UNIVERSITÄT BONN

F+F Covert Channels

Evaluation von cache-basierten FLUSH+FLUSH-Seitenkanälen in Alltagsszenarien

Bachelorarbeit

ausgearbeitet von

Lennart Hein

[redacted]

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt an der

Rheinische Friedrich-Wilhelms-Universität Bonn

Institut für Informatik IV

Arbeitsgruppe für IT-Sicherheit

im Studiengang

Informatik (B.Sc.)

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

Bonn, 29. Juni 2020

Danksagung

[redacted]

Kurzfassung

Der Flush+Flush Seitenkanalangriff wurde als Verbesserung von bestehenden Cacheangriffen wie Flush+Reload vorgestellt. Neben höheren Frequenzen wird Flush+Flush außerdem nicht von bestehenden Detektionsmechanismen entdeckt. Die Machbarkeit und Performanz des Angriffs wurde sowohl in synthetischen Szenarien als auch in der Cloud demonstriert. Dabei bleiben realistische Desktopszenarien insbesondere auf AMD Prozessoren weitgehend unerforscht. In der vorliegenden Arbeit wird erarbeitet, inwiefern der Flush+Flush Seitenkanalangriff für den Bau von verdeckten Kanälen in verschiedenen Setups in typischen Desktopumgebungen verwendet werden kann. Flush+Flush wurde dafür, zusammen mit Flush+Reload, auf Intel und AMD Prozessoren bei verschiedenen Übertragungsfrequenzen getestet. Es stellt sich heraus, dass mit Flush+Flush implementierte Kanäle auf demselben Kern nicht die maximale Kapazität von Kanälen mit Flush+Reload erreichen können. Dafür bieten Kanäle mit Flush+Flush den Vorteil, dass sie auf modernen AMD Zen Prozessoren auch dann Cross-Core realisierbar sind, wenn sich Opfer und Angreifer auf anderen Komplexen, sogenannten CCX, befinden. Zusätzlich bleibt bei einem verdeckten Kanal zumindest der Empfänger vor den Detektionsmechanismen gegen Cacheangriffe versteckt.

Inhaltsverzeichnis

1 Einleitung

Verdeckte Kanäle (engl. covert channels) sind unerlaubte Kommunikationskanäle, welche vor den Kontrollmechanismen des Systems versteckt sind. Zunehmende Bedeutung fällt dabei cache-basierten verdeckten Kanälen zu, über welche nicht nur Nutzerdaten von Angreifern ausgeschleust werden können, sie ermöglichen im Cloud Computing zudem Prozessen aus getrennten Instanzen miteinander zu kommunizieren.

Um einen cache-basierten verdeckten Kanal zu implementieren, können Seitenkanaleffekte von CPU-Caches ausgenutzt werden. Ein Cache-Angriff, der einen solchen Seitenkanal verwendet, kann Auskunft darüber geben, ob bestimmte Speicherbereiche von einem Prozess in den Cache geladen worden sind. Indem ein Kommunikationspartner Nachrichten gezielt als Cachezugriffe codiert, und der andere Kommunikationspartner diese Daten mit Hilfe des Seitenkanals empfängt, kann der verdeckte Kanal aufgebaut werden.

Obgleich die Machbarkeit solcher Kanäle gezeigt werden konnte, wird der praktische Nutzen von solchen Kanälen stark durch die Qualität der Verbindung beeinflusst. Um die Bedrohung durch cache-basierte verdeckte Kanäle besser einschätzen zu können, müssen diese in einer realistischen Umgebung getestet werden.

Flush+Flush ist ein Cache-Angriff, der sich dadurch auszeichnet, dass bestehende Maßnahmen zum Aufspüren von aktiven Seitenkanalangriffen Flush+Flush nicht entdecken. Deswegen ist Flush+Flush von besonderer Bedeutung für die Sicherheit der betroffenen Systeme.

Das Ziel dieser Arbeit besteht darin, einen verdeckten Kanal mit Hilfe von Flush+Flush zu implementieren, und in alltagsähnlichen Szenarien zu testen. Dabei soll insbesondere Flush+Flush als Cache-Angriff zum Bauen des Kanals evaluiert werden. Hierbei kann die Anfälligkeit gegenüber Rauschen für die Qualität der Verbindung entscheidend sein.

Zunächst wird in Kapitel 2 der Stand der Forschung beschrieben. In Kapitel 3 werden Grundlagen zu CPU-Caches und -Architektur, den zu testenden Cache-Angriffen sowie Kommunikationstechnik und Statistik, die zur Auswertung essenziell sind, vorgestellt. In Kapitel 4 werden Implementierungsdetails, insbesondere Hürden und Lösungen für diese, diskutiert. Kapitel 5 behandelt anschließend den Versuchsaufbau, die Messung, und den Vergleich der Performance von den verdeckten Kanälen mit Flush+Flush und Flush+Reload. In Kapitel 6 wird die Arbeit bewertet und ein Ausblick auf weitere Forschungsansätze gegeben.

2 Einordnung in den wissenschaftlichen Kontext

Seitenkanäle wurden schon seit jeher verwendet, um geheime Informationen zu erhalten. In der Wissenschaft wurden sie 1996 zum ersten Mal von Kocher als das Ableiten von Informationen aus für den Angreifer frei zugänglichen Daten definiert [Koc96]. Seitenkanaleffekte von Caches wurden 2006 erstmals durch die Techniken Evict+Time und Prime+Probe ausgenutzt [OST06]. Flush+Reload ist ein 2014 von Yarom et al. vorgestellter Seitenkanalangriff, der auch Cross-Core Angriffe ermöglicht [YF14]. 2016 wurde von Gruss der Flush+Flush Angriff als Variante des Flush+Reload Angriffs entwickelt. Flush+Flush bleibt gegenüber den üblichen Entdeckungsmechanismen, welche Cacheangriffe aufspüren sollen, versteckt [GMWM16]. DABANGG ist eine Menge von Optimierungen von Flush+Reload sowie Flush+Flush Angriffen, die 2020 von Saxena et al. veröffentlicht wurden [SP20]. Maurice et al. haben 2017 mit cache-basierten Seitenkanalangriffen gebaute verdeckte Kanäle für den Aufbau von SSH Verbindungen genutzt [MWS+17]. Cache-basierte Seitenkanäle galten bis dahin als stark fehlerbehaftet, insbesondere durch die Interferenz mit dritten Prozessen. Maurice et al. stellen vor, wie fehlerkorrigierende Codes verwendet werden können, um eine robuste Kommunikation über cache-basierte Seitenkanäle zu realisieren. Insbesondere wird in derselben Arbeit auch untersucht, inwiefern cache-basierte verdeckte Kanäle in der Cloud Anwendung finden. Neben Desktop- und Cloudszenarien wurden 2016 von Lipp et al. auch Angriffe auf Android Smartphones mit ARM Architektur demonstriert [LGS+16].

3 Grundlagen

Im folgenden Kapitel werden die benötigten Grundlagen zum Nachvollziehen der Seitenkanalangriffe sowie der Implementierung und Auswertung besprochen. Zunächst wird die Prozessisolation durch das Verwenden von virtuellem Speicher aufgezeigt. Anschließend wird die Funktionsweise von CPU-Caches erklärt, um auf die folgende Beschreibung von den Cache-Angriffen Flush+Reload und Flush+Flush vorzubereiten. Abschließend folgen Grundlagen zu Netzwerkkommunikation und Statistik.

3.1 Physischer und virtueller Speicher

Moderne `amd64` Systeme verwenden virtuellen Speicher, eine Zuordnung vom virtuellen Prozessspeicher zu den physischen Speichermedien [Int19b, Vol. 1 3-8]. Der virtuelle Speicher ist in Blöcke von 4KiB bis 64KiB[1] eingeteilt, genannt Pages [HL18, 3.3.3]. Der physische Speicher ist in gleich große Blöcke, genannt Page Frames, unterteilt. Eine Page im virtuellen Speicher verweist auf einen Page Frame im physischen Adressraum. Jeder Prozess hat eine eigene Zuordnung, dabei können dieselben Page Frames jedoch von verschiedenen Prozessen verwendet werden [Int19b, Vol. 3A 4].

Die neue Abstraktionsebene durch virtuellen Speicher bringt viele Vorteile gegenüber der ausschließlichen Verwendung von physischen Adressen. Werden geschützte Adressbereiche durch einen Prozess verletzt, ist es trivial einen Speicherzugriffsfehler zu erkennen, da in der prozessspezifischen Zuordnungstabelle klar definiert ist, auf welche Bereiche der Prozess zugreifen darf. Weiterhin ermöglicht virtueller Speicher das Positionieren eines Programms an einer beliebigen virtuellen Adresse, da jedes Programm einen eigenen virtuellen Prozessraum erhält. Dies erspart beim Laden von positionsabhängigem Code aufwendiges Umrechnen von Adressen. Für diesen Kontext insbesondere relevant ist die durch virtuellen Speicher ermöglichte Prozessisolation, wodurch ein unprivilegierter Prozess nicht auf den Speicherbereich eines anderen Prozesses zugreifen kann. [Den96]

Moderne Intel x86-64 CPUs können den virtuellen Adressraum mit 48Bit adressieren [Int19b, Vol. 1, 3.3]. Die Zuordnung, welche virtuelle Page welchem physischen Page Frame entspricht, ist durch eine Translation Table realisiert. Bei Pages von 4KiB werden $2^{48}/4096$ Einträge in der Translation Table benötigt, bei 64Bit pro Eintrag würde sich die Größe dieser Tabelle auf 512GiB belaufen. Um die Platzkosten zu reduzieren, wird eine multi-level Translation Table verwendet; statt einer großen Tabelle gibt es nun mehrere Kleine, auf die sukzessive zugegriffen wird. In Abbildung 1 ist eine solche multi-level Translation Table abgebildet: Das `CR3`-Register bestimmt

[1] Die Größe der Blöcke muss einer Potenz von 2Bytes entsprechen

für den Prozess die Basisadresse der PML4-Tabelle. Die signifikantesten neun Bits der virtuellen Adresse adressieren dann die 512 Einträge, welche wiederum Basisadressen von PDPT-Tabellen darstellen. Auch die PDPT-Tabelle hat 512, mit neun Bits adressierbare, Einträge. Diese Einträge sind entweder Basisadressen von Page Directories, welche dann Basisadressen der Page Tables beinhalten, oder 1GiB große Pages. Die Page Table enhält schließlich die Basisadresse einer 4KiB Page. Die verbleibenden 12 Bits der virtuellen Adresse verweisen dann auf eine der 4096 Bytes innerhalb der Page. [Gru18]

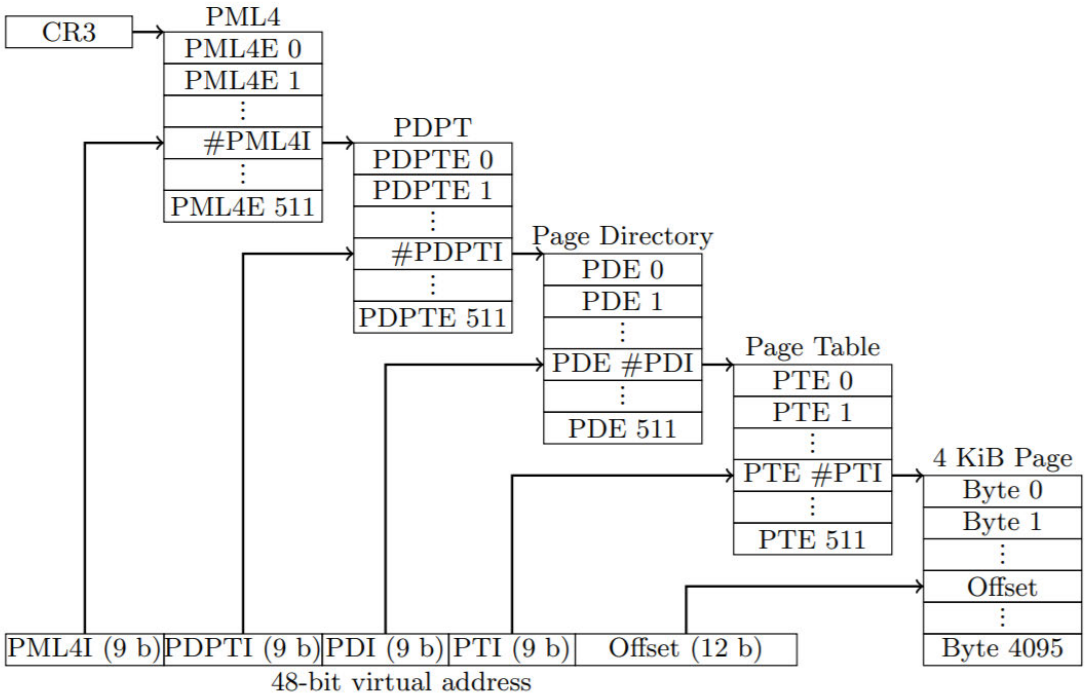

Abbildung 1: *Multi-Level Translation Table [Gru18]*

Die vorgestellte Multi-Level Translation Table hat also vier Translation Tables mit jeweils 512 Einträgen. Falls wir erneut von einer Größe von 64Bit pro Eintrag ausgehen, beläuft sich die Gesamtgröße also auf 16KiB. Der Nachteil besteht darin, dass für jede Umrechnung nun mehrere Speicherzugriffe erforderlich sind. Statt einem Zugriff auf die Translation Table werden nun vier benötigt. Deshalb wird Gebrauch von Caches (siehe Kapitel 3.2) gemacht. Die kleine Größe von 16KiB ermöglicht das Laden der Multi-Level Translation Table in den kleineren, schnelleren Cachespeicher, wodurch prinzipiell kürzere Latenzzeiten erreicht werden.

3.2 CPU-Caches

In diesem Unterkapitel werden zunächst die Funktionsweise und die Vorteile der Verwendung von Caches, insbesondere CPU-Caches, beschrieben. (Abschnitt 3.2.1) Folgend werden gängige Typen von CPU-Caches (Abschnitt 3.2.2), Replacement Policies (Abschnitt 3.2.3) und Adressmodi (Abschnitt 3.2.4) erläutert. Abschließend wird die konkrete Implementierung von modernen

Intel- und AMD-Architekturen besprochen, die in dieser Arbeit insbesondere erforscht werden. (Abschnitte 3.2.5 und 3.2.6)

3.2.1 Allgemeines zu Caches

Caches sind schnelle, meist kleine Speicher, die als Pufferspeicher vor einer anderen Datenquelle geschaltet werden, um die Latenz von Datenzugriffen zu verringern. Ein Router könnte so DNS-Abfragen in einem Cache abspeichern, um bei wiederholten Abfragen die IP direkt aus dem Cache zu entnehmen. Im Kontext dieser Arbeit betrachten wir folgend jedoch ausschließlich CPU-Caches. Der Hauptspeicher auf modernen x86-64 Systemen hat eine hohe Kapazität, würde jedoch aufgrund der verhältnismäßig hohen Latenz einer Abfrage trotzdem ein Bottleneck darstellen, wenn keine Caches eingesetzt würden. [Gru18]

Wird auf eine Adresse des Hauptspeichers zugegriffen, so erhält die CPU die Daten stets aus dem Cache. Dabei wird dann zwischen `Cache Hit` und `Cache Miss` unterschieden. Bei einem `Cache Hit` ist der angeforderte Speicherbereich bereits im Cache gespeichert. Nun kann dieser mit geringer Latenz an die CPU weitergegeben werden. Bei einem `Cache Miss` müssen die Daten erst aus dem Hauptspeicher geladen werden. In diesem Fall gibt es für den Speicherzugriff keinen Vorteil gegenüber einer Architektur ohne Cache. Nun werden jedoch die angeforderten Daten in dem Cache gespeichert, mit der Grundannahme, dass eine gerade verwendete Adresse wahrscheinlicher erneut benötigt wird, als eine beliebige andere Adresse. In welcher Cachezeile (siehe Abschnitt 3.2.2) der neue Block gespeichert wird, und was gegebenenfalls mit bereits dort gespeicherten Daten passiert, hängt von dem Typen des Caches sowie der gewählten Cache Replacement Policy ab. [Anl19]

3.2.2 Typen von CPU-Caches

Die kleinste Einheit eines CPU-Caches sind die Cachelines, in denen jeweils ein Speicherblock gespeichert wird. Aus Sicht des Caches ist der Hauptspeicher in Speicherblöcke unterteilt, welche der Größe der Cachelines entsprechen. Bei modernen Systemen (siehe Abschnitte 3.2.5 und 3.2.6) sind Cachelines und Speicherblöcke meistens 64Byte groß. Im Folgenden werden die drei wichtigsten Typen von CPU-Caches vorgestellt.

