Verdeckte Kommunikation durch Seitenkanaleffekte auf ARM-Prozessoren

Bachelorarbeit

ausgearbeitet von

Maurice Happe

2961658

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 2021

Danksagung

Ich möchte mich besonders bei meinem Erstprüfer Felix Boes bedanken, der die Rolle eines Betreuers sehr gut übernommen hat. Zudem möchte ich mich bei Matthew Smith bedanken, der meine Arbeit als Zweitprüfer korrigiert. Des Weiteren geht Dank an meine Korrekturleser Lennart Hein, Fiona Happe und Kevin Neuhöfer raus. Zusätzlich möchte ich mich bei meiner Familie bedanken, die mich emotional und finanziell bei der Bachelorarbeit unterstützt hat.

Kurzfassung

In den letzten Jahren stieg, aufgrund der zunehmenden Wichtigkeit von Smartphones und Tablets, die Relevanz von ARM Prozessoren stark an. Dies erhöht somit auch die Notwendigkeit von Sicherheitsmechanismen auf ARM Prozessoren. Eine bekannte Klasse von Sicherheitslücken, die seit 2006 auf Intel x86 Prozessoren praktiziert wird, sind cachebasierte Seitenkanalangriffe. Im Jahr 2016 wurden die ersten Arbeiten zu cachebasierten Seitenkanalangriffen auf ARM Prozessoren veröffentlicht.

Im Rahmen dieser Bachelorarbeit werden die Seitenkanalangriffe `Flush+Reload` und `Flush+Flush` auf den ARMv8 Prozessoren Cortex-A35 und Cortex-A72 ausgeführt. Mit Hilfe der benutzten Seitenkanaleffekten lassen sich verdeckte Kanäle bauen, die es zwei Prozessen ermöglicht unentdeckt miteinander zu kommunizieren. Ein weiteres Ziel ist die Auswertung der erstellten verdeckten Kanäle. Dabei wird die Implementierung der Methoden sowie mögliche Hürden vorgestellt. Es stellt sich heraus, dass ein Kanal mit `Flush+Reload` eine höhere Datenrate als ein Kanal mit `Flush+Flush` erreicht. Der Vorteil von `Flush+Flush` hingegen ist, dass es durch den Einsatz von Hardwareperformanzzählern nicht entdeckt werden kann. Zusätzlich kann in einem Cross-Core Szenario eine zuverlässigere Synchronisierungsmethode verwendet werden als bei Same-Core. Dadurch erreicht Cross-Core für den Großteil der Messungen bessere Kanalkapazitäten als Same-Core.

Inhaltsverzeichnis

1 Einleitung

Seit 2006 wurden cachebasierte Seitenkanalangriffe auf Intel x86 CPUs benutzt. Ein Seitenkanal ist eine Informationsquelle, aus der mit einem Seitenkanalangriff Informationen über ein System gewonnen werden können, die nicht aus dieser Quelle zu entnehmen sein sollten. Seitenkanäle sind zum Beispiel die Laufzeit eines Vorganges oder der Energieverbrauch eines Gerätes. Aus diesen Daten können dann Informationen über bestimmte Vorgänge im System abgleitet werden. Weiterhin lassen sich mit Seitenkanaleffekten verdeckte Kanäle bauen; Mit verdeckten Kanälen können zwei Prozesse unentdeckt miteinander kommunizieren.

Beim Beispiel eines Tresors ist der Seitenkanal das Geräusch der Drehscheibe, wenn es bei der korrekten Zahl einrastet. Dieses Geräusch sollte nicht bekannt sein und bietet somit einen Seitenkanal. Cachebasierte Seitenkanalangriffe nutzen die Implementationsdetails des Caches aus, indem sie zeitliche Unterschiede zwischen Cache Hits und Cache Misses erkennen können. Um aus diesem Seitenkanaleffekt einen verdeckten Kanal zu bauen, lassen sich zum Beispiel Hits und Misses auf bestimmte Cache-Zeilen als 1 oder 0 interpretieren. Somit entsteht ein Datenfluss und dadurch eine Kommunikation.

Erst seit 2016 wurden die ersten Arbeiten zu cachebasierten Seitenkanalangriffe auf ARM Prozessoren veröffentlicht. Trotz der unterschiedlichen Implementationsdetails des Caches von ARM Prozessoren und Intel x86 CPUs, ähneln sich die Angriffe. Das Hauptziel der Bachelorarbeit ist es auf ARM Prozessoren mit Seitenkanaleffekten verdeckte Kanäle zu bauen und die erstellten Kanäle dann zu bewerten. Dazu gehören Kriterien wie die theoretische Kanalkapazität und Fehlerrate des Kanals, unter welchen Bedingungen der Angriff ausführbar ist und wie gut der Kanal von Kontrollmechanismen des Systems versteckt ist. Als Testumgebung werden die ARMv8 Prozessoren Cortex-A35 und Cortex-A72 verwendet.

2 Benötigtes Vorwissen

Im folgenden Kapitel werden Grundlagen, die zum Verständnis dieser Arbeit notwendig sind, vorgestellt. Dazu gehört das Prinzip des virtuellen Speichers in Abschnitt 2.1. In Abschnitt 2.2 werden Caches im Allgemeinen vorgestellt. Das Angriffsszenario wird in Abschnitt 2.3 beschrieben, worauf Abschnitt 2.4 aufbaut, indem die einzelnen Seitenkanalangriffe dargestellt werden. Die Methoden `Flush+Reload` und `Flush+Flush` werden in den Abschnitten 2.5 und 2.6 nochmal detailliert erläutert. Die darauffolgenden Abschnitte beinhalten die Gegenmaßnahmen für `Flush+Reload` und `Flush+Flush` (Abschnitt 2.7) und das Prinzip von Cross-Core Angriffen (Abschnitt 2.8). In den letzten Abschnitten des Kapitels werden Grundlagen für das Evaluieren von Messwerten vorgestellt. Dazu gehören in Abschnitt 2.9 die unterschiedlichen Fehlertypen bei einer Bitübertragung und in Abschnitt 2.10 die Methoden zum Bewerten eines Kanals.

2.1 Physischer und virtueller Speicher

Das Konzept des virtuellen Speichers ist eine Zuordnung von virtuellem Adressraum auf physische Adressen [Int16, 3.3.2]. Diese Adressverwaltung nennt sich Paging und ist Aufgabe der CPU sowie des Betriebssystems. Paging findet sich auf den meisten modernen Systemen wie auf Intel x86-Prozessoren [Int16, 3.3.2] und ARM-Prozessoren [ARM15, 12] wieder. Der virtuelle Adressraum ist in Speicherblöcke sogenannte Pages eingeteilt. Jede Page im virtuellen Adressraum bildet auf einen gleich großen Page Frame im physischen Adressraum ab. Es können pro Prozess mehrere Pages auf denselben Page Frame abbilden. Genauso können auch verschiedene Prozesse auf denselben Page Frame abbilden. Dies wird zum Beispiel bei Prozessen, die auf denselben geteilten Speicher (Shared Memory) zugreifen, benutzt. Es können jedoch nicht mehrere Page Frames auf eine Page abbilden. Unter ARMv8 ist die die Größe einer Page. die sogenannte Page Granularität, einstellbar von 4 KiB bis 64 KiB und muss immer eine Potenz von 2 Byte sein [ARM15, 12.2]. Die Zuordnung von Pages auf Page Frames wird in einer Translation Table gespeichert. Bei einer Page Granularität von 4 KiB und einem 42-Bit virtuellen Adressraum entspreche das $2^{42}/4096$ Einträge. Mit 64 Bit pro Eintrag wären das 8 GiB, die in der TLB gespeichert werden müssten. Doch bei einem 48-Bit virtuellem Adressraum erreicht die Translation Table bereits 512 GiB. Um die Speicherkosten zu verringern wird eine Multi-Level Translation Table verwendet. Die genaue Funktionsweise einer Multi-Level Translation Table wird im folgenden Abschnitt an einem Beispiel erklärt.

In Abbildung 1 wird anhand eines konkreten Beispiels erklärt, wie eine virtuelle Adresse in eine physische Adresse übersetzt wird. In diesem Beispiel ist die virtuelle Adresse 42 Bit lang und es gibt zwei Level Page Table. Die höchstwertigsten 16 Bit der virtuellen Adresse müssen entweder alle 0 oder 1 sein, ansonsten schlägt die Übersetzung fehl. Sind die ersten 16 Bit alle 0 oder 1, so wird der Translation Table Base Register `TTBR0` bzw. `TTBR1` als Basisadresse für die erste Page Table benutzt (in diesem Fall L2 Page Table) [ARM16, 4.3.44 ff.]. Die `TTBR` ermöglichen schnelleren

Zugriff auf virtuelle Adressen derselben First Level Page, da der Inhalt der TTBR nicht geändert werden muss. Als Nächstes wird der Level 2 Index [41:29] in der L2 Page Table nachgeschlagen. Die gefundene Adresse bildet die Basis Adresse für die nächste Page Table (in diesem Fall die L3 Page Table). Der Level 3 Index [28:15] wird daraufhin in der L3 Page Table nachgeschlagen und der gefundene Eintrag bildet die obersten 32 Bit der physischen Adresse. Die restlichen 16 Bits der physischen Adresse werden von der virtuellen Adresse übertragen und bilden somit die gesamte physische 48-Bit Adresse. [ARM15, 12.3] Das Beispiel wäre erweiterbar durch weitere Page Tables, jedoch besitzt der Cortex A72, die gewählte Testumgebung, eine 2-Level TLB Struktur [ARM16, 5.2] und hat somit wie im Beispiel zwei Page Tables.

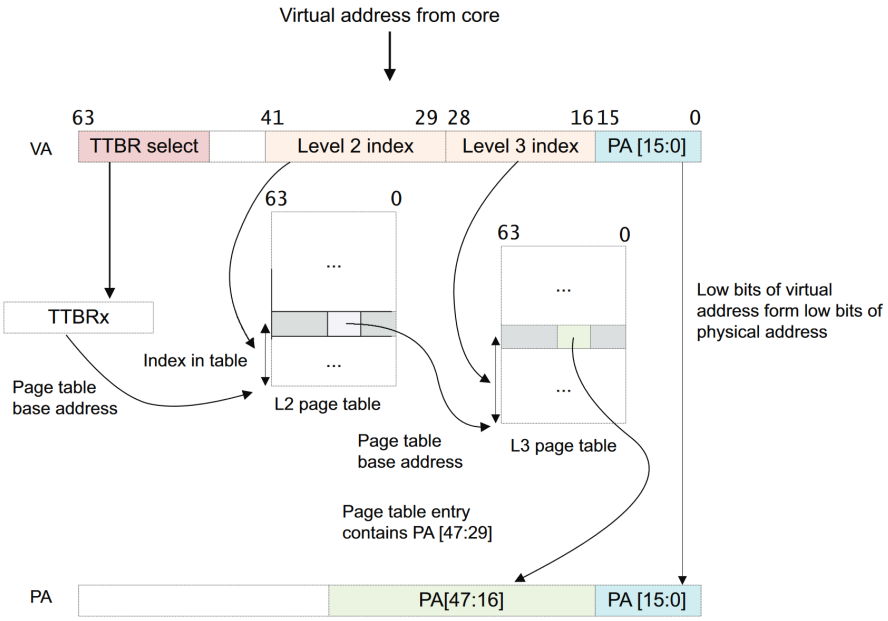

Abbildung 1: *Übersetzung von Virtueller Adresse zu Physischer Adresse auf ARMv8 Architektur am Beispiel einer 42-Bit virtuelle Adresse, einer Page Granularität von 64 KiB und zwei Level Page Tables [ARM15, 12.3]*

Die Vorteile der virtuellen Adressierung sind zum Beispiel, dass Prozesse ihren eigenen virtuellen Adressraum besitzen und unprivilegierte Speicherzugriffe von der Hardware leichter zu erkennen sind. Da durch virtuelle Adressierung Programme an eine beliebige Position im physischen Speicher platziert werden können, muss für positionsabhängigen Code bei der Ausführung relative Adressen nicht neuberechnet werden. Zudem kann zusammenhängender Code, der auf verschiedenen Kernen verteilt ist, in denselben Adressraum abgebildet werden und somit die Ladezeit verringern. [Den96]

2.2 Allgemeines zu CPU Caches

Ein CPU Cache ist ein Zwischenspeicher zwischen CPU und Arbeitsspeicher. Mit geringerer Speicherkapazität und Zugriffszeit als die des Arbeitsspeichers, werden im Cache Daten und Instruktionen abgespeichert, die für zukünftige Ausführungen wahrscheinlich benutzt werden. Dies kann durch Prefetching erfolgen oder auch durch das Abspeichern von zuletzt benutzten Daten und Instruktionen. Im Cache abgelegte Daten können bei erneutem Zugriff mit einer geringeren Latenz abgerufen werden, als wenn sie aus dem Hauptspeicher ausgelesen werden. Ein erneuter Zugriff auf Daten, die bereits im Cache liegen, nennt sich Cache Hit. Ansonsten ist es ein Cache Miss.

Intel x86, sowie ARM Prozessoren besitzen sogenannte Level-Caches, namentlich L1-Cache, L2-Cache und je nach Prozessor auch einen L3-Cache [ARM15, 11] [Int16, 3A.11.1]. Der L1-Cache hat die geringste Speicherkapazität und niedrigste Latenz. Der L3-Cache hat die größte Speicherkapazität und die höchste Latenz. Die Speicherkapazität und Latenz des L2 ist zwischen dem L1 und L3 einzuordnen. Unter ARMv8 besitzt jeder physischer Kern einen eigenen L1-Cache. Der L2- und L3-Cache sind beide Unified-Caches und werden von allen Kernen benutzt. Der Level-Cache mit dem höchsten Level wird auch Last-Level-Cache genannt. Wenn ein Cache inclusive zu einem anderem ist, dann beinhaltet er alle Cache-Zeilen des anderen Caches ebenfalls. Ein Cache ist exclusive, wenn eine Cache-Zeile in nur einem Cache-Level gleichzeitig geladen werden darf. Wenn keines der beiden Kriterien erfüllt ist, dann ist der Cache non-inclusive. [ARM15, 11] [Int16, 3A.11.1]

Datenzugriffe überprüfen als ersten Cache immer zuerst den schnellen L1-Cache des dazugehörigen Kernes. Wenn die Daten nicht gefunden wurden, wird als nächstes der L2-Cache überprüft. Sofern der L2-Cache nicht der Last-Level-Cache ist, wird nach Misserfolg noch der L3-Cache überprüft. Wenn Daten im L2 oder L3 gefunden wurden, werden diese in den L1-Cache und allen Caches, die inclusive zu dem L1-Cache sind, geladen und die Daten werden an den Prozess, der den Datenzugriff ausgeführt hat, weitergeleitet. Wenn die Daten in keinem Cache gespeichert sind, werden diese von dem sehr langsamen Hauptspeicher geladen. Zusätzlich werden die Daten in den L1-Cache und jedem Cache, zu dem der L1 inclusive ist, geladen. [ARM15, 11] [Int16, 3A.11.1]

ARMv8 Prozessoren besitzen eine sogenannte Modified-Harvard-Architektur [ARM15, 11.1]. Daten und Instruktionen werden also getrennt gespeichert. Ein L1-Cache ist somit aufgeteilt in einen L1-Datencache und L1-Instruktionscache. Ein Unified-Cache speichert jedoch Daten und Instruktionen am selben Ort ab.

2.2.1 Typen von Caches

Die bekanntesten Typen von CPU Caches sind der Direct Mapped Cache, der Fully Associative Cache und der Set Associative Cache. Alle drei teilen den Hauptspeicher in Blöcke ein. Die Größe eines Blocks ist definiert durch den Word Offset und die Länge eines Words. Ein Word Offset von 2 Bits bedeutet, dass vier Wörter unterschieden werden können. Zusammen mit einer 32-Bit Adressierung gibt es also vier 32-Bit Wörter und somit eine Blockgröße von 128 Bit. Das Beispiel wird anhand eines Direct Mapped Caches in Abbildung 2 veranschaulicht.

In den folgenden Abschnitten werden die drei Cache Typen an einer 32-Bit Architektur vorgestellt.

Direct Mapped Cache

Die einfachste Form eines Caches ist der Direct Mapped Cache (siehe Abbildung 2). Die zwei niederwertigsten Bits der Adresse bilden hier den Byte Offset und sind meistens 00, da mit 32 Bit Words gerechnet wird. Eine Cache-Zeile wird durch die Größe eines Blockes definiert. Somit beschreibt der Word Offset die Position eines Words innerhalb einer Cache-Zeile. Die mittleren Bits der Adresse, der Index, definieren in welche Cache-Zeile ein Eintrag geschrieben wird. Adressen mit demselben Index heißen kongruent zueinander. Die restlichen Bits bilden den Tag und werden zu der jeweilige Cache-Zeile eingetragen. Sobald ein Eintrag geschrieben wurde, wird das Valid Bit V auf 1 gesetzt. Wenn das Valid Bit auf 0 gesetzt wird, ist die Cache-Zeile ungültig und kann nie zu einem Cache Hit führen. So kann zum Beispiel bei Löschung einer Cache-Zeile nur das Valid

Bit geflippt werden statt, damit der langsamere Löschvorgang nicht durchgeführt werden muss. [Anl19]

Bei einem Datenzugriff wird der Index der Adresse mit allen Indizes im Cache verglichen. Wenn derselbe Index im Cache gefunden wurde, wird der Tag der Adresse mit dem Tag der Cache-Zeile verglichen. Anschließend wird geprüft, ob das Valid Bit gesetzt ist. Wenn alle Vergleiche erfolgreich waren, dann wird ein Cache Hit ausgegeben und durch das Byte und den Word Offset kann das gewünschte Word aus dem Cache gelesen werden. Wenn einer der Schritte scheitert, dann werden die Daten aus einem höheren Cache Level oder dem langsameren Hauptspeicher ausgelesen und ein neuer Eintrag für die abgerufenen Daten im Cache angelegt. Hierbei wird eine mögliche kongruente Adresse, also eine Adresse mit demselben Index, im Cache überschrieben und somit aus dem Cache gelöscht. [Anl19]

Der Vorteil des Direct Mapped Caches ist, dass räumliche Lokalität ausgenutzt wird und erneute Speicherzugriffe auf verschiedene Words eines selben Blockes zu einem Hit führen können. Zudem findet der Direct Mapped Cache Einträge bei größeren Caches im Vergleich zum folgenden Cache Typen deutlich schneller. Der große Nachteil jedoch ist, dass sich kongruente Adressen immer verdrängen und ein ständiger Wechsel zwischen zwei kongruenten Adressen nie zu einem Cache Hit führen kann. [Anl19]

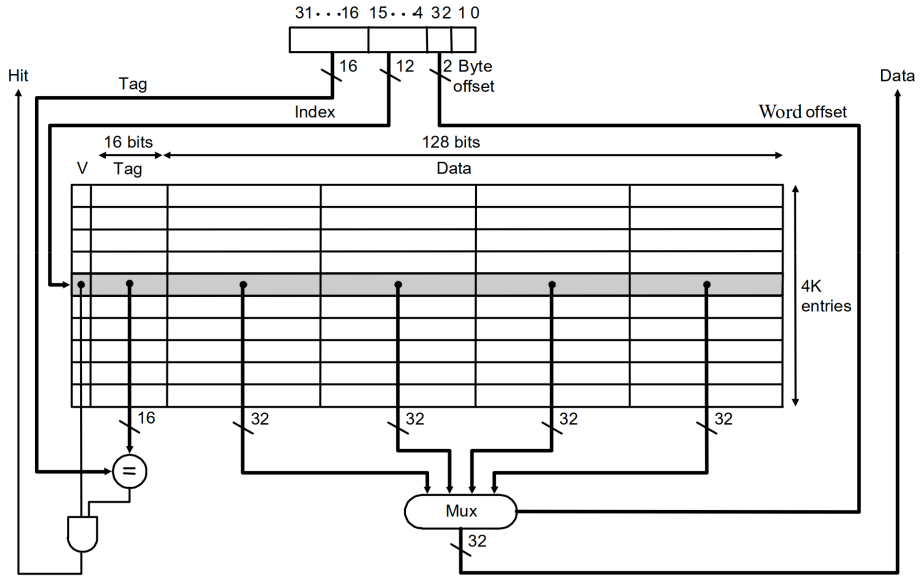

Abbildung 2: *Direct Mapped Cache (32 Bit) [Anl19]*

Fully Associative Cache

Bei dem Fully Associative Cache ist jede Cache-Zeile ein eigener Cache-Way. Daten können in jede Cache-Zeile geladen werden, da sie nicht mehr anhand der Index Bits zugeordnet werden. [Anl19] Somit können kongruente Adressen gleichzeitig in den Cache geladen werden. Da Index Bits nicht benötigt werden, bestehen die Tags beim Fully Associative Cache aus deutlich mehr Bits als beim Direct Mapped Cache.

Ähnlich wie beim Direct Mapped Cache gibt es auch beim Fully Associative Cache ein Valid Bit, welches statt einer Löschung auf 0 gesetzt werden kann. Sobald der gesamte Cache belegt ist und ein neuer Eintrag geladen werden soll, muss ein alter Eintrag überschrieben werden. Welcher Eintrag

überschrieben wird, entscheidet die Cache Replacement Policy. Verschiedene Cache Replacement Policies werden in Abschnitt 2.2.2 vorgestellt. [Anl19]

Der Vorteil des Fully Associative Caches ist, dass mehrere kongruente Adressen gleichzeitig im Cache gespeichert sein können. Der Nachteil jedoch ist, dass aufgrund des fehlenden Index, für jeden Cache-Way der Cache-Tag verglichen werden muss, was bei großen Caches zu hohen Latenzen führen kann. Fully Associative Caches sind daher nur für kleinere Caches geeignet. [Anl19]

Set Associative Cache

Der Set Associative Cache ist ein Kompromiss aus den beiden bereits vorgestellten Cache Typen. Hier besteht ein Cache aus Cache-Sets. Im Beispiel in Abbildung 3 gibt es pro Index 4 Cache-Zeilen. Daten werden ähnlich wie beim Direct Mapped Cache nach Index eingetragen. Jedoch gibt es pro Index mehrere Cache-Ways, in die die Daten geladen werden können. Somit können in Abbildung 3 vier kongruente Adressen pro Index gleichzeitig im Cache gespeichert sein. [Anl19]

Wenn Daten im Cache gesucht werden, werden ähnlich wie beim Direct Mapped Cache die Indizes verglichen. Nach erfolgreichem Finden, wird überprüft, ob in einen der Cache-Ways der passende Tag vorliegt. Wenn das Valid Bit im gefundenen Cache-Way gesetzt ist, dann wird ein Hit signalisiert und die Daten werden an die CPU weitergeleitet. Bei einem Miss wird ein Eintrag in einen freien Cache-Way geschrieben. Wenn alle Cache-Ways belegt sind, entscheidet die Cache Replacement Policy (siehe 2.2.2) welcher Cache-Way überschrieben wird. [Anl19]

Der Vorteil des Set Associative Caches ist, dass die wichtigsten Merkmale des Direct Mapped Caches und Fully Associative Caches zusammengefügt werden. Set Associative Caches haben eine schnelle Laufzeit beim Finden von Einträgen und ermöglicht das gleichzeitige Laden von kongruente Adressen. Dennoch ist der Fully Associative Cache bei sehr kleinen Cache Größen schneller als der Set Associative Cache. [Anl19]

In heutigen Systemen wie Desktop- und Smartphone-Umgebung hat sich der Set Associative Cache am meisten durchgesetzt und wird in fast allen Caches verwendet. Auf Intel x86 werden überwiegend Set Associative Caches implementiert. Bei besonders kleinen Caches wird jedoch meistens ein Fully Associative Cache benutzt [Int16, 2A, Kap3, Seite223]. Auf den Testsystemem dieser Arbeit, dem ARM Cortex-A72 und ARM Cortex-A35 ist ebenfalls ein Set Associative Cache implementiert [ARM16, 6.1 und 7.1] [ARM19, 6.1 und 7.1].

2.2.2 Cache Replacement Policies

Beim Laden von Daten in den Cache kann es dazu kommen, dass alle Blöcke für einen Index bereits Daten beinhalten. Eine weitere kongruente Adresse würde dann keinen freien Platz finden und eine der alten Cache-Zeilen müsste überschrieben werden. Jetzt stellt sich die Frage, welcher Block gelöscht werden soll. Im Optimalfall wird der Block, der am längsten nicht mehr gebraucht wird, überschrieben, aber das kann das System nicht wissen [Anl19].

Um zu bestimmen, welcher Block verdrängt wird, werden Verdrängungsstrategien, sogenannte Cache Replacement Policies, verwendet. Eine Cache Replacement Policy gibt an, welche Daten als nächstes gelöscht werden sollen. Bei einem Direct Mapped Cache ist keine Verdrängungsstrategien implementiert, da im Cache nicht zwei kongruente Adressen sein können. Im Folgenden werden die gängigsten Cache Replacement Policies vorgestellt und erklärt.

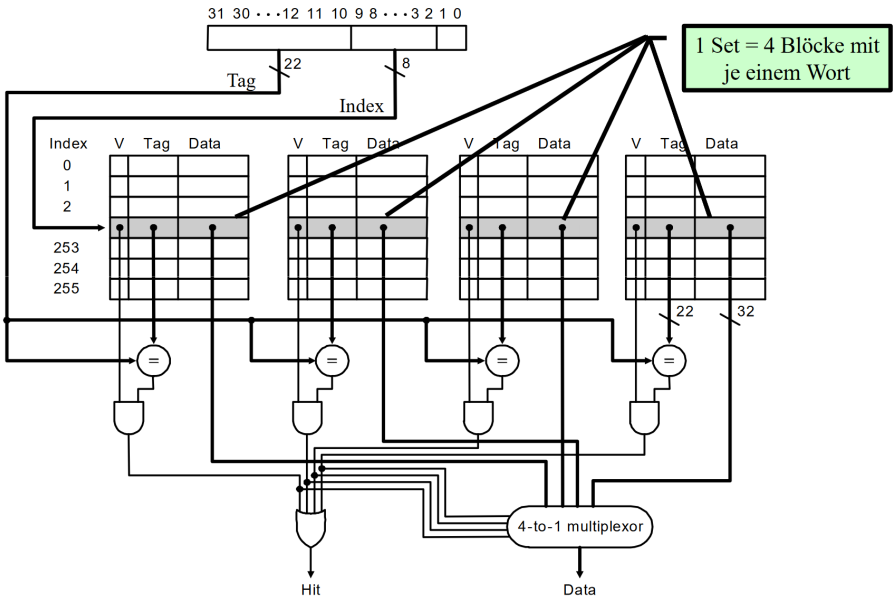

Abbildung 3: *Set Associative Cache (32 Bit) [Anl19]*

- **least recently used** (LRU): LRU entfernt den kongruenten Speicherblock, der am längsten nicht mehr benutzt wurde, aus dem Cache. In der Praxis führt dies zu einer niedrigen Miss Rate. Trotz der hohen Implementierungskosten, gilt LRU als beste Strategie heutzutage [Anl19].
- **least frequently used** (LFU): LFU entfernt den am wenigsten genutzten kongruenten Block. Annahme ist, dass der gelöschte Block auch in Zukunft nur sehr wenig genutzt wird [Anl19].
- **FIFO**: Diese Verdrängungsstrategie entfernt den kongruenten Speicherblock, der als erstes dem Cache hinzugefügt wurde. Diese Strategie implementiert die Datenstruktur einer Queue. FIFO erreicht ein ähnliches Hit zu Miss Verhältnis wie LRU. Trotz der besseren Leistung von LRU wird bei Caches mit großen Speicherkapazitäten häufiger FIFO verwendet. Grund dafür ist, dass die Implementierung von LRU in großen Caches deutlich aufwändiger als die von FIFO ist [Anl19].
- **LIFO**: LIFO entfernt den zuletzt geladenen kongruenten Speicherblock. Genau wie FIFO eine Queue implementiert, wird bei LIFO ein Stack implementiert. [Anl19].
- **random replacement** RR: RR löscht einen zufälligen kongruenten Speicherblock. Der Vorteil von RR ist, dass die Implementierungskosten sehr gering sind. Benchmarks von großen Caches mit verschiedenen Verdrängungsstrategien zeigten, dass der Unterschied von LRU und anderen Policies nur sehr gering ist. Daher wird oft RR statt LRU implementiert, um die Implementierungskosten zu verringern [Anl19].

