Verdeckte Cache Seitenkanäle

Bericht zur Projektgruppe

ausgearbeitet von

Sabrina Heidler
[redacted]

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

im Studiengang
Informatik (B.Sc.)

Erstprüfer: [redacted]
Universität Bonn

Zweitprüfer: [redacted]
Universität Bonn

Betreuer: Dr. Felix Jonathan Boes
Universität Bonn

Bonn, 2. Oktober 2020

Inhaltsverzeichnis

1 Einleitung 1

2 Grundlagen 2
- 2.1 Was ist ein Seitenkanal . . . 2
- 2.2 Optimierungen des Prozessors . . . 2
- 2.3 Funktionsweise von Hardwarecaches . . . 3
- 2.4 Zusammenhang zwischen Virtual und Shared Memory . . . 6

3 Aktueller Stand der Forschung 8
- 3.1 Historischer Verlauf . . . 8
- 3.2 Bekannte Strategien zu Cache Seitenkanälen . . . 8

4 Methodisches Vorgehen 11

5 Umsetzung und Evaluation 13
- 5.1 Setup . . . 13
- 5.2 Threshold . . . 13
- 5.3 Synchronisation . . . 14
  - 5.3.1 Empfangsfenster . . . 15
  - 5.3.2 Sendefenster . . . 16
- 5.4 Evaluation der Übertragungsqualität . . . 16

6 Fazit und Ausblick 18

Literaturverzeichnis 19

Abbildungsverzeichnis 22

Listing 23

1 Einleitung

In den letzten zehn Jahren haben cachebasierte Seitenkanal Angriffe, insbesondere auf Intel x86-CPUs, zunehmend an Aufmerksamkeit in Fachkreisen erlangt [Lip16a]. Dabei sind leistungsstarke Techniken entwickelt worden, die verdeckte Cache Seitenkanäle ausnutzen. Dadurch ist es beispielsweise möglich, an geheime Informationen zu gelangen oder übliche Mechanismen des Betriebssystems zur Kontrolle der Interprozesskommunikation zu umgehen. Dies geschieht, indem kleinste Veränderungen von Zuständen der Hardware auf mikroarchitektureller Ebene beobachtet werden. Unter anderem wegen der 2018 veröffentlichten Angriffe Meltdown [Lip18] und Spectre [Koc19], die cachebasierte Seitenkanäle nutzen, ist ein aktives neues Forschungsgebiet entstanden. Da die dabei entdeckten Verwundbarkeiten insbesondere Intel, AMD und ARM Prozessoren betreffen und diese in Milliarden von Computern verbaut sind, kann eine weitere Erforschung dieser Thematik Relevanz für eine Vielzahl von Nutzern haben [Can19].

Ziel dieser Projektgruppenarbeit ist die Untersuchung cachebasierter Seitenkanalangriffe. Insbesondere soll dabei die Strategie Flush + Flush betrachtet und angewandt werden, um unter geschickter Verwendung des Caches als Seitenkanal einen vom Betriebssystem nicht vorgesehenen Kommunikationskanal zwischen zwei Prozessen zu eröffnen. Dazu werden aus bereits bestehenden Implementierungen [Lip16b] [Gru16b], die im Zuge der Arbeit *ARMageddon: Cache Attacks on Mobile Devices* [Lip16a] und *Flush+ Flush: a fast and stealthy cache attack* [Gru16a] entstanden sind, relevante Code-Abschnitte extrahiert. Anhand dessen wird die Ableitung von Informationen aus verwendeten Daten im Cache getestet und evaluiert. In einem Sender-Empfänger Szenario sollen wiederholt mehrere Bits übertragen und darauf basierend eine Chat-ähnliche Anwendung realisiert werden.

Für die Evaluation sollen unter anderem die benötigte Zeit bzw. CPU-Zykel, die einen Cache Hit oder einen Cache Miss minimal bzw. maximal charakterisieren, betrachtet werden. Um die Performance der Seitenkanal Chat-Anwendung zu messen, werden Fehlerraten, Genauigkeit der Erkennung von Bits bzw. Zeichen und die Kapazität des Seitenkanals, also wie viele Bits bzw. Zeichen pro Zeiteinheit übertragen werden können, analysiert.

2 Grundlagen

Da zur Eröffnung eines verdeckten Seitenkanals durch Caches einiges an Hintergrundwissen benötigt wird, werden in diesem Kapitel die notwendigen Grundlagen dazu behandelt. Es wird zunächst darauf eingegangen, was überhaupt ein Seitenkanal ist (Abschnitt 2.1). Anschließend wird kurz die CPU behandelt und welche in diesem Kontext relevanten Optimierungen zur schnelleren Befehlsausführung entwickelt wurden (Abschnitt 2.2). Da der zentrale Aspekt dieser Arbeit die Ausnutzung von Cache Funktionalitäten ist, werden Grundlagen dazu im notwendigen Detailgrad besprochen (Abschnitt 2.3). Auch ein Teilverständnis von Virtual Memory und Shared Memory ist nötig, insbesondere, da Shared Memory für die Eröffnung des hiesigen Seitenkanals erforderlich ist. Daher werden zuletzt Zusammenhänge und Zweck dieser Speicherkonstrukte beschrieben (Abschnitt 2.4).

2.1 Was ist ein Seitenkanal

Ein Seitenkanal ist ein Kommunikationskanal, dessen Existenz verborgen oder verdeckt ist. Er eröffnet die Möglichkeit, geheime Informationen abzuleiten oder weiter zu geben, indem ein bestehender Kanal auf eine vom Betriebssystem oder von Hardware Herstellern unerwünschte Weise genutzt wird. Durch das Klickgeräusch eines mechanischen Zahlenschlosses den Code herzuleiten zählt vermutlich zu einem der bekanntesten Seitenkanäle in der analogen Welt. So hat bei einem Computer beispielsweise eine kryptografische Berechnung bestimmte Einflüsse auf Systemressourcen, sodass daraus geheime Informationen aus Verschlüsselungsvorgängen hergeleitet werden können, wie auch Paul C. Kocher in seiner Arbeit beschrieben hat [Koc96]. In der Vergangenheit wurden Seitenkanalangriffe auf Grundlage praktisch aller messbaren Umweltveränderungen, wie Energieverbrauch, elektromagnetischer Strahlung, Temperatur, akustischen Emissionen und vielen weiteren durchgeführt [Gru17].

Der hier betrachtete software-basierte mikroarichtekturelle Seitenkanal ist ein sogenannter Timing Angriff, der Zeit- und Verhaltensunterschiede ausbeutet und der durch Optimierungen der Hardwarestrukturen erst möglich gemacht wurde [Gru17]. Um diese Timing Attacks genauer zu verstehen, müssen jedoch zuerst einige Grundlagen zu den relevanten Hardwarestrukturen behandelt werden.

