyklen)
Basis	97,96 ± 9,721	443,9 ± 0,308	1 229 ± 17,12	94 200 ± 83,84
Flush+Flush	2,515 ± 3,918	616,8 ± 0,664	1 218 ± 16,29	48 750 ± 57,24
Load	56,27 ± 19,02	450,6 ± 0,571	1 178 ± 24,17	91 860 ± 181,8
Load Port	11,40 ± 5,126	450,6 ± 0,570	1 182 ± 23,53	91 880 ± 180,0
Signal	74,69 ± 32,20	745,5 ± 0,490	1 789 ± 11,97	94 640 ± 61,11
Transient	0,000 ± 0,000	757,1 ± 0,564	184,0 ± 0,423	94 760 ± 70,62

Tabelle 3 zeigt die ermittelten Metriken für alle betrachteten Varianten von RIDL. Die Bedeutung dieser Metriken und die Vorgehensweise ihrer Erfassung wurden bereits in Kapitel 5.4 erläutert. Die gemessenen Latenzen beinhalten dabei auch die Zeit, die für die Messung selber benötigt wird (siehe Kapitel 4.6.2).

Wie in Tabelle 3 zu sehen ist, funktioniert die Basiskonfiguration von RIDL zuverlässig mit einer Erfolgsrate von 97,96 %. Die Daten lesender Speicherzugriffe lassen sich mit einer Erfolgsrate von 56,27 % wesentlich weniger zuverlässig aus dem Line-Fill Buffer extrahieren als die Daten schreibender Speicherzugriffe. Außerdem werden Daten aus den Load Ports mit einer Erfolgsrate von 11,40 % deutlich weniger zuverlässig extrahiert als aus dem Line-Fill Buffer.

Die Variante, die Exceptions durch Transient Execution unterdrückt, extrahierte in keinem der beobachteten Fälle Daten erfolgreich. Eine mögliche Erklärung dafür ist, dass die Transient Execution bereits bei dem invaliden Speicherzugriff terminiert und so keine Daten in den Cache kodiert werden. Eine andere Möglichkeit ist, dass der Prozessorfehler, auf dem RIDL basiert, während einer Transient Execution nicht auftritt, also keine Daten aus dem Line-Fill Buffer geladen werden.

Auftretende Exceptions mittels eines Signal Handlers zu behandeln resultiert in einer verringerten Erfolgsrate (von 97,96 % auf 74,69 %) und erfordert mehr Zyklen für Angriff und Kodierung (1 229 ggü. 1 789). Dies wird durch den Aufruf des Signal Handlers verursacht, der zusätzliche Zeit benötigt und dadurch potenzielle Interferenz durch das Betriebssystem oder dritte Prozesse erhöht. Andererseits erreicht die Vermeidung von Exceptions durch ausgelagerte Pages eine geringere Datenrate als die Behandlung durch Signal Handler (443,9 ggü. 745,5 B/s), da für erstere

[3] Da aus den Load Ports nur lesende Speicherzugriffe extrahiert werden können, wird in diesem Fall auch der Opfer-Prozess angepasst.

die betreffende Page vor jedem Angriff erneut ausgelagert werden muss. Die benötigte Zeit der Auslagerung überwiegt dabei die des Signal Handlers.

Die Verwendung von Flush+Flush als Cache-basierten Seitenkanalangriff reduziert die Erfolgsrate wesentlich, von 97,96 % auf 2,515 %. Zusätzlich reduziert Flush+Flush die Dauer des Dekodierens ungefähr auf die Hälfte, von 94 200 auf 48 750 Zyklen. Beides lässt sich mit den Ergebnissen aus Kapitel 5.2 vereinbaren.

In der Literatur werden durch RIDL Daten zwischen Prozessen übertragen mit Datenraten von 100 bis 1 000 B/s [Sch+19a, Tab. 1]. In diesen Bereich fällt auch die hier beobachtete Datenrate von 443,9 B/s bei 97,96 % Erfolgsrate. Die beobachteten Ergebnisse sind in diesem Punkt also mit denen aus der Literatur vereinbar.

5.4.2 Write Transient Forwarding

In der Basiskonfiguration von Write Transient Forwarding (siehe auch Kapitel 4.7.2) führt der Opfer-Prozess schreibende Speicherzugriffe durch. Als Cache-basierter Seitenkanalangriff wird Flush+Reload verwendet. Auftretende Exceptions werden mithilfe eines Signal Handlers behandelt (siehe Kapitel 4.6.4). Der initiale Speicherzugriff erfolgt auf eine nicht-kanonische Adresse.