Directly Mapped Cache

In einem Directly Mapped Cache kann ein Speicherblock in genau einer bestimmten Cachezeile gespeichert werden. Da der Cache kleiner als der Hauptspeicher ist, teilen sich mehrere Adressen im Hauptspeicher eine Cacheline. Wenn bereits ein Speicherblock im Cache gespeichert ist, muss dieser also beim Zugriff auf einen neuen Block entfernt werden. Die untersten b Adressbits werden, bei einer Blockgröße von 2^b Bytes, ignoriert. Üblich ist die Verwendung von den verbleibenden niedrigwertigen Adressbits als Index im Cache. So wird vermieden, dass benachbarte Speicherblöcke auf dieselbe Cachezeile gemappt werden, da das konsekutive Laden von sequenziellen Speicheradressen häufig Verwendung in Programmen findet. Soll nun auf die Adresse `addr` zugegriffen werden, wird der diese Adresse beinhaltende Speicherblock angefordert. Die nächsthöherwertigen n Bits der Adresse beschreiben nun den Index der Cacheline innerhalb des Caches. Die Anzahl der

Indexbits n wird durch die Größe des Caches bestimmt, es gibt stets 2^n Cachezeilen. Die verbleibenden, höchstwertigen Bits der Adresse werden zusammen mit dem Speicherblock als Tag im Cache gespeichert. Durch den Vergleich des gespeicherten Tags in der Cachezeile mit dem Tag der gewünschten Adresse wird so entschieden, ob der angeforderte Speicherblock bereits gespeichert ist, es sich also um einen `Cache Hit` handelt. Der Nachteil des Directly Mapped Caches besteht darin, dass nur ein Speicherblock mit demselben Tag gleichzeitig gecached sein kann. Im Worst Case führt dies zu vielen `Cache Misses`. [Anl19] [Gru18]

Fully Associative Cache

Im Fully-Associative Cache kann jeder Speicherblock in jeder Cacheline gespeichert werden, dadurch kommt der Cache ohne Index aus. Da nun beim Zugriff auf eine Speicheradresse mehrere Cachelines möglicherweise den gewünschten Block enthalten, muss der gespeicherte Tag in jeder Cachezeile mit dem Tag der gewünschten Adresse verglichen werden. Ist der Vergleich mit dem Tag einer Cacheline positiv, wird ein `Cache Hit` ausgelöst. Stimmt kein Tag überein, führt die Abfrage zu einem `Cache Miss`. Im Fall des `Cache Hits` wird dann der Inhalt der entsprechenden Cachezeile an die CPU weitergegeben. Falls jede Cachezeile belegt ist, aber ein neuer Speicherblock gespeichert werden soll, entscheidet die Cache Replacement Policy (siehe Abschnitt 3.2.3) welche Cachezeile überschrieben werden soll. Der Fully Associative Cache hat den Vorteil, dass kongruente Speicherblöcke simultan im Cache gespeichert sein können. Dafür ist dieser jedoch sehr aufwendig gebaut, für jede Cachezeile wird ein Komperator-Schaltnetz benötigt. [Gru18]

Set Associative Cache

Set Associative Caches stellen einen Kompromiss zwischen der Performanz des Fully Associative Cache, und der Kosteneffizienz des Directly Mapped Cache dar. Wie Directly Mapped Caches werden auch Indizes verwendet; auf einen Index, und damit auf eine Gruppe von kongruenten Speicherblöcken, kommen aber mehrere Cachezeilen. Der Verbund von allen Cachezeilen für einen gewählten Index wird Set genannt. Enthält ein Set m Cachezeilen, wird der Cache auch *m-way set associative* genannt. Ein Set funktioniert wie ein Fully Associative Cache. Damit sind die bisher vorgestellten Cachetypen Spezialfälle des Set Associative Caches. Der Fully Associative Cache hat lediglich ein Set, beim Directly Mapped Cache enthält jedes Set nur eine Cachezeile. Wie beim Fully Associative Cache muss auch innerhalb eines Sets eine Cache Replacement Policy (siehe Abschnitt 3.2.3) verwendet werden. [Anl19][Gru18]

3.2.3 Cache Replacement Policies

Wenn bei dem Laden eines Speicherblocks alle Cachelines, denen der Speicherblock durch die Mapping-Funktion zugewiesen wurde, bereits verwendet werden, muss ein Speicherblock aus einer dieser Cachelines verdrängt werden. Diese Speicherblöcke werden kongruent genannt.

Welcher kongruente Speicherblock nun aus dem Cache verdrängt wird, ist durch die Cache Replacement Policy festgelegt. Optimalerweise sollte der Speicherblock verdrängt werden, der in Zukunft zuletzt gebraucht wird. Da technisch kein Blick in die Zukunft möglich ist, wurden verschiedene Replacement Policies entwickelt, welche sich dem theoretische Ziel möglichst annähern sollen. Bei einem `directly mapped cache` muss keine Strategie gewählt werden, da immer nur eine

kongruente Zeile gleichzeitig im Cache geladen sein kann, und somit verdrängt werden muss. Es folgen die prominentesten Replacement Policies: [Anl19]

- **least recently used (LRU)** gilt als die beste Strategie [Anl19]. Hierbei wird der am längsten nicht mehr verwendete kongruente Speicherblock aus dem Cache entfernt. Zugrunde liegt die Annahme, dass ein für lange Zeit nicht mehr verwendeter Block auch in Zukunft lange nicht mehr verwendet wird. Problematisch kann für LRU der konsekutive und sequentielle Zugriff auf kongruente Speicherblöcke sein: ist die Anzahl der kongruenten Blöcke größer als die Anzahl der Wege im Cache, beziehungsweise die Anzahl der Cachelines in einem `fully associative` cache, treten vermehrt Cachemisses auf.
- **least frequently used (LFU)** verdrängt die Cacheline, auf die am wenigsten häufig zugegriffen worden ist. Zugrunde liegt die Annahme, dass auf diese Cacheline auch in Zukunft selten zugegriffen wird.
- **FIFO** verdrängt den ältesten kongruenten Speicherblock. LRU wird zwar als beste Strategie angesehen, insbesondere bei Caches mit mehreren Wegen steigen die Kosten für die Implementierung jedoch stark. Deswegen kann FIFO auch als Annäherung von LRU verwendet werden, unabhängig von der Anzahl der Wege wird lediglich eine FIFO-Datenstruktur zur Verwaltung der Speicherblöcke benötigt.
- **most recently used (MRU)** verdrängt den zuletzt verwendeten kongruenten Speicherblock.
- **LIFO** verdrängt den zuletzt geladenen kongruenten Speicherblock.
- **random replacement (RR)** verdrängt einen zufälligen kongruenten Speicherblock. In der Praxis hat sich gezeigt, dass bei großen Caches der Performanzgewinn durch Cache Replacement Policies gering ist [Anl19], `random replacement` setzt also darauf, die Kosten der Implementierung der Replacement Policy zu minimieren.

3.2.4 Cache Addressing Modes

In Abschnitt 3.1 wurde die Unterscheidung von physischem und virtuellem Speicher besprochen. Caches können also physischen oder virtuellen Speicher für Tag und Index verwenden. Dabei sind drei Kombinationen üblich; `virtually-indexed virtually-tagged` (VIVT), `physically-index physically-tagged` (PIPT) und `virtually-indexed physically-tagged` (VIPT) [Gru18]. Diese Varianten werden im Folgenden diskutiert.

Virtually-indexed virtually-tagged (VIVT)

Ein VIVT-Cache verwendet virtuelle Adressen für Index und Tag. Dies führt zu einer geringen Latenz beim Zugriff, da der angeforderte Speicherblock ohne Umrechnung in eine physische Adresse aus dem Cache gelesen werden kann. Dafür kann Memory Deduplication nicht verlässlich verwendet werden. Wenn sich die virtuellen Adressen von gemeinsam genutzten Bibliotheken im virtuellen Speicher der Prozesse unterscheiden, werden entsprechende Speicherblöcke mehrfach in den Cache geladen, was zu einem erhöhten Speicheraufwand führt [Gru18]. Das zugrundeliegende Problem ist also, dass Speicherblöcke mit gleichen physischen Adressen unterschiedliche virtuelle Adressen, und somit Indizes und Tags, haben können, auch genannt "cache-line aliasing problem" [Bot04]. Analog können verschiedene Speicherblöcke bei verschiedenen Prozessen die gleichen

virtuellen Adressen haben. Es kann deswegen notwendig sein, bei einem Kontextswitch den Cache zu invalidieren. [Gru18]

Physically-indexed physically-tagged (PIPT)

Durch das Verwenden von physischen Adressen für Index und Tag vermeiden PIPT-Cache die Nachteile von VIVT-Caches. Memory Deduplication funktioniert wie im Hauptspeicher, und der Cache ist auch nach einem Kontextswitch sicher gültig. Dafür muss vor dem Cachezugriff die physische Adresse errechnet werden, was zu einer höheren Latenz als bei VIVT-Caches führt. [Gru18]

Virtually-indexed physically-tagged (VIPT)

VIPT-Caches verwenden lediglich eine physische Adresse für den Tag. In Abbildung 2 ist ein solcher Cache dargestellt; der virtuelle Index kann sofort verwendet werden, um das entsprechende Cacheset zu bestimmen. Während der Cache das Cacheset lädt, bestimmt eine Translation Table, hier die TLB, den physischen Tag und Offset aus der virtuellen Adresse. Dieser physische Tag wird dann mit den Tags im Cacheset verglichen, und der Offset gegebenenfalls addiert. Der VIPT-Cache ist schnell, da das Laden der Cacheline und das Bestimmen des physischen Tags parallel durchgeführt werden können. Der VIPT-Cache leidet nicht unter dem "cache-line aliasing problem", da gleiche physischen Speicherblöcke auf Grund des physischen Tags im Cache zusammengefasst werden. Damit dies funktioniert, müssen Tag und Offset die physische Adresse darstellen, weswegen mehr Bits für den Tag benutzt werden müssen. Insgesamt vereint der VIPT-Cache die geringe Latenz des VIVT-Caches mit den Vorteilen des PIPT-Caches. [Boto4]

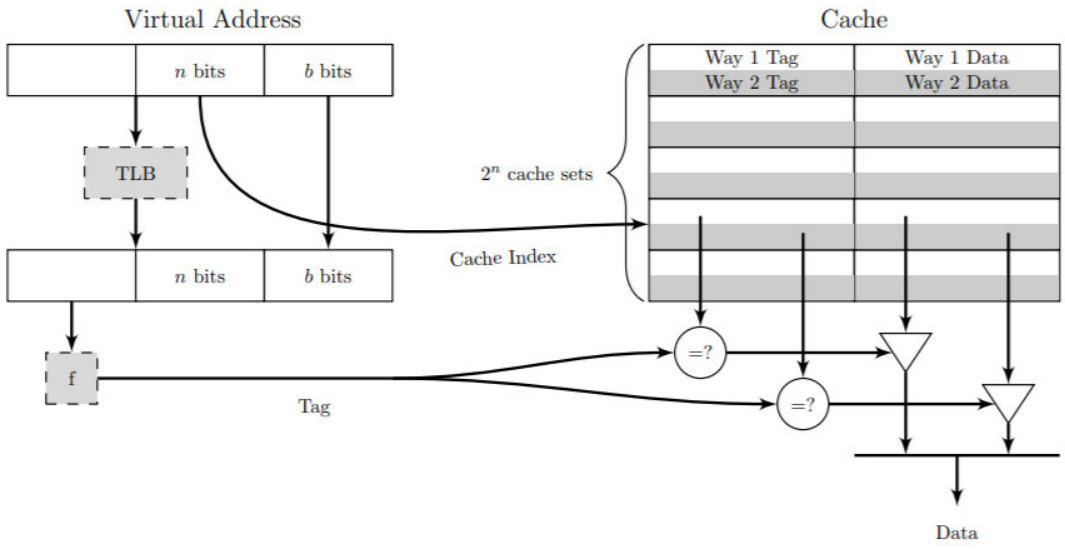

Abbildung 2: *Virtually-indexed physically-tagged Cache [Gru18]*

3.2.5 Caches moderner Intel x86-64 Desktop CPUs

Abbildung 3 zeigt die Architektur eines modernen Intel Core i7 x86-64 Prozessors mit Multi-Core Technologie. Dabei gibt es drei Typen von Caches; *L1*, *L2* und *L3* [Int19b, Vol 3A. 11-1]. *L1* Caches sind am kleinsten, haben die höchste Bandbreite und die geringste Latenz. Die *L3* Caches sind analog am größten, haben die niedrigste Bandbreite und die höchste Latenz [Gru18]. *L1* Caches sind jeweils Daten- oder Instruktionscaches, die *L2* Caches speichern sowohl Daten als auch Instruktionen, auch genannt 'unified'. Beide Caches sind privat für einen CPU-Kern. [Int19a, 1.9] Der *L3* Cache ist, anders die *L1* und *L2* Caches, von allen Kernen geteilt. Insbesondere ist der *L3* Cache inklusiv zu den *L1* und *L2* Caches, das bedeutet, dass alle in den kleineren Caches vorhandenen Cachelines auch im *L3* gecacht sein müssen. [Gru18] Es ergibt sich, dass auch der *L3* Cache sowohl Instruktionen als auch Daten speichert. Anders als bei per Core *L3* Caches , die per Bus verbunden sind, kann der gesamte Cache kann von einem einzigen Kern effizient genutzt werden. Intel nennt dies 'Intel® Smart Cache'. [Int19c] Die *L1*- und *L2* Caches haben jeweils einen TLB, siehe Abschnitt 3.2.4. Diese sind typischerweise schnelle VIVT-Caches. *L2* und *L3* selbst sind hingegen PIPT-Caches, profitieren also von Memory Deduplication. [Gru18] Durch das Vorschalten einer TLB scheint es wahrscheinlich, dass die *L1* Caches entweder als VIPT- oder PIPT-Caches implementiert sind. Als Replacement Policy, siehe Abschnitt 3.2.3, wird ein Hybrid aus Least Recently Used (LRU) und einer modifizierten LIFO verwendet. Dies kann beim Zugriff auf viele kongruente Cachelines die Hit-Rate verbessern [Gru18]. Mithilfe von `CPUID` [Int19b, Vol. 2A 3-198] kann die Größe einer Cacheline von 64Byte auf einem `i7 9700K` bestätigt werden, dies entspricht dem Wert für fast alle modernen CPUs [Gru18].