2.2 Optimierungen des Prozessors

Alle Computer haben eine Central Processing Unit (CPU), die dafür verantwortlich ist, dass ein Computer ein Programm ausführen und die Daten korrekt verarbeiten kann. Jeder Computer enthält eine interne Uhr, die durch Takte regelt, wie schnell Anweisungen ausgeführt werden können. Diese Uhr synchronisiert alle Komponenten des Systems, sodass auch der Prozessor eine

feste Anzahl an Taktzyklen benötigt, um jeden Befehl auszuführen. Daher wird der Durchsatz oft in Taktzyklen und nicht in Sekunden gemessen. [Nul14]

Aufgrund diverser technischer Errungenschaften war es im Laufe der Zeit möglich die Verarbeitungsgeschwindigkeit von Befehlen zu erhöhen. Zu den Optimierungen gehören unter anderem eine Erhöhung der Taktfrequenz, Pipelining, die Verwendung von mehreren CPU-Kernen, spekulative Ausführung oder Out-Of-Order Ausführung. Pipelining bezeichnet das Aufteilen von Instruktionen in verschiedene Phasen, sodass bestimmte Berechnungen parallel ausgeführt werden können. Bei der spekulativen Ausführung macht der Prozessor eine Vorhersage über die weitere Programmausführung und führt für diese bereits Berechnungen durch. War diese Vorhersage korrekt, liegen die berechneten Informationen bereits vor, wenn sie benötigt werden. Andernfalls verwirft die CPU die Ergebnisse wieder. Die Ausführung von Befehlen, die von noch zu berechnenden Daten abhängen zu verzögern und zuerst andere Anweisungen ohne ausstehende Datenabhänigkeiten auszuführen bezeichnet man als Out-Of-Order Execution. [Gru17]

Mit immer performanter werdenden Prozessoren, wurden die langsamen Zugriffszeiten auf den physischen Hauptspeicher (DRAM) zum neuen Flaschenhals. Aus diesem Grund wurden Caches für CPUs eingeführt. Es gibt auch noch andere Formen von Caches, jedoch liegt hier der Fokus auf der Betrachtung von CPU Hardwarecaches.

2.3 Funktionsweise von Hardwarecaches

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

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

Abbildung 1: *Die Aufteilung der Bits der Hauptspeicheradresse in die drei Felder Tag, Index und Offset*

und Offset statt, wie auch in Abbildung 1 zu sehen ist. Für die Adressierung von Fully-Associative Caches wird die Adresse nur in Tag und Offset unterteilt. [Nul14]

Bei einer Unterteilung in drei Bereiche werden die mittleren n Bits der Speicheradresse als Cache-Index verwendet, der dem Prozessor mitteilt, an welcher Stelle im Cache nach den entsprechenden Daten gesucht werden soll. Verschiedene Adressen mit denselben mittleren n Bits, bezeichnet man als kongruent, da sie derselben Speicherstelle zugeordnet werden. Somit konkurrieren die DRAM-Speicherbereiche um ihren Platz im Cache[1]. Die jeweilige Speicherstelle im Cache besteht aus einem Tag und den gepufferten Daten. Hauptspeicher und Cache sind beide in gleich große Blöcke unterteilt. Werden Daten im Cache zwischengespeichert, befindet sich an der jeweiligen Speicherstelle immer der gesamte Datenblock. Die niedrigsten b Bits der Adresse werden als Offset innerhalb dieser im Cache gespeicherten Daten verwendet. Die Anzahl der für den Offset vorgesehen Bits bestimmt mit 2^b die Größe der Daten, die zwischengespeichert werden können. Die meisten modernen Prozessoren haben pro Speicherstelle eine Größe von 64 Byte, sodass mit 6 Offset Bits 2^6 Byte an zugehörigen Daten gespeichert werden können [Gru17]. Der Tag wird dazu verwendet, um festzustellen, ob eine Speicherstelle im Cache derzeit die Daten einer bestimmten Speicheradresse enthält, indem die vordersten t Bits der Hauptspeicheradresse als Tag mit dem im Cache hinterlegten Tag verglichen werden. [Nul14]

Es wird zwischen den folgenden drei Arten von Caches unterschieden:

Directly-Mapped Caches:
Ein Directly-Mapped Cache, ist die einfachste Form eines Caches. Die Speicherstelle im Cache, an der die Daten zwischengespeichert werden, nennt man hier Cache Line. Wird nach einer bestimmten Adresse im Cache gesucht, schaut der Prozessor an der durch den Cache-Index vorgegebenen Stelle, ob die Tags der Adresse und der Cache Line übereinstimmen. Je nach Ergebnis entsteht so ein Cache Miss oder ein Cache Hit. [Gru17]

Ein großes Problem der Directly-Mapped Caches ist, dass sie nur eine einzige Speicherstelle aus allen möglichen kongruenten Speicherstellen für einen bestimmten Index in der jeweiligen Cache Line speichern können. Arbeitet der Prozessor an mehr als einer kongruenten Speicherstelle, verdrängen diese sich gegenseitig aus dem Cache, sodass viele Cache Misses entstehen und der Performancevorteil der Verwendung eines Caches verloren geht. [Gru17]

Fully-Associative Caches:
Anstatt für jeden Hauptspeicherblock einen vordefinierten Speicherort im Cache anzugeben, wie es bei Directly-Mapped Caches der Fall ist, kann bei einem Fully-Associative Cache der jeweilige Inhalt an beliebiger Stelle zwischengespeichert werden[1]. Es werden daher keine Bits für die Indizierung einer Cache Line benötigt, sodass die Bits der Adresse nur in Tag und Offset aufgeteilt werden. Die

[1] Falls kein freier Speicher mehr im Cache vorhanden ist, muss neuer Platz geschaffen werden. Dafür verwendet man sogenannte Verdrängungsstrategien, die für den Kontext dieser Arbeit jedoch nicht weiter wichtig sind.

einzelnen Einträge des Caches werden hier nicht Cache Line, sondern Cache Way, genannt, da es sich um einen assoziativen Cache-Speicher ohne Reihenfolge handelt. [Gru17]

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

Set-Associative Caches:
Ein Set-Associative Cache stellt eine Kombination aus den beiden zuvor genannten Varianten dar, der Cache Sets anstelle von Cache Lines adressiert. Die Adressierung von Speicherstellen innerhalb des Caches ähnelt somit der des Directly-Mapped Caches, indem die Hauptspeicheradresse verwendet wird, um den Speicherblock einer bestimmten Position im Cache zuzuordnen[1]. Jedes Cache Set beinhaltet eine feste Anzahl an Ways, also assoziative Speicher ohne Reihenfolge, sodass ein Cache Set mehrere kongruente Adressen beinhalten kann. Hat ein Cache Set eine feste Anzahl m an Ways, so nennt man den Cache-Speicher m-Way Set-Associative Cache. Die Anzahl der im Cache verfügbaren Sets wird in der Regel nicht mit angegeben. In Abbildung 2 ist eine Modellierung eines 3-Way Set-Associative Caches zu sehen. Wie bei einem Fully-Associative Cache kann eine Speicherstelle innerhalb des Cache Sets jedem Way zugeordnet werden. [Nul14]

Um eine bestimmte Speicherstelle im Cache zu finden, wird ein zweistufiger Prozess angewandt. Zuerst wird anhand des Cache-Index das gesuchte Cache Set bestimmt. Anschließend wird der Tag der Hauptspeicheradresse mit all den im Set befindlichen Tags der Ways verglichen, um so, im Falle eines Cache Hits, die gesuchten Daten zu erhalten. Je mehr Ways ein Speicher enthält, desto teurer ist er. Da Set-Associative Caches in der Regel nur wenige Ways in ihren Sets enthalten, ist diese Variante eines Caches ein Kompromiss aus Kosten und Performance. [Gru17]