2.2.3 Cache Adressierungen

Für die Zuweisung von Daten in den Cache wird die Adresse der Daten in Index Bits und Tag Bits eingeteilt. Die Index Bits beschreiben in welche Cache-Zeile die Daten abgelegt werden können und die Tag Bits helfen beim Wiederfinden der Daten im Cache. Die Adressierung der Tag und Index Bits kann physisch oder virtuell berechnet werden. Insgesamt folgen daraus vier Adressierungsmethoden von denen drei in heutigen Desktop- und Smartphone-CPUs verwendet werden.

Virtually-indexed and Virtually-tagged (VIVT)

VIVT benutzt virtuelle Adressierung für Index und Tag Bits. Da die virtuellen Adressen von dem Translation Lookaside Buffer (TLB) nicht in physische Adressen umgerechnet werden müssen, haben VIVT-Caches sehr geringe Latenzen. Ein Nachteil ist, dass geteilter Speicher (Shared Memory) im Cache fast nie an derselben Stelle liegt, was zu erhöhter Cache-Nutzung führt. Des Weiteren müssen nach einem Kontextwechsel also, wenn das Betriebssystem den aktuellen Prozess unterbricht, um eine andere Routine fortzusetzen, die benutzten Cache-Zeilen als ungültig markiert werden. Grund dafür ist, dass virtuelle Adressen nicht eindeutig sind. VIVT-Adressierung wird daher oft für besonders schnelle und kleine Caches verwendet. TLB werden meistens als VIVT-Caches implementiert. [Gru17].

Physically-index and Physically-tagged(PIPT)

PIPT benutzt physische Adressierung für Index und Tag Bits. Der TLB muss jede virtuelle Adresse zuerst in eine physische Adresse umwandeln, was die Latenz eines PIPT-Caches erhöht. Im Vergleich zum VIVT-Cache ist in einem PIPT-Speicher der geteilte Speicher immer an derselben Stelle. Da physische Adressen eindeutig sind, müssen auch bei einem Kontextwechsel keine Cache-Zeilen als ungültig markiert werden. Heutzutage werden PIPT-Caches als Daten- und Instruktionscaches benutzt, während der davorgestellte TLB als ein VIVT-Cache implementiert ist, um die hohen Latenzen der Adressumrechnung zu verringern. [Gru17]

Virtually-indexed Physically-tagged(VIPT)

VIPT benutzt virtuelle Adressierung für Index Bits und physische Adressierung für Tag Bits. Der Vorteil im Vergleich zu PIPT ist, dass der Index direkt im Cache gesucht werden kann, während der TLB noch den physischen Tag berechnet. Somit ist die Latenz des TLB nicht so ausschlaggebend für die gesamte Latenz des Caches. Jedoch kann der Tag erst verglichen werden, wenn er in die physische Adresse umgewandelt wurde. [Lip16b] Ein Vorteil von VIPT ist, dass ein physischer Tag benutzt wird und somit der Tag eindeutig ist, wodurch Shared Memory möglich ist. Aufgrund der niedrigen Latenz und des eindeutigen Tags, findet der VIPT-Cache auf Desktop- und Smartphone-CPUs Anwendung.

Physically-indexed Virtually-tagged (PIVT)

PIVT benutzt physische Adressierung für Index Bits und virtuelle Adressierung für Tag Bits. PIVT bringt keinen Vorteil mit sich, da der physische Tag zu Beginn von der TLB berechnet werden muss und der virtuelle Index nicht eindeutig ist. Die Implementierung eines PIVT-Caches ist in der Theorie möglich, aber findet in der Praxis keinen Anwendungsfall. [Lip16b]

2.3 Beschreibung des Angriffszenarios

Bevor das Angriffszenario beschrieben werden kann, muss das Konzept eines Seitenkanalangriffes erklärt werden. Ein Seitenkanal ist eine Informationsquelle, aus der mit einem Seitenkanalangriff Informationen über ein System gewonnen werden können, die nicht aus dieser Quelle zu entnehmen sein sollten. Seitenkanäle sind zum Beispiel die Laufzeit eines Vorganges oder der Energieverbrauch eines Gerätes. Aus diesen Daten können dann Informationen über bestimmte Vorgänge im System abgeleitet werden.

Mithilfe von cachebasierten Seitenkanalangriffen können Prozesse abgehört werden. In dieser Bachelorarbeit wird mit Seitenkanalangriffen eine verdeckte Kommunikation zwischen zwei Prozessen aufgebaut. Im Fall einer Kommunikation wird nicht von einem klassischen Angriff gesprochen, da nichts angegriffen wird. Die Rollen Angreifer-Prozess und Opfer-Prozess werden zu Empfänger-Prozess und Sender-Prozess [YF14]. Vorab wird eine bestimmte Cache-Zeile oder ein bestimmtes Cache-Set als Medium festgelegt (Genaueres in der Erklärung der einzelnen Strategien in Abschnitt 2.4). Ziel ist es über dieses Medium unentdeckt zu kommunizieren. `Flush+Reload` und `Flush+Flush` benötigen darüber hinaus einen geteilten Speicher, auf den von beiden Prozessen lesend zugegriffen werden kann [YF14] [GMWM16]. Zudem muss der Cache, über den kommuniziert wird, ein einen physischen Tag besitzen (Cache Adressierung siehe Abschnitt 2.2.3). Keiner der Prozesse besitzt Root Rechte, wodurch nur unprivilegierte Instruktionen benutzt werden können. Wie aus einem Seitenkanalangriff ein verdeckter Kanal gebaut werden kann, wird genauer in Abschnitt 3.6 erklärt.

2.4 Seitenkanalangriffe und der historische Kontext

Das erste Konzept eines Seitenkanalangriffes auf kryptografische Verfahren wurde 1996 von Paul C. Kocher vorgestellt. Durch Performance-Optimierungen benötigen kryptografische Verschlüsselungen je nach Eingabe unterschiedlich lang. Kocher stellte das Konzept von Seitenkanalangriffe vor, mit denen der private Schlüssel bei Verfahren wie RSA und Diffie-Hellman bestimmt werden kann. Der benutze Seiteneffekt ist die unterschiedliche Berechnungszeit von Verschlüsselung. [Koc96]

2.4.1 Evict+Time und Prime+Probe

Im Jahr 2006 stellten Osvik et al. die cachebasierten Seitenkanalangriffe `Evict+Time` und `Prime+Probe` vor [OST06].

Bei `Evict+Time` löst der Angreifer beim Opfer Berechnungen aus und misst wie viel Zeit diese benötigen. Daraufhin werden gezielt bestimmte Cache-Sets gelöscht. Der Angreifer löst die Berechnungen erneut aus und misst die Dauer. Ist die Dauer beider Messungen gleich, so kann der Angreifer darauf zurückschließen, dass die gelöschten Cache-Sets von dem Opfer nicht benutzt werden. Wenn die zweite Berechnung mehr Zeit beansprucht hat, dann folgt daraus, dass eines der gelöschten Cache-Sets von dem Opfer benutzt wird [OST06]. Für eine Kommunikation zwischen Empfänger und Sender wird vorher festgelegt, welches Cache-Set als Medium benutzt wird. Abhängig davon, ob der Sender das Cache-Set verwendet, kann der Empfänger dies als 0 oder 1 interpretieren.

Für einen `Prime+Probe`-Angriff füllt der Angreifer Cache-Sets mit Daten. Nach einem vom Angreifer bestimmten Zeitfenster greift der Angreifer auf die gecashten Daten zu. Ein Cache Miss hat eine deutlich längere Zugriffszeit und deutet darauf hin, dass die Daten wahrscheinlich von dem Opfer durch einen kongruenten Speicherblock verdrängt wurden. [OST06]

`Prime+Probe` ist deutlich zuverlässiger als `Evict+Time`, da `Evict+Time` nur indirekt über die Berechnungszeit die Cache-Zugriffszeiten misst. `Prime+Probe` hingegen misst die direkten Cache-Zugriffszeiten. Der Vorteil beider Seitenkanalangriffe ist, dass die Angriffe auf den meisten Systemen ausgeführt werden können. Bei `Prime` wird der Cache mit Daten befüllt und bei `Evict` kann durch Verdrängung der Daten ein ähnliches Resultat erreicht werden. Diese Funktion besitzt jedes System mit einem Cache, während hingegen eine Flush-Cache-Line-Instruktion nicht auf allen Systemen unterstützt wird [OST06]. Folgende Strategien `Flush+Reload` und `Flush+Flush` können

hingegen nicht auf jedem System ausgeführt werden, weil diese eine Flush-Instruktion benötigen. Ein Nachteil von Prime+Probe ist jedoch, dass auf neuen Systemen komplexere Cache Replacement Policies (Verdrängungsstragien: siehe 2.2.2) implementiert werden, die Prime erschweren [Gru17].

2.4.2 Flush+Reload

2014 stellte Yarom den Seitenkanalangriff Flush+Reload vor. Bei einem Flush+Reload-Angriff kann der Angreifer herausfinden, ob bestimmte Daten aus geteiltem Speicher (Shared Memory) von dem Opfer benutzt werden. [YF14] Es wird ausgenutzt, dass durch den physischen Tag ein geteilter Speicher immer in die selbe Cache-Zeilen geladen wird. Zuerst löscht der Angreifer eine bestimmte Cache-Zeile. Nach einer Wartezeit lädt der Angreifer die Daten erneut und überprüft die Zugriffszeit. Ist die Zugriffszeit sehr kurz, dann wurden die Daten in der Zwischenzeit erneut in die Cache-Zeile geladen. Bei längerer Zugriffszeit wurden die Daten von dem Angreifer aus dem Hauptspeicher geladen. [YF14] Wird über Flush+Reload eine Kommunikation geführt, so einigen sich Angreifer und Opfer vorher auf eine oder mehrere Cache-Zeilen, die von dem Angreifer beobachtet werden sollen.

Der Vorteil von Flush+Reload im Vergleich zu den Seitenkanalangriffen von Osvik et al. ist, dass Flush+Reload auf einer einzigen Cache-Zeile operieren kann und auf ganz bestimmte Daten abzielen kann. Der Nachteil jedoch ist, dass Flush+Reload nur auf geteiltem Speicher funktioniert [YF14]. Ein weiterer Nachteil von Flush+Reload ist, dass es eine sehr hohe Anzahl an Cache Misses auslöst. Dies kann das System erkennen und den Prozess daraufhin beenden. [Gru17] Trotz des Namens benötigt Flush+Reload nicht zwangsläufig eine Flush-Instruktion. Das Löschen einer Cache-Zeile kann auch durch Verdrängung erreicht werden [YF14]. Genauere Details zu Flush+Reload werden in Abschnitt 2.5 erklärt.

2.4.3 Flush+Flush

Gruss et al. stellte im Jahr 2016 den cachebasierten Seitenkanalangriff Flush+Flush als Entwicklung von Flush+Reload vor. Der Seitenkanalangriff ist sehr ähnlich zu Flush+Reload insofern, dass Flush+Flush herausfinden kann, ob bestimmte Cache-Zeilen benutzt werden. Genau wie Flush+Reload benötigt Flush+Flush einen geteilten Speicher, sodass Adressen des geteilten Speichers immer in dieselbe Cache-Zeile geladen werden. Im ersten Schritt werden die beobachteten Cache-Zeilen gelöscht. In einer vom Angreifer bestimmte Wartezeit, kann das Opfer eine der beobachteten Cache-Zeilen in den Cache laden. Nach der Wartezeit werden dieselben Cache-Zeilen erneut gelöscht. Der benutze Seitenkanaleffekt ist die Ausführungszeit von der Flush-Instruktion. Die Flush-Cache-Zeile Instruktion benötigt unterschiedlich lang abhängig davon, ob die übergebene virtuelle Adresse sich im Cache befindet oder nicht. Dauert der Flush-Vorgang länger, so wurde die übergebene Adresse wahrscheinlich von dem Opfer benutzt. [GMWM16]

Der Vorteil von Flush+Flush gegenüber den vorherigen cachebasierten Seitenkanalangriffen ist, dass Flush+Flush schneller ausgeführt werden kann. Ein durchschnittlicher Flush benötigt weniger Zeit als ein durchschnittlicher Reload. Gruss et al. erreichten mit Flush+Flush eine 6.7fach höhere Geschwindigkeit als bisherige Seitenkanalangriffe [GMWM16]. Zudem gilt Flush+Flush im Vergleich zu Flush+Reload auf Intel x86 als Stealthy, da keine Datenzugriffe ausgeführt werden, sondern nur Cache-Zeilen gelöscht werden. Datenzugriffe werden auf Intel x86 von Hardwareperformanzzählern gezählt, während das Flushen von Cache-Zeilen nicht verfolgt wird. [GMWM16] In Abschnitt 2.6 wird Flush+Flush genauer vorgestellt.

2.4.4 Seitenkanalangriffe auf ARM Prozessoren

Erst im Jahr 2016 wurden die ersten Arbeiten über cachebasierte Seitenkanalangriffe auf ARM Prozessoren veröffentlicht. Lipp et al. stellte `Evict+Time`, `Flush+Reload` und `Flush+Flush` auf verschiedenen ARM Prozessoren vor [LGS+16]. Dabei sind Lipp et al. auf Probleme und Hürden gestoßen, die es auf den herkömmlichen Intel x86 Prozessoren zuvor nicht gab. Die Cache Replacement Policies (siehe 2.2.2) von ARM Prozessoren sind in den meisten Fällen Random Replacement (`RR`) und nicht Least Recently Used (`LRU`). Durch `RR` ist nicht vorhersehbar, welche Cache-Zeile als Nächstes überschrieben wird. Dies erschwert `Prime` von `Prime+Probe`, da der Angreifer wissen muss, welche Cache-Zeile zuerst überschrieben wird und falsche Vorhersagen die Messungen verfälschen würden [LGS+16]. Zusätzlich sorgt `RR` für erhöhtes Cache-Rauschen, was bei Cache-Seitenkanalangriffen zu mehr verfälschten Messergebnissen führen kann. [LGS+16]

Ein weiteres Ziel war es einen Cross-Core Seitenkanalangriff durchzuführen. Sender und Empfänger sind also auf verschiedenen physischen Kernen und teilen somit keinen L1-Cache (mehr zu Level-Caches siehe 2.2). Ein Cross-Core-Angriff wurde üblicherweise über die Inklusivität des Last-Level-Caches bewerkstelligt. Jedoch waren Last-Level-Caches auf ARM Prozessoren nie inclusive. Neuere Generationen von ARM Prozessoren wie der ARM Cortex A72 besitzen einen inclusive Last-Level-Cache. Der L2-Cache des ARM Cortex-A35 und des ARM Cortex-A72 sind beispielsweise inclusive zum L1-Datencache [ARM16, 7.1].

2.5 Allgemeines zu Flush+Reload

`Flush+Reload` ist ein cachebasierter Seitenkanalangriff, der 2014 von Yarom vorgestellt wurde [YF14].`Flush+Reload` nutzt die unterschiedlichen Zugriffszeiten bei Speicherzugriffen als Seitenkanal aus. Wenn Daten in den Cache geladen sind, ist ein Speicherzugriff deutlich kürzer, als wenn sie aus dem Hauptspeicher geladen werden müssen. Mithilfe dieses Seitenkanals kann ein Angreifer herausfinden, ob ein Opfer-Prozess auf einen bestimmten Speicherblock zugegriffen hat. [YF14]

In Abbildung 4 wird der Verlauf eines `Flush+Reload` Angriffes dargestellt. Ein roter Strich bedeutet, dass ein bestimmter Speicherblock im Cache geladen ist. Ein blauer Strich hingegen zeigt, dass in diesem Schritt ein bestimmter Speicherblock in den Cache geladen wird. Im ersten Schritt (1) wird von dem Angreifer (attk) ein Speicherblock im virtuellen Speicher des Angreifers aus dem Cache entfernt. Dabei wird nicht nur die Cache-Zeile im L1-Cache des jeweiligen Kernes gelöscht, sondern auch in allen höheren Level des Caches (mehr zu Cache-Level in Abschnitt 2.2). In Schritt 2 kann der Opfer-Prozess (vict) entweder auf den Speicherblock zugreifen oder nicht. Bei einem Speicherzugriff (2b) wird der Speicherblock in den zugehörigen L1-Cache und in alle höheren Cache-Level geladen (blauer Strich). Greift das Opfer nicht auf den Speicherblock zu, so wird es nicht in den Cache geladen. Es ist dennoch möglich, dass Dritte auf den Speicherblock zugreifen und somit die Cache-Zeile füllen. In Schritt 3 greift der Angreifer auf den Speicherblock zu. Wenn das Opfer den Speicherblock nicht geladen hat (2a), dann muss der Angreifer in Schritt 3 den Speicherblock aus dem Hauptspeicher laden (3a). Dabei wird der Speicherblock in den zugehörigen L1-Cache und allen höheren Cache-Level geladen (blauer Strich). Wenn das Opfer den Speicherblock vorher geladen hat (2b), dann ist ein Datenzugriff kürzer, da die Daten aus dem Unified-Cache geladen werden können (hier L3-Cache) [YF14]. Befinden sich Opfer und Angreifer auf demselben physischen Kern, dann teilen sich beide denselben L1-Cache und ein Datenzugriff ist üblicherweise noch kürzer (mehr zu Cross-Core in Abschnitt 2.8).

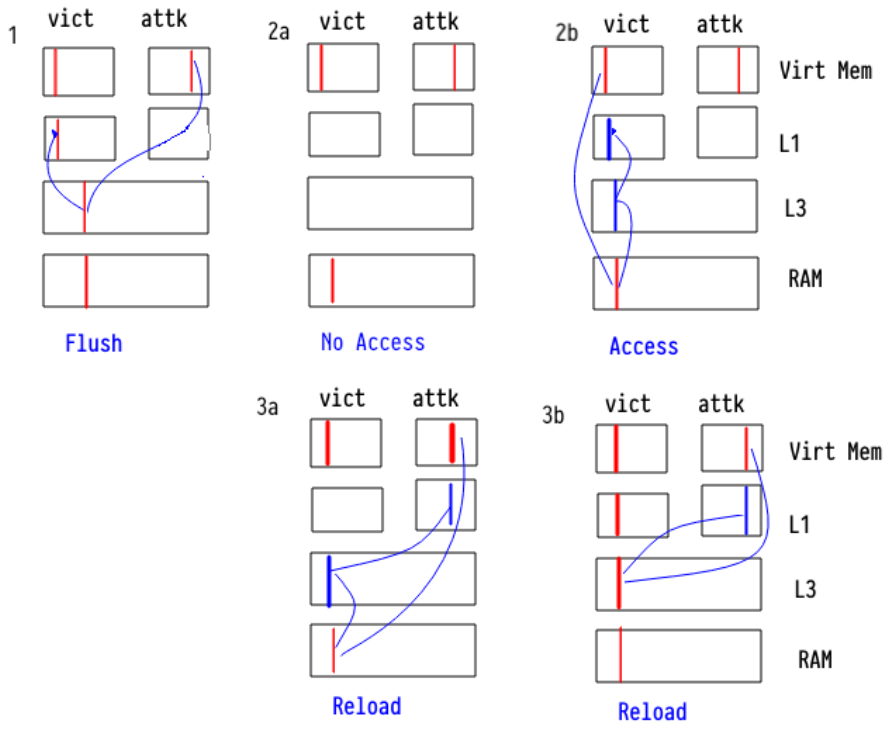

Abbildung 4: *Flush+Reload Angriff [Hei20]*

Damit sichergestellt wird, dass Angreifer und Opfer sich Cache-Zeilen teilen, muss der beobachtete Speicherblock ein geteilter Speicher sein [YF14]. In Abschnitt 3.4 wird erklärt wie dies in einem Angreiferszenario umgesetzt werden kann. Der geteilte Speicher wird jeweils in den virtuellen Adressraum des Angreifer- und Opfer-Prozesses geladen (mehr zu virtuellen Adressräumen in 2.1). Damit für die Cache-Adressierung der physische Tag benutzt wird, muss der Cache ein PIPT-Cache oder VIPT-Cache sein [Lip16b] (mehr zu Cache Adressierung in Abschnitt 2.2.3).

In Abbildung 5 sieht man die Verteilung der Timings bei der Ausführung von Flush+Reload auf einem ARM Cortex-A72. Ein Reload, der zu einem Cache Hit führt, benötigt in diesem Beispiel im Schnitt 273 CPU Zyklen. Ein Reload, der zu einem Cache Miss führt, benötigt im Schnitt 412 CPU Zyklen und ist somit deutlich länger. Die benötigten CPU Zyklen sind auf jedem System unterschiedlich und müssen vorab getestet werden [YF14]. In Abschnitt 3.5 wird erklärt wie diese Werte berechnet werden können. Zusätzlich können auf dem selbem System die Werte bei jeder Ausführung unterschiedlich sein.

Da Flush+Reload nur auf Speicher zugreifen und Cache-Zeilen löschen muss, ist Flush+Reload auf den meisten Systemen von dem Userspace ausführbar. Jedoch gibt es Systeme auf denen das Löschen einer Cache-Zeile Root Rechte erfordert. [LGS+16]

Wenn ein Angreifer ein Opfer abhören möchte, so kann der Angreifer Flush+Reload mehrfach hintereinander ausführen. In Abbildung 6 wird gezeigt wie die drei Schritte aus Abbildung 4 zyklisch ausgeführt werden. Wenn das Opfer zwischen dem Flush und Reload des Angreifers auf den Speicher zugreift (B), kann der Angreifer diesen Datenzugriff feststellen. Lädt das Opfer während der Reload Phase ebenfalls die Daten (C), dann benutzt das Opfer die von dem Angreifer gecachten Daten und der Angreifer bekommt diesen Datenzugriff nicht mit [YF14]. Wenn die Reload Phase beginnt, während das Opfer noch auf die angefragten Daten wartet (D), dann ist die

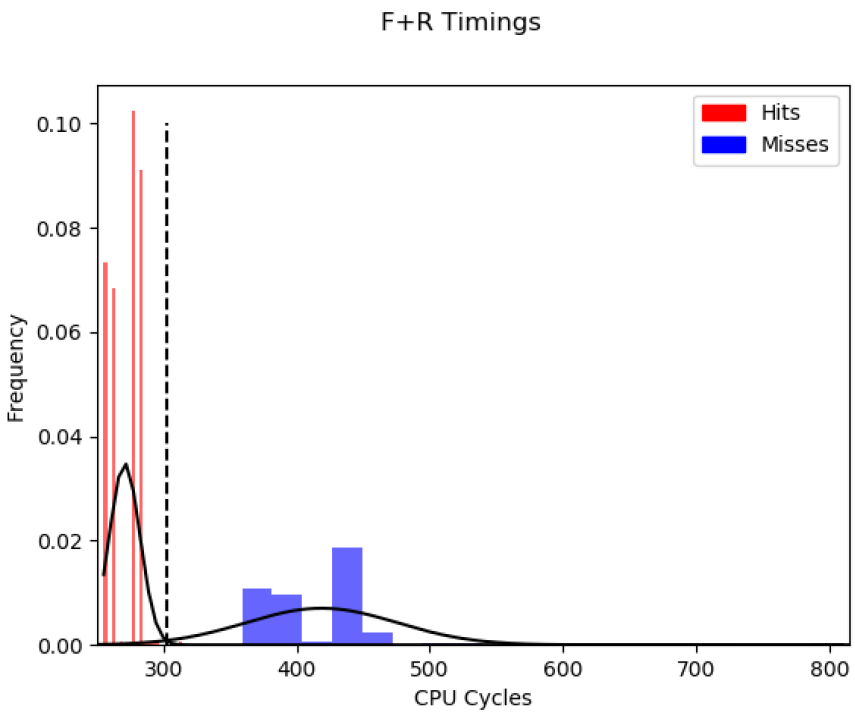

Abbildung 5: *Flush+Reload Timing*

`Reload` Zeit gekürzt, aber ein Cached `Reload`, wäre dennoch schneller. Zusätzlich kann das Opfer zwischen dem `Flush` und `Reload` mehrfach auf die Daten zugreifen (E) und der Angreifer bemerkt nur einen dieser Zugriffe [YF14].

Um Szenario C und D zu vermeiden, kann die Wartezeit zwischen `Flush` und `Reload` verlängert werden. Dadurch ist es aber wahrscheinlicher, dass Szenario E eintrifft. Durch eine längere Wartezeit ist auch wahrscheinlicher, dass andere Programme den Cache-Eintrag verdrängen, was zu falschen Negativen führen kann [YF14]. Des Weiteren kann durch eine lange Wartezeit ein Dritter auf den Speicherblock zugreifen und somit die Cache-Zeile laden und ein falsches Positives hervorrufen. Ist die Wartezeit jedoch zu kurz, so sinkt die Trefferrate, da der Datenzugriff des Opfers wahrscheinlicher auf den `Flush` oder `Reload` fallen kann. Es muss also eine passende Dauer für die Wartezeit beziehungsweise eine passende Frequenz gefunden werden. Dies wird in Kapitel 4 behandelt.

Des Weiteren kann mithilfe von `Flush+Reload` ein verdeckter Kanal gebaut werden. Dies wird in Abschnitt 3.6 genauer beschrieben.