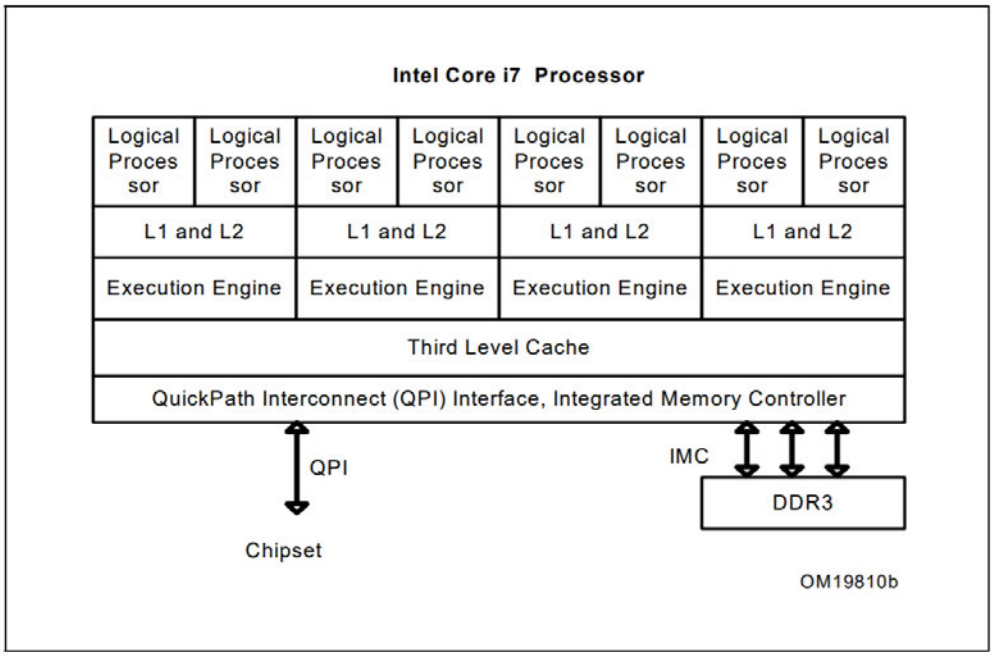

Abbildung 3: *Cache on Intel Multi-Core Technology CPUs [Int19b, Vol. 1 2-19]*

3.2.6 Caches moderner AMD Zen/Zen+/Zen2 CPUs

Wie bei modernen Intel x86-64 CPUs (siehe Abschnitt 3.2.5) gibt es die drei Caches *L1*, *L2* und *L3*. Auch sind die *L1* Caches jeweils in Instruktions- und Datencaches aufgeteilt, und der *L2* ist 'unified'. Anders als bei den modernen Intel Prozessoren ist der *L2* Cache jedoch inklusiv zu den *L1* Caches, enthält also alle Cachelines der kleineren Caches. Der *L3* Cache ist ein 'victim cache', das heißt, die aus dem *L2* verdrängten Cachelines werden im *L3* gecacht. [AMD19] Die Cachelines sind jeweils 64 Byte groß.

3.3 CLFLUSH

Die Instruktion `CLFLUSH`, für 'CacheLineFlush', invalidiert den Eintrag der angegebenen Cacheline (siehe Kapitel 3.2) aus dem Cache. [Int19b, Vol. 2A 3-145] Anschließend werden alle Kopien der Cacheline aus allen Levels im Cache gelöscht [Gru18]. Dabei können jedoch nur die virtuellen Speicherbereiche des aufrufenden Prozesses geflusht werden. Für `CLFLUSH` werden keine erhöhten Rechte benötigt. [Int19b, Vol. 2A 3-145]

Die C-Signatur sieht folgend aus:

```
1 void _mm_clflush(     // kein Rückgabewert
2     void const *p     // eine Adresse in der zu flushenden Cacheline
3 );
```

Die 64 Byte große Cacheline, welche die im Parameter `p` übergebene Adresse enthält, wird geflusht. [Int19b, Vol. 2A 3-145]

Die Instruktion benötigt weniger Taktzyklen, wenn die zu invalidierenden Daten nicht im Cache geladen sind. Da keine Kopien aus dem Cache gelöscht werden müssen, kann die Instruktion vorzeitig abgebrochen werden. Des Weiteren benötigt `CLFLUSH` mehr Taktzyklen, wenn die entsprechenden Daten modifiziert wurden, da in diesem Fall ein 'Writeback' in den Hauptspeicher stattfinden muss. [Gru18]

`CLFLUSH` ist durch `mfence` geordnet, ein Aufruf von `mfence` erzwingt die Ausführung von vorstehenden `CLFLUSH` Instruktionen. [Int19b, 4-22 Vol. 2B / Vol. 2A 3-145]

3.4 Flush+Reload

Flush+Reload ist ein cache-basierter Seitenkanalangriff, insbesondere auf modernen Intel Prozessoren. Dabei werden Unterschiede bei den benötigten Taktzyklen bei Speicherzugriffen als Seitenkanal ausgenutzt, um Informationen über den angegriffenen Prozess zu gewinnen. Grundlage hierfür ist, dass der Speicherzugriff eine deutlich kürzere Latenz aufweist, wenn die angeforderte Speicheradresse im schnellen Cachespeicher (siehe Abschnitt 3.2) liegt. [YF14] Abbildung 4 stellt gemessene Zugriffszeiten auf einem Beispielsystem[2] dar. Durch das Messen der benötigten Taktzyklen eines Speicherzugriffs wird deduziert, ob sich die angeforderte Speicherzeile bereits im Cache befindet.

[2] Archlinux auf Intel Core i7 9700K

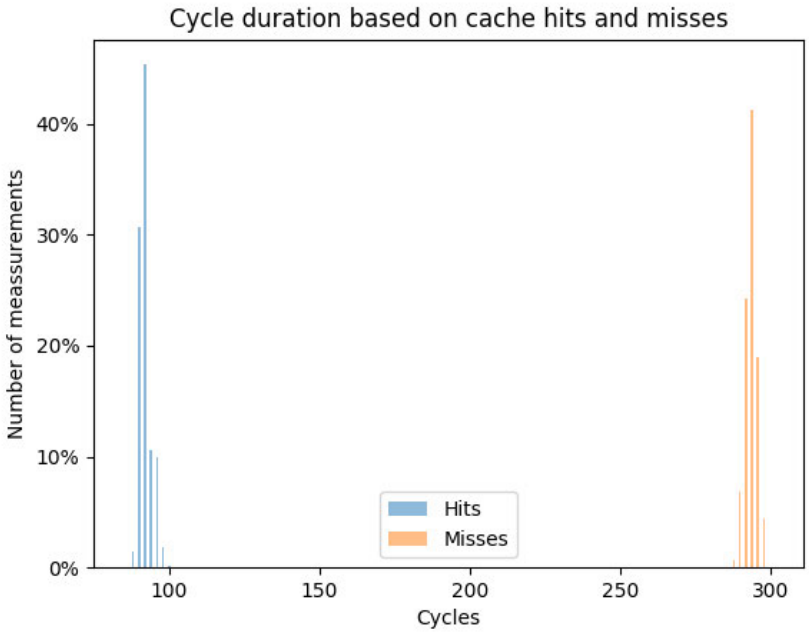

Abbildung 4: *Timingdifferenzen bei Speicherzugriff auf einem Beispielsetup*

In Abbildung 5 wird der Flush+Reload Angriff beschrieben. Hierbei wird ein Speicherblock, nachfolgend p genannt, überwacht, der sowohl in den virtuellen Speicherbereichen von Angreifer und Opfer, als auch im Physischen befindlich ist. Angreifer und Opfer befinden sich im Beispiel auf unterschiedlichen Kernen, sichtbar an den getrennten $L1$ Caches. Im ersten Schritt "Flush" ruft der Angreifer `CLFLUSH` auf (siehe Abschnitt 3.3). Dies führt dazu, dass p aus allen Cachelevels entfernt wird, auch dem $L1$ des Opfers. Im zweiten Schritt "Access/No Access" wartet der Angreifer eine festgelegte Zeitspanne. Das Opfer lädt p bei Bedarf in den Cache (2b), wobei der Speicherblock vom Hauptspeicher zunächst in den $L3$ Cache und anschließend in den $L1$ Cache geladen wird. Alternativ benötigt das Opfer p nicht (2a), p ist weiterhin nicht im Cache präsent. Zuletzt wird in der "Reload"-Phase p vom Angreifer geladen. Falls das Opfer p im zweiten Schritt nicht geladen hat (3a), muss p zunächst aus dem Hauptspeicher in $L3$ geladen werden, bevor p in den entsprechenden $L1$ geladen werden kann. Dies führt zu vielen benötigten Taktzyklen (Cache Miss). Hat das Opfer p jedoch bereits geladen, kann p direkt aus dem $L3$ geladen werden, wenige Taktzyklen sind nötig (Cache Hit). Abhängig von den benötigten Taktzyklen kann der Angreifer nun deduzieren, ob das Opfer p während der zweiten Phase verwendet hat. [Gru18] Die Grenzwerte für die Klassifikation von Cache Hits oder Cache Misses sind vom System abhängig [YF14].

Damit die vom Opfer benutzten Cacheeinträge manipuliert werden können, muss eine virtuelle Adresse des Angreifers auf dieselbe physische Adresse abbilden, wie die abzuhörende virtuelle Adresse des Opfers [YF14]. Flush+Reload setzt also geteilten Speicher voraus. Dies kann insbesondere in der Cloud die Nutzung von Flush+Reload einschränken, da dort oft memory deduplication aus Sicherheitsgründen zwischen virtuellen Maschinen deaktiviert ist [MWS+17]. Da bei modernen Intel Prozessoren, siehe Abschnitt 3.2.5, der $L3$-Cache inklusiv ist, also alle Daten aus den $L1$-Caches beinhaltet, funktioniert Flush+Reload wie oben beschrieben auch Cross-Core [YF14]. Bei den Zen-Familien von AMD, siehe Abschnitt 3.2.6, ist jedoch nicht klar, ob ein von allen geteilter,

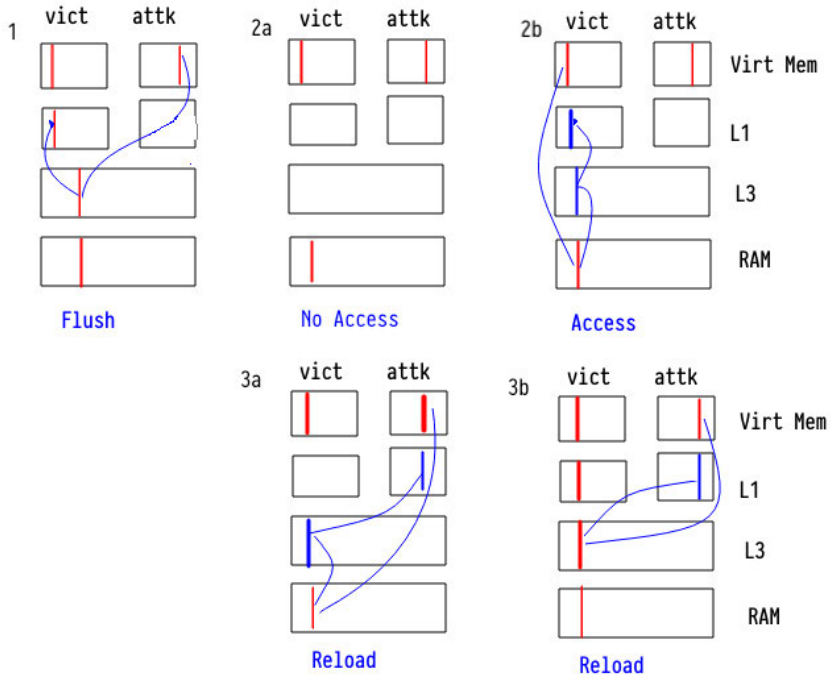

Abbildung 5: *Flush+Reload Angriff*

inklusiver Cache vorhanden ist. Somit bleibt offen, ob Flush+Reload auch Cross-Core für Angriffe verwendet werden kann. Auf diese Frage wird in Abschnitt 5.3 eingegangen. Der geteilte *L*3-Cache enthält die verdrängten Daten der privaten *L*2-Caches. Die entsprechende Cachezeile kann mit Hilfe von `CLFLUSH` invalidiert werden. Auch beim erneuten Laden wird die Cachezeile aber nicht in den geteilten *L*3 geladen, sondern nur dann, wenn der private *L*2 Cache diese verdrängt. Für Flush+Reload Angriffe auf Zen muss sich der Angreifer auf demselben Kern wie das Opfer befinden. [YSG+19] Flush+Flush benötigt keine Privilegien [YF14]; `CLFLUSH` und Speicherzugriffe sind beide unprivilegiert [Int19b, Vol. 2A 3-145].

Soll das Opfer über einen Zeitraum abgehört werden, werden mehrere Flush+Reload-Zyklen nachfolgend ausgeführt. Abbildung 6 zeigt einen solchen zeitlichen Verlauf. Nur dann, wenn ein Speicherzugriff des Opfers während der Access-Phase erfolgt, kann dieser vom Angreifer entdeckt werden. Also folgt die "Flush"-Phase des nächsten Zyklus stets möglichst direkt auf die "Reload"-Phase, damit keine Speicherzugriffe des Opfers verpasst werden. Die verbleibende Variable besteht also in der Dauer der Access-Phase. Bei einer kürzeren Access-Phase sinkt der Anteil dieser im Vergleich zur "Flush"- und "Reload"-Phase, wodurch die Trefferquote sinken kann [YF14]. Bei einer längeren Access-Phase steigt der potentielle zeitliche Abstand vom Speicherzugriff durch das Opfer und dem Reload; dadurch steigt auch die Wahrscheinlichkeit auf falsche Negative durch Cacheverdrängung durch Dritte. Analog steigt bei Non-Access auch die Zeit zwischen "Flush" und "Reload", und damit auch die Wahrscheinlichkeit von falschen Positiven, falls dritte Prozesse den beobachteten Speicherbereich cachen. Es ist also eine Balance zwischen kurzer und langer Dauer der "Access"-Phase zu finden. Weiterhin können falsch Positive auch durch das spekulative Prefetching der Speicherzugriffe des Angreifers auftreten [YF14].