Zusätzlich zu dieser Basiskonfiguration werden verschiedene Varianten betrachtet, die jeweils in einzelnen Aspekten abweichen:

- **Flush+Flush**: Als Cache-basierter Seitenkanalangriff wird Flush+Flush verwendet.
- **Kernel**: Der initiale Speicherzugriff erfolgt auf eine Adresse des Betriebssystem-Kerns (siehe Kapitel 4.7.2).
- **Transient**: Auftretende Exceptions werden durch Transient Execution unterdrückt (siehe Kapitel 4.6.4).

Der Opfer-Prozess wird nicht variiert, da Write Transient Forwarding Daten aus dem Store Buffer extrahiert und daher nur schreibende Zugriffe beobachtet werden können.

Tabelle 4 zeigt die ermittelten Metriken für alle betrachteten Varianten von Write Transient Forwarding. Die Bedeutung dieser Metriken und die Vorgehensweise ihrer Erfassung wurden bereits in Kapitel 5.4 erläutert. Die gemessenen Latenzen beinhalten dabei auch die Zeit, die für die Messung selber benötigt wird (siehe Kapitel 4.6.2).

Tabelle 4: *Die ermittelten Metriken für jede WTF-Variante, angegeben als $\mu \pm \sigma$ auf 4 gültige Ziffern.*

Variante	Erfolgsrate (%)	Datenrate (B/s)	Kodieren (Zyklen)	Dekodieren (Zyklen)
Basis	0,000 ± 0,000	752,1 ± 0,573	1 594 ± 16,30	93 990 ± 66,28
Flush+Flush	0,300 ± 0,555	1 428 ± 2,766	1 589 ± 15,92	48 710 ± 87,06
Kernel	0,000 ± 0,000	750,4 ± 0,554	1 726 ± 12,21	94 070 ± 65,21
Transient	0,000 ± 0,000	764,1 ± 0,741	269,7 ± 1,999	93 810 ± 91,53

Wie in Tabelle 4 zu sehen ist, funktioniert keine der untersuchten Varianten zuverlässig. Die meisten Varianten extrahierten in keinem der beobachteten Fälle Daten erfolgreich. Einzig die Variante, die Flush+Flush als Cache-basierten Seitenkanalangriff verwendet, weist eine Erfolgsrate von 0,3 % auf. Aufgrund der im Vergleich zu Flush+Reload niedrigen Spezifität von Flush+Flush (siehe Kapitel 5.2) wird diese jedoch wahrscheinlich durch eine falsch-positive Erkennung des übertragenen Bytes ausgelöst. Eine mögliche Erklärung für das Fehlschlagen dieses Angriffs ist, dass der verwendete Prozessor (siehe Kapitel 5.1) nicht anfällig gegen Write Transient Forwarding ist. Eine andere Erklärung ist, dass die verwendete Version des Mikrocodes auch ohne eine Unterstützung des Betriebssystems den Prozessorfehler behebt, auf dem Write Transient Forwarding basiert.

Die Ergebnisse der Literatur [Can+19b] können für Write Transient Forwarding in dieser Arbeit folglich nicht reproduziert werden.

5.4.3 ZombieLoad

In der Basiskonfiguration von ZombieLoad (siehe auch Kapitel 4.7.3) führt der Opfer-Prozess schreibende Speicherzugriffe durch. Als Cache-basierter Seitenkanalangriff wird Flush+Reload verwendet. Auftretende Exceptions werden mithilfe eines Signal Handlers behandelt (siehe Kapitel 4.6.4).

Zusätzlich zu dieser Basiskonfiguration werden verschiedene Varianten betrachtet, die jeweils in einzelnen Aspekten abweichen:

- **Flush+Flush**: Als Cache-basierter Seitenkanalangriff wird Flush+Flush verwendet.
- **Load**: Der Opfer-Prozess führt lesende Speicherzugriffe durch.
- **Transient**: Auftretende Exceptions werden durch Transient Execution unterdrückt (siehe Kapitel 4.6.4).

Tabelle 5 zeigt die ermittelten Metriken für alle betrachteten Varianten von ZombieLoad. Die Bedeutung dieser Metriken und die Vorgehensweise ihrer Erfassung wurden bereits in Kapitel 5.4 erläutert. Die gemessenen Latenzen beinhalten dabei auch die Zeit, die für die Messung selber benötigt wird (siehe Kapitel 4.6.2).