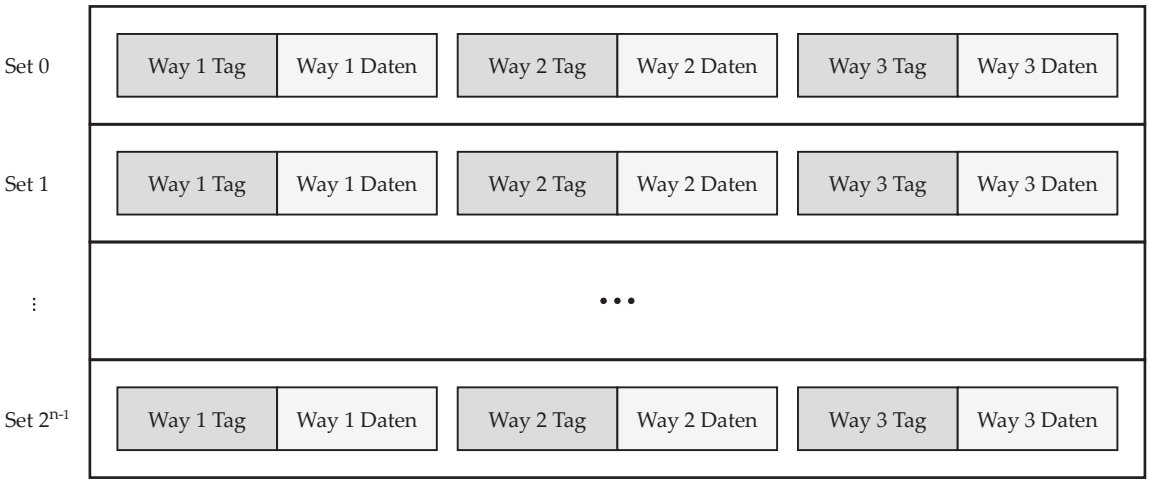


Abbildung 2: *Ein 3-Way Set-Associative Cache. Entsprechend der Anzahl n an Index Bits, gibt es 2^n Cache Sets.*

Moderne Intel Prozessoren haben verschiedene Cache-Level, typischerweise drei, die sich in Größe und Schnelligkeit unterscheiden und hierarchisch aufgebaut sind. Der L1 Cache ist der kleinste und schnellste, während der L3 Cache, auch Last-Level Cache genannt, größer und langsamer ist. Die L1 und L2 Caches sind private Caches, sodass sie in einer Architektur mit mehreren

CPU-Kernen nicht mit anderen Kernen geteilt werden. Der L3 Cache wird unter allen Kernen des Prozessors geteilt und ist inklusiv, umfasst also auch den L1 und L2 Cache. Das heißt, dass alle Daten und Instruktionen, die im L1 und L2 Cache vorhanden sind auch im Last-Level Cache enthalten sind. Im Umkehrschluss bedeutet dies ebenso, dass Cache Lines die aus dem Last-Level Cache verdrängt werden auch aus dem L1 und L2 Cache verdrängt werden. Die verwendeten Caches sind in der Regel Set-Associative Caches[Int19]. [Mau17]

Um die Performance zu verbessern, ist der Last-Level Cache bei Intel Prozessoren in Slices aufgeteilt, die jedem Kern seinen eigenen Bereich zuweisen. Diese Bereiche sind durch einen Ringbus miteinander verbunden, sodass allen Kernen der Zugriff auf jeden Bereich ermöglicht wird. Jeder CPU-Kern bekommt gleich viel Speicher mit nicht zwangsläufig zusammenhängenden Bereichen zugewiesen, auf denen der Zugriff des jeweiligen Kerns minimal schneller ist. Wie die Zuordnung der physischen Adressen zu einem bestimmten L3 Cache Speicherbereich geschieht bzw. welche Adressen vom jeweiligen Kern bevorzugt werden, ist vom Hersteller nicht dokumentiert. Jedoch wurde die sogenannte *Complex Addressing Function* für diese Adresszuordnung von Wissenschaftlern kürzlich Reverse-Engineered [Mau15][Yar15]. [Gru17]

2.4 Zusammenhang zwischen Virtual und Shared Memory

Virtual Memory erzeugt einen virtuellen Adressraum, der jedem Prozess vollständig für sich alleine zur Verfügung steht. Aus Sicht des Prozesses wird dieser Speicher genauso behandelt wie physischer Speicher. Der gesamte virtuelle und physische Speicher wird in gleich große Blöcke unterteilt, die Page genannt werden und eine feste Größe von meist 4 KB haben. Jeder Prozess hat seine eigene Page Table, die Informationen zum aktuellen Speicherort der Page beinhaltet und bekommt unabhängig von anderen Prozessen die benötigten Pages während seiner Ausführungszeit in den Hauptspeicher geladen. Prozesse kennen somit nur ihren eigenen Adressraum, sodass es möglich ist, aktive Programme besser voneinander zu isolieren, da somit keine unberechtigten Zugriffe auf den Adressraum eines anderen Prozesses geschehen können. Zudem erleichtern virtuelle Adressen auch die Organisation von Programmen mit statischen Adressen, da mehrere Programme gleichzeitig an derselben virtuellen Speicherstelle ihres virtuellen Adressraums liegen können, obwohl sie sich eigentlich an unterschiedlichen physischen Adressen befinden. Der Compiler kann die virtuellen Adressen eines Programms somit unabhängig von den benötigten Adressen anderer Programme berechnen, da sie nur für den jeweiligen Prozess gelten. Durch Virtual Memory und Swapping[2] ist es außerdem möglich, die Festplatte als Erweiterung des DRAMs zu verwenden, sodass auch Programme ausführt werden können, die mehr Speicher benötigen, als der Hauptspeicher zur Laufzeit zur Verfügung hat [Nul14]. [Tan15]

Durch Paging und Swapping können Teile des Programms je nach Bedarf in den Hauptspeicher gebracht werden[2], sodass es nicht notwendig ist ein Programm in zusammenhängenden Bereichen des Hauptspeichers abzulegen. Da ein Programm somit ungeordnet und zerstückelt gespeichert werden kann, müssen virtuelle Programmadressen in physische Speicheradressen übersetzt werden. Ein solcher Übersetzungsvorgang[2] ist in der Regel mehrstufig und kann sehr aufwendig und zeitintensiv sein. Darum wurde, ähnlich wie bei der Übersetzung einer Hauptspeicheradresse in eine Cache Speicherstelle, ein Pufferspeicher eingeführt, der vorherige Adressübersetzungen

[2] Da Paging Strategien für den Kontext dieser Arbeit nicht relevant sind, wird nicht näher darauf eingegangen.

zwischenspeichert. Der sogenannte Translation Lookaside Buffer (TLB), beinhaltet neben der Adressübersetzung auch weitere Informationen zu der jeweiligen Page, wie zum Beispiel Berechtigungen, also ob darauf lesend, schreibend oder ausführend vom Prozess zugegriffen werden darf. [Tan15]

Zur Reduzierung der physischen Speicherauslastung, sowie der Auslastung des TLBs verwenden Betriebssysteme Shared Memory. Von mehreren Prozessen genutzte Pages werden dadurch nur einmal in den physischen Speicher geladen und können von den aktiven Prozessen, die gerade darauf zugreifen wollen, gleichzeitig genutzt werden. Um die Isolation zwischen Prozessen aufrecht zu erhalten, sind die so geteilten Daten nur mit lesendem oder Copy-On-Write Zugriff verfügbar. Das Context-Aware Sharing identifiziert identische Pages, anhand des Speicherorts. Auf diese Weise können mehrfache Kopien vermieden werden, sodass beispielsweise gemeinsam genutzte Bibliotheken (Shared Libraries) nur einmal in den Speicher geladen und zwischen Prozessen geteilt werden. [Yar14]

Speicher zwischen völlig unabhängigen Prozessen zu teilen, bringt jedoch Sicherheits- und Datenschutzbedenken mit sich, die, wie sich im weiteren Verlauf zeigen wird, für das hiesige Seitenkanal Szenario ausgenutzt werden können.