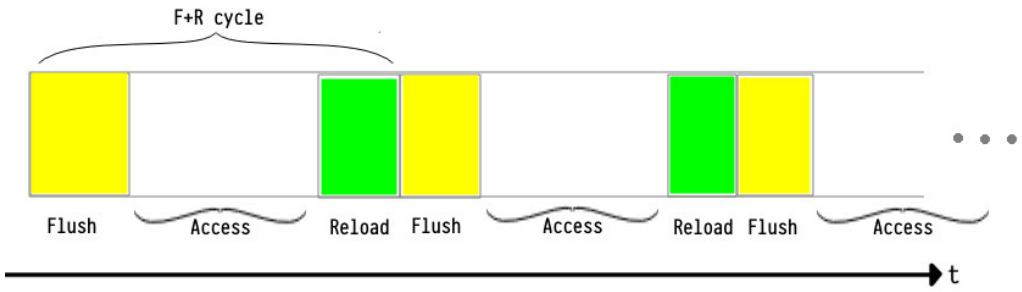

Abbildung 6: *Zetlicher Verlauf des Flush+Reload Angriffs*

3.5 Cache Attack Detection mit Hardwareperformanzanalyse

Eine Möglichkeit gegen Cacheangriffe wie Flush+Reload (siehe Kapitel 3.4) vorzugehen, ist das Entdecken von laufenden Angriffen auf dem System. Flush+Reload verursacht durch den wiederholten Speicherzugriff Cache-Hits und Cache-Misses. Hits und Misses werden, neben anderen Informationen, in speziellen Registern, so genannten Hardwareperformanzzählern, gezählt. Diese Zähler werden üblicherweise für die Performanzoptimierung verwendet, wurden aber auch zum Detektieren von Flush+Reload und anderen Cacheangriffen vorgeschlagen. Falls die `CACHE_MISSES` oder `CACHE_REFERENCES` Counter besonders viele Cache-Misses und Hits respektive auf dem Last-Level-Cache L3 anzeigen, wird ein Cacheangriff vermutet. Es wurde gezeigt, dass diese Technik zum Detektieren von verschiedenen Cacheangriffen, etwa Flush+Reload, Prime+Probe und Rowhammer Angriffen, tauglich ist [Gru18]. Da `CLFLUSH` jedoch keine Cache-Misses oder Hits verursacht, und auch sonst nicht von Hardwareperformanzzählern beobachtet werden kann, wurde subsequent Flush+Flush (siehe Abschnitt 3.6) entwickelt. Ein Flush+Flush-Angreifer kann nicht durch Cache Detection mit Hardwareperformanzanalyse entdeckt werden. [GMWM16]

3.6 Flush+Flush

Flush+Flush ist ein cache-basierter Seitenkanalangriff und eine Variante des Flush+Reload Angriffs (siehe Abschnitt 3.4). Anders als dem Flush+Reload Angriff, welcher auf unterschiedlichen Zugriffszeiten auf den Cache basiert, wird bei Flush+Flush die Timingdifferenz der `CLFLUSH`-Instruktion (siehe Abschnitt 3.3) ausgenutzt. Abhängig davon, ob die zu flushende Cacheline im Cache vorhanden ist, benötigt die Instruktion zusätzliche Taktzyklen um den Cache zu flushen. Befindet sich die Cacheline nicht im Cache, kann die Instruktion vorzeitig abschließen. In Abbildung 7 sind die Timingdifferenzen der `CLFLUSH`-Instruktion dargestellt. [GMWM16]

Analog zum Flush+Reload besteht der Flush+Flush Angriff aus drei Phasen. Im ersten Schritt wird die beobachtete Cacheline aus dem Cache verdrängt. Anschließend lädt das Opfer im zweiten Schritt die entsprechende Cacheline entweder nach, oder nicht. Im dritten Schritt unterscheidet sich Flush+Flush von Flush+Reload: der Angreifer führt erneut `CLFLUSH` aus. Wurde die Cacheline vom Opfer nicht geladen, benötigt der Angreifer lediglich eine kurze Dauer für die `CLFLUSH`-Instruktion. Hat das Opfer im zweiten Schritt die Cachline nachgeladen, benötigt der Angreifer mehr Takte. Die Interpretation von kurzen und langen Latenzen ist also invers zum Flush+Reload Angriff. [GMWM16]

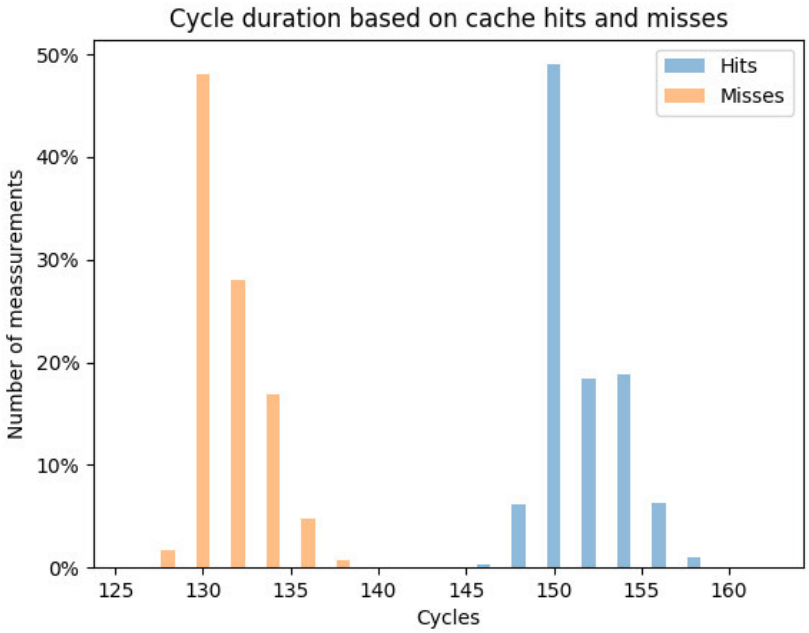

Abbildung 7: *Timingdifferenzen bei Nutzung der* CLFLUSH*-Instruktion auf einem Beispielsetup*

Anders als bei Flush+Reload Angriffen müssen diese drei Phasen jedoch nicht in jedem Zyklus wiederholt werden — das CLFLUSH im dritten Schritt genügt nicht nur der Messung, sondern flusht bereits den Cache für den nächsten Zyklus. Abbildung 8 beschreibt den modifizierten Ablauf. Auch bei Flush+Flush ist die Länge der "Access"-Phase frei wählbar, eine längere Dauer erhöht potentiell falsch Negative durch Interferenz durch Dritte. Eine kürzere Dauer kann die Genauigkeit senken, da Speicherzugriffe des Opfers zu einem größeren Anteil in die "Flush"-Phase fallen. Wie bei Flush+Reload ist also auch bei Flush+Flush eine Balance zu finden, um eine möglichst hohe Genauigkeit zu erzielen.

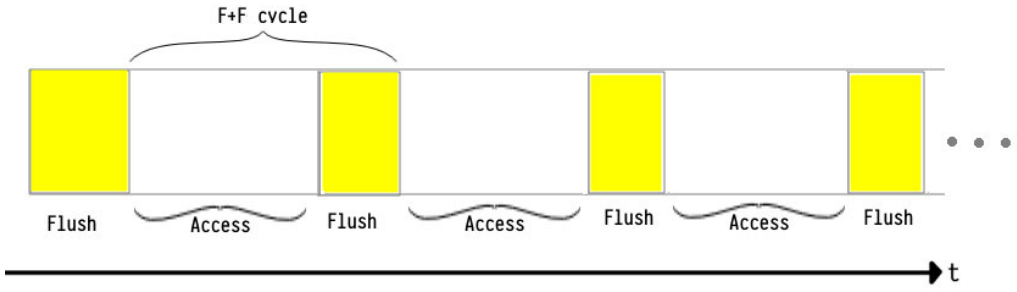

Abbildung 8: *Zetlicher Verlauf des Flush+Flush Angriffs*

Der wichtigste Unterschied zwischen Flush+Flush und Flush+Reload liegt darin, dass Flush+Flush "stealthy" ist. Da CLFLUSH weder Hits noch Misses auslöst, kann der Flush+Flush Angreifer nicht durch Hardwareperformanzanalyse entdeckt werden (siehe Abschnitt 3.5), anders als bei Flush+Reload. Zusätzlich wird die Spezifität reduziert, da CLFLUSH im Gegensatz zu Speicherzugriffen keine spekulativen Prefetches auslöst, welche zu falschen Positiven führen können.

Vergleicht man dafür Abbildung 4 mit Abbildung 7, wird ersichtlich, dass die Timingdifferenzen bei Flush+Flush kleiner sind. Dies macht es schwieriger, Hits und Misses voneinander zu unterscheiden. Da die benötigten Zugriffszeiten bei `CLFLUSH` jedoch im Worst Case deutlich geringer sind als bei Speicherzugriffen, kann Flush+Flush mit einer höheren Frequenz als Flush+Reload betrieben werden. [GMWM16]

3.7 Typen von Übertragungsfehlern

Wird für die Kommunikation ein cache-basierter verdeckter Kanal verwendet, können verschiedene Bitübertragungsfehler unterlaufen. Diese Fehlertypen müssen verhindert, korrigiert, oder entdeckt werden. Die Fehlertypen werden folgend aufgelistet: [MWS+17]

- **Substitution Errors** sind einzelne, geflippte Bits. Diese treten insbesondere dann auf, wenn dritte Programme die zur Kommunikation genutzten Cachezeilen verdrängen.
- **Insertion Errors** sind Fehler, bei denen ein oder mehrere Bits in den interpretierten Bitstrom eingefügt werden. Dies könnte auftreten, wenn der Sender vom Scheduler unterbrochen ist, und für eine ganze Zeiteinheit nicht senden kann.
- **Deletion Errors** sind Fehler bei denen, anders als bei Insertion Errors, Bits aus dem interpretierten Bitstrom ausgeschnitten werden. Dies tritt analog auf, wenn der Empfänger vom Scheduler unterbrochen ist, und für eine ganze Zeiteinheit nicht empfangen kann.
- Ein **Burst Error** ist eine Ansammlung von mehreren Errors. Fehler, die durch den Scheduler verursacht werden, sind häufig Burst Errors.

Welche Fehler genau auftreten, hängt auch von Implementierungsdetails ab. So könnten Unterbrechungen des Senders durch den Scheduler, wie oben beschrieben, entweder zu Insertion Errors führen, wenn die Taktung abhängig von der Prozesszeit ist, oder es können Substitution Errors auftreten, wenn die systemweite, monotone Realtime verwendet wird. Analog würden auch die durch den unterbrochenen Empfänger verursachten Deletion Errors zu Substitution Errors werden.

3.8 Das Ethernet Frame

Ethernet ist die am weitesten verbreitete Local Area Network-Technologie [Spuoo, Kapitel 1]. Das Ethernet Frame ist die Data Link Layer PDU[3] der Ethernet-Technologie. Ein Ethernet Frame ist folgend aufgebaut:

- **Preamble** 56Bit: Alternierend 1 und `0`. Das Preamble ermöglicht es dem Empfänger den Takt anzupassen.
- **SFD** 8Bit: Signalisiert das Übertragen eines Frames.
- **Destination Address** 48Bit: MAC-Adresse des Adressaten.
- **Source Adress** 48Bit: MAC-Adresse des Absenders.

[3] Eine *protocol data unit* (PDU) ist die kleinste Informationseinheit auf der entsprechenden OSI-Schicht.

- **Length** 16Bit: Beträgt der Wert des Felds weniger oder gleich 1500 signalisiert das Feld die Länge des Payloads. Beträgt der Wert mindestens 1536 signalisiert das Feld einen vordefinierten Datentyp des Payloads im Ethernet-Standard.[4]
- **Data** 46 bis 1500Byte: Der Payload. Beträgt die angegebene Länge weniger als 46Byte wird der Payload auf 46 gepaddet.
- **Frame Check Sequence** 32Bit: Cyclic-Redundancy-Check (CRC) Checksumme der Kontroll- und Datenfelder. Der Empfänger kann die Checksumme bilden und mit der übertragenen Checksumme vergleichen. Sind die Checksummen gleich, wurde das Frame korrekt übertragen.

3.9 Methoden um Übertragungen zu bewerten

In der vorliegenden Arbeit sollen verschiedene verdeckte Kanäle in Alltagsszenarien getestet und miteinander verglichen werden. Zum Vergleichen und Bewerten der Performance der Kanäle werden folgend Methoden vorgestellt, um die effektive Kapazität nach Shannon zu bestimmen (Abschnitt 3.9.1) sowie die Art der Fehler zu identifizieren (Abschnitte 3.9.2 und 3.9.3).

3.9.1 Channel Capacity nach Shannon

Werden Bits über einen Kanal kommuniziert, können je nach Beschaffenheit des Kanals Bitfehler auftreten. Fehlertypen wurden in Abschnitt 3.7 behandelt. Das Shannon Theorem tätigt Aussagen über solche Kanäle, in denen lediglich unabhängige Substitution Errors vorkommen.

Lemma 1: *Sei M die Anzahl der zu übertragenden Bits und p die unabhängige Wahrscheinlichkeit, dass ein Bitfehler beim Übertragen eines Bit auftritt. Sei weiterhin $H(x)$ die binäre Entropiefunktion mit $H(x) = -x\log_2 x - (1-x)\log_2(1-x)$. Dann existiert ein Code der Länge $N = M + N * H(p) = \frac{M}{1-H(p)}$ für den die Nachricht mit einer gegen 1 gehende Wahrscheinlichkeit wiederhergestellt werden kann. Weiterhin muss ein Code mit dieser Eigenschaft mindestens N Bits lang sein.*

[Kleo8]

Sei die Effizienz des Codes definiert als $\frac{M}{N}$, also die Datenbits pro Bit des Codes. Abbildung 9 zeigt die Effizienz abhängig von der Genauigkeit[5]. Es gilt:

$$\text{Effizienz} = \frac{M}{N} = \frac{M}{\frac{M}{1-H(p)}} = (1 - H(p))$$

Für einen Kanal der Bandbreite f und einer Fehlerrate von p gilt nun: die effektive Kapazität C, des Kanals beträgt:

$$C = f * \text{Effizienz} = f * (1 - H(p))$$

Über die effektive Kapazität lassen sich nun Kanäle mit unterschiedlichen Bandbreiten und Fehlerraten vergleichen.

[4] Der Bereich zwischen 1501 und 1535 ist absichtlich undefiniert [Spuoo, Kapitel 1].
[5] Es lässt sich zeigen, dass $H(p) = H(1-p)$, also gilt durch $acc = 1 - err$ auch $H(err) = H(acc)$

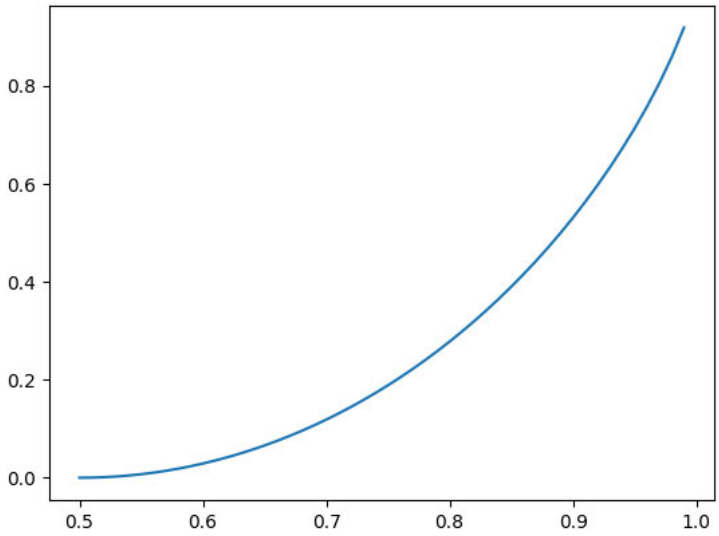

Abbildung 9: *Effizienz des Codes abhängig von der Genauigkeit*

3.9.2 Konfusionsmatrix

Die Konfusionsmatrix schlüsselt die Performance einer Klassifikation detailliert auf. Sie ist ein Tupel aus Richtige Negative (RN), Falsche Positive (FP), Falsche Negative (FN) und Richtige Positive (RP).

- RN: korrekt als '0' klassifiziert
- FP: fälschlicherweise als '1' klassifiziert, eigentlich '0'
- FN: fälschlicherweise als '0' klassifiziert, eigentlich '1'
- RP: korrekt als '1' klassifiziert

Die Genauigkeit wird durch $\frac{RN+RP}{RN+RP+FN+FP}$ berechnet.

3.9.3 Wasserstein-Distanz

Die Wassersteindistanz gibt an, wie unterschiedlich zwei Verteilungen sind. Anders als die binäre Fehlerrate, die jedes Bit einzeln betrachtet, bezieht die Wassersteindistanz auch etwaige horizontale Verschiebung mit ein. Abbildung 10 veranschaulicht die Besonderheit der Wassersteindistanz.