Tabelle 5: *Die ermittelten Metriken für jede ZombieLoad-Variante, angegeben als $\mu \pm \sigma$ auf 4 gültige Ziffern.*

Variante	Erfolgsrate (%)	Datenrate (B/s)	Kodieren (Zyklen)	Dekodieren (Zyklen)
Basis	94,31 ± 23,14	747,1 ± 0,542	1 722 ± 15,45	94 510 ± 66,87
Flush+Flush	67,51 ± 43,55	1 424 ± 2,597	1 720 ± 15,63	48 690 ± 83,72
Load	93,25 ± 24,98	768,2 ± 1,413	1 636 ± 19,49	91 950 ± 169,6
Transient	99,70 ± 5,472	780,3 ± 1,366	263,1 ± 5,020	91 870 ± 161,4

Wie in Tabelle 5 zu sehen, funktioniert die Basiskonfiguration von ZombieLoad zuverlässig mit einer Erfolgsrate von 94,31 %. Die Daten lesender Speicherzugriffe lassen sich mit einer Erfolgsrate von 93,25 % nur unwesentlich weniger zuverlässig extrahieren als die Daten schreibender Speicher-

zugriffe. Der gleiche Effekt wurde bei RIDL beobachtet (siehe Kapitel 5.4.1), ist dort aber wesentlich stärker ausgeprägt.

Ebenfalls im Gegensatz zu den RIDL-Varianten funktioniert bei ZombieLoad die Variante, die Exceptions durch Transient Execution unterdrückt, sehr zuverlässig. Die Erfolgsrate dieser Variante liegt hier bei 99,70 % und damit über der Erfolgsrate der Basiskonfiguration. Außerdem benötigt die Transient-Execution-Variante wesentlich weniger Zyklen für den Angriff und die Kodierung in den Cache (263,1 ggü. 1 722), wodurch sich die Datenrate leicht erhöht[4]. Beides lässt sich durch die Abwesenheit einer architekturell auftretenden Prozessor-Exception erklären. Das Betriebssystem muss die Exception nicht behandeln, wodurch der Ablauf beschleunigt und potenzielle Quellen der Interferenz reduziert werden.

Die Verwendung von Flush+Flush als Cache-basierten Seitenkanalangriff reduziert die Erfolgsrate von 94,31 % auf 67,51 %. Zusätzlich reduziert Flush+Flush die Dauer des Dekodierens ungefähr auf die Hälfte, von 94 510 auf 48 690 Zyklen. Da die Dauer des Dekodierens in beiden Fällen die Gesamtdauer dominiert, erhöht sich dadurch die Datenrate ungefähr auf das Doppelte, von 747,1 auf 1 424 B/s. Beide Effekte stimmen mit den Ergebnissen aus Kapitel 5.2 überein.

In der Literatur werden durch ZombieLoad Daten zwischen Prozessen übertragen mit einer Datenrate von 5,30 kB/s und einer Richtig-positiv-Rate von 85,74 %, unter Verwendung von TSX [Sch+19b, Kap. 5.4]. Die hier implementierte Transient-Execution-Variante von ZombieLoad erreicht eine Datenrate von 780,3 B/s bei einer Erfolgsrate von 99,70 %. Diese ist also um eine Größenordnung langsamer, aber wesentlich zuverlässiger als der Angriff aus der Literatur. Für diese Unterschiede gibt es zwei mögliche Erklärungen. Einerseits können sie ausgelöst werden durch den Unterschied zwischen TSX und Transient Execution. Andererseits wurden die in dieser Arbeit implementierten Angriffe nicht auf ihre Datenrate optimiert und die verwendete Hardware ist unterschiedlich leistungsstark.

Insgesamt stimmen die hier erlangten Ergebnisse also im Wesentlichen mit denen aus der Literatur überein.

5.4.4 Übergreifende Phänomene

Bei einem Vergleich der Ergebnisse zwischen den untersuchten Angriffen fällt auf, dass Flush+Reload als Cache-basierter Seitenkanalangriff immer zuverlässiger als Flush+Flush ist. Andererseits benötigt Flush+Flush nur rund halb so viele Zyklen zum Dekodieren wie Flush+Reload. Dies deckt sich mit den Beobachtungen aus Kapitel 5.2. Auch die konkrete Zahl der zum Dekodieren benötigten Zyklen stimmt über verschiedene Angriffe hinweg größtenteils überein, sowohl für Flush+Reload als auch für Flush+Flush. Außerdem wird die Gesamtdauer des Angriffs stets durch die Dauer der Dekodierung dominiert. Eine Beschleunigung der Dekodierung führt damit auch zu einer Verbesserung der erreichten Datenrate.