2.6 Allgemeines zu Flush+Flush

Im Jahr 2016 stellte Gruss et al. den cachebasierten Seitenkanalangriff `Flush+Flush` vor [GMWM16]. `Flush+Flush` ist eine Variante von `Flush+Reload` (siehe Abschnitt 2.5). Der ausgenutze Seitenkanaleffekt bei `Flush+Flush` ist die unterschiedliche Ausführungszeit der Flush-Instruktion. Wird die Flush-Instruktion mit einer virtuellen Adresse ausgeführt, die nicht im Cache geladen ist, so beendet die Instruktion schon früher. Hintergrund ist, dass der eigentliche `Flush` Vorgang aus Optimierungsgründen übersprungen werden kann [GMWM16].

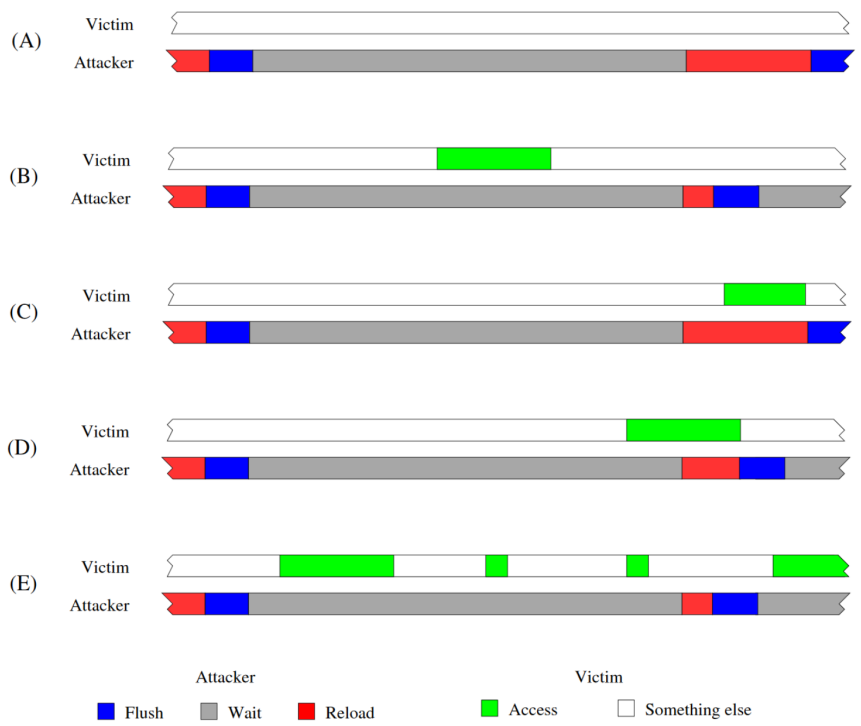

Abbildung 6: *Mögliche Flush+Reload Szenarien [YF14]*

Ähnlich wie `Flush+Reload` kann `Flush+Flush` in drei Phasen eingeteilt werden. In der ersten Phase löscht der Angreifer einen bestimmten Speicherblock, der beobachtet werden soll, aus dem Cache. In der zweiten Phase wird eine, vom Angreifer bestimmte Zeit, gewartet, in der das Opfer den Speicherblock erneut in den Cache laden kann. In der dritten Phase führt der Angreifer die Flush-Instruktion aus und deduziert aus der gemessenen Zeit, ob das Opfer in der zweiten Phase den Speicherblock geladen hat oder nicht. [GMWM16] Damit ein Angreifer ein Opfer abhören kann, muss der Angreifer `Flush+Flush` nicht alle drei Phasen zyklisch ausführen. Nach der ersten Ausführung von `Flush+Flush` ist der Cache bereits geleert und es kann direkt mit Phase 2 fortgesetzt werden. Beim Abhören eines Prozesses kann also die Wartezeit und `Flush` abwechselnd ausgeführt werden. Ähnlich wie bei `Flush+Reload` kann die Wartezeit angepasst werden, um die Genauigkeit zu optimieren. Ist das Zeitfenster zwischen den `Flush` Instruktionen zu groß, dann besteht Gefahr, dass mehrere Datenzugriffe des Opfers als nur einen erkannt werden. Außerdem ist es wahrscheinlicher, dass Dritte die geladene Cache-Zeile verdrängen, was somit zu falschen Negativen führen kann. Bei einem zu kleinen Zeitfenster ist es wahrscheinlicher, dass der Datenzugriff des Opfers während der `Flush` Phase durchgeführt wird und somit von dem Angreifer nicht erkannt werden kann.

In Abbildung 7 sieht man die Zeitmessungen eines `Flush+Flush` Angriffes. Ein durchschnittlicher Cache Miss liegt bei 219 CPU Zyklen und ein durchschnittlicher Cache Hit liegt bei 248 CPU Zyklen. Wenn man dies mit `Flush+Reload` in Abbildung 5 vergleicht, sieht man, dass die Zeitunterschiede zwischen Hit und Miss bei `Flush+Flush` deutlich geringer sind. Das Unterscheiden von Hits und Misses ist dadurch bei `Flush+Flush` schwieriger. Ein Vorteil von `Flush+Flush` gegenüber `Flush+Reload` ist, dass `Flush+Flush` deutlich schneller ausgeführt werden kann [GMWM16]. Dies liegt daran, dass ein durchschnittlicher `Reload` mehr Zeit benötigt als ein durchschnittlicher `Flush`. Zudem kann bei erneuten Ausführung von `Flush+Flush` auf das erste `Flush` verzichtet werden, da die Cache Zeile bereits leer ist. Genau wie mit `Flush+Reload` kann auch aus `Flush+Flush` ein ver-

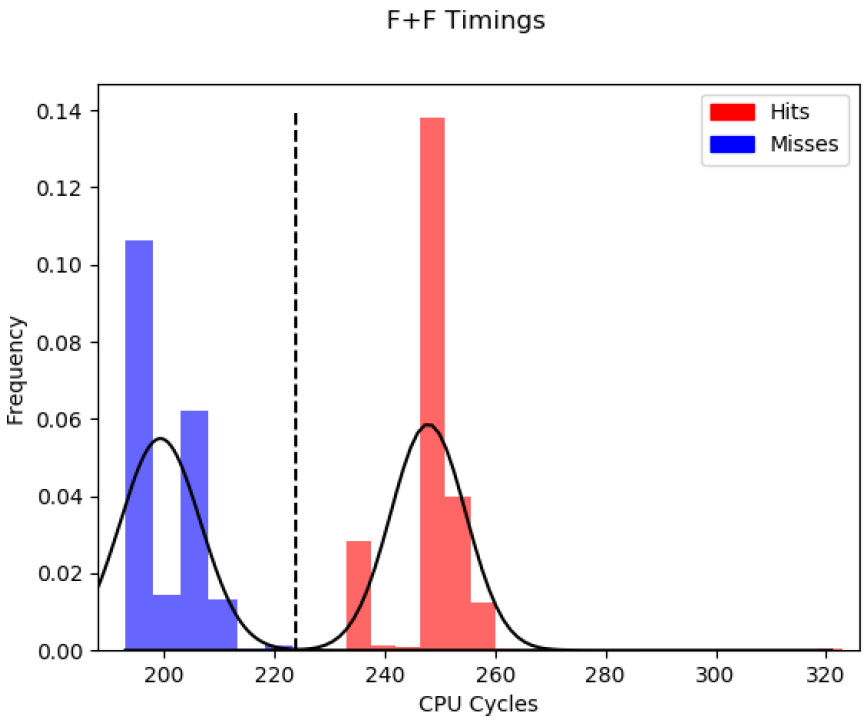

Abbildung 7: *Flush+Flush Zeiten auf einem ARM Cortex-A72*

deckter Kanal gebaut werden (siehe Abschnitt 3.6). Eine schnellere Ausführung von `Flush+Flush` führt dann zu einer höheren Datenrate des verdeckten Kanals.

Darüber hinaus gilt `Flush+Flush` auf Intel x86 als Stealthy, da es nicht von Hardwareperformanzzählern erkannt werden kann. [GMWM16] Auf dem ARM Cortex-A72 hingegen kann der Hardwareperformanzzähler Cache-Invalidierungen (Invalidierung siehe Abschnitt 3.2.2) erkennen. Diese werden für den L1- und den L2-Cache separat pro Prozess gezählt [ARM16]. Nach Dokumentation ist unklar, ob der Zähler nur bei einer Invalidierung, die zu einem Hit führt, erhöht wird. Genaueres zu Gegenmaßnahmen wird in Abschnitt 2.7 erklärt.

Ähnlich wie bei `Flush+Reload` muss der beobachtete Speicherblock ein geteilter Speicher sein [GMWM16]. Dieser muss dann mithilfe eines `PIPT`-Caches über seine physische Adresse in den Cache geladen werden, damit sichergestellt wird, dass Angreifer und Opfer dieselbe Cache-Zeile benutzen.

2.7 Gegenmassnahmen für Flush+Reload und Flush+Flush

Bei der Vorstellung von Sicherheitslücken stellt sich immer die Frage, wie man diese Lücke schließen kann oder wie man es Angreifern erschweren kann diese auszunutzen. Der naive Ansatz wäre die Zeitunterschiede von `Reload` oder `Flush` zu eliminieren. Das würde bei `Reload` bedeuten, dass der Cache deaktiviert wird, was zu sehr großen Performance Verlusten führen würde. Bei `Flush+Flush` würde dies bedeuten, dass der `Flush` Vorgang immer vollständig ausgeführt wird, auch wenn die Cache-Zeile bereits leer ist. `Flush+Reload` und `Flush+Flush` würden mit diesem Ansatz nicht mehr funktionieren, aber das System würde ohne einem Cache deutlich langsamer laufen. Statt diese hohen Performance Verluste in Kauf zu nehmen, können die Zeitunterschiede auch so angepasst werden, dass es zu Rauschen führen kann. [MWS+17] Einzelne Cache Hits

könnten künstlich verlangsamt werden und die Klassifizierung von Hit und Miss wären nicht mehr eindeutig. Mit dieser Methode wäre der Performance Verlust nicht so hoch, wie wenn die Zeitunterschiede komplett eliminiert werden. Wenn mit `Flush+Reload` oder `Flush+Flush` über einen verdeckten Kanal kommuniziert wird können Sender und Empfänger jedoch trotzdem Methoden verwenden, um dennoch fehlerfrei zu kommunizieren. Eine Methode wäre es die Nachricht so oft zu senden, bis diese fehlerfrei übertragen wurde und durch ein `ACK` von dem Empfänger bestätigt wurde. Eine fehlerfreie Nachricht könnte zum Beispiel durch passende Parity Bits oder einen übereinstimmenden CRC erkannt werden. [MWS+17] Je nach Stärke des Rauschens kann Sender und Empfänger sich auch auf ein Protokoll einigen, was Fehler in der Übertragung korrigiert. Eine weitere mögliche Einschränkung von `Flush+Reload` und `Flush+Flush` wäre es die `Flush`-Instruktion nur im privilegierten Modus auszuführbar zu machen. Somit wären für die Ausführung von `Flush+Reload` und `Flush+Flush` Root Rechte benötigt. Dies wäre jedoch nur mit einem Microcode-Update möglich, da die Hardware überarbeitet werden müsste [YF14]. Ein weiterer Ansatz wäre es Memory Deduplication auszustellen, wodurch geteilter Speicher nicht mehr möglich wäre. Da `Flush+Reload` und `Flush+Flush` nur auf geteiltem Speicher agieren können, gäbe es somit kein brauchbares Medium. [YF14] Der Performanzverlust durch das Ausstellen von Memory Deduplucation wäre jedoch hoch, da besonders Standard-Bibliotheken in den Speicher der meisten Prozesse geladen werden müssten.

Ein weiterer naiver Ansatz ist es die Angreifer-Prozesse zu beenden. Dafür müssen die Prozesse von dem System zuerst erkannt werden können. Um `Flush+Reload` zu entdecken können Hardwareperformanzzählern benutzt werden. Diese ermöglichen es unter anderem die Anzahl von Cache Hits und Cache Misses, die ein jeweiliger Prozess verursacht hat, zu beobachten [ARM20, 4.4]. Bei einer sehr hohen Anzahl an Hits und Misses kann das System auf Verdacht den Prozess beenden. `Flush` wird nicht durch Hardwareperformanzzählern auf Intel x86 verfolgt und Angreifer bleiben so unerkannt. [GMWM16] Auf dem ARM Cortex-A72 hingegen können Cache-Invalidierungen von Hardwareperformanzzählern gezählt werden [ARM16]. Somit ist es als Sicherheitsmaßnahme möglich `Flush+Flush`, durch eine erhöhte Auslösung von Invalidierungen, zu entdecken und den Prozess zu beenden. `Flush+Flush` ist dennoch schwieriger zu detektieren als `Flush+Reload`, da der Empfänger pro Iteration nur einmal `Flush` ausführt. Bei `Flush+Reload` führt der Empfänger je Iteration einen `Flush` und zusätzlich einen `Reload` aus [YF14] und führt somit doppelt so viele Aktionen pro Sekunde aus, die von Hardwareperformanzzählern [ARM16] erkannt zu werden können. Damit `Flush+Flush` und `Flush+Reload` von Hardwareperformanzzählern nicht als Bedrohung erkannt werden, können bewusst weniger `Flush` und `Reload` pro Sekunde ausgeführt werden. Dafür muss zuerst der Schwellwert für `Flush` beziehungsweise `Reload` pro Sekunde gefunden werden, ab welchem ein Prozess als Bedrohung gilt. Dies ist jedoch nicht Teil der Bachelorarbeit und bietet sich für weitere Forschung an.

2.8 Same-Core und Cross-Core Angriffe

Für `Flush+Reload` und `Flush+Flush` muss zwischen Angreifer und Opfer ein Cache geteilt werden. Da jeder physische CPU Kern einen eigenen L1-Cache besitzt, muss für eine Kommunikation über den L1-Cache sichergestellt werden, dass Sender und Empfänger auf demselben physischen CPU Kern, auch genannt Same-Core, ausgeführt werden. Bei der Ausführung von Sender und Empfänger kann mit `taskset` [Fre14] festgelegt werden, auf welchem CPU Kern ein Prozess ausgeführt werden soll.

```
1 taskset 0x5 ./sender       // execute sender on virtual cores 0 and 2
```

```
2 taskset 0x7 ./receiver     // execute receiver on virtual cores 0, 1 and 2
```

taskset nimmt als erstes Argument eine Bitmaske, die beschreibt auf welchen virtuellen Kernen der Prozess, der im zweiten Argument spezifiziert wird, ausgeführt werden darf. Das Betriebssystem kann mit den gegebenen Einschränkungen den Prozess auf einem oder mehreren beliebigen Kernen ausführen. Eine Bitmaske von 0x7 also 0111 bedeutet, dass der Prozess auf einen oder mehreren der ersten drei virtuellen Kernen ausgeführt wird. Unterstützt das System Hyperthreading, so können mehrere virtuelle Kerne auf demselben physischen Kern sein. Für einen geteilten L1-Cache reicht es aus, wenn Sender und Empfänger nur auf demselben physischen Kern ausgeführt werden. Mit folgenden Befehlen kann dies erreicht werden. [Fre14]

```
1 taskset 0x1 ./sender       // execute sender on virtual core 0
2 taskset 0x1 ./receiver     // execute receiver on virtual core 0
```

Besitzt das System einen Unified-Cache, der inclusive zum L1-Cache ist, so können Sender und Empfänger auch auf verschiedenen physischen Kernen ausgeführt werden. Ein Unified-Cache ist ein Cache, der von allen Kernen geteilt wird. Wenn der Unified-Cache inclusive zum L1-Cache ist, so werden alle Cache-Zeilen von jedem L1-Cache ebenfalls in den Unified-Cache geladen. Werden Sender und Empfänger auf verschiedenen physischen Kernen, also Cross-Core, ausgeführt, dann wird über den Unified-Cache kommuniziert. Ist der Unified-Cache nicht inclusive zum L1, so ist es dennoch möglich eine Cross-Core Kommunikation aufzubauen. Lipp et al. stellten 2016 vor wie durch das Ausnutzen von Cache Coherence Protocols dennoch Cache-Zugriffe über Cross-Core wahrgenommen werden können [LGS+16]. Die in dieser Bachelorarbeit betrachteten Prozessoren ARM Cortex-A35 und ARM Cortex-A72 besitzen jedoch einen unified Last-Level-Cache Mit den folgenden Befehlen werden Sender und Empfänger Cross-Core ausgeführt.

```
1 taskset 0x1 ./sender       // execute sender on virtual core 0
2 taskset 0x8 ./receiver     // execute receiver on virtual core 3
```

Dies setzt voraus, dass die virtuelle Kerne 0 und 3 nicht auf demselben physischen Kern liegen.

Ein Cache-Zugriff auf den L1-Cache hat nach Spezifikation eine geringere Latenz als ein Zugriff auf den L2- oder L3-Cache. Die Dokumentation über die Testsysteme ARM Cortex-A35 und ARM Cortex-A72 beinhalten keine genauen Zugriffszeiten der Caches. Zu einer ähnlichen CPU, dem ARM Cortex-A53, wurden Benchmarks erstellt. Die Cache-Zugriffszeiten des ARM Cortex-A53 könnten vergleichbar mit denen der Testsysteme sein. Ein L1-Daten-Cache Zugriff dauert auf dem Cortex-A53 3 CPU Zyklen, während ein L2 Daten-Cache 15 CPU Zyklen benötigt. Bei einem Datenzugriff aus dem Hauptspeicher werden zusätzlich zu der L2 Zugriffszeit nach Dokumentation weitere 128 Nanosekunden benötigt [LZM20]. Durch die, im Vergleich zum L2 Cache, kürzere Latenz wird erwartet, dass in einem verdeckten Kanal, der im Same-Core ausgeführt wird, eine höhere Datenrate erreicht wird als mit Cross-Core.

2.9 Fehler bei der Bitübertragung

Beim Übertragen von Bits über einen cachebasierten Seitenkanal können verschiedene Fehler auftreten. Im Folgenden werden diese vorgestellt und erklärt [MWS+17].

- **Substitution-Errors**: Diese Art von Fehler kommt vor, wenn einzelne Bits geflippt sind.Bitflips können zum Beispiel hervorgerufen werden, wenn die benutzte Cache-Zeile von anderen Prozessen verdrängt wird oder ein Bit falsch interpretiert wird.

- **Insertion-Errors**: Ein Insertion-Error tritt auf, wenn ein Bit in den Bitstream eingefügt wird. Dies passiert zum Beispiel, wenn der Scheduler den Sender-Prozess anhält und der Empfänger weiter Bits empfängt.
- **Deletion-Errors**: Ein Deletion-Error beschreibt ein fehlendes Bit im Bitstream. Dies kann ähnlich wie beim Insertion-Error durch den Scheduler hervorgerufen werden. Wenn der Empfänger-Prozess in der Übertragung pausiert wird und der Sender weiterhin sendet, dann kann der Empfänger ein oder mehrere Bits verpassen, wodurch fehlende Bits im Bitstream entstehen.

Zur Erkennung der Bitfehler müssen abhängig von dem Fehlertypen verschiedene Methoden angewendet werden. In Abschnitt 2.10 werden diese vorgestellt und auch erklärt für welche Art von Fehlertypen jede Methode jeweils geeignet ist.

2.10 Methoden zu der Bewertung von Kanälen

Ein Ziel dieser Bachelorarbeit ist es einen verdeckten Kanal zu evaluieren. Dies kann mit den folgenden gelisteten Methoden erreicht werden. Mit der Konfusionsmatrix und der Wasserstein-Distanz können Fehlertypen erkannt werden. Mit dem Satz von Shannon kann die effektive Kapazität eines Kanals bestimmt werden, um den Kanal anschließend aussagekräftig mit anderen Kanälen vergleichen zu können.

2.10.1 Konfusionsmatrix

Eine Konfusionsmatrix beschreibt wie korrekt binäre Daten klassifiziert werden. Im Fall eines Bitstreams beschreibt die Konfusionsmatrix wie die Bits 0 und 1 jeweils von dem Empfänger erkannt wurden. Dabei wird zwischen den vier folgenden Punkten unterschieden. [Ste20]

- **True Positive** (TP): Eine 1 wird korrekt als eine 1 erkannt.
- **False Positive** (FP): Eine 1 wird fälschlicherweise als eine 0 erkannt.
- **True Negative** (TN): Eine 0 wird korrekt als eine 0 erkannt.
- **False Negative** (FN: Eine 0 wird fälschlicherweise als eine 1 erkannt.

Die Genauigkeit kann wie folgt berechnet werden [Ste20]:

$$\frac{\#TP + \#TN}{\#TP + \#TN + \#FP + \#FN} \tag{2.1}$$

Basierend auf der Genauigkeit, kann die Fehlerrate bestimmt werden.

$$\text{Fehlerrate} = 1 - \text{Genauigkeit} \tag{2.2}$$

Mithilfe der Konfusionsmatrix können Substitution-Errors entdeckt werden, indem diese als FP oder als FN klassifiziert werden. Insertion-Errors und Deletion-Errors können jedoch zu einer Verschiebung des Bitstreams führen, wodurch jedes folgende Bit nicht mit dem originalen Bit verglichen wird. Bei einer alternierenden Bit Reihe 1010...10 würde ein Insertion oder Deletion-Error dafür sorgen, dass jedes darauf folgende Bit als Substitution-Error klassifiziert wird.

2.10.2 Wasserstein-Distanz

Da die Konfusionsmatrix so anfällig für Insertion- oder Deletion-Errors ist, wird die Wasserstein-Distanz als zusätzliches Hilfsmittel hinzugezogen. Im Vergleich zu der Konfusionsmatrix, die die vertikale Distanz berechnet, berechnet die Wasserstein-Distanz die horizontale Verschiebung. In Abbildung 8 ist visualisiert wie die Wasserstein-Distanz im Vergleich zu einer vertikalen Distanz berechnet wird. Bei einer alternierenden Bitreihe `1010...10`, die von dem Empfänger als `0101...01` interpretiert wird, beträgt die Genauigkeit 0 und die Fehlerrate ist 100%. Da die Bitreihen nur horizontal verschoben sind, berechnet die Wasserstein-Distanz jedoch 0.

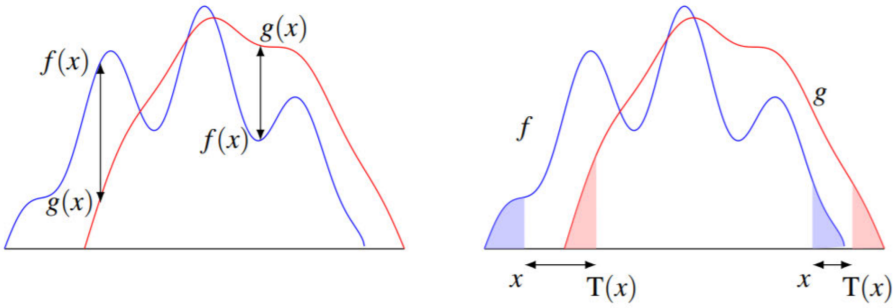

Abbildung 8: *Vertikale Distanz (links) und horizontale Distanz (rechts) [San15]*

Mithilfe der Wasserstein-Distanz kann überprüft werden, ob ein empfangener Bitstream Insertion- und Deletion-Errors enthalten kann. Dafür muss die erwartete Wasserstein-Distanz bestimmt werden. Diese wird berechnet, indem dem gesendeten Bitstream normalverteilte Substitution-Errors künstlich hinzugefügt werden. Die Anzahl der Substitution-Errors basiert auf der Fehlerrate des Kanals. Die Wasserstein-Distanz des ursprüngliche Bitstreams und des bearbeiteten Bitstreams ist dann die erwartete Wasserstein-Distanz. Wird die Wasserstein-Distanz von dem gesendeten Bitstream und dem empfangenen Bitstream bestimmt, so kann der Wert mit der erwarteten Wasserstein-Distanz verglichen werden. Ist die erwartete Wasserstein-Distanz höher, so sind Insertion- und Deletion-Errors aufgetreten. Ansonsten sind die Fehler unabhängig voneinander. Diese Methode kann für den Satz von Shannon in Abschnitt 2.10.3 benutzt werden, da eine Vorbedingung des Satzes von Shannon ist, dass jeder Fehler unabhängig voneinander ist.

2.10.3 Kanalkapazität nach Shannon

Mit dem Satz von Shannon kann die Kapazität eines Kanals berechnet werden. Genauer beschreibt der Satz von Shannon die Mindestlänge eines zuverlässigen fehlerkorrigierenden Codes abhängig von der Fehlerrate und Länge der Nachricht. Die mathematische Definition sieht wie folgt aus [Kle04]:

Lemma 1: *Sei M die Anzahl der übertragenden Bits und p die unabhängige Wahrscheinlichkeit, dass ein Bit durch einen Substitution-Error falsch übertragen wird. Zusätzlich sei $H(p)$ die binäre Entropiefunktion $H(p) = -p * \log_2 * p - (1-p) * \log_2 * (1-p)$. Dann folgt daraus $N = M + N * H(p) = \frac{M}{1-H(p)}$, wobei N die minimale Länge eines Codes ist, der mit einer Wahrscheinlichkeit gegen 1 wiederhergestellt werden kann.*

Die Effizienz eines Kanals ist definiert durch $\frac{M}{N}$ also die Länge der Nachricht in Bits geteilt durch die Länge des Codes in Bits. Mit dieser Gleichung ist die Effizienz abhängig von einer bestimmten Codelänge. Mit dem obigen Lemma lässt sich Folgendes bilden [Hei20]:

$$\text{Effizienz} = \frac{M}{N} = \frac{M}{\frac{M}{1-H(p)}} = 1 - H(p) \tag{2.3}$$

Wodurch die Effizienz nur noch von der Fehlerrate abhängig ist. Basierend auf der Effizienz und der Bandbreite f kann nun die effektive Kapazität C wie folgt berechnet werden [Hei20].

$$\text{Kapazität} = \text{Effizienz} * f = (1 - H(p)) * f \tag{2.4}$$

Mithilfe der berechneten Kanalkapazität lassen sich Kanäle mit unterschiedlicher Frequenz und Fehlerrate miteinander vergleichen.

3 Implementierung

In diesem Kapitel werden die nötigen Implementierungen für das Bauen eines verdeckten Kanals mithilfe von `Flush+Reload` und `Flush+Flush` vorgestellt. In Abschnitt 3.1 wird das Konzept und die Implementierung von Barriers vorgestellt. Das Löschen von Cache-Zeilen unter Intel x86 und ARM wird in Abschnitt 3.2 beschrieben. Methoden zur akkuraten Zeitmessung unter Intel x86 und ARM werden in Abschnitt 3.3 dargestellt. In Abschnitt 3.4 werden verschiedene Arten von geteiltem Speicher vorgestellt. Abschnitt 3.5 beschreibt wie aus Messwerten ein Schwellwert berechnet werden kann. In Abschnitt 3.6 wird erklärt wie aus einem Seitenkanal ein verdeckter Kanal erstellt werden kann. Wie Empfänger und Sender-Prozess synchronisiert werden können wird in Abschnitt 3.7 vorgestellt. Der letzte Abschnitt (3.8) beschreibt das Ethernet Protokoll.