Angenommen wir haben A als n-lange Folge von $[1,1,...,1]$ und B als $[0,0,...,0]$. In diesem Fall ist die Verschiebung vertikal, und sowohl Fehlerrate als auch die Wassersteindistanz sind 1.

Angenommen wir haben A als n-lange Folge von $[1,0,1,0,...1,0]$. Sei nun B die Folge $[0,1,0,1,...0,1]$ sodass $A_i \neq B_i \forall i : 0 \leq i < n$. Die Fehlerrate beträgt folgend 1. Da die Verschiebung jedoch rein horizontal ist, folgt bei Verwendung der Python SciPy Bibliothek:

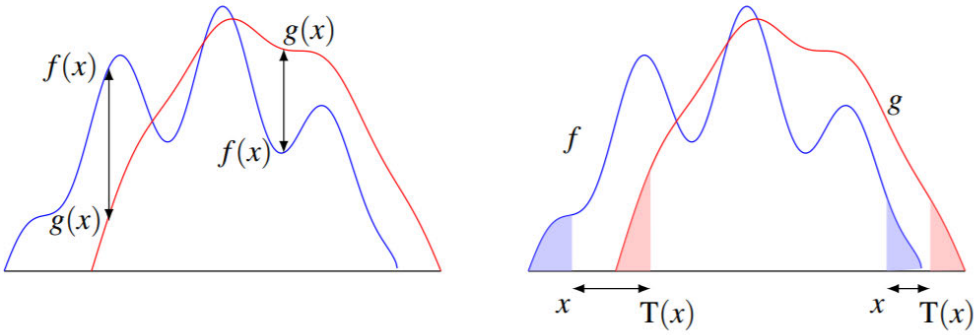

Abbildung 10: *Vertikale und Horizontale Distanz [San15]*

```
from scipy.stats import wasserstein_distance
wasserstein_distance([1,0]*5, [0,1]*5)
0.0
>>>
```

Die Wassersteindistanz bleibt also niedrig, wenn Insertion- oder Deletionfehler (siehe Abschnitt 3.7) auftreten.

4 Implementierung

In diesem Kapitel werden Implementierungsdetails zum Projekt erläutert. So werden Techniken zur präzisen Zeitmessung in amd64 Assembler (siehe Abschnitt 4.1), der Algorithmus zum Klassifizieren eines Messwerts (Abschnitt 4.2), Details zum Herstellen von Kommunikation verschiedener Prozesse über Caches (Abschnitt 4.3) sowie Synchronisierung der Kommunikation auf Bit und Paketebene (Abschnitt 4.4) vorgestellt. Abschließend werden weitere Probleme und die verwendete Lösungsstrategie besprochen (Abschnitte 4.5 und 4.6).

4.1 Präzise Zeitmessung

Die präzise Zeitmessung der Zeiten, welche von `CLFLUSH` (bei Flush+Flush) und Speicherzugriffen (bei Flush+Reload) benötigt werden, ist zwingend notwendig, um die Seitenkanaleffekte des Caches auszunutzen. Insbesondere bei `CLFLUSH` sind die Differenzen gering (siehe Abbildung 7). Um die benötigte Zeit zu bestimmen, wird die aktuelle Systemzeit vor und nach der `CLFLUSH`-Instruktion, beziehungsweise des Speicherzugriffs bei Flush+Reload, gemessen. Als Pseudocode lässt sich dies folgend beschreiben:

```
1 start = meassure_time()
2 CLFLUSH(p)
3 end = meassure_time()
4 delta = end-start
```

Für die Zeitmessung wird die Instruktion `RDTSC` (read time stamp counter) verwendet. Dabei wird der `EDX` auf die höheren 32Bit des Time Stamp Counters (TSC) gesetzt, der `EAX` auf die Niedrigeren. [Int19b, Vol. 2B 4-545]

Zunächst muss sichergestellt werden, dass die Reihenfolge der Instruktionen eingehalten wird. Instruction-Level Parallelism[1] könnte dazu führen, dass `CLFLUSH` vor der ersten, oder nach der zweiten, Zeitmessung ausgeführt wird, was das Bestimmen der korrekten Zeitspanne unmöglich machen würde. Das Intel Software Developer Manual [Int19b, Vol. 3A 8.3] nennt insbesondere folgende unprivilegierte Methoden, um auf die Ordnung von Instruktionen Einfluss zu nehmen:

- `CPUID` (CPU Identification) ist eine Instruktion, mit deren Hilfe Informationen und Features über den Prozessor abgefragt werden können. Für Zweck der Zeitmessung wird `CPUID` auch häufig wegen der Eigenschaft als *serialising instruction* verwendet. Durch den Einsatz von `CPUID` wird garantiert, dass alle vorstehenden Instruktion vor, und alle nachstehenden Instruktionen nach `CPUID` ausgeführt werden. [Int19b, Vol. 3A 8.3]

[1] ILP beschreibt das Umordnen von Instruktionen zur Steigerung des Durchsatzen an Instruktionen pro Zyklus. [Anl19]

- `LFENCE` (Load Fence) ordnet das Ausführen von lesenden Speicherzugriffen. Vorstehende Load-Instruktionen müssen zuerst die Daten erhalten, bevor `LFENCE` abschließt. Nachfolgende Load-Instruktionen können erst ausgeführt werden, wenn `LFENCE` abschließt. [Int19b, Vol. 2A 3-558]

  Zusätzlich schließt `LFENCE` erst ab, wenn alle vorstehenden Instruktionen abgeschlossen sind [Int19b, Vol. 3A 8.3]. Dies ist eine Form von Serialisierung, die von anderen Fence-Instruktionen nicht geteilt wird.
- `SFENCE` (Store Fence) ordnet das Ausführen von schreibenden Speicherzugriffen. `SFENCE` verhindert, dass Store-Instruktionen nach `SFENCE` den Speicher modifizieren, bevor Store-Instruktionen vor der `SFENCE` dies tun. [Int19b, Vol. 2B 4-609]
- `MFENCE` (Memory Fence) ordnet das Ausführen von lesenden und schreibenden Speicherzugriffen. Daten von vorstehenden lesenden Speicherzugriffen müssen lokal erhalten worden sein, und Daten von vorstehenden schreibenden Speicherzugriffen global sichtbar sein, bevor `MFENCE` abschließt und Speicherzugriffe ausführt. [Int19b, Vol. 2B 4-22] [Adv20a, 7.6.3]

Eine Option ist nun das Verwenden von `CPUID` vor und nach `RDTSC`:

```
1  asm volatile ("CPUID");
2  asm volatile ("RDTSC": "=a" (lo), "=d" (hi));
3  start = (hi<<32) | lo;
4  asm volatile ("CPUID");
5  asm volatile ("clflush 0(%0)\n":: "c" (p): "rax");
6  asm volatile ("CPUID");
7  asm volatile ("RDTSC": "=a" (lo), "=d" (hi));
8  end = (hi<<32) | lo;
9  asm volatile ("CPUID");
10 delta = end - start;
```

Aus dem Intel Manual geht jedoch hervor, dass Fence-Instruktionen effizienter als `CPUID` sind [Int19b, Vol. 3A 8-16]. Daher kann die Zeitmessung noch optimiert werden. Alle vorangehenden Instruktionen müssen ausgeführt werden, bevor das erste `RDTSC` ausgeführt wird. Dies kann mit `LFENCE` erreicht werden [Int19b, Vol. 2B 4-545]. Für das zweite `RDTSC` müsste jedoch zunächst ein `MFENCE` verwendet werden — nur `MFENCE` kann nach der `CLFLUSH`-Instruktion verhindern, dass nachfolgende lesende Speicherzugriffe vor `CLFLUSH` geordnet werden können [Adv20b][2]. Dafür kann auf das `LFENCE` verzichtet werden, da nur `CLFLUSH` dem `MFENCE` vorgestellt ist, und `CLFLUSH` in Bezug auf `MFENCE` geordnet ist [Int19b, Vol. 2A 3-145]. Weiterhin muss auf beide `RDTSC` eine `LFENCE`-Instruktion folgen, um zu verhindern, dass nachfolgende Instruktionen vor `RDTSC` ausgeführt werden [Int19b, Vol. 2B 4-545]. Es gilt also die optimierte Zeitmessung:

[2] Dies gilt nur bei Flush+Flush. Bei Flush+Reload wird durch den lesenden Speicherzugriff auch das `LFENCE` geordnet.

```
asm volatile ("LFENCE");
asm volatile ("RDTSC": "=a" (lo), "=d" (hi));
start = (hi<<32) | lo;
asm volatile ("LFENCE");
asm volatile ("CLFLUSH 0(%0)\n":: "c" (p): "rax");
asm volatile ("MFENCE");
asm volatile ("RDTSC": "=a" (lo), "=d" (hi));
end = (hi<<32) | lo;
asm volatile ("LFENCE");
delta = end - start;
```

4.2 Messzeiten von CLFLUSH Klassifizieren

Werden mit den Methoden aus Abschnitt 4.1 die Latenzen für Hits und Misses bei `CLFLUSH` (beziehungsweise Speicherzugriffen bei Flush+Reload) gemessen, muss auf Basis dieser Trainingsdaten eine Klassifikation von ungelabelten Daten durchgeführt werden. Zunächst müssen starke Ausreißer bereinigt werden. Dazu werden Werte außerhalb des 10σ-Streuintervalls eliminiert.

Wir nehmen an, dass Hits und Misses jeweils normalverteilt sind. Das arithmetische Mittel (μ) und die Standardabweichung (σ) der bereinigten Daten können berechnet werden, und stellen die Parameter der Dichtefunktion der Normalverteilung dar:

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad -\infty < x < \infty$$

Abbildung 11 zeigt eine Messung und die resultierenden Normalverteilungen für Hits und Misses auf einem nativen Archlinux mit einem Intel Core i7 9700K bei einer Frequenz von 10KHz.

Die einfachste Klassifizierung besteht im Bestimmen eines Threshholds — so werden alle Werte über `X` als Hit, alle Werte unter `X` als Miss klassifiziert. Im Fall der Normalverteilung muss dazu nun der Schnittpunkt der Dichtefunktionen bestimmt werden, dazu wird folgende Gleichung nach x gelöst:

$$f(x \mid \mu_{miss}, \sigma^2_{miss}) = f(x \mid \mu_{hit}, \sigma^2_{hit})$$

Im oben genannt Beispiel gilt mit den Werten $\mu_{miss} = 150$, $\sigma_{miss} = 2.12$, $\mu_{hit} = 171.3$, $\sigma_{hit} = 2.55$:

$$f(x \mid 150, 2.12^2) = f(x \mid 171.3, 2.55^2) \implies x = 160 \vee 43$$

Auf der Trainingsmenge kann nun bestimmt werden, welcher Schnittpunkt als Threshhold geeignet ist. Im vorstehenden Beispiel ergeben sich jeweils Genauigkeiten von 99.98% für 160 und 50% für 40. 160 ist also der Threshhold für `CLFLUSH` auf dem vorliegenden System bei 10KHz.

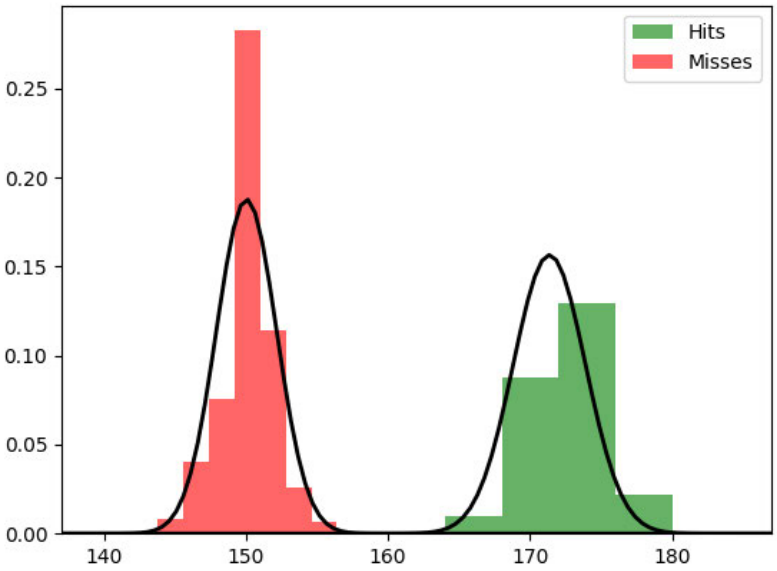

Abbildung 11: *Timingdifferenzen bei Nutzung der* CLFLUSH*-Instruktion bei einer Frequenz von* 10*KHz auf Arch auf einem i*7 9700*K*

4.2.1 CLFLUSH vs CLFLUSHOPT

Neben CLFLUSH (Abschnitt 3.3) existiert noch eine weitere verwandte Instruktion auf der amd64 Architektur. Die Instruktion CLFLUSHOPT unterscheidet sich von CLFLUSH indem sie nur von mfence und anderen Fence-Instruktionen geordnet wird, nicht jedoch durch andere CLFLUSH und CLFLUSH_OPT. [Int19b, Vol. 2A, 3-147]

Für die Kommunikation auf einer einzelnen Cacheline sind beide äquivalent — in Abschnitt 4.1 wurde gezeigt, dass CLFLUSH stets von fence-Instruktionen eingeschlossen ist, es gibt also keine CLFLUSH-Instruktionen des Angreifers, die umsortiert werden können. Sollen mehrere Cachelines gleichzeitig für die Kommunikation verwendet werden, zum Beispiel um die Bandbreite zu erhöhen, dann kann das Verwenden von CLFLUSHOPT die Performanz erhöhen, da die Flushes parallel durchgeführt werden können.

4.3 Verwendung des Caches als Medium

Im Rahmen der Bachelorarbeit sollen mit Flush+Flush und Flush+Reload verdeckte Kanäle gebaut werden. Verdeckte Kanäle sind vor Kontrollmechanismen versteckte, unerlaubte Kommunikationskanäle. Dabei war der verdeckte Kanal von der Domäne nicht als Kommunikationskanal vorgesehen. Dies findet beispielsweise Anwendung in isolierten Cloudumgebungen, oder zur Kommunikation einer Malware mit einem Angreifer. Um einen Kommunikationskanal mit Flush+Flush oder Flush+Reload zwischen Sender und Empfänger aufzubauen, einigen sich die Kommunikationspartner auf eine physische Adresse. Durch das gezielte Cachen oder nicht Cachen dieser physischen Adresse kann der Sender nun Symbole (hier “im Cache” für 1 und “nicht im Cache” für

0) kommunizieren. Der Empfänger startet einen Flush+Flush oder Flush+Reload Angriff auf die vereinbarte physische Adresse. Dieser funktioniert methodisch wie in den Abschnitten 3.4 und 3.6 beschrieben. Anschließend interpretiert der Empfänger die Symbole und erhält Datenpakete. In der vorliegenden Arbeit wird das Ethernet Protokoll verwendet. Anders als bei einem Angriff gibt es keinen Angreifer und Opfer, sondern einen Sender und Empfänger. Im Gegensatz zum Opfer ist der Sender so konstruiert, dass der Empfänger die Daten möglichst einfach, und in hoher Qualität und Bandbreite erhält. Einige Techniken werden in Abschnitt 4.5 besprochen.