Insgesamt konnten einige Ergebnisse der Paper, die RIDL [Sch+19a] und ZombieLoad [Sch+19b] beschreiben, in dieser Arbeit reproduziert werden. Write Transient Forwarding [Can+19b, Kap. 3.1] hingegen konnte in der hier angefertigten Implementierung keine Daten erfolgreich extrahieren.

[4] Da in beiden Fällen die Dauer des Dekodierens dominiert, erhöht auch eine wesentliche Reduktion der Dauer des Kodierens die Datenrate nur leicht.

5.5 Store-to-Leak

Anders als die drei in Kapitel 5.4 untersuchten Angriffe extrahiert Store-to-Leak keine Daten direkt aus Puffern der Mikroarchitektur. Daher wird Store-to-Leak separat in diesem Kapitel betrachtet. Wie in Kapitel 4.7.2 dargestellt, kann Store-to-Leak verwendet werden, um die Anwesenheit von Speicherzuordnungen im Adressbereich des Betriebssystem-Kerns festzustellen. Auf diese Weise kann mittels Store-to-Leak die Basisadresse des Betriebssystem-Kerns bestimmt und damit KASLR umgangen werden.

Zum Zweck der Evaluation wird Store-to-Leak wie in Kapitel 4.7.2 beschrieben ausgeführt. Um zu ermitteln, ob eine Speicherzuordnung für eine anvisierte virtuelle Adresse existiert, werden acht Store-to-Leak Iterationen ausgeführt. Dabei wird erfasst, wie oft Store-to-Leak einen validen TLB Eintrag für diese Adresse detektiert. Erreicht diese Zahl einen festen Threshold von 2, so wird die Speicherzuordnung als vorhanden angenommen. Auf diese Art und Weise werden eine vorhandene und eine nicht vorhandene Speicherzuordnung jeweils 10 000-mal klassifiziert. Die erhobenen Metriken werden über diese Klassifikationen gemittelt, um einen einzelnen Messpunkt zu erhalten. Dies wird wiederholt, bis nacheinander 1 000 zeitlich separierte Messpunkte ermittelt wurden. Aus diesen Messpunkten werden anschließend die Mittelwerte und Standardabweichungen der Metriken berechnet.

Dabei werden die Fehlerrate der Klassifikation und die zur Klassifikation benötigte Zeit erfasst. Diese Messungen werden getrennt für den Fall der vorhandenen und der nicht vorhandenen Speicherzuordnung. Aus allen erfassten Messungen werden anschließend die Sensitivität und Spezifität der Klassifikation berechnet. Die Bedeutung und Berechnung dieser Metriken wurde bereits in Kapitel 5.2 erläutert.

Wie die in Kapitel 5.4 betrachteten Angriffe, wird auch Store-to-Leak anhand verschiedener Varianten evaluiert. Dabei kann einerseits der verwendete Cache-basierte Seitenkanalangriff variiert werden, sowie andererseits die Technik zur Vermeidung von Exceptions. Die Basiskonfiguration von Store-to-Leak verwendet Flush+Reload als Cache-basierten Seitenkanalangriff und behandelt Exceptions durch einen Signal Handler. Zusätzlich werden folgende Varianten untersucht:

- **Flush+Flush**: Als Cache-basierter Seitenkanalangriff wird Flush+Flush verwendet.
- **Transient**: Auftretende Exceptions werden durch Transient Execution unterdrückt (siehe Kapitel 4.6.4).