3.1 Memory Barrier

Für cachebasierte Seitenkanalangriffe ist es wichtig, dass die Instruktionen in derselben Reihenfolge ausgeführt werden, wie sie im Programmcode stehen. Wenn eine Cache-Zeile gelöscht und anschließend neu geladen wird, dann führt eine Vertauschung dieser Instruktionen zu falschen Ergebnissen der Zeitmessung. Bei sogenannten Out-of-order-Ausführungen kann der Compiler während der Kompilierzeit die Reihenfolge der Instruktionen ändern. Die Instruktionsreihenfolge wird so angepasst, dass der Code effizienter ist, aber immer noch dasselbe Ergebnis ausgibt. Jedoch werden bei cachebasierten Seitenkanalangriffen die zeitineffzienten Datenzugriffe und Instruktionen gemessen und Out-of-order-Ausführungen würden das Ergebnis verfälschen. Während der Laufzeit können durch spekulative Ausführung weiterhin Optimierungen durchgeführt werden. Bei spekulativer Ausführung bearbeitet die CPU Daten und Instruktionen, bevor diese überhaupt im Programm benötigt werden. So können zum Beispiel Daten in den Cache geladen werden während noch andere Instruktionen in der Ausführung sind. Dies wäre das zwar ein Zeitgewinn, aber bei cachebasierten Seitenkanalangriffen würde dies die Zeitmessung verfälschen.

Um die Neuanordnung von Instruktionen zu verhindern, gibt es auf Intel x86 Prozessoren die CPU-Instruktionen `sfence`, `lfence` und `mfence` [Int16, 11.4.4.3]. Diese stellen sicher, dass alle im Code vorstehenden Instruktionen bereits ausgeführt worden sind, bevor eine der folgenden Instruktion ausgeführt wird. Diese Technik wird auch Serialisierung genannt. `sfence` (Store Fence) ordnet dabei nur schreibende Speicherzugriffe [Int16, 10.4.6.4]. `lfence` ordnet nur lesende Speicherzugriffe. `mfence` (Memory Fence) ordnet sowohl lesende als auch schreibende Speicherzugriffe. [Int16, 11.4.4.3]

Auf ARMV8-A Prozessoren gibt es `sfence`, `lfence` und `mfence` nicht, aber die Assembler Befehle `ISB`, `DMB` und `DSB` [ARM17], die im Folgenden erläutert werden.

- Instruction Synchronization Barrier (ISB): Der Befehl ISB serialisiert Instruktionen. Nach Ausführung von ISB kann keine weitere Instruktion in die Instruction Pipeline geladen werden und erst wenn diese leer ist, dürfen weitere Instruktionen geladen werden. [ARM17]
- Data Memory Barrier (DMB): Der Befehl DMB serialisiert Datenzugriffe. Nach Ausführung von DMB können keine weitere lesende und schreibende Speicherzugriffe ausgeführt werden bis alle vorherigen durchgeführt worden sind. Instruktionen ohne Speicherzugriffe können trotzdem auch weiterhin ausgeführt werden. [ARM17]
- Data Synchronization Barrier (DSB): Der Befehl DSB ist eine striktere Version von DMB. DSB serialisiert genau wie DMB Datenzugriffe, aber DSB blockiert alle folgende Instruktion bis alle vorherigen Datenzugriffe ausgeführt worden sind. [ARM17]

Für das Messen der Cachezugriffszeit bei einem cachebasierten Seitenkanalangriff müssen Barriers vor dem Beginn der Zeitmessung und vor dem Ende der Zeitmessung stehen. Nur so kann garantiert werden, dass alle Instruktionen innerhalb der Zeitmessung komplett ausgeführt werden und keine zusätzliche Instruktion in der Zeitmessung dazwischen kommt.

```
1 asm volatile ("DSB SY");
2 asm volatile ("ISB");
```

Als Barrier reicht das obige Code-Beispiel. In Zeile 1 wird eine DSB mit dem Parameter SY aufgerufen. Der Parameter SY bedeutet Full System Operation, also ist jede System Operation inklusive. Statt SY kann sich eine DSB auch nur auf Load oder Store Operationen beziehen. Im Rahmen der Zeitmessung sollen jedoch alle System-Operationen durch Barriers gestoppt werden. In Zeile 2 wird eine ISB ausgeführt, die dafür sorgt, dass Instruktionen serialisiert werden. für die Implementierung von Flush+Reload und Flush+Flush ist sowohl DSB alös auch ISB notwendig. Für die Serialisierung von Cache-Reloads wird DSB benötigt und ISB für die Serialisierung von Flush-Instruktionen.

3.2 Löschung von Cache-Zeilen

Für cachebasierte Seitenkanalangriffe kann es notwendig sein, bestimmte Cache-Zeilen zu löschen. Je nach Prozessor werden bestimmte Flush-Instruktionen unterstützt. In diesem Unterkapitel werden die Flush-Instruktionen auf Intel x86 und ARM Prozessoren vorgestellt.

3.2.1 Flush-Instruktionen auf Intel x86

Auf Intel x86 Prozessoren wird die unprivilegierte Instruktion clflush unterstützt. clflush invalidiert die Cache-Zeile, die den Wert der übergebenen virtuellen Adresse beinhaltet, auf jedem Cache-Level. [Int16, Vol. 1:11.4.4.1] Wenn einer der Cache-Zeilen als dirty markiert ist, dann wird der Inhalt durch einen write back zurück in den Speicher geschrieben, bevor die Cache-Zeile invalidiert wird. [Int16, 3.2] Zudem gibt es noch die unprivilegierte Instruktion clflushopt, welche eine optimierte Version von clflush ist, da sie mehrfach parallel ausgeführt werden kann. Die Vorraussetzung ist, dass die übergebenen virtuellen Adressen unkorreliert sind. [Int16, 3.2] Beide Instruktionen sind für cachebasierte Seitenkanalangriffe geeignet, da sie unprivilegiert sind.

3.2.2 Flush-Instruktionen auf ARM

Auf ARMv8 Prozessoren können Flush-Instruktionen Cache-Zeilen invalidieren und reinigen, indem die dazugehörige virtuelle Adresse oder der Cache-Way und das Cache-Set übergegeben werden.

Invalidierung

Bei Invalidierung einer Cache-Zeile wird das Valid Bit auf 0 gesetzt [ARM15, 11.5]. Falls jedoch der Inhalt durch das Dirty Bit als `dirty` markiert ist, wird der Inhalt durch `write back` nicht in den Speicher geladen. Die Änderung der Daten verfallen somit. [Lip16b]

Reinigung

Beim Reinigen einer Cache-Zeile wird bei einem gesetzten Dirty Bit der Inhalt durch `write back` in allen höheren Cache-Level und im Hauptspeicher aktualisiert. Das Dirty Bit wird daraufhin auf 0 gesetzt. Ist das Dirty Bit bereits auf 0 gesetzt, so muss nichts aktualisiert werden. In beiden Fällen bleiben die Daten im Cache. Diese Instruktion ist nur für Datencaches möglich. [ARM15, 11.5]

Wenn eine Cache-Zeile anhand ihrer virtuellen Adresse invalidiert oder gereinigt wird, dann wird zwischen zwei Punkten unterschieden.

- Der Point of Coherency beschreibt den Punkt, an dem alle CPU Kerne und DMAs denselben Inhalt der gewählten virtuellen Adresse sehen können. Dies ist in den meisten Fällen im Hauptspeicher [ARM15, 11.4]. Bei dem ARM Cortex-A72 ist dies ebenfalls im Hauptspeicher.[ARM16, 4.4.8]
- Der Point of Unification beschreibt den Punkt, an dem der Instruktions- und Datencache sowie `TLB` desselben CPU Kernes dieselbe Kopie der gewählten virtuellen Adresse sehen können. Wenn der Prozessor einen Unified-Cache hat, dann ist der Lowest-Level Unified-Cache der Point of Unification. Ansonsten ist der Point of Unification im Hauptspeicher. Der ARM Cortex-A72 besitzt einen unified L2-Cache und dieser ist somit der Point of Unification. [ARM15, 11.4]

Da ARMv8 Prozessoren eine Modified-Harvard-Architektur besitzen, werden Daten und Instruktionen getrennt gespeichert. Es muss also zu jeder Flush-Instruktion der Cache spezifiziert werden. Der Datencache wird hier mit `DC` und der Instruktionscache mit `IC` gekennzeichnet (siehe Abbildung 9).

Cache	Operation	Description
DC	CIVAC	Clean and Invalidate by Virtual Address to Point of Coherency
	CVAC	Clean by Virtual Address to Point of Coherency
	CVAU	Clean by Virtual Address to Point of Unification
	IVAC	Invalidate by Virtual Address, to Point of Coherency
IC	IVAU	Invalidate by Virtual Address to Point of Unification

Abbildung 9: *Flush-Instruktionen der ARMv8 Architektur [ARM15, 11.5]*

In Abbildung 9 werden alle relevanten unprivilegierten Flush-Instruktionen, die eine virtuelle Adresse als Eingabe nehmen, dargestellt. Im Rahmen dieser Bachelorarbeit wird ausschließlich `DC CIVAC` verwendet.

3.3 Akkurate Zeitmessung

Für das Ausführen von cachebasierten Seitenkanalangriffen ist eine sehr genaue Zeitmessung erforderlich. Die Zeiten, die miteinander verglichen werden, unterscheiden sich besonders bei `Flush+Flush` nur um wenige CPU Zyklen. Zudem gilt als weiteres Kriterium, dass die Zeitmessung unprivilegiert werden kann.

3.3.1 Zeitmessung auf Intel x86

Intel x86 Prozessoren besitzen die unprivilegierte Instruktion `rdtsc`, die den exakten CPU Zyklus ausgibt. [Int16, Vol. 2B: 4.3] `rdtsc` wird für die meisten cachebasierten Seitenkanalangriffe auf Intel x86 verwendet, da die Genauigkeit auf einzelne CPU Zyklen rückzuführen ist.

3.3.2 Zeitmessung auf ARM

ARMv8 Prozessoren besitzen ein Register `PMCCNTR`, welches die genauen Zyklen des Prozessors ausgibt. Dies ist ähnlich zur x86 Instruktion `rdtsc` [ARM16, 11.3]. Jedoch ist der Zugriff auf das Register nur mit Root Rechten möglich.

Als Alternative bietet sich die unprivilegiert POSIX Funktion `clock_gettime()` an. Die Genauigkeit von `clock_gettime()` liegt jedoch nur im Millisekunden- bis Nanosekundenbereich [Fre01], was deutlich ungenauer als `rdtsc` und `PMCCNTR` ist.

Eine weitere Funktion auf ARMv8 Prozessoren ist der unprivilegierte Syscall `perf_event_open`, mit dem Informationen über die CPU ausgegeben werden können. Einer dieser Informationen ist der genaue CPU Zyklus (`PERF_COUNT_HW_CPU_CYCLES`) [Fre20c]. Die Genauigkeit liegt somit, ähnlich wie bei `rdtsc` und `PMCCNTR`, unter dem Nanosekundenbereich. Ein Nachteil ist jedoch, dass der Syscall für einen Overhead sorgt, was die genaue Zeitmessung erschwert. [LGS+16]

3.3.3 Implementierung der Zeitmessung

Im Rahmen dieser Bachelorarbeit wird die akkurate Zeitmessung mit `perf_event_open` implementiert, da `perf_event_open` unprivilegiert ist und auf einen CPU Zyklus akkurat ist. Da der Overhead bei jeder Zeitmessung mit `perf_event_open` ungefähr gleich ist, ist die Differenz zwischen unterschiedlichen Messungen mit Overhead dieselbe wie ohne Overhead.

Zu Beginn müssen die nötigen Einstellungen in einen gegebenen Struct gesetzt werden. Da können die Einstellungen im Beispiel auf der Linux Manual Page von `perf_event_open` übernommen werden [Fre20c]. Wichtig ist, dass `config` auf `PERF_COUNT_HW_CPU_CYCLES` gesetzt ist, damit die CPU Zyklen ausgegeben werden.

```
1 asm volatile ("DSB SY");              // data sync barrier
2 asm volatile ("ISB");                 // instruction sync barrier
3 ioctl(fd, PERF_EVENT_IOC_RESET, 0);   // reset timer
4 ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);  // start timer
```

```
5 // code
6 asm volatile ("DSB SY");               // data sync barrier
7 asm volatile ("ISB");                  // instruction sync barrier
8 ioctl(fd, PERF_EVENT_IOC_DISABLE, 0); // stop timer
9 read(fd, &count, sizeof(long long));  // load measured time
```

Damit die Reihenfolge der Instruktionen bei Laufzeit nicht geändert wird, werden in Zeile 1-2 und 6-7 Data Synchronization Barriers und Instruction Synchronization Barriers gesetzt (mehr zu Barriers in Abschnitt 3.1). So kann garantiert werden, dass jede Instruktion vor der Zeitmessung abgearbeitet ist. `SY` steht im Zusammenhang mit `DSB` für Full System Operation. Es werden also Load und Store Operationen abgewartet. In Zeile 3-4 wird der CPU Zyklus auf 0 gesetzt und der CPU Zyklus Zähler wird gestartet. In Zeile 5 wird der Code ausgeführt, von dem die Zeit gemessen werden soll. In Zeile 8 wird der CPU Zyklus Zähler gestoppt und in Zeile 9 wird die gemessene Zyklenanzahl auf die Variable `count` geladen. [Hap21]

3.4 Geteilter Speicher

`Flush+Flush` und `Flush+Reload` benötigen miteinander geteilten Speicher, damit sich zwei Prozesse garantiert eine oder mehrere Cache-Zeile teilen. Als Kriterium gilt, dass beide Prozesse nur Leserechte auf den geteilten Speicher haben, da dies in einem Angriffsszenario praktikabel ist [YF14] [GMWM16]. Im Folgenden werden drei Möglichkeiten vorgestellt.

Mit der Funktion `mmap` lässt sich ein Speicherbereich reservieren, der als geteilter Speicherbereich klassifiziert werden kann [Fre20b]. Jedoch erfordert dies Root Rechte, welche in einem Angriffsszenario nicht gegeben sind. Als weitere Möglichkeit für geteilten Speicher ist es eine speziell angefertigte Bibliothek zu erstellen. Jedoch ist auch dies in einem Angriffsszenario nicht immer möglich. Statt eine Bibliothek anzufertigen, kann auch eine bereits existierende Bibliothek benutzt werden. So kann zum Beispiel die C-Standard-Bibliothek `libc` verwendet werden. Der Nachteil beim Benutzen einer bereits existierender Bibliothek ist jedoch, dass andere Prozesse auf diese zugreifen können und somit unvorhergesehen bestimmte Daten in den Cache ablegen. Dennoch ist das Benutzen einer bereits existierender Bibliothek die einzig praktikable Lösung in einem Angriffsszenario. Für Testzwecke und Messungen eignen sich jedoch selbst angefertigte geteilte Speicher mehr, da andere Prozesse nicht auf diesen Speicherbereich zugreifen werden.

Mit den oben genannten Methoden können mehrere Prozesse auf denselben Speicherbereich zugreifen. Jeder Prozess hat einen eigenen virtuellen Adressraum für den geteilten Speicherbereich, die jedoch alle auf denselben physischen Adressraum mappen. Wenn die Daten über den physischen Tag in den Cache geladen werden (mehr zu Cache-Adressierung in Abschnitt 2.2.3), dann lädt jeder Prozess dieselben physischen Adressen in dieselbe Cache-Zeile. Jeder Prozess teilt sich somit die Cache-Zeilen für den geteilten Speicher. Zudem hat jeder physische CPU Kern einen eigenen L1-Cache. Für einen Same-Core Angriff (mehr zu Same-Core in Abschnitt 2.8) müssen die Prozesse auf demselben physischen Kern ausgeführt werden. Mit `taskset` kann beim Ausführen bestimmt werden auf welchem virtuellen Kern und somit auf welchen physischen Kern der Prozess ausgeführt wird [Fre14]. Für `Flush+Flush` und `Flush+Reload` muss der Cache also ein `PIPT`-Cache oder `VIPT`-Cache sein [Lip16b] und die kommunizierenden Prozesse müssen auf demselben Kern ausgeführt werden oder die Cache-Eigenschaften für einen Cross-Core Angriff müssen erfüllt sein (mehr zu Cross-Core in Abschnitt 2.8). In den benutzen Testumgebungen, dem ARM Cortex-A72 und Arm Cortex-A35, sind die L1-Datencaches sowie der geteilte L2-Cache `PIPT`-Caches. Des Weiteren sind

die Last-Level-Caches unified und inclusive, also ist Cross-Core ebenfalls möglich. [ARM19, 6.1 & 7.1] [ARM16, 6.1 & 7.1].

3.5 Kalibrierung des Schwellwertes

Für die Klassifizierung von Cache Hits und Cache Misses wird ein Schwellwert berechnet. Mit dem Schwellwert können gemessene Zeiten als Hit oder Miss interpretiert werden. Dieser Schwellwert ist abhängig von dem System und kann sich bei jeder erneuten Ausführung ändern. Er muss also zu Beginn eines Verbindungsaufbaus berechnet werden.

Bei Ausführung des Empfängers werden mehrere Cache Hits und Cache Misses simuliert und Zugriffszeiten gemessen. Mit der Standardabweichung σ für jeweils Hits und Misses werden Ausreißer in einem Streuintervall von 10σ entfernt. Für die bereinigten Daten werden mit der Annahme, dass Hits und Misses normalverteilt sind, für jeweils Hits und Misses Durchschnitt μ und Standardabweichung σ berechnet. Daraus folgt die folgende Dichtefunktion der Normalverteilung [Hei20].

$$f(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} * e^{-\frac{(x-\mu)^2}{2\sigma^2}} \text{ mit } -\infty < x < \infty \tag{3.1}$$

Werden die gleichgesetzten Dichtefunktionen von Hit und Miss nach x aufgelöst, so bildet x den Schnittpunkt beider Funktionen. Dieser Schnittpunkt kann dann als Schwellwert benutzt werden [Hei20].

$$f(x|\mu_{miss},\sigma^2_{miss}) = f(x|\mu_{hit},\sigma^2_{hit}) \tag{3.2}$$

Mit Beispielwerten $\mu_{miss} = 198.74$ $\mu_{hit} = 242.75$ $\sigma_{miss} = 5.65$ $\sigma_{hit} = 5.17$ folgt:

$$f(x|198.74,5.65^2) = f(x|242.75,5.17^2) \Rightarrow x \approx 222 \vee 725 \tag{3.3}$$

Da die Gleichung zwei Schnittpunkte berechnet, muss als letzter Schritt entschieden werden, welcher Schnittpunkt sich besser als Schwellwert eignet. Dafür wird für jeden Schnittpunkt die Genauigkeit berechnet (siehe Abschnitt 2.10.1). Schnittpunkt 725 ergibt eine Genauigkeit von 49,9% und Schnittpunkt 222 eine Genauigkeit von 99.8% . Somit erweist sich 222 als besserer Schwellwert. In Abbildung 10 wird das Beispiel mit berechnetem Schwellwert veranschaulicht.

Mit berechnetem Schwellwert können neue Messungen nach Hit und Miss klassifiziert werden. Bei `Flush+Flush` werden alle Werte unter dem Schwellwert als Miss und alle Werte gleich oder über dem Schwellwert als Hit interpretiert. Bei `Flush+Reload` ist die Klassifizierung invers. Bei `Flush+Reload` werden alle Werte unter oder gleich dem Schwellwert als Hit und Werte über dem Schwellwert als Miss eingeordnet.

3.6 Vom Seitenkanal zum verdeckten Kanal

Im Rahmen der Bachelorarbeit soll mit `Flush+Flush` und `Flush+Reload` ein verdeckter Kanal gebaut werden. Verdeckte Kanäle sind Kanäle, die ursprünglich nicht zur Kommunikation gedacht waren

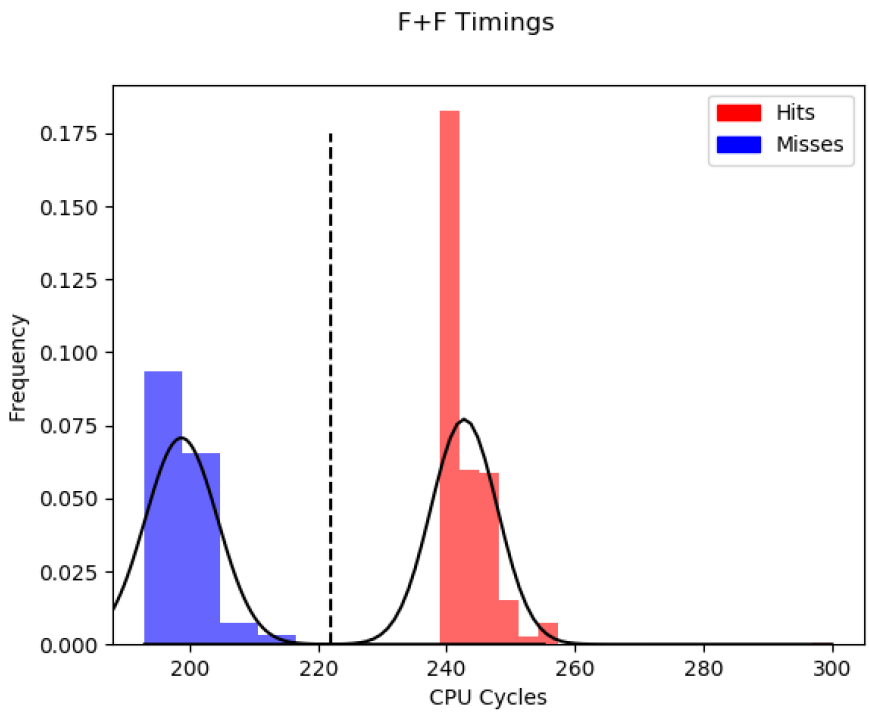

Abbildung 10: *Flush+Flush auf einem ARM Cortex-A72 im selben Prozess und Thread*

und somit von dem System nur schwer oder gar nicht erkennbar sind. Mithilfe von Seitenkanalangriffen können aus einem Seitenkanal Informationen abgeleitet werden, die ursprünglich nicht zu entnehmen sein sollten. Die Seitenkanaleffekte, die in dieser Bachelorarbeit ausgenutzt werden, sind die unterschiedliche Zugriffszeit von Daten abhängig davon, ob diese im Cache vorliegen oder nicht, sowie die unterschiedlich lange Ausführungszeit der Flush-Instruktion.

In dem gewählten Szenario gibt es einen Sender- und Empfänger-Prozess. Der Sender legt Daten in den Cache ab und der Empfänger überprüft, ob diese im Cache liegen oder nicht. Bei einem Cache Hit wird dies als eine 1 interpretiert und bei einem Cache Miss als eine 0.

3.6.1 Senden von Bits

Für Sender und Empfänger muss vorher festgelegt werden, welche Daten im geteilten Speicher als Medium benutzt werden sollen. Diese werden, da sie im geteilten Speicher liegen über den PIPT-Cache von beiden Prozessen in dieselbe Cache-Zeile geladen. Im Folgenden wird eine Funktion `function` benutzt. Diese befindet sich in einem geteilten Speicher und ist das Medium, über das Sender und Empfänger kommunizieren. Der Empfänger überprüft nach einem festgelegten Intervall die Cache-Zeile für `function` und ob sie geladen wurde oder nicht. Um eine 1 zu senden, lädt der Sender mit folgendem Befehl die Funktion `function` in den Datencache. [Hap21]

```
1 store = *((uint64_t*)&function);        // load first 64 bit of function to store
```

Damit dieser Code korrekt ausgeführt wird, muss Folgendes beachtet werden. Wird eine Funktion aus einer importierten Bibliothek verwendet, so muss sie beim ersten Aufruf erst über die Procedure Linkage Table (PLT) dynamisch eingebunden werden. Die PLT gibt den Programmfluss weiter an die Global Offset Table (GOT). Da der Eintrag für die aufgerufene Funktion in der GOT leer ist, ruft die PLT den Resolver auf, der die Funktion dynamisch einbindet. Daraufhin wird die Adresse der

Funktion in die GOT eingetragen. Jeder weiterer Aufruf der Funktion wird über den Eintrag in die GOT über einen relativen Sprung direkt in die Funktion in der geteilten Bibliothek weitergeleitet. [Ben11]

Damit die korrekte Zeile in den Cache geladen wird, muss `&function` die virtuelle Adresse von `function` in der geteilten Bibliothek sein. `function` muss vorher mindestens einmal aufgerufen werden, damit die korrekte virtuelle Adresse geladen wird. Ist die Vorbedingung erfüllt, werden die ersten 64 Bit an der virtuellen Adresse von `function` auf eine beliebige Variable (hier `store`) zugewiesen. Da `function` nicht aufgerufen wird, sondern wie Daten behandelt wird, werden die 64 Bit in den Datencache geladen.

Um eine 0 zu senden, muss die Cache-Zeile leer sein. Dies wird sichergestellt, indem der Sender eine Flush-Instruktion ausführt (mehr zu Flush-Instruktion in Abschnitt 3.2). Um eine Cache-Zeile im Datencache zu reinigen und zu invalidieren, wird `DC CIVAC` wie folgt ausgeführt. [Hap21]

```
1 asm volatile (" DC CIVAC, %0;":: "r" (&function)); // flush &function from data cache
```

Als Eingabewert wird hier `&function`, die virtuelle Adresse von `function`, die aus dem Cache entfernt werden soll, übergeben.

Wenn während einer Iteration die Adresse nur einmal in den Cache geladen wird, ist es möglich, dass die Adresse innerhalb dieser Iteration aus dem Cache verdrängt wird (mehr zu Cache-Verdrängung in Abschnitt 2.2.2). Genauso kann eine bereits gelöschte Cache-Zeile erneut geladen werden. Damit der Zustand der Cache-Zeile bis zur Messung erhalten bleibt, kann der Sender innerhalb einer Iteration durchgehend die Cache-Zeile laden beziehungsweise flushen.

3.6.2 Empfangen von Bits

Bei dem Empfangen der Bits unterscheidet sich die Methodik abhängig davon, ob `Flush+Flush` oder `Flush+Reload` benutzt wird. Daher wird die Implementierung beider Methoden vorgestellt.