4.4 Synchronisierung der kommunizierenden Prozesse

In den vorstehenden Abschnitten wurde erklärt, wie ein Kommunikationsmedium mit zwei Zuständen hergestellt werden kann. Um auf diesem Medium kommunizieren zu können, bedarf es jedoch einem Protokoll, und Synchronisierung. Dabei müssen zunächst einzelne Bits synchronisiert werden, um einen Bitstream herzustellen. Aufbauend wird ein Protokoll verwendet, um die Datenpakete zu synchronisieren. Das Protokoll legt außerdem fest, wie aus den übertragenen Bits ein Datenpaket interpretiert wird, sowie Details zur Übertragung, etwa Checksummen, maximale Paketlängen, oder Ziel-und Absenderadressen. [Mar19]

4.4.1 Synchronisierung von Bits

Bei der Synchronisieren von Bits in klassischer Kommunikation werden insbesondere *self-clocked signals* verwendet. Diese gleichen kleine Abweichungen bei der Frequenz von Sender und Empfänger aus, und stellen sicher, dass die Kommunikation auf Dauer synchronisiert bleibt [Mar19]. Für den verdeckten Kanal über Cacheangriffe kann jedoch eine einfachere Lösung verwendet werden — die Systemzeit. In den durchzuführenden Experimenten existiert mit der Systemzeit stets ein präziser und von beiden geteilter Frequenzgeber. [3]

Clocken des Empfängers mit clock_nanosleep

Für den Sender wird die Standardfunktion `clock_nanosleep` [Fre20a] aus `<time.h>` verwendet. `clock_nanosleep` ermöglicht das Suspendieren des aufrufenden Threads mit Präzision im Nanosekundenbereich. Die Signatur lautet wie folgt:

```
1 int clock_nanosleep(clockid_t clockid,
2                      int flags,
3                      const struct timespec *request,
4                      struct timespec *remain);
```

- In `clockid` wird die zu verwendende Zeitmessung übergeben. Neben der Systemzeit (`CLOCK_REALTIME`) oder der von dem Prozess verwendeten CPU-Zeit (`CLOCK_PROCESS_CPUTIME_ID`) ist auch eine monotone Zeitmessung (`CLOCK_MONOTONIC`) verfügbar. Die monotone Zeit ist für diesen Anwendungsfall vorzuziehen, da die Systemzeit während der Kommunikation geändert werden könnte.

[3] Insbesondere in Cloudszenarien existiert keine geteilte und zuverlässige Systemzeit, hier müssen andere Lösungen angewendet werden.

- `flags` kann entweder `0` oder `TIMER_ABSTIME` sein. Im ersten Fall wird die angegebene Zeit als Dauer für die Suspendierung verwendet, im zweiten Fall wird suspendiert, bis der angegebene Zeitpunkt erreicht ist. Falls bei Verwendung von `TIMER_ABSTIME` der angegebene Zeitpunkt bereits vergangen ist, wird der Thread nicht suspendiert.
- `request` beschreibt Zeit, entweder eine Zeitdauer oder einen Zeitpunkt.
- In `remain` kann die verbleibende Zeit zurückgegeben werden, falls die Suspendierung von einem Signal unterbrochen wird.

Nach dem Interpretieren eines Bits wird vom Empfänger der nachfolgende Programmcode ausgeführt. In jedem Zyklus wird die Periodendauer auf den Zeitpunkt addiert. In Kombination mit dem Suspendieren bis zu einem Zeitpunkt durch `TIMER_ABSTIME`, können im Fall eines Descheduling für eine ganze, oder mehrere Zyklen folgende Durchläufe durchgeführt werden, ohne erneut zu warten. Somit führt Descheduling nur zu Burst-Errors (siehe Abschnitt 3.7), und nicht zur Nachhaltigen Desynchronisation.

```
1 nanosecs = INTERVAL;
2 nanosecs += t0.tv_nsec;
3 if(unlikely(nanosecs > 999999999))  // overflow der Nanosekunden
4     {nanosecs -= 1000000000;
5     t0.tv_sec++;}
6 t0.tv_nsec = nanosecs;
7 clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t0, NULL);
```

Clocken des Senders mit `ualarm`

Der Sender könnte prinzipiell auch identisch zum Empfänger mit `clock_nanosleep` synchronisiert werden. In Abschnitt 4.5 wird jedoch empfohlen, dass der Sender über den ganzen Zyklus wiederholt das aktuelle Bit sendet. Insofern ist die Verwendung von `ualarm` [Fre20c] sinnvoll. Dies ist eine Standardfunktion der `<unistd.h>` und sendet nach der angebenen Zeit ein Signal an den aufrufenden Prozess. `ualarm` hat Präzision im Mikrosekundenbereich. Die Funktion hat zwei Parameter, `usecs` und `interval`. Nach `usecs` Mikrosekunden, und danach alle `interval` Mikrosekunden, sendet `ualarm` ein `SIGALRM`-Signal an den aufrufenden Prozess. Der Sender fängt nun alle `SIGALRM`-Signale ab, und aktualisiert dann das zu sendende Bit.

4.4.2 Synchronisierung von Frames

Das Interpretieren der Bitstreams wird mit dem Ethernet-Protokoll (siehe Abschnitt 3.8) realisiert. Das Ethernet-Protokoll ist das am weitesten verbreitete lokale Data Link Layer Protokoll [Spu00], und cache-basierte verdeckte Kanäle sind zu Ethernetmedien ähnlich.

4.5 Reduzierung von Interferenz durch dritte Prozesse

In Abschnitt 3.4 und 3.6 wurde für einen Flush+Flush- oder Flush+Reload-Angriff erklärt, dass eine niedrige Frequenz zu mehr falschen Negativen (FN) durch Cacheverdrängen von Dritten, sowie potentiell auch falschen Positiven (FP) durch Caching von Dritten führen kann. Im Szenario

des verdeckten Kanals können beide Probleme minimiert werden. Die Cacheverdrängung durch Dritte tritt auf, wenn zwischen dem Speicherzugriff des Senders und dem Auswerten des Caches durch den Empfänger ein dritter Prozess durch einen Speicherzugriff die beobachtete Cacheline verdrängt, resultierend in einem falschen Negativen. Falls der Sender nun über den ganzen Zyklus (siehe Abbildungen 6 und 8) Speicherzugriffe auf die beobachtete Cacheline durchführt, wird dadurch die Wahrscheinlichkeit auf falsche Negative durch Verdrängung durch Dritte reduziert, und bleibt insbesondere unabhängig von der Frequenz. Die falschen Positiven durch Caching von Dritten treten nur dann auf, wenn Dritte dieselbe physische Adresse verwenden, die beobachtet wird. Für die in dieser Arbeit entwickelten verdeckten Kanäle wurde eine eigene geteilte Bibliothek verwendet, wodurch die falschen Positive eliminiert werden. Abhängig von der Anwendung in einem Realszenario ist eine solche Bilbiothek jedoch nicht immer gegeben, und es muss auf eine Standardbibliothek zurückgegriffen werden, welche möglichst wenig benutzt, sowie auf möglichst vielen Systemen vorhanden ist.

4.6 Varianz durch Unterschiede bei der Ausführzeit bei same-core und remote-core

Bei Flush+Flush und Flush+Reload Angriffen wird die benötigte Zeit für `CLFLUSH` und Speicherzugriffe gemessen. Abhängig davon, ob sich Sender und Empfänger auf demselben oder einem anderen Kern befinden, können die Latenzen abweichen [GMWM16]. Um aussagekräftige Daten zu erhalten, sollte Varianz durch das Scheduling vermieden werden. Daher werden für alle Experimente Sender und Empfänger auf demselben Kern ausgeführt. Eine Möglichkeit dies zu erreichen, ist die Linuxfunktion `taskset`. Mit `taskset` wird der Linux Kernel angewiesen, einen Prozess nur auf den angegebenen logischen Kernen auszuführen [Fre20b]. Folgend wird die Benutzung veranschaulicht:

```
1 taskset -c 0-0 ./sender
2 taskset -c 0-0 ./receiver-same-core
3 taskset -c 1-7 ./receiver-remote-core
```

5 Performance-Messung

Im folgenden Kapitel wird die Messung und Evaluation der Kanäle behandelt. Zunächst werden dazu die Testsysteme und die Testumgebung beschrieben (Abschnitt 5.1). Anschließend wird die Durchführung der Tests behandelt, und die resultierenden Daten interpretiert (Abschnitt 5.2). Nachfolgend werden die Kanäle in ihrer Kapazität bewertet und verglichen (Abschnitt 5.2.4). Anschließend die Fragen behandelt, ob Flush+Reload auch Cross-Core auf AMD Prozessoren funktional ist (Abschnitt 5.3). Zu diesem Zweck werden zusätzliche Tests durchgeführt.

5.1 Systeme und Umfeld

Das Ziel der vorliegenden Arbeit ist die Evaluation von mit Flush+Flush und Flush+Reload gebauten verdeckten Kanälen in Alltagsszenarien. Um die Daten konsistent und aussagekräftig zu halten wird folgendes Szenario für alle Messungen angewendet: Laufende Internettelefonie über das Programm Discord, sowie ein offener Chrome-Tab in dem die GitHub-Seite der vorliegenden Arbeit [Hei20] geladen ist.

Als Linux Distribution wird `Arch` genutzt. Die Kernelversion ist `5.4.44-1`. Folgende Software ist installiert:

- Python3 3.8.3-1
- Python-Pip 20.0.2-1
- Numpy 1.18.5
- SciPy 1.4.1
- Maplotlib 3.2.1
- Sklearn 0.0
- Gnu Compiler Collection (GCC) 10.1.0-1
- CMAKE 3.17.3-1
- make 4.3-3

5.1.1 System 1: Intel i7 9700K

Das erste Testsystem basiert auf einem Intel Core i7 9700K der Coffee Lake Refresh Architektur mit 8 Kernen. Die L1 Instruktions- und Datencaches sind jeweils 8-Wege Set-Associative der Größe von 32KiB pro Kern. Die L2 Caches sind 256KiB große 4-Wege Set Associative Caches. Der von allen Kernen geteilte L3 Cache hat eine Kapazität von 12MiB und ist als 12-Wege Set Associative

realisiert, und unterstützt Intel Smart Cache (siehe Abschnitt 3.2.5). Wie bei modernen Prozessoren üblich beträgt die Größe einer Cacheline 64 Bytes. Der 9700K unterstützt kein Multi-Threading. Der Prozessor ist auf allen Kernen auf eine Taktfrequenz von 5.0 GHz übertaktet. Auf dem System sind 32GiB DDR4 Hauptspeicher verfügbar. Das Betriebssystem wurde auf einer 250GB großen SSD installiert. [Int19c]

5.1.2 System 2: AMD Ryzen 2600

Das zweite Testsystem basiert auf einem AMD Ryzen 5 2600 der Zen+ Architektur mit 6 Kernen, die in zwei Komplexen mit jeweils drei Kernen strukturiert sind. Die L1 Instruktions- und Datencaches sind jeweils 8-Wege Set-Associative, die Datencaches habe eine Größe von 32KiB pro Kern, die Instruktionscaches 64KiB pro Kern. Die L2 Caches sind 512KiB große 8-Wege Set Associative Caches. Der $L3$ Cache ist durch die zwei Komplexe geteilt. Jeder Komplex hat eine Kapazität von 12MiB und ist als 12-Wege Set Associative realisiert. Kerne beider Komplexe können den $L3$ des anderen Komplexes durch einen Ringbus verwenden. Wie bei modernen Prozessoren üblich beträgt die Größe einer Cacheline 64 Bytes. Der 2600 unterstützt Multi-Threading, und verfügt somit über 12 Threads. Der Prozessor läuft auf allen Kernen mit einer Taktfrequenz von 3.6 GHz. Auf dem System sind 16GiB DDR4 Hauptspeicher verfügbar. Das Betriebssystem wurde auf einer 250GB großen SSD installiert.

5.2 Durchführung und Interpretation

Auf den zwei vorliegenden Systemen wird im definierten Testumfeld Flush+Flush und Flush+Reload getestet, und miteinander verglichen. Hierbei werden 3 Frequenzen, $1kHz$, $10kHz$, $100kHz$ verwendet.

5.2.1 Threshholds bestimmen

Zunächst müssen die Threshholds für die Klassifikation von Messdaten bestimmt werden. Dazu werden vom Sender alternierend '1' und '0' über den Cache gesendet. Der Empfänger misst die benötigte Zeit für `CLFLUSH` (beziehungsweise Speicherzugriff) für 10000 Bits. Anschließend wird der Threshhold, wie in Abschnitt 4.2 beschrieben, bestimmt. Abbildung 12 zeigt die Gaußkurven von für Flush+Reload auf dem Intel System. Dabei sind die Threshholds für alle getestete Frequenzen praktisch identisch.

Abbildung 13 zeigt die Kurven für Flush+Flush, ebenfalls auf dem Intel System. Für die Frequenzen von $1kHz$ und $10kHz$ sind die Kurven erneut praktisch kongruent, die bei $100kHz$ scheint jedoch keine klare Seperation der beiden Cluster möglich zu sein. Trotzdem berechnet sich der Threshhold für alle Frequenzen auf 141.3 ± 0.4.

Auf dem AMD System sind für Flush+Reload, wie in Abbildung 14 erkennbar, ähnliche Muster zu erkennen. Für $100kHz$ scheint es auch hier eine Vermischung der Gruppen zu geben, die niedrigeren Frequenzen sind klar separiert. In Abbildung 15 wird Flush+Flush auf AMD dargestellt, hier sind die Timingdifferenzen, die bei Flush+Flush ohnehin gering ausfallen, noch schlechter zu trennen, als auf dem Intel System.

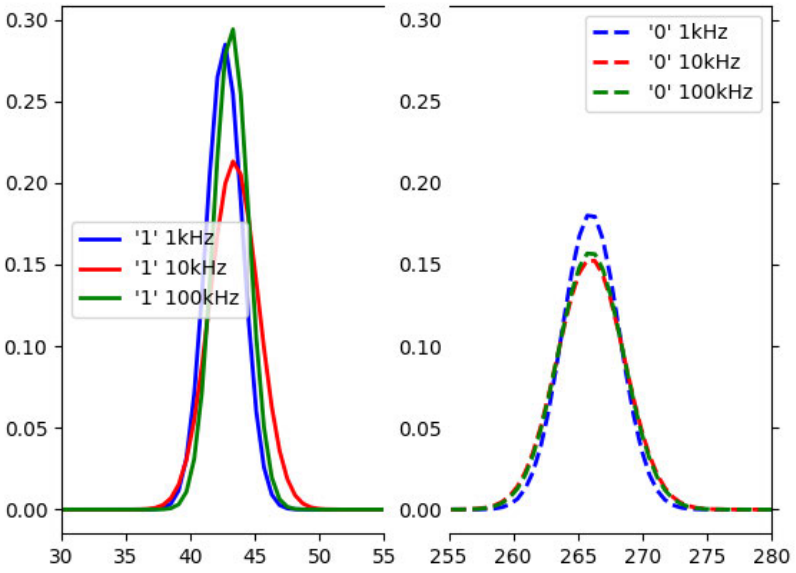

Abbildung 12: *Timingdifferenzen bei Flush+Reload auf Intel*

5.2.2 Datenübertragung

Mit den bestimmten Threshholds können nun Datenpakete übertragen werden. Hier werden Ethernet Frames (siehe Abschnitt 3.8) verwendet, um die Synchronisation auf dem *data link layer* zu realisieren. Dabei werden die Kontrollinformationen ignoriert, ebenso die Checksumme. Für die Datenauswertung wird lediglich der Payload mit dem Soll verglichen. Dies ist ausdrücklich keine Evaluation von Ethernet als Protokoll für die Kommunikation mit Flush+Flush und Flush+Reload. Bei der Messung werden 10 Frames mit jeweils 1000 Bits Payload übertragen. In Abbildung 16 sind die Genauigkeiten[1] der Übertragungen dargestellt. Zunächst ist erkennbar, dass die Genauigkeit bei zunehmender Frequenz monoton zu fallen scheint. Insbesondere bei Intel ist die Genauigkeit bei niedrigeren Frequenzen sehr hoch und fällt bei Frequenzen von $100kHz$ stärker ab, als es auf AMD der Fall ist. Flush+Reload auf AMD profitiert kaum von einer Frequenz von weniger als $10kHz$. Flush+Flush ist auf AMD sehr ungenau.