Tabelle 6: *Die ermittelten Metriken für jede Store-to-Leak-Variante, angegeben als $\mu \pm \sigma$ auf 4 gültige Ziffern.*

Variante	Sensitivität (%)	Spezifität (%)	Mapped (µs)	Unmapped (µs)
Basis	69,22 ± 7,399	100,0 ± 0,01168	8,752 ± 0,06659	8,951 ± 0,03694
Flush+Flush	2,185 ± 4,657	99,19 ± 3,017	8,126 ± 0,04591	8,186 ± 0,04292
Transient	99,89 ± 2,355	100,0 ± 0,0003162	1,673 ± 0,01000	1,799 ± 0,008132

Tabelle 6 zeigt die ermittelten Metriken für jede der untersuchten Varianten von Store-to-Leak. *Mapped* und *Unmapped* bezeichnen dabei die zur Klassifikation einer vorhandenen und einer nicht vorhandenen Speicherzuordnung benötigte Zeit. Dabei ist zu sehen, dass die Unterdrückung von Exceptions durch Transient Execution zu einer wesentlich zuverlässigeren und wesentlich schnelleren Klassifikation führt. Dies ist durch die Abwesenheit einer architekturell auftretenden Prozessor-Exception zu erklären. Das Betriebssystem muss die Exception nicht behandeln, wodurch der Ablauf beschleunigt und potenzielle Quellen der Interferenz reduziert werden. Der gleiche Effekt wurde bereits bei ZombieLoad in Kapitel 5.4.3 beobachtet.

Außerdem erfolgt die Klassifikation von vorhandenen Speicherzuordnungen etwas schneller als die Klassifikation nicht vorhandener Zuordnungen. Dies lässt sich dadurch erklären, dass bei vorhandenen Zuordnungen die wiederholten Zugriffe durch den TLB beschleunigt werden. Im Falle einer nicht vorhandenen Zuordnung wird hingegen kein Eintrag im TLB erstellt.

Des Weiteren wird die Klassifizierung durch die Verwendung von Flush+Flush wesentlich weniger zuverlässig und etwas schneller. Dies stimmt mit den Ergebnissen aus Kapitel 5.2 und ähnlichen Beobachtungen bei den in Kapitel 5.4 betrachteten Angriffen überein. Die Auswirkung auf die Ausführungszeit ist bei Store-to-Leak wesentlich geringer als bei den in Kapitel 5.4 betrachteten Angriffen, da Store-to-Leak nur eine einzelne Cachezeile beobachtet.

Wie in Kapitel 3.1.2 beschrieben, besitzt die Basisadresse des Linux Kernels eine Entropie von 9 Bits[5]. Um diese Basisadresse zu bestimmen, müssen von den 512 möglichen Adressen im Schnitt 256 geprüft werden. Diese Klassifizierungen benötigen mit der Transient-Variante von Store-to-Leak ungefähr $256 \cdot 1{,}799 = 460{,}544$ Mikrosekunden. Store-to-Leak kann folglich genutzt werden, um in einer halben Millisekunde die Basisadresse des Linux Kernels zu bestimmen und dadurch KASLR zu umgehen.

In der Literatur wird für die mithilfe von Store-to-Leak durchgeführte Klassifikation ein *F1-Score* von 0,9996 angegeben [Can+19b, Kap. 6.1]. Der F1-Score ist das harmonische Mittel von der *Genauigkeit* und der Sensitivität der Klassifikation [Tha20, Kap. 2.8]. Die Genauigkeit, auch *positiver Vorhersagewert*, ist der Anteil der als vorhanden klassifizierten Speicherzuordnungen, die tatsächlich vorhanden sind [Tha20, Kap. 2.5]. Für die Transient-Variante von Store-to-Leak beträgt die Genauigkeit hier 1,000 und der F1-Score 0,9994. Die Ergebnisse der Literatur können hier also reproduziert werden.

5.6 Zusammenfassung

In Kapitel 5.4 und Kapitel 5.5 wurden RIDL, Write Transient Forwarding, ZombieLoad und Store-to-Leak in verschiedenen Varianten evaluiert. Bei dieser Evaluation zeigten drei von den vier betrachteten Angriffen die erwarteten Ergebnisse in den meisten ihrer Varianten. RIDL, ZombieLoad und Store-to-Leak extrahierten erfolgreich anvisierte Daten. Durch Write Transient Forwarding hingegen konnten in keiner der evaluierten Varianten die anvisierten Daten erfolgreich extrahiert werden.

[5] Die Basisadresse des Linux Kernels wird zwischen `0xffffffff80000000` und `0xffffffffc0000000` gewählt, als ein Vielfaches von `0x200000`.