3 Aktueller Stand der Forschung

In diesem Kapitel wird auf die Entwicklung des Forschungsfelds zu cachebasierten Seitenkanälen und auf verwandte Arbeiten eingegangen. In Abschnitt 3.1 wird der Historische Verlauf beleuchtet. Die im Kontext relevanten Strategien zu Cache Seitenkanälen werden in Abschnitt 3.2 beschrieben.

3.1 Historischer Verlauf

Der erste verdeckte Cache Seitenkanal wurde 1992 von Hu [Hu92] theoretisch nachgewiesen. Percival [Per05] war der Erste, der auch praktisch einen verdeckten Kanal auf dem L1 Cache erzeugte. Die ersten Erforschungen von cachebasierten Seitenkanälen wurden zunächst auf kryptografischen Algorithmen durchgeführt, wie die theoretischen Angriffe von Kocher [Koc96] und Kelsey u. a. [Kel98] zeigen. Im weiteren Verlauf führten Page [Pag02], sowie Tsunoo u. a. [Tsu03] praktische Angriffe auf DES durch. Im Jahr 2004 zeigte Bernstein [Ber05] dies auch für den nachfolgenden Verschlüsselungsalgorithmus AES. Während zahlreichen Studien von Cache Seitenkanälen, wurden in Fachkreisen Strategien entwickelt, die sich die verschiedenen Architekturen und damit verbundene Möglichkeiten und Restriktionen besser zunutze machen. So wurde beispielsweise der L1 Cache angegriffen [Gul11], auf Basis von Shared Memory der L3 Cache ausgebeutet [Yar14] oder Seitenkanäle in Cloud Umgebungen aufgebaut. Ristenpart u. a. [Ris09] demonstrierten dies zuerst. 2017 wurden von Maurice u. a. [Mau17] Seitenkanäle in Cloud Umgebungen noch verfeinert. Sogar Angriffe auf modernen mobilen Endgeräten konnten auf Basis von cachebasierten Seitenkanälen bereits durchgeführt werden, wobei Bildschirmberührungen, Tastenanschläge, sowie die Länge der auf dem Touchscreen eingegebenen Wörter vom Angreifer erfasst werden konnten [Lip16a]. Im Jahr 2018 wurden die Angriffe Meltdown [Lip18] und Spectre [Koc19] vorgestellt, die auf Ausnutzung von Out-Of-Order Execution und spekulativer Ausführung beruhen und auf Cache Seitenkanälen basieren. Da durch das Bekanntwerden dieser Verwundbarkeiten bereits einige weitere entdeckt werden konnten, hat dies ein aktives neues Forschungsfeld eröffnet [Can19]. Bisher sind jedoch keine Publikationen von Malware bekannt, die Techniken cachebasierter Seitenkanäle in allgemeiner Form ausnutzen. Auf einen Teil der verwendeten Strategien zu cachebasierten Seitenkanälen, unter anderem auch die von Gruss u. a. vorgestellte Strategie Flush + Flush [Gru16a], wird im Folgenden eingegangen.

3.2 Bekannte Strategien zu Cache Seitenkanälen

Cachebasierte Seitenkanäle bauen auf der Ausnutzung von Timing Unterschieden auf, sodass ein Zeitunterschied zu messen ist, je nachdem ob die benötigten Daten gecached sind oder nicht. Sind die Daten nicht im Cache vorhanden, ist die Zugriffszeit, aufgrund größerer Latenzzeiten beim Zugriff auf den Hauptspeicher, deutlich höher.

Es gibt im Allgemeinen die drei klassischen Strategien Prime + Probe, Evict + Time [Osv06] und Flush + Reload [Yar14] zur Eröffnung eines verdeckten Kanals durch Hardwarecaches. Zu all diesen Strategien existieren zahlreiche Variationen, wovon im weiteren Verlauf Flush + Flush, eine Abwandlung von Flush + Reload, näher betrachtet wird.

Flush + Reload:
Diese Strategie zielt auf den Last-Level Cache. Da dieser von allen Prozessor-Kernen geteilt wird, müssen der Sender- und der Empfängerprozess nicht zwangsläufig auf demselben Kern ausgeführt werden. Für diese Strategie wird Shared Memory, insbesondere Context-Aware Sharing, benötigt. Die Flush + Reload Strategie besteht aus drei Phasen, die in der Regel mehrfach wiederholt werden:

Annahmen und Vorraussetzungen:

- Sender und Empfänger befinden sich auf gleichem physischen Gerät
- Das OS verwendet einen Hardware Cache als CPU-Puffer
- Sender und Empfänger benutzen den Last-Level Cache
- Shared Memory wird verwendet
- Die `clflush` Instruktion ist auf der Architektur nutzbar

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

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

Flush + Flush:
Flush + Flush nutzt die gleichen Hard- und Software-Eigenschaften aus wie Flush + Reload. Das heißt, diese Variation benötigt ebenso wie Flush + Reload unter anderem Shared Memory. Diese Strategie besteht effektiv nur aus einer einzigen Phase, die für das mehrfache Auslesen eines Speicherbereichs mittels `clflush` Anweisung in einer Endlosschleife ausgeführt wird. Natürlich muss in der Zwischenzeit auch hier genug Zeit für den Sender zur Verfügung stehen, um seine Berechnungen zwischen den Flush + Flush Iterationen durchzuführen. [Gru16a]

```
1 sync_with_receiver()  //Synchronisation
2 while true            //Endlosschleife
3 do_computation()      //Durch Berechnungen werden  ggf. jew. Daten in Cache geladen
4 wait_for_receiver()   //Zeit für Empfänger zum auslesen der Daten
```

Listing 3.1: *Pseudocode Flush + Flush Sender*

```
1 sync_with_sender()                              //Synchronisation
2 while true                                      //Endlosschleife
3   time_start=measure_time()                     //beginne Zeitmessung
4   flush(address)                                //leere Speicherstelle
5   time_needed=measure_time()-time_start         //berechne benötigte Zeit zum flushen
6   cache_status=check_hit_or_miss(time_needed)   //prüfe anhand Zeit ob Hit oder Miss
7   process_data(cache_status)                    //verarbeite Hit/Miss Information
8   wait_for_sender()                             //Zeit für Sender Berechnungen
```

Listing 3.2: *Pseudocode Flush + Flush Empfänger*

Dieses Verfahren baut auf der Beobachtung auf, dass der `clflush` Befehl im Fall eines Cache Misses vorzeitig abbricht. Bei einem Cache Hit müssen auch alle anderen hierarchisch abhängiger Caches geleert werden. Da hiermit ein Seitenkanal auf dem geteilten Last-Level Cache aufgebaut wird und dieser inklusive ist, müssen somit bei einem Cache Hit auch die entsprechenden Einträge aus dem L1 und L2 Cache entfernt werden. Das kostet im Vergleich zu einem Abbruch selbstverständlich deutlich mehr Zeit. Diese Strategie beruht also einzig und allein auf der Ausführungszeit des `clflush` Befehls. [Gru16a]

Da das Leeren des Caches, um zu überprüfen, ob ein Speicherbereich geladen ist, gleichzeitig auch die Vorbereitung für die nächste Iteration darstellt, können die einzelnen Durchläufe mit einer höheren Frequenz stattfinden. Daher ist Flush + Flush schneller als alle bisher bekannten Cache Strategien zur Eröffnung von Seitenkanälen. Mit 496 KB/s in einem verdeckten Cross-Core Kanal ist er 6,7 mal schneller als jede zuvor veröffentlichte Strategie. [Gru16a]