Flush+Flush

Bei `Flush+Flush` wird ausgenutzt, dass die Flush-Instruktion, abhängig von dem Inhalt der Cache-Zeile, unterschiedlich lange Ausführungszeiten hat. Ist die Ausführungszeit länger, so befand sich die übergebene virtuelle Adresse im Cache. In Abbildung 7 werden unterschiedliche Ausführungszeiten von `Flush` auf einem ARM Cortex-A72 dargestellt. Eine durchschnittliche Ausführungszeit bei einem Cache Miss liegt bei 219 CPU Zyklen während bei einem Cache Hit die durchschnittliche Ausführungszeit bei 248 CPU Zyklen liegt. Eine Implementierung von `Flush+Flush` sieht wie folgt aus [Hap21].

```
1  asm volatile ("DSB SY");                          // data sync barrier
2  asm volatile ("ISB");                             // instruction sync barrier
3  ioctl(fd, PERF_EVENT_IOC_RESET, 0);               // reset timer
4  ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);              // start timer
5  asm volatile (" DC CIVAC, %0;":: "r" (&function)); // flush from data cache
6  asm volatile ("DSB SY");                          // data sync barrier
7  asm volatile ("ISB");                             // instruction sync barrier
8  ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);             // stop timer
9  read(fd, &count, sizeof(long long));              // load measured time
10 if(count>=threshold){hit = 1;}                    // classify if hit
11 else{hit = 0;}                                    // or miss
```

Damit in Zeile 3-4 die Zeitmessung gestartet werden kann (mehr zu Zeitmessung in Abschnitt 3.3), muss vorher eine Barrier in Zeile 1-2 ausgeführt werden (mehr zu Barriers in 3.1). Die Barrier sorgt dafür, dass keine vorherigen Instruktionen während der Laufzeit aus Optimierungsgründen nach Zeile 4 ausgeführt werden und die Zeitmessung verfälschen. In Zeile 5 wird die Flush-Instruktion für die virtuelle Adresse von Funktion `function` ausgeführt. Wie in Abschnitt 3.6.1 beschrieben muss sichergestellt werden, dass die virtuelle Adresse von `function` in der geteilten Bibliothek benutzt wird und nicht die virtuelle Adresse von `function` in der PLT. Damit das Beenden der Zeitmessung in Zeile 8-9 nicht parallel zu der Flush-Instruktion ausgeführt wird, wird in Zeile 6-7 eine weitere Barrier eingefügt. In Zeile 10 wird die gemessene Zyklusanzahl mit einer vorher berechneten Threshold verglichen (mehr zu Schwellwert-Berechnung in Abschnitt 3.5). Wenn die gemessene Zyklusanzahl höher oder gleich dem Schwellwert ist, dann befand sich die Adresse tatsächlich in der Cache-Zeile und es wird ein Hit ausgegeben und somit eine 1 empfangen. Ansonsten ergibt es einen Miss und das empfangene Bit ist eine 0.

Flush+Reload

Bei `Flush+Reload` wird ausgenutzt, dass ein Datenzugriff schneller ausgeführt wird, wenn die Daten im Cache geladen sind. Ein Codebeispiel zu `Flush+Reload` könnte wie folgt aussehen [Hap21].

```
1  asm volatile ("DSB SY");               // data sync barrier
2  asm volatile ("ISB");                  // instruction sync barrier
3  ioctl(fd, PERF_EVENT_IOC_RESET, 0);    // reset timer
4  ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);   // start timer
5  store = *((uint64_t*)&function);       // load into data cache
6  asm volatile ("DSB SY");               // data sync barrier
7  asm volatile ("ISB");                  // instruction sync barrier
8  ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);  // stop timer
9  read(fd, &count, sizeof(long long));   // load measured time
10 if(count<=threshold){hit = 1;}         // classify if hit
11 else{hit = 0;}                        // or miss
```

Der Aufbau der Implementierung ist dem von `Flush+Flush` sehr ähnlich. Statt dem Flushen bei `Flush+Flush` werden in Zeile 5 die ersten 64 Bit der Funktion `function` zugewiesen. Wenn diese im Cache geladen sind, dann ist die gemessene Zeit kleiner als `threshold` (Zeile 10) und führt somit zu einem Hit und einer empfangenen 1. Wenn die gemessene Zeit höher als `threshold` ist, dann wird eine 0 empfangen.

3.7 Synchronisierung der Prozesse

Damit eine stabile Kommunikation zwischen zwei Prozessen aufgebaut werden kann, müssen die Prozesse zeitlich synchronisiert sein. Um einen Bitstream zu erzeugen, müssen die Bits synchronisiert sein. Aufbauend darauf bedarf es einem Protokoll, um den Bitstream zu interpretieren.

3.7.1 Synchronisierung von Bits

Zur Synchronisierung von Bits werden überlicherweise mit self-clocking Codes wie dem Manchester-Code gearbeitet [MF19, 2.9]. Im Rahmen von cachebasierten verdeckten Kanälen sind beide Prozesse jedoch auf derselben CPU und teilen sich somit auch die Systemzeit. Somit kann eine zuverlässige Synchronisierung gewährleistet sein.

Als wichtiges Kriterium gilt, dass eine fein granulare Zeitmessung möglich ist.. Unter Linux ermöglicht die Funktion clock_nanosleep das Halten eines Prozesses für eine angegebene Zeitspanne oder bis die Systemzeit einen absoluten Punkt erreicht. [Fre20a]

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

- clockid beschreibt die Art der Clock, die verwendet werden soll. CLOCK_REALTIME ist die Systemuhr, welche aber während der Laufzeit dynamisch geändert werden kann. CLOCK_MONOTONIC hingegen beschreibt die vergangene Zeit seit einem willkürlichem Zeitpunkt und kann nicht geändert werden. [Fre20a]
- flags ist entweder 0 oder TIMER_ABSTIME. Mit dem Wert 0 wartet clock_nanosleep um den angegebenen Wert. Bei TIMER_ABSTIME wird bis zu einem angegebenen Zeitpunkt gewartet. Ist dieser bereits vergangen, so führt das Programm direkt fort. [Fre20a]
- request ist der relative beziehungsweise absolute Zeitpunkt bis zu dem gewartet werden soll.
- remain kann im Fall einer Suspendierung die verbliebene Zeit speichern.

Um nun um eine gewünschte Zeit interval zu warten, kann folgender Code implementiert werden [Hap21].

```
1 clock_gettime(CLOCK_MONOTONIC, &time);
2 time.tv_nsec += interval;
3       if(time.tv_nsec > 999999999) // nanoseconds overflow
4       {
5           time.tv_nsec -= 1000000000;
6           time.tv_sec++;
7       }
8       clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &time, NULL);
```

In Zeile 1 wird mit clock_gettime die MONOTONIC Systemzeit in den Struct time geladen. Der Struct besitzt die zwei Fields tv_nsec und tv_sec. In Zeile 2 wird die zu wartende Zeit, also hier das Intervall in Nanosekunden, auf time gesetzt. Da time.tv_nsec nicht größer als 999999999 sein darf, muss der Überlauf auf die Sekundenzahl übertragen werden. Dies wird in Zeile 3-7 behandelt. Wenn die Nanosekunden überlaufen, wird somit die Sekundenzahl erhöht und die Nanosekunden werden um 1000000000, also eine Sekunde, reduziert. In dem letzten Schritt wird clock_nanosleep ausgeführt und hält den Prozess bis die berechnete Zeit erreicht ist.

Wird die Synchronisierung der Bits für das Empfangen und Senden der Bits implementiert, sollte clock_gettime nicht in jedem Schleifendurchlauf aufgerufen werden. Wird clock_gettime in jedem Schleifendurchlauf zu Beginn erneut aufgerufen, so entsteht ein Overhead, wodurch jede Iteration unterschiedlich länger andauert. Daher sollte clock_gettime nur zu Beginn des Programmes ausgeführt werden und in jedem Schleifendurchlauf das Periodendauer auf die Startzeit addiert werden. Zusätzlich zur Eliminierung des Overheads verhindert die Methode Fehleranfälligkeit durch Descheduling. Wird ein Prozess durch den Scheduler descheduled, so wird der Programmfluss vorübergehend pausiert. Nachdem der Prozess fortgesetzt wird, erkennt die Methode, die clock_gettime in jeder Iteration ausführt, nur, ob die zu wartende Zeit für die nächste Schleifeniteration überschritten wurde. Die Methode, die clock_gettime nur einmal aufruft, könnte erkennen, ob die zu wartende Zeit für mehrere Iterationen überschritten wurde. Diese Methode würde zu Substitution-Errors führen bis das Zeitdefizit vom angehaltenen Prozess

eingeholt wurde. Die Methode, die clock_gettime in jeder Iteration ausführt, könnte zu Insertion- und Deletion-Errors führen, wodurch die folgenden Bits verschoben wären.

clock_nanosleep arbeitet nur zuverlässig, wenn die Wartezeit über 2 Millisekunden ist [Saioo, 4]. Ist die Wartezeit unter 2 Millisekunden, so wird eine Busy-Waiting-Schleife eingesetzt, welche sehr unzuverlässig wartet. Dadurch ist bei kleinen Zeitintervallen eine Synchronisation nicht gewährleistet. Stattdessen kann eine selbst implementierte Busy-Waiting-Schleife implementiert werden, welche bei Intervallen unter 2 Millisekunden präziser arbeitet. Statt Zeile 8 von dem obigen Code kann folgender Code benutzt werden.

```
1 while(1)       // Busy Waiting Loop
2 {
3    clock_gettime(CLOCK_MONOTONIC, &towait);    // load current time on towait
4    // check if time elapsed
5    if(towait.tv_sec >= time.tv_sec && towait.tv_nsec >= time.tv_nsec)
6    {
7        break;  // leave loop
8    }
9 }
```

In Zeile 3 wird die aktuelle Zeit mit clock_gettime auf towait geladen. In Zeile 4 wird verglichen, ob die Zeit von time bereits vergangen ist. Wenn dies zutrifft, dann wird in Zeile 7 die While Schleife verlassen. Wenn nicht, dann beginnt der Schleifendurchlauf erneut.

Der Vorteil durch die Implementierung einer Busy-Waiting-Schleife ist, dass, bei Zeitintervallen von unter 2 Millisekunden, die Schleife zuverlässiger verlassen wird als bei einer Implementierung von Nanosleep. Bei Nanosleep ist der Prozess jedoch in einem wartenden Zustand, während der Prozess bei einer Busy-Waiting-Schleife durchgehend Berechnungen durchführt. Bei Systemen mit geringerer Auslastung kann diese Busy-Waiting-Schleife den kompletten physischen CPU Kern, auf dem der Prozess läuft, auslasten. Dadurch werden auch alle anderen Prozesse, die auf dem physischen CPU Kern laufen, beeinträchtigt. Wenn Sender und Empfänger beide auf dem selbem physischen Kern ausgeführt werden, kann die Busy-Waiting-Schleife des Empfängers somit auch die Performance des Senders beeinflussen. Dies kann zur Störung der Kommunikation führen. Für die Implementierung einer Busy-Waiting-Schleife sollte Sender und Empfänger somit auf verschiedenen physischen Kernen ausgeführt werden (mehr dazu in Abschnitt 2.8).

3.7.2 Synchronisierung von Frames

Durch eine Synchronisierung von Bits existiert ein Bitstream zwischen den kommunizierenden Prozessen. Dieser kann von einem Protokoll interpretiert und in Frames eingeteilt werden. Im Rahmen dieser Bachelorarbeit wird das Ethernet Protokoll benutzt. Mit dem Ethernet Protokoll wird definiert wie das Datenpaket zu interpretieren ist. Zusätzlich gibt das Ethernet Protokoll allgemeine Details zur Übertragung an. Genauere Informationen zum Ethernet Protokoll werden in Abschnitt 3.8 aufgelistet.

3.8 Das Ethernet Protokoll

Für Kommunikation in Netzwerken werden Protokolle verwendet, die einen Bitstream als mehrere Paketinformationen interpretieren können. Da sich mithilfe von cachebasierten Seitenkanalangriffen ein Bitstream erstellen lässt, kann auch hier ein Netzwerkprotokoll angewendet werden. Da

der Fokus der Bachelorarbeit nicht auf die Optimierung eines Netzwerkprotokolls liegt, wird ein simples Netzwerkprotokoll verwendet werden. Dafür bietet sich das Ethernet Protokoll an, da es das am weitesten verbreitete Protokoll ist, was den gegebenen Anforderungen genügt.

Das Ethernet Protokoll gilt nach dem OSI-Schichten Modell als Schicht 1 & 2 Protokoll. Es wurde 1976 von Metcalfe und Boggs entwickelt [MB76] und wird bis heute am meisten in Netzwerken als Protokoll verwendet. Im Jahr 2012 legte die IEEE Standards Association fest wie ein 802.3 Ethernet Paket aufgebaut sein muss [IEE12]. Folgende Bytes stellen ein Ethernet-Paket dar. Als Ethernet-Frame gelten die Bytes beginnend mit Desination Adress bis zur Frame Check Sequence.

- **Preamble** (7 Bytes): Um den Beginn der Datenübertragung zu signalisieren, werden von dem Sender 56 alternierende Bits, beginnend mit 1, gesendet.
- **Start Frame Delimiter** (1 Byte): Die SFD Bits sind festgelegt auf die Bitreihenfolge 10101011. Der Empfänger liest durchgehend auf dem Kanal und sobald die Preamble und SFD Bits komplett erkannt werden, werden folgende Bit als Ethernet Frame interpretiert. [MF19, 3.3]
- **Destination Adress** (6 Bytes): In einem Netzwerk mit mehreren Geräten muss eindeutig, sein welches Ethernet Paket an welchen Rechner gesendet werden soll.
- **Source Adress** (6 Bytes): In einen Ethernet Frame gehört die MAC Adresse des Absenders. In einem Netzwerk mit mehreren Geräten, kann somit differenziert werden, wer der Absender des Ethernet Pakets ist. [MF19, 3.3]
- **Ethertype** (2 Bytes): Befindet sich der Wert unter 1501, so ist Ethertype die Größe der Payload in Bytes. Wenn der Wert über 1535 ist, beschreibt Ethertype welches Protokoll für die Payload benutzt wird [IEE12]. Im Rahmen dieser Bachelorarbeit wird Length immer als die Länge der Payload interpretiert.
- **Payload** (46 - 1500 Bytes): Die Payload ist der Inhalt des Ethernet Pakets und ist nie größer als 1500 Bytes. Wenn die gegebene Länge in Ethertype unter 46 ist, so wird Payload mit 0 Bytes gepaddet, sodass die Payload 46 Bytes groß ist. [MF19, 3.3]
- **Frame Check Sequence** (4 Bytes): Das Ende des Ethernet Pakets ist ein Cyclic Redundancy Check, der von dem Sender für das gesamte Ethernet Paket erstellt wurde [MF19, 3.3]. Der Empfänger kann für das empfangene Ethernet Paket ebenfalls eine Frame Check Sequence (FCS) erstellen und ihn mit dem empfangenen FCS vergleichen. Sind die FCS nicht identisch, dann wurden wahrscheinlich ein oder mehrere Bits nicht korrekt empfangen.
- **Interpacket Gap** (12 Bytes): Zwischen Ethernet Paketen muss der Sender mindestens 12 Bytes senden, bevor das nächste Ethernet Paket gesendet werden darf. [IEE12]

Innerhalb dieser Bachelorarbeit soll die Kommunikation zwischen einem Sender und Empfänger über einen verdeckten Kanal implementiert werden. Die Felder Destination Address und Source Address sind in diesem Spezialfall überflüssig und werden mit 0 Bits gefüllt.

4 Messungen der verdeckten Kanäle

In diesem Kapitel wird die Implementierung eines verdeckten Kanals aus Kapitel 3 mithilfe von Messmethoden dargestellt. Dafür werden in Abschnitt 4.1 die zwei Testumgebungen und die installierte Software vorgestellt. In Abschnitt 4.2 werden die Messmethoden präsentiert. Der verdeckte Kanal wurde mit den Messmethoden in verschiedenen Szenarien ausgewertet und in Abschnitt 4.3 vorgestellt. Eine Auswertung der Ergebnisse folgt in Kapitel 5.

4.1 Die Testumgebungen

Ziel der Bachelorarbeit ist es, aus den Seitenkanalangriffen `Flush+Reload` und `Flush+Flush`, einen verdeckten Kanal zu bauen und diesen Kanal zu bewerten. Für das Berechnen der Messwerte wurden zwei verschiedene ARMv8 Prozessoren benutzt. In diesem Abschnitt werden die genauen Details zu jedem Prozessor und der benutzten Software vorgestellt. Da die Seitenkanalangriffe nur den Daten-Cache abhören, werden die Details zu dem L1-Instruktionscache jeweils ausgelassen.

Raspberry Pi 4 B

- Cortex-A72, 8 Kerne mit jeweils 1.5 GHz [Ras20]
- 4 GiB LPDDR4-3200 SDRAM Arbeitsspeicher [Ras20]
- L1-Daten-Cache: 32 KiB pro Kern, 2-Way Set Associative mit 64 Byte Cache-Zeile [ARM16, 6.1]
- L1-Daten-Cache benutzt Least Recently Used [ARM16, 6.1]
- L2 Unified-Cache: 512 KiB, 16-Way Set Associative mit 64 Byte Cache-Zeile [ARM16, 7.1]
- L2 Unified-Cache benutzt eine Kombination von Least Recently Used und Random Replacement
- L2 ist inclusive zum L1-Daten-Cache [ARM16, 7.1]
- L1-Daten-Cache sowie unified L2-Cache sind `PIPT`-Caches [ARM16, 7.1]

Als Betriebssystem ist Manjaro 20.2.1 auf einer 64 GiB Micro SD-Karte installiert.

Rock Pi S

- Cortex-A35, 4 Kerne mit jeweils 1.3 GHz [Rad18]
- 512MB DDR3 Arbeitsspeicher [Rad18]
- L1-Daten-Cache: 8 KiB pro Kern, 4-Way Set Associative mit 64 Byte Cache-Zeile [ARM19, 6.1]
- L1-Daten-Cache benutzt eine Kombination von Least Recently Used und Random Replacement [ARM19, 6.1]

- L2 Unified-Cache: 128 KiB, 8-Way Set Associative mit 64 Byte Cache-Zeile [ARM19, 7.1]
- L2 Unified-Cache benutzt eine Kombination von Least Recently Used und Random Replacement [ARM19, 7.1]
- L2 ist inclusive zum L1-Daten-Cache [ARM19, 7.1]
- L1-Daten-Cache sowie unified L2-Cache sind PIPT-Caches [ARM19, 7.1]

Als Betriebssystem ist Ubuntu 20.04.02[1] auf einer 8 GiB Micro SD-Karte installiert.

Installierte Software

Auf beiden vorgestellten Systemen ist folgende Software installiert:

- GNU Compiler Collection 10.2.0
- GNU Make 4.3
- Python 3.9.1
- Python pip 20.3
- Numpy 1.20.0
- Matplotlib 3.3.3
- scipy 1.6.0

4.2 Vorstellen der Messmethoden

Damit vorgestellte Messwerte nachvollziehbar sind, muss die Messmethode vorgestellt werden. Es folgen zwei Messmethoden, die in den nachfolgenden Texten mit Messmethode 1 und Messmethode 2 referenziert werden. Um ungewollte Cache-Verdrängungen zu vermindern, laufen nur Sender- und Empfänger-Prozess. Beide Messmethoden wurden auf einem System mit möglichst wenig Rauschen im Cache ausgeführt. Es laufen also nur Sender- und Empfänger-Prozess, um ungewollte Cache-Verdrängungen zu vermindern.

Die erste Messmethode wird benutzt, um den Schwellwert von `Flush+Reload` oder `Flush+Flush` zu berechnen. Dafür wird `Flush+Reload` oder `Flush+Flush` im selben Prozess und Thread (folgend Same-Thread genannt) 2000 Mal ausgeführt. Dabei ist keine Frequenz vorgegeben und es werden nur die Hit- und Miss-Timings berechnet. Damit sowohl Hits als auch Misses entstehen, werden vor den Zeitmessungen Speicherzugriffe oder Flush-Instruktionen durchgeführt. Als Speicherzugriff wird eine Funktion aus einer speziell angefertigte Read-Only Bibliothek, wie in Abschnitt 3.6.1 beschrieben, geladen. Ziel ist es die unterschiedlichen Zeiten von Hits und Misses zu erfassen und grafisch darzustellen. Darauf aufbauend werden zwischen Sender- und Empfänger-Prozess 2000 alternierende Bits gesendet. Dies wird für `Flush+Reload` und `Flush+Flush` auf drei Frequenzen 100 Hz, 10 kHz und 1 MHz durchgeführt. Basierend auf den gemessenen Zeiten des Empfängers können Schwellwerte berechnet werden, die für die zweite Messmethode benutzt werden können. Diese Messmethode wird folgend als **Messmethode** 1 bezeichnet.

Als zweite Messmethode sendet der Sender-Prozess mit `Flush+Reload` und `Flush+Flush` ein Ethernet Frame 20 Mal auf den drei Frequenzen 100 Hz, 10 kHz und 1 MHz an den Empfänger-Prozess. Der Empfänger benutzt den Schwellwert, der in der ersten Messmethode für die jeweilige Frequenz berechnet wurde, um zwischen Hits und Misses zu unterscheiden. Die Cache-Zeile

[1] Aktuell nicht für den Rock PI S verfügbar, aber kann durch ein Upgrade erlangt werden.

wird von dem Sender mit einer Funktion aus einer speziell angefertigten Read-Only Bibliothek in den Cache geladen. Der Inhalt der Payload ist in jeder Messung derselbe String der Länge 53 Byte, wodurch jedes abgesendete Ethernet Frame 73 Byte lang und identisch ist. Grund für die identischen Ethernet Frames ist, dass die Ergebnisse besser vergleichbar sind. Der Sender sendet vor jedem Ethernet Frame 56 Preamble Bits und 8 SFD Bits, um den Empfänger zu signalisieren, dass ein Ethernet Frame folgt. Die empfangenen Bits nach den Preamble und SFD Bits werden bitweise mit den gesendeten Bits verglichen.

Aus der Menge der 73 gesendeten Bytes und der Menge der 73 empfangenen Bytes werden folgend die Wasserstein-Distanz, die erwartete Wasserstein-Distanz, die Fehlerrate, die Konfusionsmatrix und die theoretische Kanalkapazität berechnet (siehe Abschnitt 2.10). Für die Berechnung der theoretische Kanalkapazität wird die Fehlerrate und die verwendete Frequenz benutzt. Damit der Satz von Shannon (siehe Abschnitt 2.10.3) angewendet werden kann, müssen die Bitfehler unabhängig voneinander sein. Dies wird überprüft, indem die erreichte Wasserstein-Distanz größer oder gleich der erwarteten Wasserstein-Distanz ist (siehe Abschnitt 2.10.2. Somit wird garantiert, dass die Fehler in dem angepassten Bit String nur Substitution-Errors sind. Ein Overhead durch das Ethernet Protokoll wird dabei vernachlässigt, da der Fokus nicht auf das Bewerten des Ethernet Protokolls liegt. Für die Kanalkapazität wird angenommen, dass ein fehlerkorrigierender Code benutzt wird (siehe Abschnitt 2.10.3). Demnach werden folgende Kanalkapazitäten als theoretische Kanalkapazitäten bezeichnet.

Über die 20 gesendeten Ethernet Frames wird für jede Einstellung der das arithmetische Mittel sowie die Varianz der Vergleichskriterien erstellt. Das arithmetische Mittel wird hier statt des Medians gewählt, da für die Berechnung der Shannon-Kapazität das arithmetische Mittel der Fehlerrate benötigt ist. Diese Messmethode wird folgend als **Messmethode 2** bezeichnet. Messmethode 2 wurde mit Same-Core und Cross-Core ausgeführt. Dabei ist anzumerken, dass wie in Abschnitt 3.7 beschrieben, Cross-Core mit einer Busy-Waiting-Schleife als Synchronisierungsmethode ausgeführt wird. Würde Same-Core mit einer Busy-Waiting-Schleife implementiert werden, so wäre der Kern, auf dem beide Prozesse ausgeführt werden, überladen und eine Synchronisation wäre nicht möglich.

4.3 Die Messergebnisse

In diesem Abschnitt werden die Messergebnisse, die mit den oben genannten Messmethoden erlangt wurden, präsentiert. Dies wird seperat für `Flush+Reload` und für `Flush+Flush` vorgestellt. Eine Einordnung der Ergebnisse folgt in Kapitel 5.

4.3.1 Flush+Reload

In diesem Abschnitt werden die Ergebnisse der oben vorgestellten Messmethoden für `Flush+Reload` vorgestellt. Es wird zwischen Same-Core und Cross-Core unterschieden, da die Cache-Zugriffszeit zum L1- und zum unified L2-Cache laut Spezifikation unterschiedlich lang sind (siehe Abschnitt 2.8). Zusätzlich wird bei Cross-Core für die Synchronisation eine Busy-Waiting-Schleife implementiert, während bei Same-Core stattdessen Nanosleep verwendet wird (Details in Abschnitt 3.7).

Same-Core

In Abbildung 11 sind die erstellten Graphen für die Schwellwert-Berechnung von `Flush+Reload` auf dem ARM Cortex-A72 dargestellt. Die Verteilungen von Hits und Misses bei einer Ausführung im selben Thread sind wie erwartet eindeutig getrennt. Dabei ähneln die Verteilungen von Hits und Misses nicht die einer Normalverteilung. Die Varianz der Hits ist sehr gering im Vergleich zu der Varianz der Misses. Aufgrund dessen ist der Schwellwert näher am Durchschnitt der Hits. Die Verteilung bei zwei Prozessen, die sich auf demselben Kern ausgeführt werden, ist auf 100 Hz sehr ähnlich zur Ausführung im selben Thread, abgesehen davon, dass die Verteilung der Hits bei 100 Hz eher einer Normalverteilung entsprechen. Die klare Trennung von Hits und Misses spiegelt sich in Messmethode 2 in Abbildung 12 wider. Mit einer Fehlerrate von 2% würde `Flush+Reload` eine theoretische Kanalkapazität von 93.97 bps erreichen. Jedoch ist die erwartete Wasserstein-Distanz mit 0.0077 höher als die erreichte Wasserstein-Distanz von 0.0005. Der Satz von Shannon ist somit nicht anwendbar. Wird die Taktfrequenz jedoch auf 10 kHz erhöht, so steigt die Fehlerrate sehr stark an. Die erwartete Wasserstein-Distanz für 10 kHz von 0.1258 ist niedriger als die erreichte Wasserstein-Distanz von 0.1688 ist. Der Satz von Shannon ist somit anwendbar. Mit einer Fehlerrate von 45% erreicht der Kanal nur eine Kapazität von 72.25 bps. Die Grafik in Abbildung 11 zeigt, dass bei 10 kHz ein kleiner Anteil von Hits und Misses falsch klassifiziert wird. Abgesehen von den falsch klassifizierten Bits, ähneln die Verteilungen von Hits und Misses in Abschnitt 11 einer Normalverteilung. Bei einer Frequenz von 1 MHz kann zwischen einem Hit und Miss nicht mehr unterschieden werden. Dies wird an Abbildung 11 dargestellt, da die Zeiten von Hits und Misses fast komplett identisch sind. Der Schwellwert für die anderen Szenarien liegt bei ungefähr 300 CPU Zyklen.