6 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit bestand zunächst in einer Erklärung der Funktionsweise moderner Spectre-Angriffe, gefolgt von einer Implementierung und Evaluation ausgewählter konkreter Spectre-Angriffe. Als konkrete Spectre-Angriffe wurden dabei RIDL, ZombieLoad, Write Transient Forwarding und Store-to-Leak gewählt. Zur Erfüllung dieses Ziels wurden zu Beginn die Grundlagen der Funktionsweise moderner Prozessoren erläutert, die für ein Verständnis von Spectre-Angriffen benötigt werden. Dies umfasst die Funktionsweise von Prozessor-Caches und den Aufbau der Mikroarchitektur moderner Intel-Prozessoren. Anschließend wurden Flush+Reload und Flush+Flush als Cache-basierte Seitenkanalangriffe beschrieben. Diese bilden einen wichtigen Teil der Implementierung von Spectre-Angriffen. Danach wurde die Methodik von Spectre-Angriffen im Allgemeinen dargestellt. Diese Angriffe wurden zunächst in zwei Kategorien unterteilt: Spectre-Angriffe basierend auf Branch Prediction und Spectre-Angriffe basierend auf Prozessor-Exceptions. Daraufhin wurde die Funktionsweise der oben genannten konkreten Spectre-Angriffe erklärt und Details ihrer Implementierung beschrieben. Im Zuge der Evaluation wurden zunächst einzelne Techniken ausgewertet, die in Spectre-Angriffen Verwendung finden. Dazu gehören die beiden beschriebenen Cache-basierten Seitenkanalangriffe sowie eine Auswahl an Techniken zur Unterdrückung von Prozessor-Exceptions. Schließlich wurden die konkreten Spectre-Angriffe selbst evaluiert. Von jedem Angriff wurden dabei verschiedene Varianten nach einheitlichen Metriken ausgewertet und anschließend verglichen.

Die Evaluation hat ergeben, dass Flush+Reload wesentlich zuverlässiger, aber in vielen Fällen langsamer ist als Flush+Flush. Die untersuchten Techniken zur Unterdrückung von Prozessor-Exceptions haben zuverlässig funktioniert. Jede dieser Techniken erreichte eine Erfolgsrate von 99,93 % oder mehr. Von den vier betrachteten Spectre-Angriffen zeigten drei die erwarteten Ergebnisse in den meisten ihrer Varianten. RIDL und ZombieLoad extrahierten erfolgreich Daten aus einem anderen Prozess. RIDL erreichte dabei eine Datenrate von 443,9 B/s bei einer Erfolgsrate von 97,96 %, ZombieLoad eine Datenrate von 780,3 B/s bei einer Erfolgsrate von 99,70 %. Mithilfe von Store-to-Leak gelang es, Speicherzuordnungen des Betriebssystem-Kerns korrekt zu klassifizieren mit einer Sensitivität von 99,98 % und einer Spezifität von 100,0 %. Durch Write Transient Forwarding hingegen konnten in keiner der evaluierten Varianten die anvisierten Daten erfolgreich extrahiert werden. Insgesamt konnten damit Techniken aus den Papern zu Flush+Reload [YF14], Flush+Flush [Gru+16], RIDL [Sch+19a], ZombieLoad [Sch+19b] und Store-to-Leak [Can+19b] erfolgreich implementiert und Ergebnisse dieser Paper reproduziert werden.

Für aufbauende Arbeiten bietet sich beispielsweise eine Optimierung der implementierten Angriffe an. Außerdem wurden in dieser Arbeit ausschließlich konkrete Spectre-Angriffe betrachtet, die auf Prozessor-Exceptions basieren, sowie von diesen nur ausgewählte Varianten. Weiterführende Forschung kann demnach Spectre-Angriffe basierend auf Branch Prediction oder weitere Kombinationen der beschriebenen Techniken untersuchen. Zusätzlich fand die Evaluation im Rahmen dieser

Arbeit auf einem einzigen System statt. Um die betrachteten Varianten in einem größerem Kontext zu evaluieren, können die Untersuchungen auf weiteren Systemen wiederholt werden. Desweiteren wurden in dieser Arbeit nur Angriffe betrachtet, die von unprivilegierten Nutzerprozessen auf Linux-Systemen durchgeführt werden können. In weiterführenden Arbeiten können folglich andere Angriffsszenarien betrachtet werden. Dies umfasst Angreifer in anderen Kontexten (z.B. im Betriebssystem-Kern oder in Trusted Execution Environments wie Intel SGX), andere Angriffsziele (z.B. Trusted Execution Environments, Hypervisor oder benachbarte virtuelle Maschinen) sowie andere Hardwarekonfigurationen (z.B. Cross-Core Angriffe oder Unterstützung von Intel TSX). Da in dieser Arbeit außerdem keine Gegenmaßnahmen für Spectre-Angriffe betrachtet wurden, können zukünftige Arbeiten die hier implementierten Angriffe im Kontext aktivierter Gegenmaßnahmen untersuchen.