Im Gegensatz zu anderen cachebasierten Seitenkanal Strategien findet bei Flush + Flush auf der Empfänger Seite kein direkter Speicherzugriff statt, da von ihm keine Speicherbereiche geladen werden. Folglich werden vom Empfänger auch keine Cache Misses verursacht, wodurch State-Of-The-Art Erkennungsmechanismen und Gegenmaßnahmen des Systems zu Cache Angriffen unwirksam sind. Flush + Flush ist somit besonders unauffällig. Zudem löst es keine Prefetches aus und ermöglicht so die Überwachung mehrerer Adressen innerhalb eines Page Speicherbereichs, im Gegensatz zu Flush+Reload, das in diesen Szenarien fehlschlägt. [Gru16a]

Flush + Flush hat von Natur aus eine etwas geringere Genauigkeit als Flush + Reload, aufgrund geringerer Zeitdifferenzen zwischen einem Cache Hit und einem Cache Miss und wegen einer im Durchschnitt geringeren Zugriffszeit. Während andere Strategien also einen deutlich größeren Unterschied an CPU-Zykeln zwischen einem Cache Hit und einem Cache Miss präsentieren, liegt die messbare Anzahl an verwendeten CPU-Zykeln bei Flush + Flush deutlich näher aneinander. Dies liegt mitunter daran, dass bei Flush + Flush die großen Latenzzeiten beim Zugriff auf den DRAM nicht zum tragen kommen, da vom Empfänger keine Speicherbereiche in den Cache geladen werden. Außerdem ist die Ausführungszeit der `clflush` Instruktion im Allgemeinen geringer, als beim Laden von Daten, ob nun gecached oder nicht. Somit kann es hier leichter passieren, dass, aufgrund der nah aneinander liegenden Werte der verbrauchten Zeit, fälschlicherweise auf einen Cache Hit bzw. Cache Miss geschlossen wird. [Gru16a]

4 Methodisches Vorgehen

Es wird die Flush + Flush Strategie verwendet, um über den Cache einen verdeckten Seitenkanal aufzubauen. Auf Basis dessen wird durch gezielte und wiederholte Übertragung einzelner Bits in einem Sender-Empfänger Szenario ein verdeckter Kommunikationskanal zwischen zwei Prozessen eröffnet.

Um einen Seitenkanal auf Basis der Flush + Flush Strategie zu eröffnen, muss der in Listing 3.1 und 3.2 vorgestellte Algorithmus zunächst implementiert werden. Dazu werden aus bereits vorhandenen Quellen [Lip16b][Gru16b] relevante Codeabschnitte extrahiert und ein funktionsfähiges Programm aufgebaut. Für dieses Szenario wird ein Sender und ein Empfänger benötigt. Beide verwenden eine oder mehrere bestimmte gemeinsame Speicherstellen über die kommuniziert wird. Der Sender greift je nach zu übermittelnden Daten auf eine solche Speicherstelle zu, sodass diese in den Cache geladen wird. Der Empfänger flusht die entsprechende Position im Cache, um darüber verdeckte Informationen zu erhalten. Hat der Sender Daten an die vereinbarte Stelle geladen, repräsentiert dies eine „1“. Befinden sich keine vom Sender abgelegten Daten an dieser Position, stellt dies eine „0“ dar.

Damit eindeutig eine „1“ bzw. eine „0“ erkannt werden kann, muss klar zwischen Cache Hits und Cache Misses unterschieden werden können. Dazu werden die CPU-Zykel, die bei einem erwarteten Cache Hit bzw. Miss zur Verarbeitung benötigt werden, analysiert und statistisch aufbereitet. Anhand eines oberen bzw. unteren Schwellwerts, auch Threshold genannt, wird somit festgelegt, ob es sich bei einer bestimmten Flushzeit einer Cache Speicherstelle um einen Cache Hit oder einen Cache Miss handelt.

Durch das Übersetzen des Vorhandenseins von Speicherstellen im Cache zu Einsen und Nullen können Sender und Empfänger ohne vom Betriebssystem vorgesehene Wege zur Interprozesskommunikation Bit-Strings miteinander austauschen. Damit diese Kommunikation in beide Richtungen funktioniert, müssen Sender und Empfänger zu gegebener Zeit ihre Rolle wechseln. Darauf basierend wird eine Chat-ähnliche Anwendung implementiert, die übertragene Bit-Strings in ASCII-Zeichen umwandelt und auf der Konsole ausgibt. Werden Empfänger und Sender aus unterschiedlichen Terminals heraus gestartet, kann so im Wechsel eine menschenlesbare Text-Kommunikation stattfinden.

Da eine fehlerarme Übertragung von Bits in beiden Richtungen stattfinden soll, ist es vorteilhaft, einen Duplex-Kanal zu errichten, indem mehrere Speicherstellen adressiert werden. Eine Speicherstelle soll dabei jeweils nur zum Senden oder Empfangen verwendet werden, sodass mindestens zwei voneinander verschiedene Speicheradressen vonnöten sind. Damit diese Speicherstellen sich nicht gegenseitig aus dem Cache verdrängen, ist es zwingend erforderlich, dass sie nicht kongruent zueinander sind. Eine notwendige Voraussetzung, um so kommunizieren zu können, ist, dass das

Betriebssystem Shared Memory verwendet. So können beispielsweise Adressen von gemeinsam genutzten Shared Libraries verwendet werden. Diese Shared Library soll, um falsche Ergebnisse durch Zugriffe anderer Prozesse ausschließen zu können, exklusiv von Sender und Empfänger genutzt werden. Diese Einschränkung wird für das Test-Setup gemacht, da eine Testumgebung unter vollkommen realen Bedingungen den Rahmen dieser Projektgruppe übersteigen würde.

Wenn Sender und Empfänger nicht richtig miteinander synchronisiert sind, kann es passieren, dass ein über den Cache übertragenes Bit gar nicht oder fälschlicherweise erneut ausgelesen wird. Um diese Fehler möglichst minimal zu halten, muss eine feste Taktung zwischen Sender und Empfänger vorgegeben werden. Dies kann durch eine bestimmte Anzahl an vergangenen CPU-Zykeln oder einer festgelegten Zeit realisiert werden. Da zudem eine möglichst hohe Geschwindigkeit der Übertragung wünschenswert ist, gilt es die Synchronisation so zu gestalten, dass nicht übermäßig viel Zeit verschwendet wird und gleichzeitig eine akzeptable Fehlerrate erreicht wird.

Im Anschluss der praktischen Durchführung findet eine Evaluation anhand interessanter Kenngrößen statt. Diese soll einen Überblick darüber geben, wie performant oder fehleranfällig eine Bit-Übertragung von Textzeichen über einen Flush + Flush Seitenkanal ist. Da die Eröffnung eines cachebasierten Seitenkanals stark auf der zugrunde liegenden Hardware Architektur aufbaut, muss diese ebenso in ihren Details erfasst werden.

5 Umsetzung und Evaluation

Ziel dieser Projektgruppenarbeit ist das Ausnutzen eines cachebasierten Seitenkanals mit der Strategie Flush + Flush, sodass die übliche Interprozesskommunikation umgangen wird, um darauf aufbauend im verborgenen Bit-Strings zu versenden. Auf Basis der funktionierenden Übertragung von Bit-Strings über den Cache enthalten Sender und Empfänger Funktionalitäten, die ASCII-Zeichen in einen Bit-String wandeln und umgekehrt und diese Zeichen als Nutzereingabe von der Konsole einlesen und diese beim jeweiligen Empfänger ausgeben. Dadurch wird ein wechselseitiger Chat in menschenlesbarer Form ermöglicht. Der Programmcode zu dieser Arbeit und die Evaluationsergebnisse liegen dieser Abgabe bei und sind auch auf GitHub zu finden [Hei20]. Im Folgenden werden die Ergebnisse der im vorangegangenen Kapitel vorgestellten Vorgehensweisen besprochen.