Auf dem ARM Cortex-A35 ist die Differenz zwischen Hit und Miss bei einer Same-Thread Ausführung deutlich höher als beim ARM Cortex-A72. In Abbildung 13 wird gezeigt, dass ein Großteil der Misses in Messmethode 1 über 2000 CPU Zyklen benötigt. Die Varianz der Hits ist deutlich geringer als die Varianz der Misses wodurch der Schwellwert näher an dem Durchschnitt der Hits liegt. Trotz der eindeutigen Trennung von Hits und Misses im Same-Thread Szenario, sind die Ergebnisse bei Messmethode 2 unerwartet schlecht. Aufgrund der hohen Fehlerrate ist die theoretische Kanalkapazität auf Same-Core sehr gering, mit der Ausnahme von Same-Core auf 1 MHz, welche bei 49.32 kbps liegt. Der Satz von Shannon ist für 1 MHz anwendbar, da die erwartete Wasserstein-Distanz mit 0.1766 niedriger als die erreichte Wasserstein-Distanz von 0.5394 (siehe Abbildung 14) ist. Die Verteilungen der Hits und Misses in Abbildung 13 für 10 kHz ähneln der einer Normalverteilung. Der Schwellwert liegt bei circa 200 CPU Zyklen und mit ungefähr gleich großem Abstand zu dem Durchschnitt beider Verteilungen. Für 100 Hz und 1 MHz trifft dies nicht zu, da sich ein Großteil der Verteilungen überschneiden und somit eine Differenzierung zwischen Hit und Miss nicht mehr möglich ist.

Cross-Core

`Flush+Reload` erreicht mit Cross-Core auf dem ARM Cortex-A72 ein deutlich höhere theoretische Kanalkapazität auf 10 kHz als mit Same-Core (vgl. Abbildung 12). Die erwartete Wasserstein-Distanz mit 0.0267 ist jedoch höher als die Wasserstein-Distanz in Abbildung 12 von 0.0032, wodurch der Satz von Shannon nicht anwendbar ist. Ließe sich der Satz von Shannon anwenden, so würde mit einer Fehlerrate von 10% eine theoretische Kanalkapazität von 7.48 kbps erreicht werden. Dies wird auch in Abbildung 15 deutlich, da bei 10 kHz die Trennung von Hit und Miss klarer ist als in Abbildung 11 für 10 kHz. Für 100 Hz ist die erwartete Wasserstein-Distanz mit

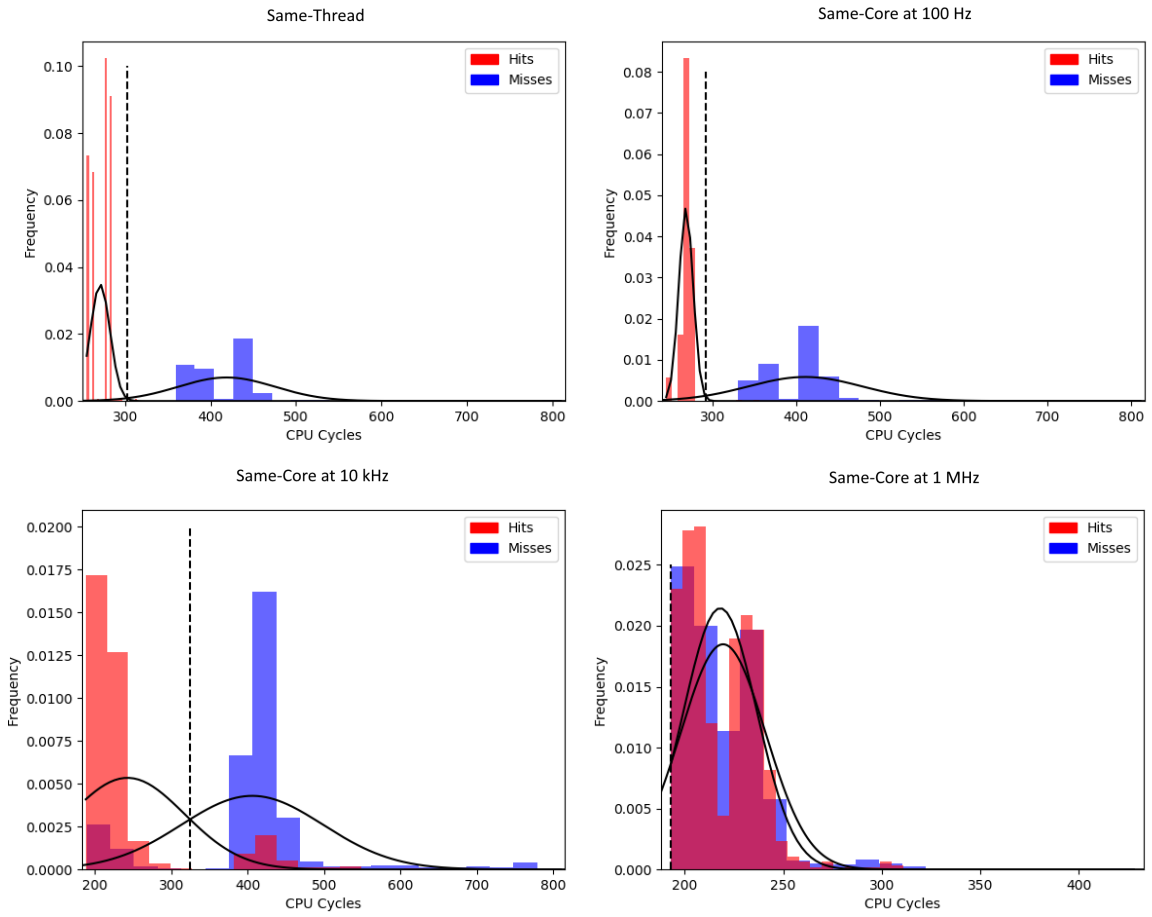

Abbildung 11: *Messmethode 1: Schwellwert-Berechnung für Flush+Reload Same-Core auf einem ARM Cortex-A72*

0.0077 ebenfalls höher als die erreichte Wasserstein-Distanz von 0.0021. Die Verteilungen von Hits und Misses in Abbildung 15 für 100 Hz gleichen einer Normalverteilung. Aufgrund der hohen Varianz der Misses für 100 Hz und 10 kHz liegt der Schwellwert nah am Durchschnitt der Verteilung der Hits. Ähnlich wie bei der Same-Core Ausführung, ist bei 1 MHz in Abbildung 15 kein Unterschied zwischen Hit und Miss mehr zu erkennen. Die Verteilungen überlappen sich und eine sinnvolle Schwellwert-Berechnung ist nicht mehr möglich. Der Schwellwert für die anderen Szenarien liegt ähnlich wie für Same-Core bei 300 CPU Zyklen.

Auch auf dem ARM Cortex-A35 erzielt Cross-Core eine geringere Fehlerrate für 100 Hz und 10 kHz. Besonders auf 10 kHz erreicht der Kanal mit einer Fehlerrate von 17% eine Kapazität von 3.42 kbps. Die erwartete Wasserstein-Distanz auf 10 kHz beträgt 0.1196, was geringer als die berechnete Wasserstein-Distanz von 0.1512 ist. Der Satz von Shannon ist somit anwendbar. Die Fehlerrate von 17% spiegelt sich in Abbildung 16 wider, da die Überschneidung von Hits und Misses für 10 kHz nicht so groß sind wie in Abbildung 13 für 10 kHz. Die Verteilung der Hits und Misses gleichen beide der einer Normalverteilung mit ähnlicher Varianz. Dadurch ist der Schwellwert mit 250 CPU Zyklen mittig zwischen den arithmetischen Mitteln von Hits und Misses gelegen. Auf 1 MHz erzielt Cross-Core eine ähnlich hohe Fehlerrate wie Same-Core, was zu einer theoretische Kanalkapazität von 41.96 kbps führt. Die erwartete Wasserstein-Distanz ist mit 0.1766 geringer als die erreichte Wasserstein-Distanz von 0.4242, wodurch der Satz von Shannon anwendbar ist.

Flush+Reload A72	Wasserstein-Distanz Durchschnitt / Varianz	Fehlerrate Durchschnitt / Varianz	True Positives Durchschnitt / Varianz	False Positives Durchschnitt / Varianz	True Negatives Durchschnitt / Varianz	False Negatives Durchschnitt / Varianz	Kanalkapazität [bps]
Same-Core, 100 Hz	0.0005 / 0.0001	0.02 / 0.001	202.05 / 837.04	6.95 / 837.04	367.85 / 861.13	7.15 / 861.13	93.97
Cross-Core, 100 Hz	0.0021 / 0.0001	0.05 / 0.0189	196.6 / 1576.84	14.4 / 1576.84	361.5 / 1640.25	13.5 / 1640.25	71.36
Same-Core, 10 kHz	0.1688 / 0.0253	0.45 / 0.015	128.5 / 1191.35	80.5 / 1191.35	196.0 / 5719.8	179.0 / 5719.8	72.25
Cross-Core, 10 kHz	0.0032 / 0.0001	0.09 / 0.14	180.65 / 2546.23	28.35 / 2546.23	348.05 / 2577.05	26.95 / 2577.05	7482.77
Same-Core, 1 MHz	0.1525 / 0.0034	0.42 / 0.0006	42.1 / 165.79	166.9 / 165.79	297.15 / 527.93	77.85 / 527.93	18546.1
Cross-Core, 1 MHz	0.4565 / 0.0008	0.61 / 0.0002	165.4 / 88.74	43.6 / 88.74	64.8 / 88.06	310.2 / 88.06	35200.45

Abbildung 12: *Messmethode 2 mit Flush+Reload getestet auf einem ARM Cortex-A72*

4.3.2 Flush+Flush

Im folgendem Abschnitt werden die Ergebnisse der oben vorgestellten Messmethoden für `Flush+Flush` vorgestellt. Aufgrund der laut Spezifikation unterschiedlich langen Zugriffszeiten von einem L1- und einem unified L2-Cache, wird zwischen Same-Core und Cross-Core unterschieden. Zusätzlich ist bei Cross-Core für die Synchronisation eine Busy-Waiting-Schleife implementiert, während bei Same-Core stattdessen Nanosleep verwendet wird (Details in Abschnitt 3.7).

Same-Core

Die Verteilungen der Hits und Misses auf dem ARM Cortex-A72 mit Messmethode 1 im selben Thread ähneln einer Normalverteilung. Die Varianz beider Verteilungen ist fast identisch, wodurch der Schwellwert zentral zwischen beiden Durschnitten platziert ist. Die zeitliche Differenz der Flush-Instruktion abhängig davon, ob die angegebene Adresse im Cache geladen ist, ist groß genug, dass keine Überschneidungen der Verteilungen vorkommen. Die Schwellwert-Berechnung zwischen zwei Prozessen resultiert jedoch in Überschneidungen von Hits und Misses, was sich in Messmethode 2 in Abbildung 18 widerspiegelt. Mit einer hohen Fehlerrate werden deutlich schlechtere Ergebnisse als mit `Flush+Reload` erzielt. Die Verteilungen in Abbildung 17 für 100 Hz und 10 kHz gleichen beide der Verteilung bei der Same-Thread Ausführung. Der Unterschied ist, dass die Varianz der Verteilungen bei 100 Hz und 10 kHz größer sind und es somit zu Überschneidungen von Hits und Misses führt. Der Schwellwert ist in allen Szenarien unterschiedlich. Bei Same-Thread liegt er bei circa 220 CPU Zyklen, bei 100 Hz ist der Schwellwert 260 CPU Zyklen und bei 10 kHz ist der Schwellwert 290 CPU Zyklen. Für 1 MHz ist eine sinnvolle Schwellwert-Berechnung nicht möglich. Für 100 Hz liegt die erwartete Wasserstein-Distanz bei 0.0652, was höher als die erreichte Wasserstein-Distanz von 0.0323 ist. Ebenso ist für 1 MHz die erwartete Wasserstein-Distanz mit 0.1270 höher als die erreichte Wasserstein-Distanz von 0.2854. Somit gilt für 100 Hz und 1 MHz, dass der Satz von Shannon nicht anwendbar ist. Für 10 kHz ist die erwartete Wasserstein-Distanz mit 0.1103 niedriger als 0.1883 (siehe Abbildung 18). Somit lässt sich mit dem Satz von Shannon eine theoretische Kanalkapazität von 352 bps berechnen.

Auf dem ARM Cortex-A35 ist die Schwellwert-Berechnung mit Messmethode 1 für Same-Thread (siehe Abbildung 19) bis auf kleine Überschneidungen ähnlich zum selben Szenario auf dem ARM Cortex-A72 (siehe Abbildung 17). Der Schwellwert für Same-Thread liegt bei 120 CPU Zyklen, während bei den anderen Szenarien keine Schwellwert-Berechnung möglich ist. Die Fehlerrate bei Messmethode 2 in Abbildung 20 sind alle nah an 50%, was zu geringeren theoretischen Kanalkapazitäten führt. Mit einer Fehlerrate von 50% erreicht ein Kanal eine theoretische Kanalkapazität von 0 bps, was den Kanal unbrauchbar machen würde. Die beste theoretische Kanalkapazität erzielt die Frequenz 1 MHz mit 7.22 kbps. Die erwartete Wasserstein-Distanz von 0.1564 ist geringer als die

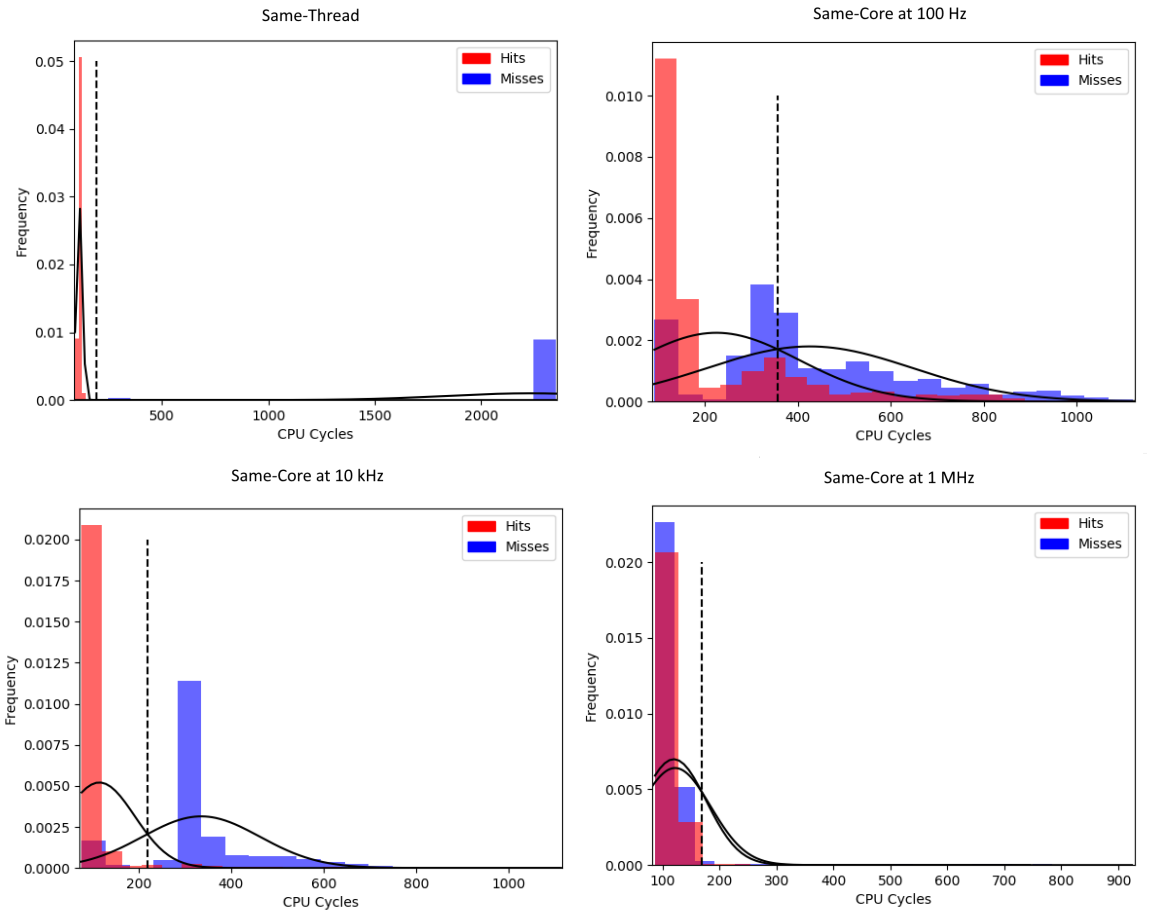

Abbildung 13: *Messmethode 1: Schwellwert-Berechnung für Flush+Reload Same-Core auf einem ARM Cortex-A35*

berechnete Wasserstein-Distanz von 0.2920. Daraus folgt, dass die Fehler unabhängig voneinander sind und der Satz von Shannon anwendbar ist. Die hohe Fehlerrate spiegelt sich in Abbildung 19 wider, da sich Hits und Misses stark überlappen.

Cross-Core

Auf dem ARM Cortex-A72 wird mit `Flush+Flush` bei einer Frequenz von 100 Hz mit Cross-Core eine niedrigere Fehlerrate als mit `Flush+Reload` erreicht. Mit einer Fehlerrate von 4% würde Cross-Core in Abbildung 18 eine theoretische Kanalkapazität von 75.77 bps erzielen. Jedoch ist die erwartete Wasserstein-Distanz mit 0.0108 höher als die erreichte Wasserstein-Distanz von 0.0046. Der Satz von Shannon ist somit nicht anwendbar. Die Verteilungen für 100 Hz und 10 kHz in Abbildung 21 haben, ähnlich zu den Verteilungen bei Same-Core, eine so hohe Varianz, dass es zu Überschneidungen von Hits und Misses kommt. Der Schwellwert für 100 Hz und 10 kHz liegt ungefähr bei 250 CPU Zyklen.

Die Cross-Core Ergebnisse auf dem ARM Cortex-A35 sind den Ergebnissen mit Same-Core sehr ähnlich. Die Überschneidungen in Abbildung 22 sind genauso groß, wie die Überschneidungen in Abbildung 19 für Same-Core.

Flush+Reload A35	Wasserstein-Distanz Durchschnitt / Varianz	Fehlerrate Durchschnitt / Varianz	True Positives Durchschnitt / Varianz	False Positives Durchschnitt / Varianz	True Negatives Durchschnitt / Varianz	False Negatives Durchschnitt / Varianz	Kanalkapazität [bps]
Same-Core, 100 Hz	0.3482 / 0.0005	0.53 / 0.0015	156.05 / 79.35	52.95 / 79.35	118.7 / 493.61	256.3 / 493.61	2.6
Cross-Core, 100 Hz	0.0553 / 0.0143	0.37 / 0.0164	102.55 / 1598.05	106.45 / 1598.05	269.95 / 1301.15	105.05 / 1301.05	4.93
Same-Core, 10 kHz	0.3572 / 0.0004	0.51 / 0.0213	145.55 / 2268.71	63.55 / 2268.71	141.65 / 5760.85	233.35 / 5760.85	2.89
Cross-Core, 10 kHz	0.1512 / 0.0276	0.17 / 0.0509	172.8 / 2479.06	36.2 / 2479.06	310.15 / 8728.03	64.85 / 8728.03	3422.95
Same-Core, 1 MHz	0.5394 / 0.0019	0.63 / 0.0003	185.0 / 37.2	24.0 / 37.2	36.0 / 101.0	339.0 / 101.0	49327.91
Cross-Core, 1 MHz	0.4242 / 0.0004	0.62 / 0.0002	165.1 / 30.49	43.9 / 30.49	82.65 / 84.33	292.35 / 84.33	41957.97

Abbildung 14: *Messmethode 2 mit Flush+Reload getestet auf einem ARM Cortex-A35*

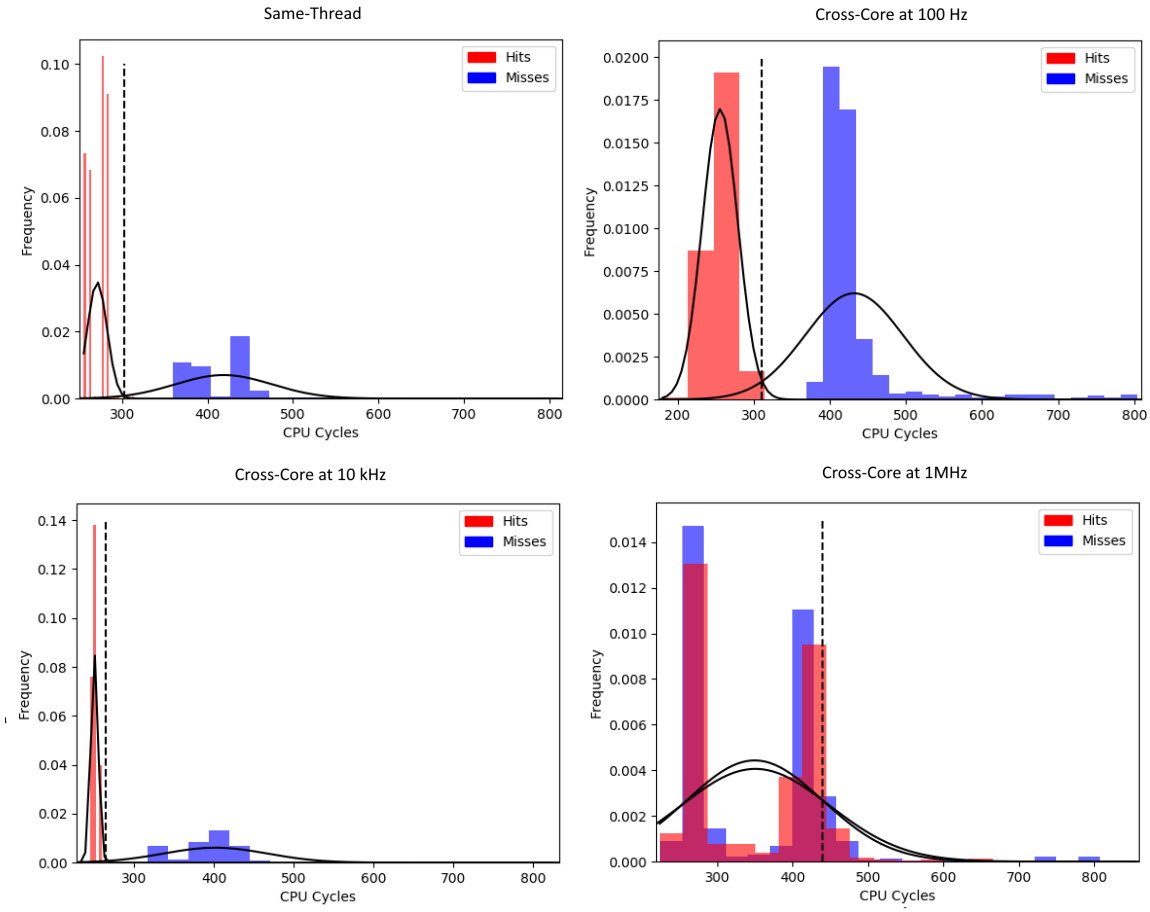

Abbildung 15: *Messmethode 1: Schwellwert-Berechnung für Flush+Reload Cross-Core auf einem ARM Cortex-A72*

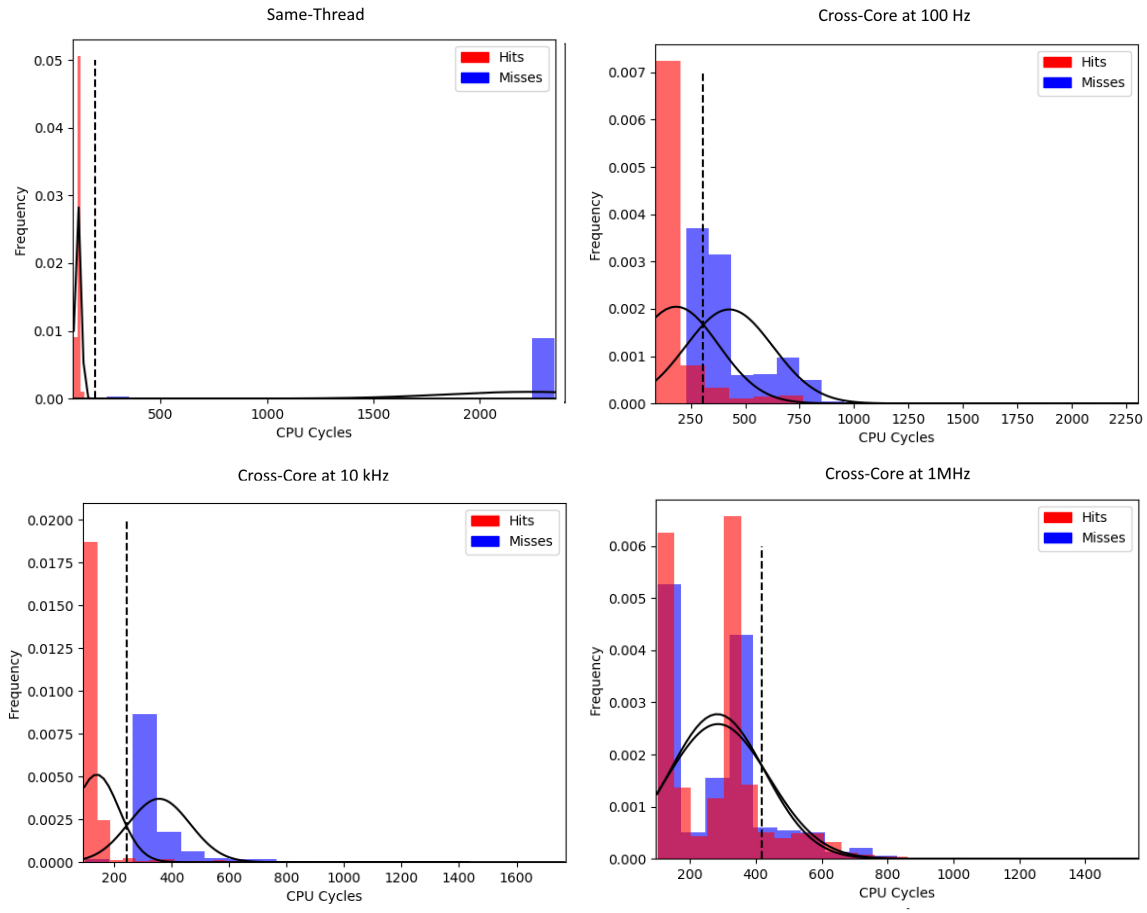

Abbildung 16: *Messmethode 1: Schwellwert-Berechnung für Flush+Reload Cross-Core auf einem ARM Cortex-A35*