Literaturverzeichnis

[Bul+18] Bulck, Jo Van u. a.: „Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution“. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Hrsg. von William Enck ; Adrienne Porter Felt. USENIX Association, 2018, S. 991–1008. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[Bul+20] Bulck, Jo Van u. a.: „LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection“. In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, S. 54–72. URL: https://doi.org/10.1109/SP40000.2020.00089.

[Can+19a] Canella, Claudio u. a.: „A Systematic Evaluation of Transient Execution Attacks and Defenses“. In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. Hrsg. von Nadia Heninger ; Patrick Traynor. Erweiterter Klassifizierungsbaum unter https://transient.fail/. USENIX Association, 2019, S. 249–266. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

[Can+19b] Canella, Claudio u. a.: „Fallout: Leaking Data on Meltdown-resistant CPUs“. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Hrsg. von Lorenzo Cavallaro u. a. ACM, 2019, S. 769–784. URL: https://doi.org/10.1145/3319535.3363219.

[Che+19] Chen, Guoxing u. a.: „SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution“. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*. IEEE, 2019, S. 142–157. URL: https://doi.org/10.1109/EuroSP.2019.00020.

[Evt+18] Evtyushkin, Dmitry u. a.: „BranchScope: A New Side-Channel Attack on Directional Branch Predictor“. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*. Hrsg. von Xipeng Shen u. a. ACM, 2018, S. 693–707. URL: https://doi.org/10.1145/3173162.3173204.

[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 2018, Baltimore, MD, USA, August 15-17, 2018*. Hrsg. von William Enck ; Adrienne Porter Felt. USENIX Association, 2018, S. 955–972. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/gras.

[Gru+16] Gruss, Daniel u. a.: „Flush+Flush: A Fast and Stealthy Cache Attack“. In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*. Hrsg. von Juan Caballero ; Urko Zurutuza ; Ricardo J. Rodríguez. Bd. 9721. Lecture Notes in Computer Science. Springer, 2016, S. 279–299. URL: https://doi.org/10.1007/978-3-319-40667-1_14.

[Gru17] Gruss, Daniel: „Software-based Microarchitectural Attacks“. In: *CoRR* abs/1706.05973 (2017). arXiv: 1706.05973. URL: http://arxiv.org/abs/1706.05973.

[Gru20] Gruss, Daniel: „Transient-Execution Attacks“. 2020. URL: https://gruss.cc/files/habil.pdf (besucht am 15. 01. 2021).

[Hor18] Horn, Jann: *speculative execution, variant 4: speculative store bypass*. 2018. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528 (besucht am 27. 03. 2021).

[HWH13] Hund, Ralf ; Willems, Carsten ; Holz, Thorsten: „Practical Timing Side Channel Attacks against Kernel Space ASLR“. In: *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 2013, S. 191–205. URL: https://doi.org/10.1109/SP.2013.23.

[Int21a] Intel Corporation: *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2021. URL: https://software.intel.com/content/dam/develop/public/us/en/documents/64-ia-32-architectures-optimization-manual.pdf (besucht am 21. 01. 2021).

[Int21b] Intel Corporation: *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1*. 2021. URL: https://software.intel.com/content/dam/develop/external/us/en/documents-tps/253665-sdm-vol-1.pdf (besucht am 20. 01. 2021).

[Int21c] Intel Corporation: *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2*. 2021. URL: https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325383-sdm-vol-2abcd.pdf (besucht am 20. 01. 2021).

[Int21d] Intel Corporation: *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3*. 2021. URL: https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325384-sdm-vol-3abcd.pdf (besucht am 20. 01. 2021).

[Int21e] Intel Corporation: *Intel® Intrinsics Guide*. 2021. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide (besucht am 04. 03. 2021).

[Koc+19] Kocher, Paul u. a.: „Spectre Attacks: Exploiting Speculative Execution“. In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, S. 1–19. URL: https://doi.org/10.1109/SP.2019.00002.