5.1 Setup

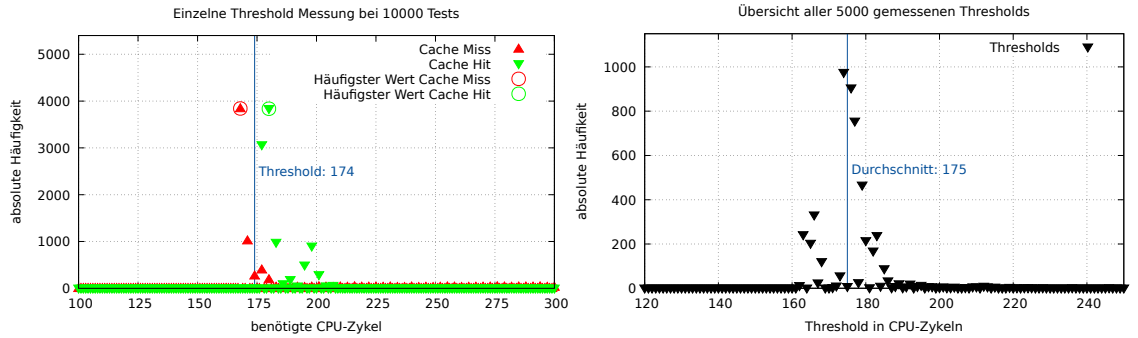
Das Testsystem hat einen Intel Core i7-4800MQ Prozessor mit vier Kernen, 8 GB Hauptspeicher und die für Intel übliche Cache-Hierarchie [Int13]. Der Last Level Cache ist ein 12-Way Set-Associative Cache mit 8192 Sets mit einer Line Size von 64 Byte (überprüft durch den Befehl `$ getconf -a | grep CACHE`). Die Tests werden auf dem Betriebssystem Ubuntu 20.04 LTS durchgeführt. Für die Tests wurden keine Sicherheitsmechanismen des Betriebssystems ausgeschaltet. Die erstellten Programme sind in der Programmiersprache C geschrieben.

Damit alle zeitkritischen Prozesse wie die Kalibrierung zum Bestimmen eines Thresholds oder der Sender- und Empfängerprozess vom Scheduler nicht während der Ausführungszeit im Wechsel verschiedenen Prozessorkernen zugewiesen werden und so Unterschiede in der Zugriffszeit auf den Last-Level Cache je nach ausführendem Prozessorkern entstehen, wurden alle im Zuge dieser Projektgruppenarbeit implementierten zeitkritischen Programme in der Ausführung jeweils an einen festen Kern gebunden.

5.2 Threshold

Zunächst muss ein Threshold bestimmt werden, der die mittlere Grenze der benötigten Zeit zwischen einem Cache Hit und einem Cache Miss festlegt. Bei der automatisierten Wiederholung von aufeinanderfolgenden Messungen von je 100.000 Cache Hits und Misses zeichnen sich stets Thresholds zwischen 160 und 189 benötigten CPU-Zykeln ab. Gemittelt über alle Messungen lag der Threshold bei etwa 176 mit einer Varianz von etwa 22,5 und einer Standardabweichung von ca. 4,7. In der Anwendung zur korrekten Biterkennung haben sich Thresholds zwischen 172 und 176 CPU-Zykeln als am zuverlässigsten bewährt. In Abbildung 2 ist eine exemplarische Messung eines einzelnen Thresholds und die Bestimmung des durchschnitts Thresholds bei wiederholten

Messungen zu sehen. Abbildung 4 zeigt die Verteilung aller gemessenen Thresholds über mehrere Tests hinweg. Damit zu starke Abweichungen bei der Messung den Durchschnitt nicht verfälschen, werden nur die Thresholds bis zu 250 CPU-Zykeln berücksichtigt.



(a) Eine einzelne Threshold Bestimmung

(b) Bestimmung eines durchschnitts Thresholds anhand von 5000 einzelnen Threshold Messungen

Abbildung 3: *Threshold Messungen*

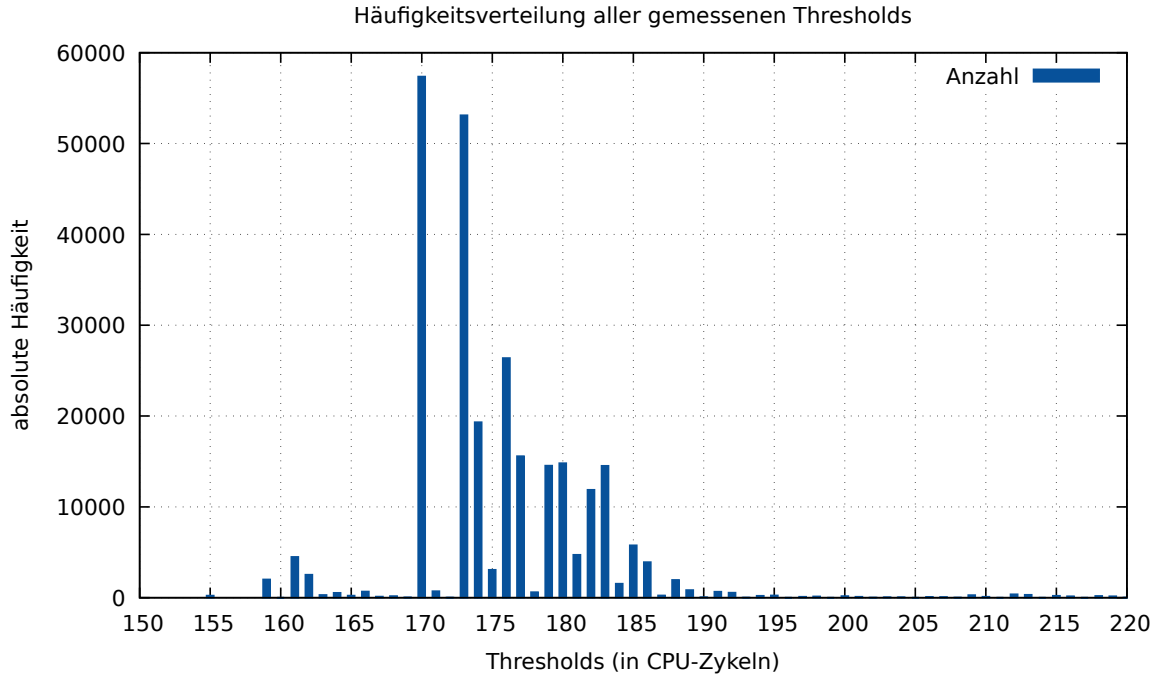


Abbildung 4: *Verteilung aller gemessenen Thresholds*

5.3 Synchronisation

Da eine zuverlässige Synchronisation zwischen Sender und Empfänger ausschlaggebend für die erfolgreiche Übertragung von Bit-Strings ist, muss diesem Aspekt besondere Aufmerksamkeit gewidmet werden. Dabei sind zwei Probleme zu lösen: der zeitgleiche Start von Sender und Empfänger und das Aufrechterhalten der Synchronisationstaktung, ohne dabei Kommunikationswege des Betriebssystems wie z.B. Signale zu verwenden.

Für die Synchronisation wird eine bis auf Nanosekunden genaue PC-interne Uhr verwendet, indem Sender und Empfänger an die Zeit gebundene Empfangs- bzw. Sendefenster erhalten. Ein solches Aktionsintervall ist stets kleiner als eine Sekunde. Pro Intervall wird immer nur ein Bit übertragen. Um Ungenauigkeiten bei der Messung zu minimieren, wird die benötigte Zeit für das Leeren des Caches so oft wie möglich vom Empfänger gemessen, um anhand der Durchschnittszeit zu bestimmen, ob eine „1“ oder eine „0“ gesendet wurde. Der Sender greift beim Übertragen einer „1“ ebenso möglichst oft auf die Speicherstelle zu.