5.2.3 Interpretation der Fehlertypen

In Abschnitt 4.5 wurden Designentscheidungen erläutert, die bestimmte Fehlerquellen minimieren sollten. So soll das wiederholte Cachen durch den Sender die erhöhte Anzahl von falschen Negativen bei niedrigen Frequenzen eliminieren. Das Verwenden einer eigenen geteilten Bibliothek soll analog die erhöhte Anzahl von falschen Positiven verhindern. In Abbildung 17 werden exemplarisch die Fehlerklassen bei Flush+Flush Übertragungen auf Intel abhängig von der Frequenz aufgetragen. Dabei sind keine negativen Effekte durch niedrigere Frequenzen innerhalb des Intervalls von $1kHz$ bis $100kHz$ beobachtbar. Für Flush+Reload Übertragungen, sowie Messungen auf dem AMD

[1] Wie in Abschnitt 3.9.2 definiert, beträgt die Genauigkeit $\frac{\text{richtig klassifizierte Bits}}{\text{Anzahl der Bits}}$.

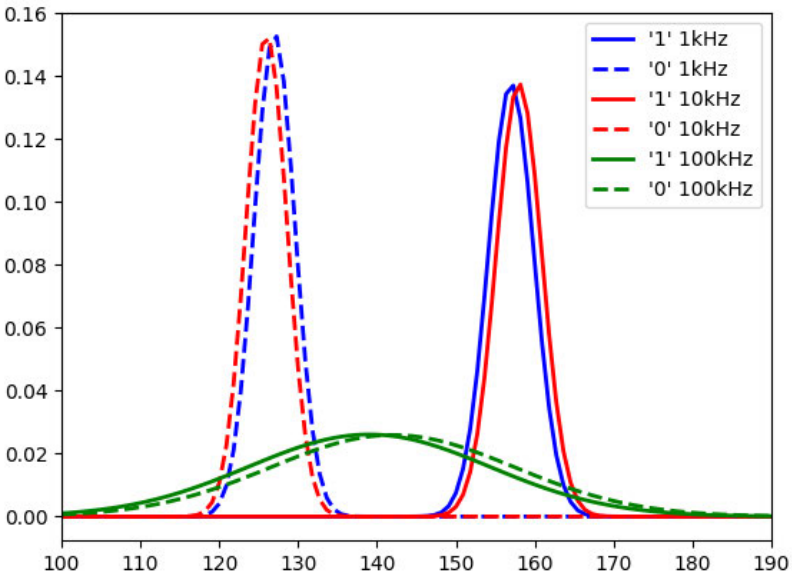

Abbildung 13: *Timingdifferenzen bei Flush+Flush auf Intel*

System stellen sich identische Ergebnisse ein. Zumindest bei den gewählten Frequenzen sind die oben genannten Fehlerquellen also entweder ohnehin nicht relevant, oder das Design des Kanals ist bei der Reduzierung dieser Fehler erfolgreich.

Weiterhin soll gezeigt werden, dass die Fehler voneinander unabhängig sind, und keine systematischen Burst-, Deletion- oder Insertionerrors auftreten (siehe Abschnitt 3.7). Exemplarisch wird hier erneut Flush+Flush bei einer Frequenz von $100kHz$ auf Intel beschrieben. Abbildung 18 stellt die Messdaten für '1' und '0' Symbole bei einer Frequenz von $100kHz$ dar. Zwar sind die gleichen Häufungen wie bei Abbildung 13 erkennbar, durch die hohe Frequenz scheint die Zuordnung jedoch stark fehlerbehaftet zu sein. Eine mögliche Erklärung ist das Descheduling von Sender oder Empfänger, wodurch Insertion- und Deletionerrors respektive auftreten können [MWS+17]. In Abschnitt 4.4 wurde jedoch ein Design vorgestellt, dass Insertion- und Deletionerrors zu verhindert sucht. Um zu überprüfen, inwiefern Insertion- und Deletionerrors auftreten, wird die Wassersteindistanz (siehe Abschnitt 3.9.3) berechnet. Für Flush+Flush bei $100kHz$ auf Intel beträgt die Wassersteindistanz 0.0054 bei einer Genauigkeit von 0.624. Dies ist vergleichbar mit dem erwarteten Wert von 0.0048 bei einer unabhängigen Fehlerrate von $1 - 0.624$ für jedes Bit. Systematische Insertion- und Deletionerrors scheinen durch das Design des Kanals eliminiert zu werden. In Abbildung 19 ist die Anzahl der Fehler in einer Anhäufung dargestellt. Hier wird verdeutlicht, dass Bursterrors keine systematische Ursache zu haben scheinen, und die Fehler tatsächlich voneinander unabhängig sind. Die hier am Beispiel des Flush+Flush Kanals mit $100kHz$ auf Intel erwähnten Eigenschaften gelten ebenso für die anderen evaluierten Kanäle [Hei20].

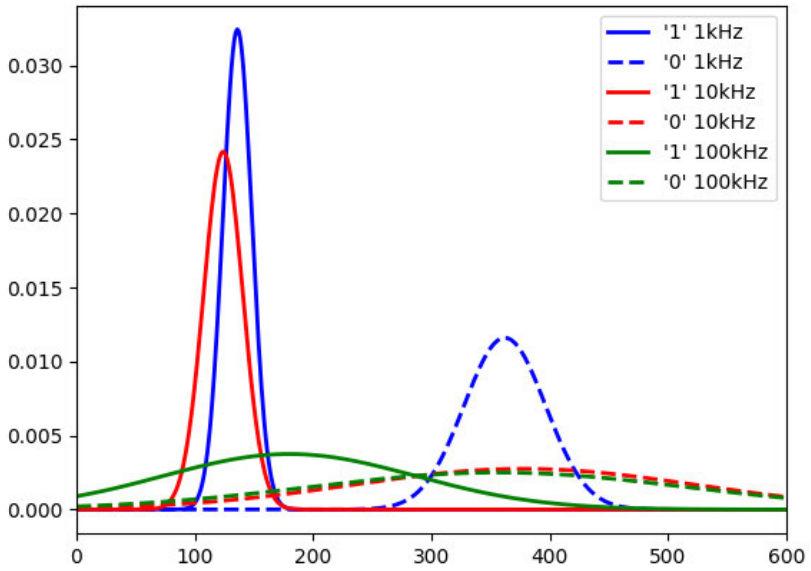

Abbildung 14: *Timingdifferenzen bei Flush+Reload auf AMD*

5.2.4 Kapazität der Kanäle und Evaluation

Ein Kommunikationskanal ist insbesondere durch die Kapazität zu bewerten. Da die Fehlerraten bei verdeckten Kanälen mit Flush+Flush und Flush+Reload signifikant sind, werden fehlerkorrigierende Codes verwendet, um die effektive Fehlerrate zu reduzieren [GMWM16]. Da es nicht Ziel dieser Arbeit ist, für die unterschiedlichen Kanäle einen konkreten Code zu entwickeln, muss die theoretische Kapazität berechnet werden, um zwischen Bandbreite und Fehlerrate abzuwägen. Da wir im vorstehenden Abschnitt 5.2.3 gezeigt haben, dass die Fehler voneinander unabhängig auftreten, kann Shannons Theorem (siehe Abschnitt 3.9.1) angewendet werden. Abbildung 20 zeigt die Kapazität nach Shannon auf.

Auf Intel werden mit Flush+Flush und Flush+Reload ähnliche Ergebnisse erzielt — bis zu $7.88 kBit/s$ mit Flush+Reload bei $100 kHz$, aber lediglich marginal weniger Kapazität mit Flush+Flush ($7.52 kBit/s$ bei $10 kHz$). Im Bereich von $10 kHz$ bis $100 kHz$ erzielen dabei sowohl Flush+Flush, als auch Flush+Reload, hohe Kapazitäten von $5.47 kBit/s$ (Flush+Reload bei $10 kHz$), beziehungsweise $4.48 kBit/s$ (Flush+Flush bei $100 kHz$). Für Flush+Flush scheint um $10 kHz$ das Maximum der Kapazität erreicht zu werden, für Flush+Reload ist nicht geklärt, ob Kanäle von Frequenzen von über $100 kHz$ profitieren würden.

Auf AMD wird mit Flush+Reload die höchste gemessene Kapazität dieser Arbeit erzielt, bei $100 kHz$ erreicht der Kanal eine Kapazität von $22.29 kBit/s$. Der niedrige Abfall der Genauigkeit bei $100 kHz$ gegenüber der Kanäle auf Intel führt hier zu einer besonders hohen Kapazität. Bei $1 kHz$ bis $10 kHz$ ist die Kapazität ähnlich zur auf Flush+Flush und Flush+Reload Kanälen auf Intel Erzielten. Mit Flush+Flush auf AMD können lediglich bis zu $2.31 kBit/s$ erzielt werden (bei $100 kHz$). Auch bei AMD bleibt unklar, ob Frequenzen jenseits der $100 kHz$ die Kapazität weiter erhöhen könnten.

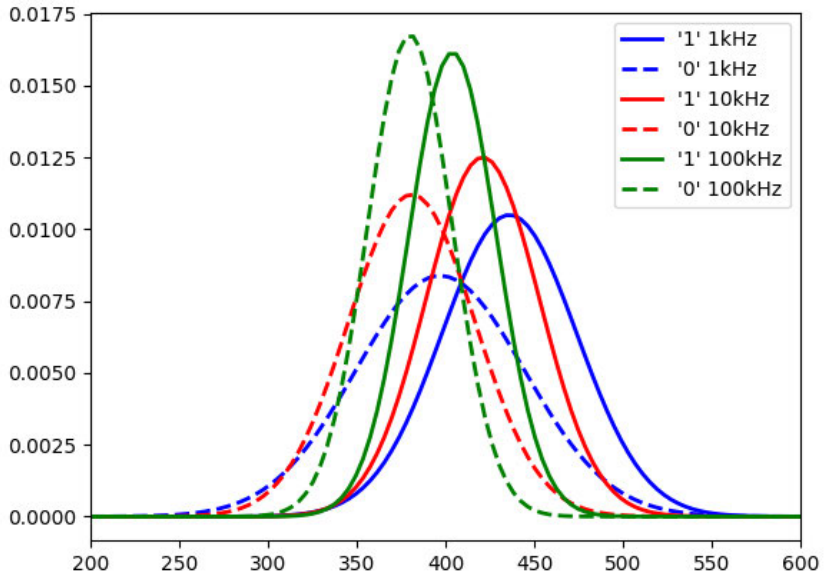

Abbildung 15: *Timingdifferenzen bei Flush+Flush auf AMD*

5.3 Flush+Reload auf AMD: Remotecore und -CCX

Flush+Reload Angriffe benötigen einen geteilten, inklusiven Cache [YF14]. Des Weiteren muss es eine Timingdifferenz bei Cache Hits und Cache Misses geben. In Abschnitt 3.2.6 wurde erwähnt, dass die genaue Funktionsweise der AMD Zen, Zen+ und Zen2 nicht bekannt ist. Folgend ist auch die Machbarkeit von Flush+Reload auf Remote-Cores nicht bekannt. Bei Remote-Core wird weiterhin zwischen Same- und Remote-CCX unterschieden. Ein CCX, oder **C**PU **C**omple**X**, ist ein Verbund mehrerer Kerne, die sich bestimmte Ressourcen teilen, etwa den L3 Cache. Bei dem vorliegenden 6-Kern AMD Prozessor Ryzen 2600 bilden jeweils drei Kerne einen CCX. Um zu überprüfen, ob Flush+Reload auch Cross-Core und Cross-CCX funktioniert, wurde die Threshholdmessung aus Abschnitt 5.2 für verschiedene Prozessorkerne wiederholt. Dabei werden auf dem 6-Kerner die Kombinationen aus dem ersten Kern für den Sender, und dem zweiten und vierten Kern für den Empfänger[2] getestet. Abbildung 21 zeigt die benötigten Zyklen für den Speicherzugriff bei Same-CCX und Cross-CCX. Für Same-CCX ist die benötigte Timingdifferenz zwischen ’1’ und ’0’ vorhanden. Bei Cross-CCX fehlt diese Differenz anscheinend. Werden die Mediane mit den Werten der Same-Core Messung aus Abbildung 14 verglichen, stimmen diese mit den Same-Core Cache Misses überein. Die Same-CCX Messungen resultieren in derselben benötigten Zeit für übertragene ’1’ Symbole. Die benötigte Zeit für Speicherzugriffe auf vom Sender nicht geladene Speicherblöcke (’0’ Symbole) ist überraschenderweise länger als auf Cross-CCX. Eine Erklärung für die Messdaten bleibt offen, es folgt jedoch, dass Flush+Flush lediglich auf einem CCX, dafür auf allen Kernen des CCX, funktioniert.

[2] Da der Ryzen 2600 Multithreading unterstützt, muss bei Verwendung von `taskset` beachtet werden, dass der zweite phsische Kern die logischen Kerne 3 und 4 umfasst.