[Kor+18] Koruyeh, Esmaeil Mohammadian u. a.: „Spectre Returns! Speculation Attacks using the Return Stack Buffer“. In: *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*. Hrsg. von Christian Rossow ; Yves Younan. USENIX Association, 2018. URL: https://www.usenix.org/conference/woot18/presentation/koruyeh.

[Kos+20] Koschel, Jakob u. a.: „TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs“. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020*. IEEE, 2020, S. 309–321. URL: https://doi.org/10.1109/EuroSP48549.2020.00027.

[Lin21a] Linux Kernel Organization: „arch/x86/Kconfig“. In: *Linux Kernel Source Code*. 2021. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/Kconfig?h=v5.10 (besucht am 14. 03. 2021).

[Lin21b] Linux Kernel Organization: „CPU Performance Scaling“. In: *The Linux kernel user's and administrator's guide*. 2021. URL: https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html (besucht am 20. 03. 2021).

[Lin21c] Linux Kernel Organization: „Documentation for /proc/sys/vm/“. In: *The Linux kernel user's and administrator's guide*. 2021. URL: https://www.kernel.org/doc/html/latest/admin-guide/sysctl/vm.html (besucht am 14. 03. 2021).

[Lin21d] Linux Kernel Organization: „setjmp(3)“. In: *The Linux man-pages project*. 2021. URL: https://man7.org/linux/man-pages/man3/setjmp.3.html (besucht am 05. 03. 2021).

[Lin21e] Linux Kernel Organization: „sigaction(2)“. In: *The Linux man-pages project*. 2021. URL: https://man7.org/linux/man-pages/man2/sigaction.2.html (besucht am 05. 03. 2021).

[Lin21f] Linux Kernel Organization: „The kernel's command-line parameters“. In: *The Linux kernel user's and administrator's guide*. 2021. URL: https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html (besucht am 20. 03. 2021).

[Lip+18] Lipp, Moritz u. a.: „Meltdown: Reading Kernel Memory from User Space“. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Hrsg. von William Enck ; Adrienne Porter Felt. USENIX Association, 2018, S. 973–990. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[MR18] Maisuradze, Giorgi ; Rossow, Christian: „ret2spec: Speculative Execution Using Return Stack Buffers“. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Hrsg. von David Lie u. a. ACM, 2018, S. 2109–2122. URL: https://doi.org/10.1145/3243734.3243761.

[PS05] Pagiamtzis, Kostas ; Sheikholeslami, Ali: „Using cache to reduce power in content-addressable memories (CAMs)“. In: *Proceedings of the IEEE 2005 Custom Integrated Circuits Conference, CICC 2005, DoubleTree Hotel, San Jose, California, USA, September 18-21, 2005*. IEEE, 2005, S. 369–372. URL: https://doi.org/10.1109/CICC.2005.1568682.

[Qur+07] Qureshi, Moinuddin K. u. a.: „Adaptive insertion policies for high performance caching“. In: *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*. Hrsg. von Dean M. Tullsen ; Brad Calder. ACM, 2007, S. 381–391. URL: https://doi.org/10.1145/1250662.1250709.

[Sch+19a] Schaik, Stephan van u. a.: „RIDL: Rogue In-Flight Data Load“. In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, S. 88–105. URL: https://doi.org/10.1109/SP.2019.00087.

[Sch+19b] Schwarz, Michael u. a.: „ZombieLoad: Cross-Privilege-Boundary Data Sampling“. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Hrsg. von Lorenzo Cavallaro u. a. ACM, 2019, S. 753–768. URL: https://doi.org/10.1145/3319535.3354252.

[SP18] Stecklina, Julian ; Prescher, Thomas: „LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels“. In: *CoRR* abs/1806.07480 (2018). arXiv: 1806.07480. URL: http://arxiv.org/abs/1806.07480.

[Tha20] Tharwat, Alaa: „Classification assessment methods“. In: *Applied Computing and Informatics* 17.1 (2020), S. 168–192. URL: https://doi.org/10.1016/j.aci.2018.08.003.

[Wei+18] Weisse, Ofir u. a.: „Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution“. In: *Technical report* (2018). URL: https://foreshadowattack.eu/foreshadow-NG.pdf (besucht am 20. 03. 2021).

[YF14] Yarom, Yuval ; Falkner, Katrina: „FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack“. In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. Hrsg. von Kevin Fu ; Jaeyeon Jung. USENIX Association, 2014, S. 719–732. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Bonn, 19. April 2021 _____

Jan-Niklas Sohn