Um den zeitgleichen Start der Kommunikation zu ermöglichen, wird zuerst der Empfangsprozess gestartet, welcher dann auf ein Bereitschaftssignal vom Sender wartet. Ist auch der Sender bereit zur Kommunikation, überträgt er über eine bestimmte Speicherstelle ein „1“-Bit, indem auf die entsprechende Speicherstelle zugegriffen wird. Da die benötigten Flushzeiten bei einem Cache Hit oder Miss mit Flush + Flush sehr nah am Threshold liegen, kann es dabei auch vorkommen, dass einzelne Zeitmessungen knapp über dem Threshold liegen, obwohl vom Sender das Bereitschaftsbit noch nicht abgelegt wurde und somit das Bereitschaftssignal zu früh erkannt wird. Um dies zu verhindern, wird die Bereitschaft des Senders erst dann erkannt, wenn zweimal hintereinander vom Empfänger ein Cache Hit gemessen wird.

Damit zur Zeit der Messung die richtigen Daten im Cache vorhanden sind, beginnt das Sendefenster etwas früher und hört etwas später auf als das Empfangsfenster. Dass die vorangegangene Übertragung nicht die Übertragung des nächsten Bits beeinflusst, wird durch eine kleine Pausenzeit in jedem Übertragungsintervall verhindert, in welcher der Empfänger zusätzlich auch genügend Zeit hat, die erhaltenen Daten weiter zu verarbeiten. So ergeben sich ein Sendefenster vom Intervallstart bis 80 % der Intervallzeit und ein Empfangsfenster von 10 % bis 75 % der Intervallzeit.

5.3.1 Empfangsfenster

Eine große Schwierigkeit stellt die Einhaltung der jeweiligen Aktionsfenster dar. Die C-Funktion clock_nanosleep der Bibliothek unistd.h, die dazu verwendet werden kann einen Prozess bis zu einem bestimmten Zeitpunkt schlafen zu legen, führt zu permanent unerwartet hohen Zugriffszeiten, wenn sie verwendet wird, um das jeweilige Empfangsfenster einzuhalten. Da die Zeitmessungen somit nicht verwertbar sind, scheidet diese Funktion als Kontrollmechanismus zur Einhaltung des Empfangsfensters aus. Für Beginn und Ende des Empfangsfensters wird das Warten auf den Start- bzw. Endzeitpunkt daher mit einer Busy-Waiting Schleife realisiert. Das zu häufige Messen der aktuellen Zeit mit der C-Funktion clock_gettime führt ebenfalls zu nicht verwertbaren Messergebnissen, wenn sie zwischen den einzelnen Zeitmessungen eines Intervalls aufgerufen wird und kostet zudem viel Zeit, die innerhalb eines Intervalls nicht auf die Messung weiterer Cache Flushzeiten verwendet werden kann. Daher wird eine in Abhängigkeit der Intervallgröße feste Anzahl an Wiederholungen der Zeitmessung bzw. Speicherzugriffe bestimmt, welche in ihrer Ausführungszeit genau so lange brauchen sollen, dass das jeweilige Aktionsfenster ideal ausgenutzt wird. Die richtige Anzahl an Wiederholungen zu bestimmen, die auch bei verschieden großen Intervallen zu einer idealen Ausnutzung des Empfangsfensters führt, ist jedoch nicht trivial, da in Abhängigkeit einer empfangenen „1“ oder „0“ unterschiedlich viel Zeit benötigt wird und die Ausführungszeit somit je nach übertragenem Bit oder sonstiger Prozessorlast leicht variieren kann.

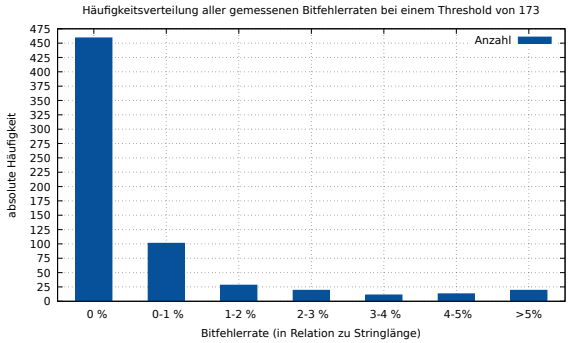
Bei einer Intervallgröße I, die in Nanosekunden angegeben ist und von der Sendefrequenz bestimmt wird, hat sich in Praxistests jedoch eine Anzahl von $\frac{7.000 * I}{100.000.000}$ bewährt.

5.3.2 Sendefenster

Auch wenn die Funktion clock_nanosleep beim Empfängerprogramm Probleme verursacht, scheint es essenziell, dass der Sendeprozess diese Funktion verwendet, wenn ein „0“-Bit gesendet und keine Daten im Cache abgelegt werden sollen. Wird der Sendeprozess nämlich nicht schlafen gelegt, misst der Empfänger, obwohl der Sender dauerhaft die Speicherstelle flusht, dennoch Zugriffszeiten auf den Cache, die einem Cache Hit entsprechen, was vermutlich auf Mechanismen wie Prefetching und spekulative Ausführung beim Sender zurückzuführen ist. Bei der Verwendung von clock_nanosleep werden meist etwa 70.000 ns benötigt, bis die Kontrolle an den Sendeprozess zurückgegeben wird, was dazu führt, dass bei zu hohen Übertragungsfrequenzen das dadurch kleine Aktionsfenster nicht eingehalten werden kann. Daher funktionieren Bit-String Übertragungen bei Frequenzen von mehr als etwa 7.100 Bit pro Sekunde nicht mehr zuverlässig.

5.4 Evaluation der Übertragungsqualität

Für die Evaluation werden randomisierte Bit-Strings erzeugt und unter anderem die Fehlerrate der Bit-Erkennung analysiert. Bei Frequenzen von 10 bis 7.100 Bit pro Sekunde liegt mit einem Threshold von 173 die Fehlerrate der einzelnen Bit-String Übertragungen im Schnitt bei 0,14%. Die empirische Varianz der Bitfehlerrate liegt bei 3,22 % und die durchschnittliche mittlere Abweichung der Fehlerraten vom Fehlerratendurchschnitt beträgt 1,79 %. Die Anzahl an fehlerhaft erkannten



(a) Verteilung aller gemessenen Bitfehlerraten

	Durchschnitt	Varianz	Stdabw.
Fehlerhafte Bits	0,91	21,04	4,59
Fehlerrate Bits	0,14 %	3,23	1,80
Falsche „1“-Bits	0,13	5,79	2,41
Falsche „0“-Bits	0,78	6,53	2,55
Fehlerhafte Zeichen	0,56	2,45	1,56
Fehlerrate Zeichen	0,48 %	13,37	3,66

(b) Fehler Tabelle

Abbildung 5: *Fehler Analyse*

Bits liegt im Durchschnitt insgesamt bei 0,91 Bit. In Relation zur mittleren Bit-Stringlänge von ca. 1.046 Bits bedeutet dies, dass etwa 99,913 %, also 99.913 von 100.000 übertragenen Bits korrekt sind. Die empirische Varianz der Anzahl von fehlerhaften Bits beträgt 21,04, sowie die Standardabweichung einen Wert von 4,59 fehlerhaften Bits hat. In Abbildung 5 ist die Verteilung dieser Bitfehlerraten und eine tabellarische Übersicht der Fehleranalyse zu sehen. Die Anzahl an fehlerhaft erkannten „1“-Bits liegt im Durchschnitt bei 0,13 Bits und bei 0,78 fehlerhaften „0“-Bits. Gibt es ein fehlerhaftes Bit, so ist es also deutlich häufiger eine „0“, die als „1“ interpretiert wurde. Dies kann mehrere Gründe haben, wie zum Beispiel, dass der Threshold zu niedrig ist, dass durch

Mechanismen der Performance-Optimierung dennoch Daten im Cache abgelegt werden oder dass sonstige Arbeitslast zu Verzögerungen führt.