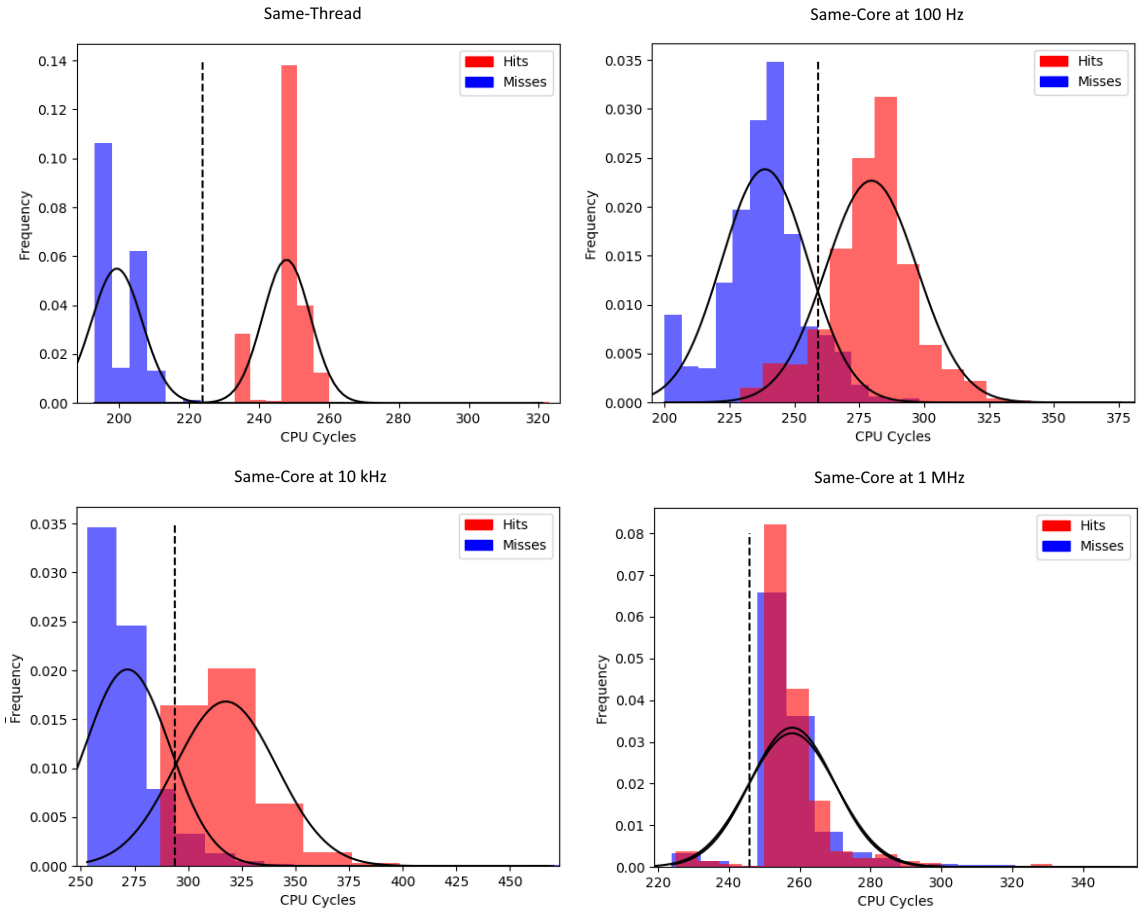

Abbildung 17: *Messmethode 1: Schwellwert-Berechnung für Flush+Flush Same-Core auf einem ARM Cortex-A72*

Flush+Flush A72	Wasserstein-Distanz Durchschnitt / Varianz	Fehlerrate Durchschnitt / Varianz	True Positives Durchschnitt / Varianz	False Positives Durchschnitt / Varianz	True Negatives Durchschnitt / Varianz	False Negatives Durchschnitt / Varianz	Kanalkapazität [bps]
Same-Core, 100 Hz	0.0325 / 0.0002	0.23 / 0.038	132.9 / 2903.59	76.1 / 2903.59	317.6 / 3571.44	57.4 / 3571.44	22.2
Cross-Core, 100 Hz	0.0046 / 0.0001	0.04 / 0.0075	199.6 / 645.44	9.4 / 645.44	364.1 / 635.69	10.9 / 635.69	75.77
Same-Core, 10 kHz	0.1883 / 0.0092	0.39 / 0.0004	40.05 / 707.65	168.95 / 707.65	316.0 / 932.0	59.0 / 932.0	352
Cross-Core, 10 kHz	0.1580 / 0.0055	0.52 / 0.0007	103.25 / 362.49	105.75 / 362.49	176.95 / 693.05	198.05 / 693.05	11.54
Same-Core, 1 MHz	0.0658 / 0.0004	0.45 / 0.0005	60.0 / 29.6	149.0 / 29.6	264.45 / 113.85	110.55 / 113.85	7225.55
Cross-Core, 1 MHz	0.2854 / 0.0023	0.58 / 0.0002	165.1 / 30.49	43.9 / 30.49	82.65 / 84.33	292.35 / 84.33	18546.1

Abbildung 18: *Messmethode 2 mit Flush+Flush getestet auf einem ARM Cortex-A72*

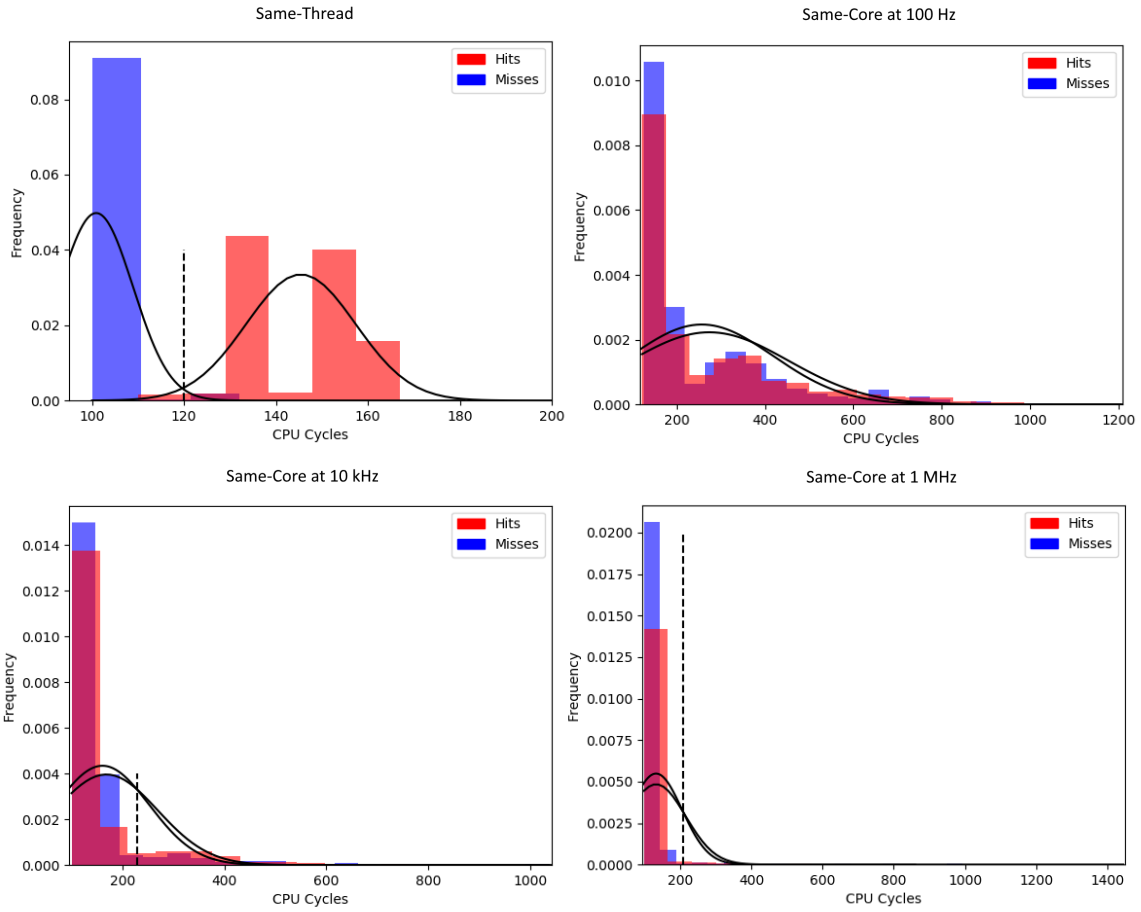

Abbildung 19: *Messmethode 1: Schwellwert-Berechnung für Flush+Flush Same-Core auf einem ARM Cortex-A35*

Flush+Flush A35	Wasserstein-Distanz Durchschnitt / Varianz	Fehlerrate Durchschnitt / Varianz	True Positives Durchschnitt / Varianz	False Positives Durchschnitt / Varianz	True Negatives Durchschnitt / Varianz	False Negatives Durchschnitt / Varianz	Kanalkapazität [bps]
Same-Core, 100 Hz	0.2657 / 0.0076	0.41 / 0.0021	32.45 / 48.27	176.55 / 48.27	313.8 / 3272.68	61.2 / 3272.68	2.34
Cross-Core, 100 Hz	0.2646 / 0.0098	0.44 / 0.0003	59.1 / 49.33	149.9 / 49.33	270.1 / 194.19	104.9 / 194.19	1.04
Same-Core, 10 kHz	0.2788 / 0.0469	0.42 / 0.0012	25.05 / 881.85	183.95 / 881.85	314.5 / 2147.85	60.5 / 2146.85	185.46
Cross-Core, 10 kHz	0.2509 / 0.0174	0.4 / 0.0034	31.05 / 49.33	177.95 / 49.33	317.9 / 5320.89	33.55 / 5320.89	290.49
Same-Core, 1 MHz	0.2920 / 0.0002	0.55 / 0.0003	133.75 / 49.49	75.25 / 49.49	129.2 / 52.06	245.8 / 52.06	7225.55
Cross-Core, 1 MHz	0.3021 / 0.0041	0.48 / 0.0002	83.45 / 33.55	125.55 / 33.55	217.35 / 68.23	157.65 / 68.23	1154.46

Abbildung 20: *Messmethode 2 mit Flush+Flush getestet auf einem ARM Cortex-A35*

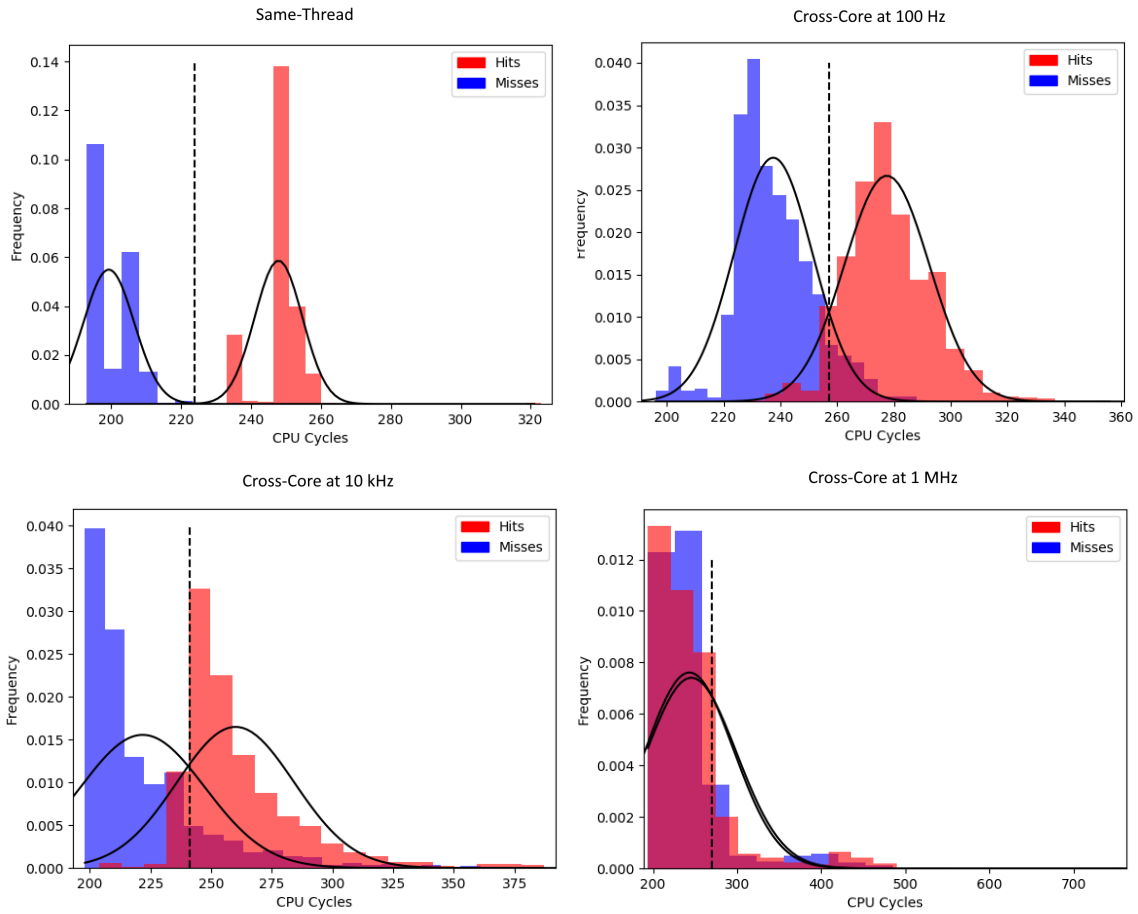

Abbildung 21: *Messmethode 1: Schwellwert-Berechnung für Flush+Flush Cross-Core auf einem ARM Cortex-A72*

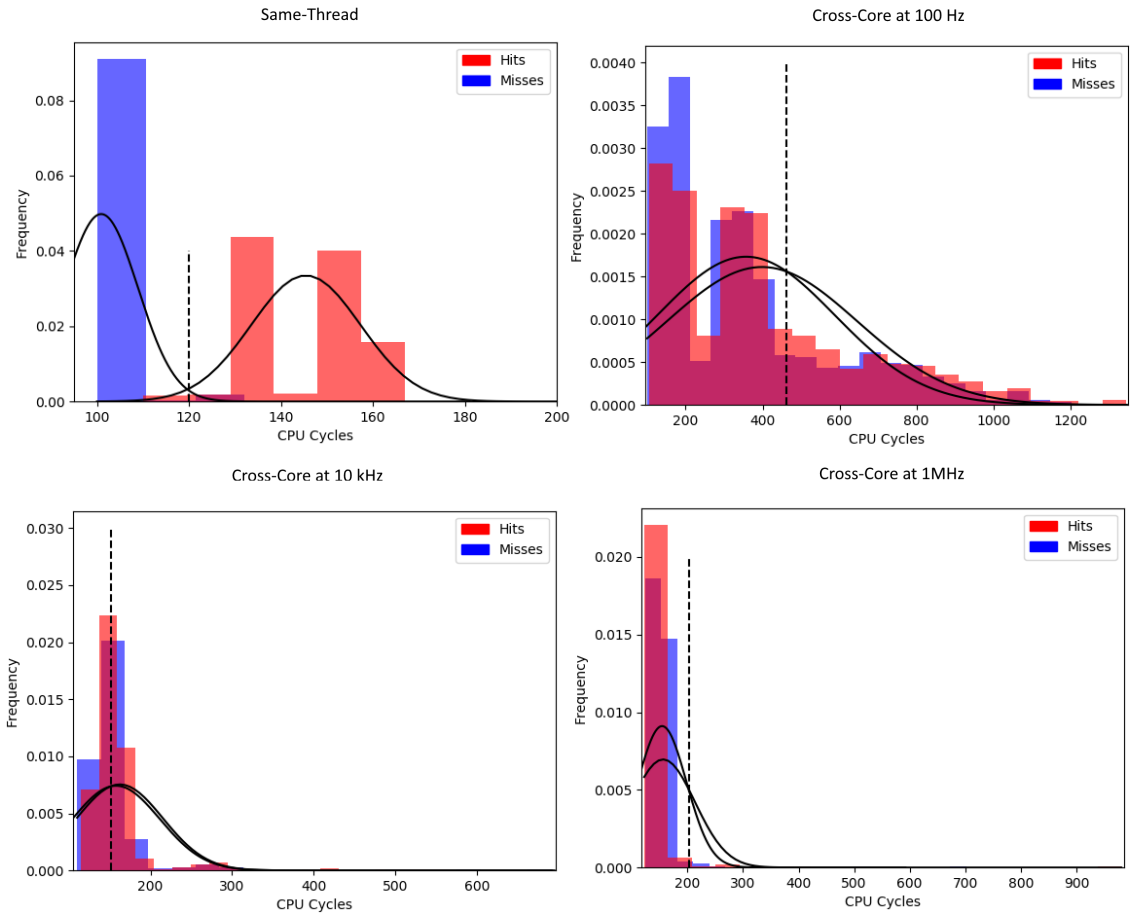

Abbildung 22: *Messmethode 1: Schwellwert-Berechnung für Flush+Flush Cross-Core auf einem ARM Cortex-A35*

5 Einordnung der Messwerte

In diesem Kapitel werden die Messergebnisse aus Kapitel 4 eingeordnet, in Abschnitt 5.1 gedeutet und miteinander verglichen. In Abschnitt 5.2 werden die Messungen und Deutungen mit denen von „ARMageddon: Cache Attacks on Mobile Devices“[LGS+16] verglichen.

5.1 Klassifizierung der Ergebnisse

Für die Klassifizierung der Ergebnisse ist zu erwähnen, dass die Messmethoden mit einer Testmenge, die sich in retroperspektiv als zu klein herausstellte, durchgeführt. Dies kann zu Ausreißern und somit zu unerwarteten Testergebnissen führen. Für den Großteil der Einordnung wird angenommen, dass die Messmethoden mit einer ausreichend großen Testmenge gemessen wurden, sodass die Ergebnisse zuverlässig sind.

5.1.1 Flush+Reload auf dem ARM Cortex-A72

Bei einem Kanal, der über Cross-Core ausgeführt wird, wird statt dem L1-Cache der unified L2-Cache verwendet. Dies hat zur Folge, dass nicht nur die Speicherzugriffszeit auf den Cache erhöht ist (siehe Abschnitt 2.8), sondern auch mehr Dritte die benutze Cache-Zeile verdrängen können. Eine erhöhte Last durch Dritte könnte somit zu einer erhöhten Anzahl an False Negatives führen. Dies spiegelt sich bei einer Frequenz von 100 Hz wider, da die Anzahl an False Negatives für Cross-Core minimal höher ist als die für Same-Core (siehe Abbildung 12). Da mehr Prozesse Zugriff auf den unified L2-Cache haben als auf den L1 eines einzelnen Kernes, kann eine Verdrängung von Cache-Zeilen eher eintreffen. Die Messmethoden wurden auf einem möglichst rauschfreiem System durchgeführt. Dennoch ist es möglich, dass Systemdienste Cache-Zeilen verdrängen können.

Bei einer Frequenz von 10 kHz steigt die Fehlerrate von Same-Core sehr stark an, während die von Cross-Core nur leicht ansteigt (vgl. Abbildung 12). Dies kann an den unterschiedlichen Synchroniserungsmethoden für Same-Core und Cross-Core liegen. Nanosleep ist für eine Wartezeit unter 2 Millisekunden nicht sehr zuverlässig (siehe Abschnitt 3.7). Eine Periodendauer bei 10 kHz beträgt 0.1 Millisekunden. Cross-Core hingegen wurde mit einer Busy-Waiting-Schleife implementiert, welche bei Wartezeiten unter 2 Millisekunden zuverlässiger als Nanosleep arbeitet. Verlässt Nanosleep die Wartezeit zu spät, so kann dies Substitution-Errors führen. Dies spiegelt sich ebenfalls in der Schwellwert-Berechnung in Abbildung 11 wider, da vereinzelte Messungen falsch klassifiziert werden. Aus der erwarteten Wasserstein-Distanz von 0.1103 lassen sich keine Insertion- und Deletion-Errors ableiten. Die erreichte Wasserstein-Distanz ist mit 0.1580 höher als die erwartete Wasserstein-Distanz, wodurch die Fehler unabhängig voneinander sind und somit Substitution-Errors sind. Für 100 Hz jedoch beträgt die erwartete Wasserstein-Distanz 0.0077, was höher als die erreichte Wasserstein-Distanz von 0.0005 ist. Dies ist ein Indikator dafür, dass

Insertion- und Deletion-Errors aufgetreten sind. Da bei 10 kHz keine Insertion- und Deletion-Errors auftreten, wird vermutet, dass bei einer größeren Testmenge die erreichte Wasserstein-Distanz größer als die erwartete Wasserstein-Distanz ist. Wodurch sich keine Insertion- und Deletion-Errors zeigen lassen könnten.

Für eine Cross-Core Ausführung auf 10 kHz wird eine theoreotische Kanalkapazität von 7.48 kbps erreicht. Jedoch ist die erwartete Wasserstein-Distanz mit 0.0267 höher als 0.0032. Aufgrund der höheren Zuverlässigkeit der benutzten Busy-Waiting-Schleife wird erwartet, dass weniger Insertion- und Deletion-Errors auftreten. Daher wird vermutet, dass sich die berechnete Wasserstein-Distanz bei einer größeren Testmenge der zu erwartenden Wasserstein-Distanz nähert.

Auf 1 MHz ist für Same-Core und Cross-Core eine Schwellwert-Bestimmung nicht mehr möglich, da wie in Abbildungen 11 und 15 zu erkennen, zwischen Hit und Miss nicht mehr unterschieden werden kann. Folgend ist die Fehlerrate erwartet 50%. In der Tabelle in Abbildung 12 ist die Fehlerrate bei 1 MHz jedoch besser als erwartet. Dies ist aufgrund der schlechten Klassifizierung von Hit und Miss in Abbildung 15 eher unwahrscheinlich. Es wird vermutet, dass bei einer größeren Testmenge, die Fehlerrate gegen 50% konvergiert.

Der Schwellwert für Same-Core sowie für Cross-Core liegt bei allen Szenarien, wo ein Schwellwert sinnvoll berechnet werden kann, bei circa 300 CPU Zyklen. Ein durschnittlicher Hit liegt bei circa 250 CPU Zyklen. Dies ist deutlich höher als eine erwartete Cache Hit Zeit. Da für den ARM Cortex-A72 in der Dokumentation keine genauen Angaben für Cache-Zugriffszeiten angegeben sind, werden stattdessen von einem vergleichbarem System die Zugriffszeiten zur Orientierung vorgestellt. Bei dem ARM Cortex-A53 liegt ein L1 Hit bei 3 Zyklen und ein L2 Hit bei 15 Zyklen [LZM20]. Die großen Unterschiede zwischen dem erwarteten und dem berechneten Wert können unter anderem durch die verwendete Zeitmessung hervorgerufen werden. `perf_event_open` sorgt für einen kleinen Overhead bei dem Auslesen von CPU Zyklen [LGS+16] (siehe Abschnitt 3.3.3). Ein weiterer Grund könnte sich auf die Implementierung rückführen lassen. Die Performance des Programmes lässt sich durch die Verwendung von mehr Inline-Assembler und Makros verbessern, was zu kürzeren Zugriffszeiten führen kann.

Da `Flush+Reload` mit Same-Core auf 10 kHz eine hohe Fehlerrate hat, scheint die optimale Frequenz zwischen 100 Hz und 10 kHz zu liegen. Bei einer Cross-Core Implementierung ist durch die Verwendung einer Busy-Waiting-Schleife eine bessere Performance möglich. Die optimale Frequenz für Cross-Core scheint aufgrund der hohen theoretische Kanalkapazität bei circa 10 kHz zu liegen.

5.1.2 Flush+Reload auf dem ARM Cortex-A35

Die Schwellwert-Berechnung von `Flush+Reload` auf dem ARM Cortex-A35 (vgl. Abbildungen 13 und 16) ist deutlich ungenauer als auf dem ARM Cortex-A72 (siehe Abbildungen 11 und 15). Auf dem ARM Cortex-A35 überlappen die Verteilungen von Hits und Misses besonders bei Same-Core in Abbildung 13 sehr stark. Dies ist für `Flush+Reload` eher ungewöhnlich, da der Unterschied zwischen einem Cache Hit und Cache Miss vergleichsweise groß ist. Yarom zeigte in seiner Arbeit zu `Flush+Reload`, dass Cache Hits und Misses auf einer ausgewählten Intel x86 CPU ungefähr 50 und 300 respektive benötigen [YF14]. Die Differenz zwischen den Zeiten mit den meisten Hits und Misses ist vergleichbar mit denen auf dem ARM Cortex-A72. Auf dem ARM Cortex-A35 mit 100 Hz brauchen einige Hits jedoch deutlich mehr Zeit, was zu Überschneidungen in höheren Latenzen (300 CPU Zyklen und mehr) führt. Überschneidungen in geringeren Latenzen (200

CPU Zyklen und weniger), können durch Substitution-Errors folgen. Die erwartete Wasserstein-Distanz von 0.1507 für Same-Core ist geringer als die erreichte Wasserstein-Distanz von 0.3482. Für Cross-Core auf 100 Hz ist die erwartete Wasserstein-Distanz mit 0.0486 ebenfalls kleiner als die erreichte Wasserstein-Distanz von 0.0543. Somit gilt für beide Ausführungen, dass keine Deletion- und Insertion-Errors aufgetreten sind. Demnach ist die hohe Anzahl an False Positives und False Negatives für 100 Hz in Abbildung 14 nur aus Substitution-Errors bestehend. Dies deckt sich mit der Annahme, dass die Überschneidungen aus Substitution-Errors folgt.

Cross-Core erreicht auf dem ARM Cortex-A35 mit `Flush+Reload` niedrigere Fehlerraten als mit Same-Core. Diese besseren Ergebnisse könnten durch die Implementierung der Busy-Waiting-Schleife verursacht worden sein. Aufgrund der niedrigen Zuverlässigkeit von Nanosleep bei Wartezeiten von unter 2 Millisekunden (siehe Abschnitt 3.7), können Insertion- und Deletion-Errors reduziert werden.

Ähnlich wie auf dem ARM Cortex-A72 ist die Schwellwert-Berechnung für eine Frequenz von 1 MHz nicht mehr möglich und folgend liegt die erwartete Fehlerrate bei 50%. Die errechnete Fehlerrate liegt jedoch für Same-Core und Cross-Core bei ungefähr 62%. Es wird wie beim ARM Cortex-A72 vermutet, dass bei einer größeren Testmenge die Fehlerrate gegen 50% konvergiert.