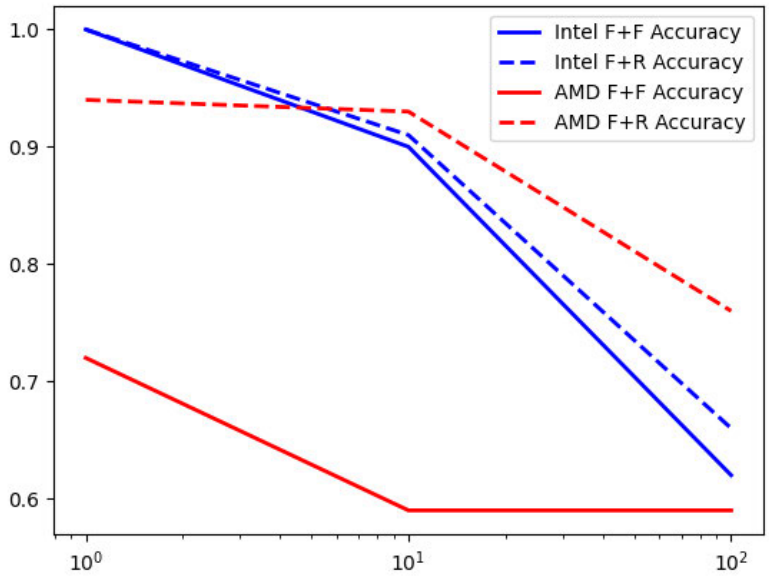

Abbildung 16: *Genauigkeit abhängig von der Frequenz bei F+F und F+R auf Intel und AMD*

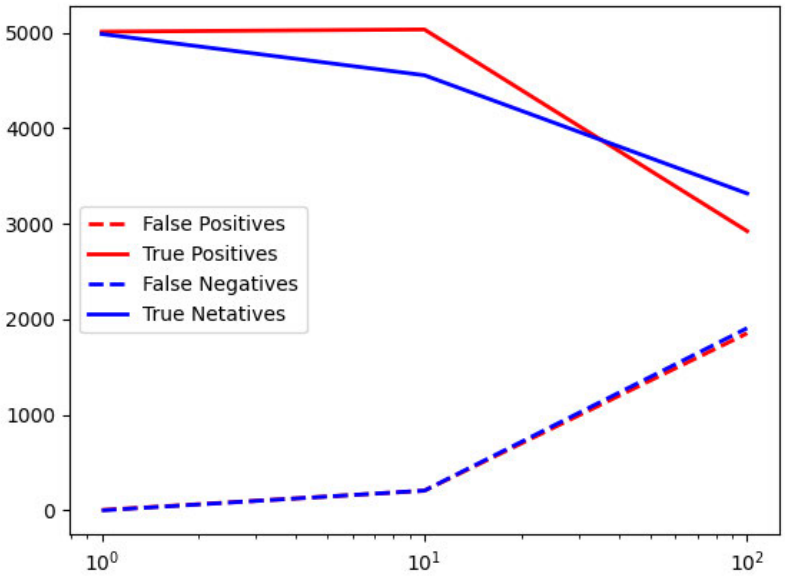

Abbildung 17: *Verschiedene Fehlerklassen abhängig von der Frequenz bei Flush+Flush Übertragungen auf Intel*

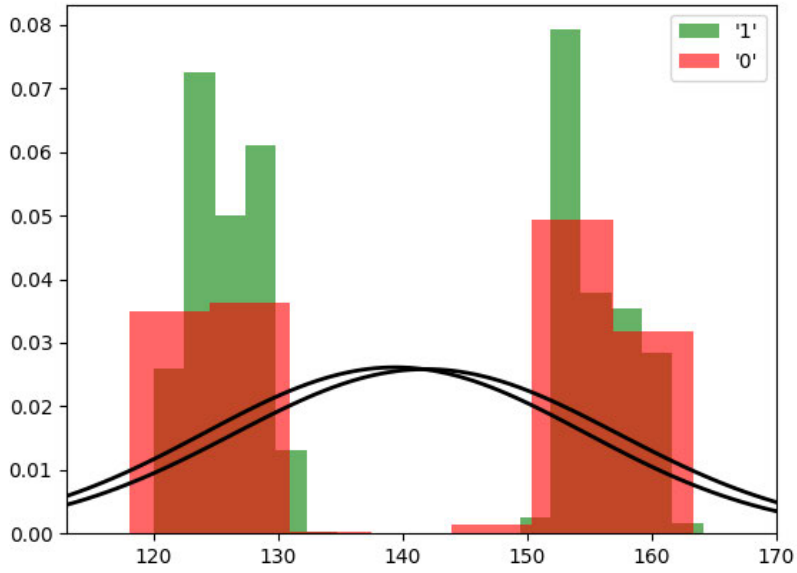

Abbildung 18: *Timingdifferenzen bei Flush+Flush bei 100kHz auf Intel*

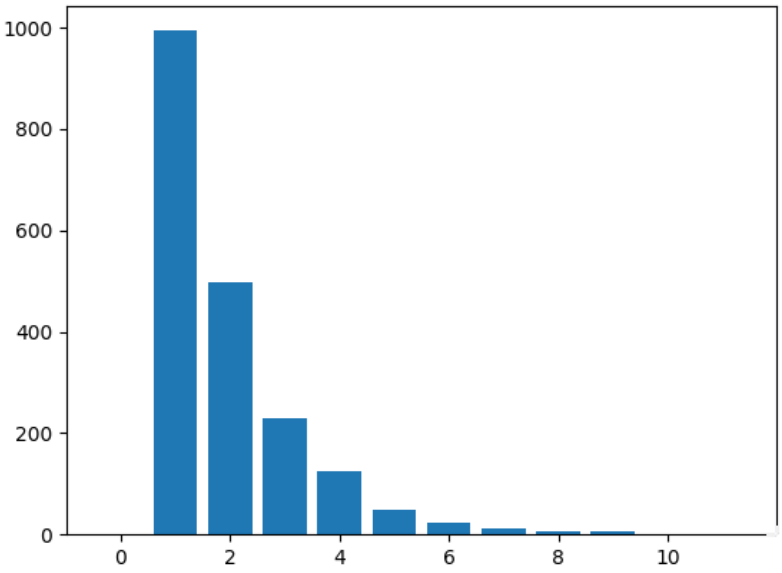

Abbildung 19: *Anzahl von Fehler in einer Anhäufung*

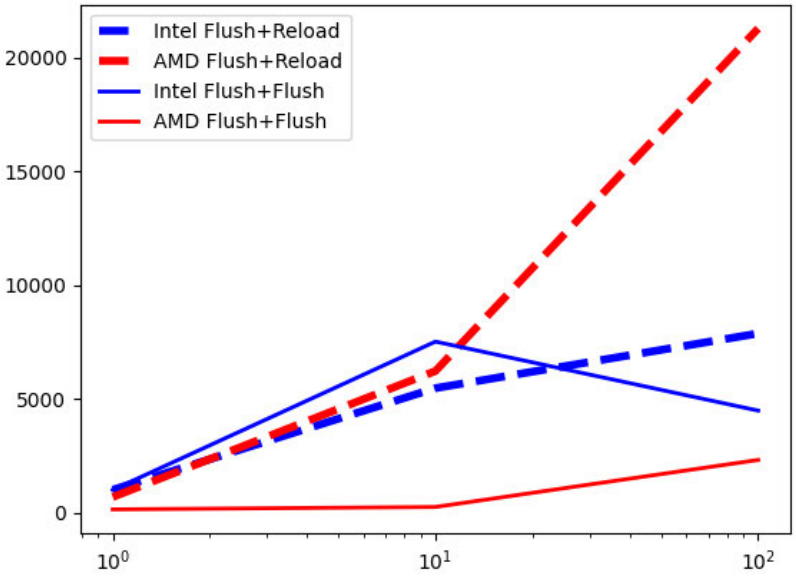

Abbildung 20: *Kapazität der Kanäle nach Shannon, abhängig von der Frequenz*

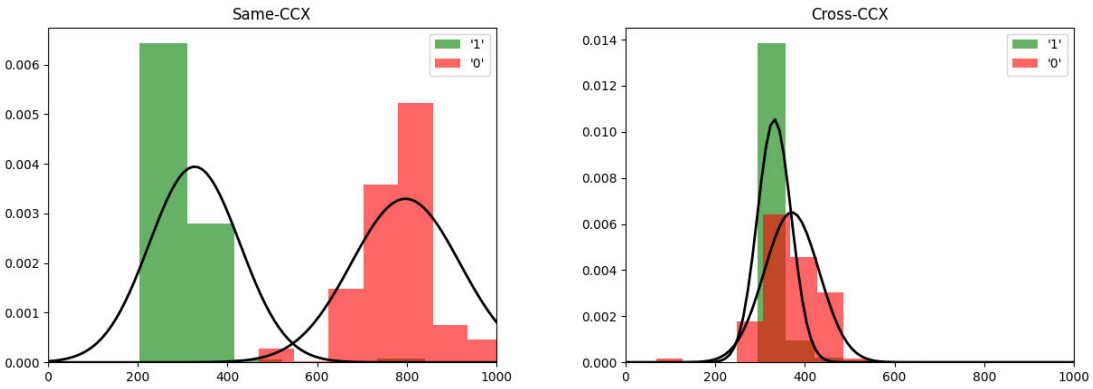

Abbildung 21: *Timings bei Flush+Reload bei Same-CCX und Cross-CCX*

6 Fazit und Ausblick

Ziel dieser Arbeit war die Implementierung und Evaluation von verdeckten Kanälen, die Seitenkanaleffekte ausnutzen. Dabei wurden typische Fehlerquellen durch Descheduling und Interferenz erfolgreich minimiert. Es wurde gezeigt, dass Flush+Flush und Flush+Reload in Alltagsszenarien sowohl auf Intel als auch auf AMD durchführbar sind. Weiterhin sollte beantwortet werden, ob Flush+Flush eine sinnvolle Fortentwicklung von Flush+Reload darstellt.

In den Tests hat sich gezeigt, dass Flush+Flush den Flush+Reload Angriff nicht ersetzt, jedoch in einigen Anwendungen ergänzt. Durch die höhere Timingdifferenz von Cache Hits und Misses erreicht Flush+Reload höhere Genauigkeiten und damit Kapazitäten von bis zu $21 kBit/s$. Die Timingdifferenzen von `CLFLUSH` sind zwar im Allgemeinen niedriger, dafür ist Flush+Flush als `stealthy` Angriff konzipiert, das heißt der Empfänger bleibt vor Detektionsmaßnahmen versteckt, nicht jedoch der Sender. Weiterhin wurde gezeigt, dass Flush+Reload auf der AMD Zen Architektur Cross-Core funktioniert, wenn Sender und Empfänger auf demselben CPU-Complex ausgeführt werden. Cross-CCX stellt sich der Flush+Reload Angriff in den Messungen dieser Arbeit als nicht durchführbar heraus. Dies macht den Flush+Flush Angriff zur vielseitigsten Alternative, wenn die Exekution auf demselben CCX nicht gewährleistet werden kann. In allen getesteten Szenarien erreicht der Flush+Flush Angriff bei $100 kHz$ mindestens eine Kapazität von $2.31 kBit/s$.

In dieser Arbeit wurde der Fokus weniger auf die Optimierung der Kapazitäten gelegt, als auf die Evaluation von verschiedenen Systemen. So ist das Bestimmen der optimalen Frequenz eine potentielle Möglichkeit, die Kapazität zu erhöhen, und bietet sich deshalb für eine weiterführende Forschung an. State of the Art Implementierungen [GMWM16][MWS+17][SP20] verwenden weiterhin mehrere Cachelines gleichzeitig — dies resultiert in mehreren Bits pro Wort, und folglich in höheren Kapazitäten. Die Frage, ob die Verwendung der `CLFLUSHOPT` Instruktion für solche Kanäle einen Vorteil bieten würde, wurde bisher nicht erforscht. Gleiches gilt für die Verwendung von `CLFLUSHOPT` bei Fullduplex Kanälen, hier sind beide Kommunikationspartner Sender und Empfänger und verwenden ebenfalls mehrere, mindestens zwei, Cachelines.

Literaturverzeichnis

[Adv20a] Advanced Micro Devices: AMD64 Architecture Programmer's Manual Volume 2: System Programming. 2020. – Online erhältlich unter https://www.amd.com/system/files/TechDocs/24593.pdf; abgerufen am 28. Juni 2020.

[Adv20b] Advanced Micro Devices: AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. 2020. – Online erhältlich unter https://www.amd.com/system/files/TechDocs/24594.pdf; abgerufen am 28. Juni 2020.

[AMD19] AMD: AMD NEXT HORIZON: GAMING TECH DAY. 2019. – Online erhältlich unter https://www.computerbase.de/2019-06/amd-zen-2-ryzen-3000-architektur/; abgerufen am 28. Juni 2020.

[Anl19] Anlauf, Prof. Dr. Joachim K.: Rechnerorganisation. 2019. – Online erhältlich unter https://www.ti.uni-bonn.de/teaching/ss19/rechnerorganisation; abgerufen am 28. Juni 2020.

[Bot04] Bottomley, James: Understanding Caches. (2004). – Online erhältlich unter https://www.linuxjournal.com/article/7105; abgerufen am 28. Juni 2020.

[Den96] Denning, Peter J.: BEFORE MEMORY WAS VIRTUAL. (1996). – Online erhältlich unter https://web.archive.org/web/20120224183050/http://cs.gmu.edu/cne/pjd/PUBS/bvm.pdf; abgerufen am 28. Juni 2020.

[Fre20a] Free Software Foundation: clock_nanosleep manual page. Website, 2020. – Online erhältlich unter https://man7.org/linux/man-pages/man2/clock_nanosleep.2.html; abgerufen am 28. Juni 2020.

[Fre20b] Free Software Foundation: taskset manual page. Website, 2020. – Online erhältlich unter https://linux.die.net/man/1/taskset; abgerufen am 28. Juni 2020.

[Fre20c] Free Software Foundation: ualarm manual page. Website, 2020. – Online erhältlich unter https://www.man7.org/linux/man-pages/man3/ualarm.3.html; abgerufen am 28. Juni 2020.

[GMWM16] Gruss, Daniel ; Maurice, Clémentine ; Wagner, Klaus ; Mangard, Stefan: Flush+Flush: a fast and stealthy cache attack. In: International Conference on Detection of

Intrusions and Malware, and Vulnerability Assessment Springer, 2016, S. 279–299

[Gru18] Gruss, Daniel: Software-based microarchitectural attacks. In: IT-Information technology 60 (2018), Nr. 5-6, S. 335–341

[Hei20] Hein, Lennart: F+F Covert Channels - Evaluation von cache-basierten FLUSH+FLUSH-Seitenkanälen in Alltagsszenarien. https://github.com/lennihein/BA/tree/54173f8d084cbcc255bea4b1d6f174bbb7a9b2d0, 2020

[HL18] H.J. Lu, Milind Girkar Jan Hubicka Andreas Jaeger Mark M. Michael Matz M. Michael Matz: System V Application Binary Interface - AMD64 Architecture Processor Supplement. 2018. – Online erhältlich unter https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf; abgerufen am 28. Juni 2020.

[Int19a] Intel Corporation: 8th and 9th Generation Intel® Core™ Processor Families Datasheet, Volume 1 of 2. 2019. – Online erhältlich unter https://www.intel.com/content/www/us/en/products/docs/processors/core/8th-gen-core-family-datasheet-vol-1.html

[Int19b] Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual. 2019. – Online erhältlich unter https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf; abgerufen am 28. Juni 2020.

[Int19c] Intel Corporation: Intel® Core™ i7-9700K Prozessor Spezifikation. 2019. – Online erhältlich unter https://ark.intel.com/content/www/de/de/ark/products/186604/intel-core-i7-9700k-processor-12m-cache-up-to-4-90-ghz.html; abgerufen am 28. Juni 2020.

[Kle08] Kleitman, Prof. D.: Principles of Discrete Applied Mathematics. 2008. – Online erhältlich unter http://www-math.mit.edu/~djk/18.310/18.310F04/shannon_bound.html; abgerufen am 28. Juni 2020.

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

[LGS+16] Lipp, Moritz ; Gruss, Daniel ; Spreitzer, Raphael ; Maurice, Clémentine ; Mangard, Stefan: Armageddon: Cache attacks on mobile devices. In: 25th {USENIX} Security Symposium ({USENIX} Security 16), 2016, S. 549–564

[Mar19] Martini, Prof. Dr. P.: Kommunikation in Verteilten Systemen. 2019. – Online erhältlich unter https://net.cs.uni-bonn.de/wg/cs/teaching/wt-201920/kommunikation-in-verteilten-systemen/; abgerufen am 28. Juni 2020.

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

[OST06] Osvik, Dag A. ; Shamir, Adi ; Tromer, Eran: Cache attacks and countermeasures: the case of AES. In: Cryptographers' track at the RSA conference Springer, 2006, S. 1–20

[San15] Santambrogio, Filippo: Optimal transport for applied mathematicians. In: Birkäuser, NY 55 (2015), Nr. 58-63, S. 94

[SP20] Saxena, Anish ; Panda, Biswabandan: DABANGG: Time for Fearless Flush based Cache Attacks. (2020)

[Spu00] Spurgeon, Charles E.: Ethernet: the definitive guide. Ö'Reilly Media, Inc.", 2000

[YF14] Yarom, Yuval ; Falkner, Katrina: FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: 23rd {USENIX} Security Symposium ({USENIX} Security 14), 2014, S. 719–732

[YSG+19] Yan, Mengjia ; Sprabery, Read ; Gopireddy, Bhargava ; Fletcher, Christopher ; Campbell, Roy ; Torrellas, Josep: Attack directories, not caches: Side channel attacks in a non-inclusive world. In: 2019 IEEE Symposium on Security and Privacy (SP) IEEE, 2019, S. 888–904

Abbildungsverzeichnis