Da ein ASCII-Zeichen im Sender-/Empfängerprogramm durch sieben Bits codiert wird, führt bereits eins von diesen sieben Bits zu einem falschen Zeichen, wenn das Bit fehlerhaft ist. Daher entspricht die Fehlerrate der Bits nicht gleich der Fehlerrate der Zeichen, da mehrere aufeinander folgende falsche Bits, die ein Zeichen codieren auch nur zu einem fehlerhaften Zeichen führen. Dennoch wirkt sich ein fehlerhaftes Zeichen aufgrund der kürzeren Länge der Zeichenfolge im Vergleich zum Bit-String stärker auf die Fehlerraten aus. Bei einer durchschnittlichen ASCII-Stringlänge von 149 Zeichen zeichnen sich im Mittel Fehlerraten von 0,48 % an fehlerhaften Zeichen ab. Bei im Schnitt 0,56 fehlerhaften Zeichen pro Übertragung sind also 371 von 100.000 Zeichen nicht korrekt.

6 Fazit und Ausblick

In dieser Arbeit wurde gezeigt, dass es sogar auf modernen Betriebssystemen möglich ist, einen funktionierenden cachebasierten Seitenkanal mit der Strategie Flush + Flush zu eröffnen. Es konnte veranschaulicht werden, dass der zentrale Aspekt, ein Bit anhand der Cache Flushzeiten korrekt zu erkennen gelingt. Auch eine zweiseitige Kommunikation ist über diesen Kanal möglich, wie die funktionierende Chat-Anwendung zeigt.

Der Fokus dieser Arbeit lag deutlich auf der Machbarkeit und weniger auf der Optimierung dieses Seitenkanals. Daher lassen sich weitere Optimierungen durchführen. Da die Kapazität des Kanals durch die Ausführungszeiten der C-Funktion clock_nanosleep beim Sender, wie bereits beschrieben, beschränkt ist, sollte weitergehend untersucht werden einen anderen Weg für die korrekte Übertragung von „0"-Bits bei gleichzeitiger Einhaltung des Sendefensters zu finden. Auch eine Optimierung der Relation vom Sendefenster zum Empfangsfenster und der eingeplanten Pausenzeit wäre möglich, um die Kapazität des Kanals zu erhöhen.

Auch wenn die empirische Analyse der Fehlerraten verhältnismäßig gut ausfällt, sind längere und wiederholte Übertragungen selten fehlerfrei. Dies kann durch eine Erweiterung der Übertragung durch Checksummen oder Fehlerkorrekturverfahren verbessert werden.

Die Arbeit an diesem Thema wurde ursprünglich auf einem anderen Testsystem gestartet. Dieses hat einen Intel Core i7-4790K Prozessor mit vier Kernen, 16 GB Hauptspeicher und die für Intel übliche Cache-Hierarchie [Int14]. Der Last Level Cache ist ein 16-Way Set-Associative Cache mit 8192 Sets mit einer Line Size von 64 Byte (überprüft durch den Befehl `$ getconf -a | grep CACHE`). Das installierte Betriebssystem ist Ubuntu 14.04. Leider ist es auf diesem Testsystem nicht möglich, zuverlässige Zeitmessungen durchzuführen, die eine korrekte Biterkennung zulassen. Die gemessenen benötigten CPU-Zykel für das Leeren des Caches schwanken übermäßig stark, übersteigen teils den Threshold um das 10-Fache und lassen somit keine Erkennung eines direkten Zusammenhangs zwischen gecachten oder ungecachten Daten zu. Auch die Untersuchung dessen Ursache kann interessante Erkenntnisse liefern, die Aufschluss über die Machbarkeit von cachebasierten Seitenkanälen auf verschiedenen Soft- und Hardwarearchitekturen geben.

Literaturverzeichnis

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

[Can19] Canella, Claudio and Van Bulck, Jo and Schwarz, Michael and Lipp, Moritz and Von Berg, Benjamin and Ortner, Philipp and Piessens, Frank and Evtyushkin, Dmitry and Gruss, Daniel: „A systematic evaluation of transient execution attacks and defenses“. In: *28th USENIX Security Symposium*. 2019, S. 249–266.

[Gru16a] Gruss, Daniel and Maurice, Clémentine and Wagner, Klaus and Mangard, Stefan: „Flush+ Flush: a fast and stealthy cache attack“. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, S. 279–299.

[Gru16b] Gruss, Daniel and Maurice, Clémentine and Wagner, Klaus and Mangard, Stefan: *GitHub-Repository zu: „Flush+ Flush: a fast and stealthy cache attack"*. https://github.com/IAIK/flush_flush. Zugriff am 29.09.2020. 2016.

[Gru17] Gruss, Daniel: „Software-based microarchitectural attacks“. Diss. Graz University of Technology, 2017.

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

[Hei20] Heidler, Sabrina: *GitHub-Repository zu dieser Arbeit*. https://github.com/SabrinaHei/PG_Cachebasierte_Seitenkanaele.git. Hochgeladen am 01.10.2020. 2020.

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

[Int13] Intel Corporation: *Intel® Core™ i7-4800MQ Prozessor Spezifikation*. https://ark.intel.com/content/www/de/de/ark/products/75128/intel-core-i7-4800mq-processor-6m-cache-up-to-3-70-ghz.html. Zugriff am 29.09.2020. 2013.

[Int14] Intel Corporation: *Intel® Core™ i7-4790K Prozessor Spezifikation*. https://ark.intel.com/content/www/us/en/ark/products/80807/intel-core-i7-4790k-processor-8m-cache-up-to-4-40-ghz.html. Zugriff am 29.09.2020. 2014.

[Int19] Intel Corporation: *Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual*. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. Zugriff am 29.09.2020. 2019.

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

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

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

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

[Lip16a] Lipp, Moritz and Gruss, Daniel and Spreitzer, Raphael and Maurice, Clémentine and Mangard, Stefan: „Armageddon: Cache attacks on mobile devices“. In: *25th USENIX Security Symposium*. 2016, S. 549–564.

[Lip16b] Lipp, Moritz and Gruss, Daniel and Spreitzer, Raphael and Maurice, Clémentine and Mangard, Stefan: *GitHub-Repository zu: „ARMageddon: Cache Attacks on Mobile Devices"*. https://github.com/IAIK/armageddon. Zugriff am 29.09.2020. 2016.

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

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

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

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

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

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

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

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

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

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

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

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

Abbildungsverzeichnis

1 Die Aufteilung der Bits der Hauptspeicheradresse in die drei Felder Tag, Index und Offset 4
2 Ein 3-Way Set-Associative Cache. Entsprechend der Anzahl n an Index Bits, gibt es 2^n Cache Sets. 5

3 Threshold Messungen 14
4 Verteilung aller gemessenen Thresholds 14
5 Fehler Analyse 16

Listings

3.1 Pseudocode Flush + Flush Sender 9
3.2 Pseudocode Flush + Flush Empfänger 10

Selbstständigkeitserklärung

Hiermit versichere ich, das vorliegende Laborprotokoll 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, 2. Oktober 2020

Sabrina Heidler