Die optimale Frequenz für `Flush+Reload` auf dem ARM Cortex-A35 scheint für Cross-Core bei circa 10 kHz zu liegen, da es die höchste theoretische Kanalkapazität erreicht. Aufgrund der sehr hohen Fehlerraten bei Same-Core für die getesteten Frequenzen lässt sich keine optimale Frequenz finden.

5.1.3 Flush+Flush auf dem ARM Cortex-A72

Die erwarteten Differenzen zwischen einem Hit und Miss bei `Flush+Flush` sind deutlich geringer als die Unterschiede bei `Flush+Reload` [GMWM16]. Aufgrund der geringeren Differenz ist `Flush+Flush` jedoch anfälliger für Fehler, da eine präzise Differenzierung zwischen Hits und Misses notwendig ist [GMWM16]. Diese Annahme lässt sich für das Same-Thread Szenario in 17 bestätigen, da Hit und Miss im Vergleich zu `Flush+Reload` in Abbildung 11 einen geringeren Abstand haben. Zusätzlich liegt der Schwellwert bei `Flush+Flush` bei circa 220 CPU Zyklen, während der Schwellwert bei `Flush+Reload` bei ungefähr 300 CPU Zyklen liegt. Da die Zeiten von Hit und Miss für `Flush+Flush` geringer als die von `Flush+Reload` sind, sollte `Flush+Flush` eine höhere Datenrate erreichen können als `Flush+Reload`. Diese Annahme lässt sich anhand der Werte in der Tabelle in Abbildung 18 jedoch nicht bestätigen. Die Fehlerrate für 10 kHz ist für Same-Core sowie für Cross-Core nah an 50%, was entgegen der Annahme zu einer niedrigen theoretische Kanalkapazität führt.

Die Hit und Miss Verteilung in Abbildungen 17 und 21 für jeweils 100 Hz und 10 kHz zeigen im Vergleich zu der Same-Thread Ausführung eine höhere Varianz. Diese hohe Varianz sorgt für eine Überschneidung der Verteilungen, was die Klassifizierung von Hit und Miss erschwert. Dies kann Grund für die höhere Fehlerrate in Abbildung 18 sein.

Ähnlich wie bei `Flush+Reload` ist die Schwellwert-Berechnung für 1 MHz nicht möglich, da sich die Verteilungen von Hits und Misses fast komplett überlappen. Da eine Klassifizierung von Hit und Miss nicht mehr möglich ist, wird eine Fehlerrate von 50% erwartet. Aufgrund der kleinen Testmenge wird die erwartete Fehlerrate nicht erreicht. Daher wird wie bei `Flush+Reload` vermutet, dass bei einer größeren Testmenge die Fehlerrate gegen 50% konvergiert und die Kanalkapazität somit gegen 0 bps fällt.

Für 100 Hz und 1 MHz auf Same-Core ist die erwartete Wasserstein-Distanz höher als die erreichte Wasserstein-Distanz. Dies deutet darauf hin, dass Insertion- und Deletion-Errors aufgetreten sind und der Bit String verschoben wurde. Ein verschobener Bit String kann mehrere Ursachen haben. Zum einen kann er durch Descheduling hervorgerufen werden. Jedoch entstehen durch die Implementierung, die in Abschnitt 3.7 beschrieben wurde, durch den Scheduler nur Substitution-Errors, womit dies ausgeschlossen ist. Eine andere Ursache könnte sein, dass bereits beim Senden der Preamble Bits ein Substitution-Error auftritt und die Preamble Bits zu früh oder zu spät erkannt werden. Somit wäre der gesamte empfangene Ethernet Frame verschoben. Die Klassifizierung von Cache Hit und Cache Miss ist, wie bei 100 Hz (siehe Abbildung 17), nicht präzise genug. Dadurch könnten wie oben beschrieben Substitution-Errors in den Preamble Bits aufgetreten sein. Diese Substitution-Errors werden in Abbildung 18 durch False Negatives und False Positives dargestellt.

Aufgrund der hohen Fehlerrate für Same-Core 100 Hz und 10 kHz ist das Bestimmen einer optimalen Frequenz nicht eindeutig. Die Überschneidungen in Abbildung 17 für 10 kHz sind verglichen mit den anderen Frequenzen nicht so stark. Daher wird vermutet, dass die optimale Frequenz bei ungefähr 10 kHz liegt. Cross-Core erreicht dennoch auf 100 Hz die höchste theoretische Kanalkapazität.

5.1.4 Flush+Flush auf dem ARM Cortex-A35

Der Schwellwert für das `Flush+Flush` Same-Thread Szenario auf dem ARM Cortex-A35 trennt Hits und Misses bis auf kleine Überschneidungen eindeutig (siehe Abbildung 19). Die durschnittliche Flush-Instruktion ist verglichen mit den Zeiten auf dem ARM Cortex-A72 sehr niedrig. Mit einem durchschnittlichen Flush-Miss von ungefähr 110 CPU Zyklen und einem durchschnittlichem Flush-Hit von circa 150 CPU Zyklen, ist die Flush-Instruktion des ARM Cortex-A35 knapp 100 CPU Zyklen schneller ausgeführt. Aus diesen beobachteten Werten wird erwartet, dass `Flush+Flush` auf dem ARM Cortex-A35 zu einer höheren Datenrate und, aufgrund der Überschneidungen in Abbildung 19, zu einer höheren Fehlerrate führt als auf dem ARM Cortex-A72.

In Abbildungen 19 und 22 ist aber entgegen der Annahme zu erkennen, dass für kein Same-Core oder Cross-Core Szenario zwischen Hits und Misses unterschieden werden kann. Die Verteilung von Hits und Misses überlappen sich fast komplett und eine Schwellwert-Berechnung ist nicht mehr möglich. Daraus folgend sind die Fehlerraten in Abbildung 20 nah an 50%.

Für `Flush+Flush` bietet sich keine der getesteten Frequenzen als optimale Frequenz an. Testes halber wurde `Flush+Flush` auf dem A35 mit einer Frequenz von 10 Hz und 1 Hz ausgeführt, aber die Ergebnisse waren entweder gleich oder noch schlechter als auf den höheren Frequenzen. Daraus lässt sich schließen, dass `Flush+Flush` zum Bau eines verdeckten Kanals auf dem ARM Cortex-A35 nicht geeignet ist.

5.1.5 Zusammenfassung der gemessenen Ergebnisse

In folgendem Abschnitt werden die besprochenen Ergebnisse aus Abschnitt 5.1 zusammengefasst. Von allen Szenarien ergab sich die höchste theoretische Kanalkapazität für `Flush+Reload` auf dem ARM Cortex-A72 mit einer Frequenz von 10 kHz und Cross-Core. Diese liegt bei 7.48 kbps und einer Fehlerrate von 9%. Im selben Szenario mit einer Same-Core Ausführung ist durch die hohe Fehlerrate von 45% eine deutlich geringere theoretische Kanalkapazität vorhanden. Dies kann sich

auf die beschriebene Implementierung von Nanosleep im Vergleich zu einer Busy-Waiting-Schleife in Abschnitt 3.7 rückführen lassen.

Verglichen mit der `Flush+Reload` Ausführung auf dem ARM Cortex-A35 erreicht der ARM Cortex-A72 eine mehr als doppelt so hohe theoretische Kanalkapazität. Die höchste theoretische Kanalkapazität auf dem A35 ist ebenfalls bei 10 kHz und Cross-Core, aber liegt nur bei 3.42 kbps. Somit lässt sich für beide Prozessoren zusammenfassen, dass eine Frequenz von ungefähr 10 kHz als optimale Frequenz gilt.

Die höchste Kanalkapazität von `Flush+Flush` auf dem ARM Cortex-A72 liegt mit 352 bps auf Same-Core und 10 kHz. Dies ist eine 21 Mal schlechtere Kapazität als die beste von `Flush+Reload`. Grund für die schlechtere Kapazität ist die erhöhte Fehlerrate von `Flush+Flush`, die durch die geringere Timing-Differenz zwischen Hit und Miss von `Flush` entsteht. Da die Schwellwert-Berechnung auf dem ARM Cortex-A35 für `Flush+Flush` für Same-Core sowie Cross-Core nicht möglich ist, ist die höchste theoretische Kanalkapazität dementsprechend schlechter als die auf dem A72. Die beste Kapazität auf dem A35 wird mit 10 kHz Cross-Core erreicht und beträgt 290 bps, was fast 12 Mal schlechter als auf dem A72 ist. Für beide Systeme gilt, dass die optimale Frequenz zwischen 100 Hz und 10 kHz liegt, da die Werte ab 10 kHz eine vergleichsweise hohe Fehlerrate haben.

Somit lässt sich zusammenfassen, dass `Flush+Reload` auf beiden Systemen besser performiert als `Flush+Flush`. Eine optimale Frequenz liegt bei ungefähr 10 kHz, während die optimale Frequenz von `Flush+Flush` bei einer Frequenz im Bereich von 100 Hz bis 10 kHz liegt. Da auf dem ARM Cortex-A72 Cache-Invalidierungen von Hardwareperformanzzählern erkannt werden können, ist `Flush+Flush` auf dem A72 erkennbar. Da, wie in Abschnitt 2.7 beschrieben, `Flush+Flush` halb so viele Aktionen ausführt, die von Hardwareperformanzzählern erkannt werden können, gilt `Flush+Flush` immer noch als schwieriger zu entdecken. Der ARM Cortex-A35 schließt grundsätzlich schlechter ab als der ARM Cortex-A72. `Flush+Reload` ist auf dem A35 realisierbar, aber verglichen mit dem ARM Cortex-A72 mit einer hohen Fehlerrate. Eine optimale Frequenz liegt bei unter 10 kHz, da die Fehlerrate noch relativ hoch ist. `Flush+Flush` auf dem A35 besitzt so hohe Fehlerraten, dass ein Kanal schwierig mit `Flush+Flush` zu implementieren wäre. Für eine optimale Frequenz eignet sich keine der gemessenen Frequenzen. In der Dokumentation zum ARM Cortex-A35 wird nicht spezifiziert, dass der Hardwareperformanzzähler Cache-Invalidierungen erkennen kann. Somit wäre der Vorteil von `Flush+Flush` auf dem A35, dass es von Hardwareperformanzzählern nicht erkannt werden kann.

5.2 Vergleich mit fremden Ergebnissen

In Abschnitt 5.1.5 wurden die Messergebnisse miteinander verglichen und zusammengetragen. In diesem Abschnitt werden die Ergebnisse mit den Resultaten von „ARMageddon: Cache Attacks on Mobile Devices“vorgezogen [LGS+16] verglichen.

Für den Vergleich von Kanälen basierend auf `Flush+Reload` und `Flush+Flush` muss sichergestellt werden, dass die getesteten CPUs eine ähnliche Architektur besitzen. Wichtige Faktoren, die zuvor verglichen werden müssen, sind unter anderem die Taktfrequenz, sowie die Implementationsdetails des Caches und Cache-Latenzen. Aufgrund der starken Einschränkung und der geringen Anzahl an veröffentlichten Arbeiten zu cachebasierten Seitenkanalangriffen auf ARM Prozessoren, fällt die Auswahl an Vergleichsarbeiten sehr kurz aus.

Die Arbeit von Lipp et al. „ARMageddon: Cache Attacks on Mobile Devices“benutzte ähnliche Testsyteme zu dieser, wodurch sie sich als Vergleichsarbeit anbietet [LGS+16]. Lipp et al. testeten unter anderem `Flush+Reload` und `Flush+Flush` auf ARM basierten CPUs. Das vorgestellte Testsystem ist ein Samsung Galaxy S6, welches aus den zwei Prozessoren ARM Cortex-53 und ARM Cortex-57 besteht. Mit etwas höheren Taktfrequenzen und einer ähnlichen Implementierung des Caches lässt sich das Samsung Galaxy S6 als Vergleichssystem darstellen. Eine Aussage zu den Cache-Latenzen kann nicht getroffen werden, da in der Dokumentation für den A35 und den A72 keine Latenzen angegeben sind. Zudem benutzen Lipp et al. dieselbe Zeitmessung (`perf_event_open`) und Flush-Instruktionen (`CIVAC`) [LGS+16] wie die Implementierung für diese Bachelorarbeit. Mit `Flush+Reload` und Cross-Core erreicht das S6 eine Kanalkapazität von 1.14 mbps und eine Fehlerrate von 1.10%. Mit `Flush+Flush` erreicht der Kanal eine Kapazität von 257 kbps mit einer Fehlerrate von 0.48%.

Verglichen mit den Ergebnissen dieser Arbeit erreicht Lipp et al. eine deutlich höhere Kanalkapazität. Gründe für eine 152 Mal bessere Kanalkapazität bei `Flush+Reload` als die dieser Bachelorarbeit, kann unter anderem sein, dass der Code von Lipp et al. besser optimiert wurde. Im Vergleich zu dem Code dieser Arbeit, benutzen Lipp et al. mehr Makros und Inline Code [Lip16a], was zu besserer Performance führen kann. Ein weiterer Faktor kann sein, dass das benutze Testsystem S6 eine bessere Leistung als der A35 und A72 erbringen kann. Trotz der höheren Kanalkapazitäten lassen sich Ähnlichkeiten zu den Resultaten aus Abschnitt 5.1.5 finden. Auf dem A72 hat `Flush+Flush` eine 21 Mal schlechtere Kapazität als `Flush+Reload` und auf dem A35 ist `Flush+Flush` 12 Mal schlechter als mit `Flush+Reload`. Dies ähnelt den Ergebnissen von Lipp et al., da `Flush+Flush` auf dem S6 eine 6 mal schlechtere Kanalkapazität erreicht als `Flush+Reload`. Jedoch ist die angegebene Fehlerrate für `Flush+Flush` auf dem S6 geringer als die für `Flush+Reload`. Dies gleicht sich nicht mit den Ergebnissen dieser Arbeit. Da die angegebene Kanalkapazität für `Flush+Flush` von 257 kbps geringer ist als die von `Flush+Reload` mit 1.14 mbps, folgt daraus, dass die Fehlerrate von `Flush+Flush` bei einer ähnlich hohen Frequenz von `Flush+Reload` stark ansteigt. Somit wäre die Fehlerrate von `Flush+Reload` im Vergleich zu `Flush+Flush` auf derselben Frequenz deutlich geringer. Diese Deduktion gleicht sich mit den Ergebnissen dieser Arbeit.

Zusammengefasst gleichen die Ergebnisse von Lipp et al. den Ergebnissen von dieser Arbeit. Auf den Testsystemen eignet sich `Flush+Reload` besser für hohe Kanalkapazitäten und `Flush+Flush` für einen besser versteckten Kanal, aber einer geringeren Datenrate.

6 Fazit und Ausblick

Ein Ziel dieser Arbeit war es mithilfe von `Flush+Reload` und `Flush+Flush` auf einem ARM basierten System einen verdeckten Kanal zu bauen. Dies ist in Kapitel 3 beschrieben. Ein weiteres Ziel war es die erstellten Kanäle miteinander zu vergleichen, sowie Messergebnisse mit anderen Arbeiten zu vergleichen. Dies führte zu folgenden Ergebnissen.

Es wurde gezeigt, dass `Flush+Reload` und `Flush+Flush` auf den ARMv8 Prozessoren Cortex-A72 und Cortex-A35 im selben Thread ausgeführt werden können. `Flush+Reload` erreicht in einem verdeckten Kanal eine theoretische Kanalkapazität von 7.22 kbps auf dem ARM Cortex-A72 und eine Kapazität von 3.42 kpbs auf dem ARM Cortex-A35. `Flush+Flush` hingegen erreicht nur eine theoretische Kanalkapazität von 352 bps auf dem ARM Cortex-A72. Auf dem ARM Cortex-A35 ließ sich `Flush+Flush` für einen verdeckten Kanal nicht nutzen, da eine Schwellwert-Berechnung in einem Same-Core und Cross-Core Szenario nicht möglich war. Grundsätzlich erzielte der Kanal in einem Cross-Core Szenario überwiegend bessere Kapazitäten als in einem Same-Core Szenario. Grund dafür ist eine zuverlässigere Synchronisierung, die nur für ein Cross-Core Szenario sinnvoll eingesetzt werden konnte (siehe Abschnitt 3.7). Grund dafür ist, dass die Synchronisierungsmethode den Kern, auf dem der Prozess ausgeführt wird, stark auslastet. Somit würde der Empfänger-Prozess die Performanz des Sender-Prozesses beeinträchtigen. Aufgrund der geringeren Fehlerrate und der somit höheren Kapazität von `Flush+Reload` eignet sich `Flush+Reload` besser für den Bau eines verdeckten Kanals als `Flush+Flush`. Der Vorteil von `Flush+Flush` ist jedoch, dass es von Hardwareperformanzzählern nicht erkannt werden kann. Somit bietet sich `Flush+Reload` für einen verdeckten Kanal mit höherer Datenrate an, während `Flush+Flush` mit geringerer Datenrate dann für einen besser versteckten Kanal geeignet ist.

Da der Fokus nicht auf die Optimierung der Kanalkapazität lag, bietet sich dies für weitere Forschung an. Dies kann zum Beispiel durch besser optimierten Code oder das Finden einer optimalen Frequenz erreicht werden. Eine optimale Frequenz wurde in Abschnitt 5.1 nur angedeutet, aber werden Tests auf mehreren Frequenzen durchgeführt, so kann sich der optimalen Frequenz genähert werden. Die optimale Frequenz ist dadurch definiert, dass es mit der erreichten Fehlerrate zu der höchsten Kanalkapazität führt.

Zudem gehört zu der Forschung von Sicherheitslücken immer die Suche nach Gegenmaßnahmen. In Abschnitt 2.7 wurden Gegenmaßnahmen vorgestellt. Als weiteres Forschungsziel bietet sich die Suche nach neuen Gegenmaßnahmen, sowie der Verringerung des Performanzverlust bei bereits bekannten Gegenmaßnahmen, an. Auf dem ARM Cortex-A72 können Cache-Invalidierungen und Cache-Zugriffe von Hardwareperformanzzählern erkannt werden. Somit ist auf dem A72 `Flush+Reload` und `Flush+Flush` erkennbar. Für weitere Forschungszwecke bietet sich an die Anzahl der Aktionen, die von Hardwareperformanzzählern erkannt werden können, zu reduzieren, sodass `Flush+Reload` und `Flush+Flush` nicht als Bedrohung klassifiziert werden.

Literaturverzeichnis

[Anl19] Anlauf, Joachim K.: Rechnerorganisation Uni Bonn. Vorlesung, 2019. – Online erhältlich unter https://www.ti.uni-bonn.de/teaching/ss19/rechnerorganisation; abgerufen im Oktober 2020.

[ARM15] ARM Limited Company: ARM Cortex-A Series Programmer's Guide for ARMv8-A. Website, 2015. – Online erhältlich unter https://developer.arm.com/documentation/den0024/a/; abgerufen im September 2020.

[ARM16] ARM Limited Company: ARM® Cortex®-A72 MPCore Processor. Website, 2016. – Online erhältlich unter https://developer.arm.com/documentation/100095/0003; abgerufen im Oktober 2020.

[ARM17] ARM Limited Company: ARMv8-A Memory systems. Website, 2017. – Online erhältlich unter https://developer.arm.com/documentation/100941/0100/Barriers; abgerufen im Oktober 2020.

[ARM19] ARM Limited Company: Arm Cortex-A35 Processor Technical Reference Manual. Website, 2019. – Online erhältlich unter https://developer.arm.com/documentation/100236/latest/; abgerufen im Januar 2021.

[ARM20] ARM Limited Company: Armv8.1-M Performance Monitoring User Guide. Website, 2020. – Online erhältlich unter https://developer.arm.com/documentation/arm051-799564642-251/latest; abgerufen im Januar 2021.

[Ben11] Bendersky, Eli: Position Independent Code (PIC) in shared libraries. Website, 2011. – Online erhältlich unter https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/; abgerufen im Februar 2021.

[Den96] Denning, Peter J.: BEFORE MEMORY WAS VIRTUAL. Website, 1996. – Online erhältlich unter http://160592857366.free.fr/joe/ebooks/ShareData/Before%20Memory%20was%20Virtual%20By%20Peter%20J.%20Denning%20from%20George%20Mason%20University.pdf; abgerufen im Oktober 2020.

[Fre01] Free Software Foundation: clock_gettime(3) - Linux man page. Website, 2001. –

Online erhältlich unter
https://linux.die.net/man/3/clock_gettime; abgerufen im Dezember 2020.

[Fre14] Free Software Foundation: taskset(1) — Linux manual page. Website, 2014. – Online erhältlich unter
https://man7.org/linux/man-pages/man1/taskset.1.html; abgerufen im Januar 2021.

[Fre20a] Free Software Foundation: clock_nanosleep(2) — Linux manual page. Website, 2020. – Online erhältlich unter
https://man7.org/linux/man-pages/man2/clock_nanosleep.2.html; abgerufen im Januar 2021.

[Fre20b] Free Software Foundation: mmap(2) — Linux manual page. Website, 2020. – Online erhältlich unter
https://man7.org/linux/man-pages/man2/mmap.2.html; abgerufen im Januar 2021.

[Fre20c] Free Software Foundation: perf_event_open(2) — Linux manual page. Website, 2020. – Online erhältlich unter
https://man7.org/linux/man-pages/man2/perf_event_open.2.html; abgerufen im Dezember 2020.

[GMWM16] Gruss, Daniel ; Maurice, Clémentine ; Wagner, Klaus ; Mangard, Stefan: Flush+Flush: A Fast and Stealthy Cache Attack. Detection of Intrusions and Malware, and Vulnerability Assessment pp 279-299, 2016. – Online erhältlich unter
https://gruss.cc/files/flushflush.pdf; abgerufen im Novemeber 2020.

[Gru17] Gruss, Daniel: Software-based Microarchitectural Attacks. IT-Informationtechnology Volume 60: Issue 5-6, 2017. – Online erhältlich unter
https://gruss.cc/files/phd_thesis.pdf; abgerufen im Oktober 2020.

[Hap21] Happe, Maurice: Verdeckte Kommunikation durch Seitenkanaleffekte auf ARM-Prozessoren - Gitlab. https://git.cs.uni-bonn.de/boes/ba_happe_2020, 2021. – Commit: 14cfde4a

[Hei20] Hein, Lennart: F+F Covert Channels. Rheinische Friedrich-Wilhelms-Universität Bonn, 2020. – Online erhältlich unter
https://ba.lennihein.com/BA_Hein_Redacted.pdf; Commit = 1cb2edd

[IEE12] IEEE Standards Association: IEEE 802.3-2012 - IEEE Standard for Ethernet. 2012. – Online erhältlich unter
https://standards.ieee.org/standard/802_3-2012.html; abgerufen im Januar 2021.

[Int16] Intel Corperation: Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 1. Website, 2016. – Online erhältlich unter
https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf; abgerufen im September 2020.

[Kle04] Kleitman, Daniel: Optimal transport for applied mathematicians. Vorlesung, 2004. – Online erhältlich unter
http://www-math.mit.edu/~djk/18.310/18.310F04/shannon_bound.html; abgerufen

im Januar 2021.

[Koc96] Kocher, Paul C.: Timing Attacks on Implementations of Diffie-Hellman , RSA, DSS, and Other Systems. Advances in Cryptology — CRYPTO '96: pp 104-113, 1996. – Online erhältlich unter https://paulkocher.com/doc/TimingAttacks.pdf; abgerufen im November 2020.

[LGS+16] Lipp, Moritz ; Gruss, Daniel ; Spreitzer, Raphael ; Maurice, Clémentine ; Mangard, Stefan: ARMageddon: Cache Attacks on Mobile Devices. 25th USENIX Security Symposium (USENIX Security 16) pp 549-564, 2016. – Online erhältlich unter https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_lipp.pdf; abgerufen im Novemeber 2020.

[Lip16a] Lipp, Moritz: ARMageddon: Cache Attacks on Mobile Devices. https://github.com/IAIK/armageddon, 2016. – Commit = 96ebc2d

[Lip16b] Lipp, Moritz: Cache Attacks on ARM. Graz University of Technology, 2016. – Online erhältlich unter https://mlq.me/download/master_thesis.pdf; abgerufen im Novemeber 2020.

[LZM20] LZMA benchmark: 7-Zip LZMA Benchmark ARM Cortex-A53. Website, 2020. – Online erhältlich unter https://www.7-cpu.com/cpu/Cortex-A53.html; abgerufen im Februar 2021.

[MB76] Metcalfe, Robert M. ; Boggs, David R.: Ethernet: distributed packet switching for local computer networks. Communications of the ACM 1976, 1976. – Online erhältlich unter https://dl.acm.org/doi/10.1145/360248.360253; abgerufen im Januar 2021.

[MF19] Martini, Peter ; Frank, Matthias: Vorlesung: Kommunikation in Verteilten Systemen Uni Bonn. Vorlesung, 2019. – Online erhältlich unter https://net.cs.uni-bonn.de/de/wg/kommunikationssysteme/lehre/ws-201920/kommunikation-in-verteilten-systemen/; abgerufen im Januar 2021.

[MWS+17] Maurice, Clémentine ; Weber, Manuel ; Schwarz, Michael ; Giner, Lukas ; Gruss, Daniel ; Boano, Carlo A. ; Römer, Kay ; Mangard, Stefan: Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. Network and Distributed System Security Symposium, 2017. – Online erhältlich unter https://cmaurice.fr/pdf/ndss17_maurice.pdf; abgerufen im Januar 2021.

[OST06] Osvik, Dag A. ; Shamir, Adi ; Tromer, Eran: Cache Attacks and Countermeasures: the Case of AES. Topics in Cryptology – CT-RSA 2006: pp 1-20, 2006. – Online erhältlich unter https://eprint.iacr.org/2005/271.pdf; abgerufen im November 2020.

[Rad18] Radxa Limited: Rock Pi S: Getting started. Website, 2018. – Online erhältlich unter https://wiki.radxa.com/RockpiS/getting_started; abgerufen im Januar 2021.

[Ras20] Raspberry Pi Foundation: Raspberry Pi 4 Tech Specs. Website, 2020. – Online erhältlich unter https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/; abgerufen im Januar 2021.

[Sai00] Saikkonen, Riku: IO-Port-Programming. Website, 2000. – Online erhältlich unter https://tldp.org/HOWTO/IO-Port-Programming-4.html; abgerufen im Januar 2021.

[San15] Santambrogio, Filippo: Optimal transport for applied mathematicians. Birkhäuser, Cham, 2015. – Online erhältlich unter https://link.springer.com/book/10.1007%2F978-3-319-20828-2; abgerufen im Januar 2021.

[Ste20] Steinhage, Volker: Grundlagen der Künstlichen Intelligenz Uni Bonn. Vorlesung, 2020. – abgerufen im Februar 2021.

[YF14] Yarom, Yuval ; Falkner, Katrina: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. 23rd USENIX Security Symposium, 2014. – Online erhältlich unter https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom; abgerufen im November 2020.

Abbildungsverzeichnis

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 2021

_____

Maurice Happe