UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Browser-basierte Cache-Angriffe auf die RSA-Schlüsselgenerierung

Browser-based Cache Attacks on RSA key generation

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Moritz Krebbel

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Ida Bruhns

Lübeck, den 6. Oktober 2018

Kurzzusammenfassung

Mikroarchitekturangriffe haben sich als sehr mächtig erwiesen, erfordern aber meist die Ausführung von einem nativen Code auf dem System des Opfers. Im Jahr 2015 wurde gezeigt [OKSK15], dass Browser-basierte Angriffe, bei denen bereits der Besuch einer Webseite durch das Opfer ausreicht, möglich sind. Ein solcher Angriff wird in Scriptsprachen wie Javascript implementiert und nutzt den geteilten L3-Cache der CPU aus, um Informationen über die Speicherzugriffe des Opferprogramms zu erhalten. Der Angriff ist weder auf einen Exploit im Browser noch auf fahrlässiges Verhalten der Nutzer angewiesen.
In dieser Arbeit wird ein Browser-basierter Prime-and-Probe-Angriff implementiert und die Portierung eines nativen Angriffs auf die RSA-Primzahlgenerierung [CAPGATBB18] in OpenSSL hin zu einem Browser-basierten Angriff auf Mozilla NSS untersucht. Zusätzlich werden neue Leakages in der RSA-Primzahlgenerierung von Mozilla NSS und OpenPGP.js beschrieben und analysiert. Des Weiteren wird ein verbesserter Algorithmus zum Finden von Eviction-Sets vorgestellt, der die in der Praxis wichtige Initalisierungsphase eines Angriffs beschleunigt.

Abstract

It is evident that microarchitectural attacks are very powerful, but they mostly need the original native code of the victim's system. In 2015 it was shown that browser-based attacks are even possible if the victim has only visited a website [OKSK15]. Such an attack is implemented in scripting languages such as Javascript and uses the shared CPU's L3 cache to obtain information about the victim's memory accesses. The attack neither requires an exploit in the browser nor a user's negligent behavior.
This work implements a browser-based prime-and-probe-attack and examines the porting of a native attack on RSA prime generation [CAPGATBB18] in OpenSSL to a browser-based attack on Mozilla NSS. New leakages in the RSA prime number generation of Mozilla NSS and OpenPGP.js are also described and analyzed. Furthermore an improved algorithm for finding eviction sets is presented which accelerates the important initialization phase of an attack.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 6. Oktober 2018

Danksagungen

Als erstes möchte ich allen Personen danken die mich bei der Anfertigung dieser Arbeit unterstützt haben. Besonders hervorheben möchte ich:

- meine Betreuer für die freundliche Unterstützung und die vielen hilfreichen Tipps
- meine Familie für das ausgiebige Korrekturlesen meiner Arbeit und der Gabe nützlicher Anregungen sowie der immerwährenden Unterstützung während meines Studiums
- meine Kommilitonin für einerseits anregende Diskussionen und andererseits der visuellen Detektierung von Fehlern
- den Pool, dessen Rechner ein mächtiges Werkzeug für die Implementation ausgiebiger Benchmarks sind

Inhaltsverzeichnis

1 Einführung

Seitenkanalangriffe sind ein mächtiges Werkzeug für die Kryptanalyse, da sie es unter anderem erlauben, geheime Informationen wie den kryptografischen Schlüssel aus einem Gerät wie einer Chipkarte zu extrahieren. Hierfür wird beispielsweise der Stromverbrauch oder die Berechnungsdauer verschiedener Ein- und Ausgaben gemessen. Bei solchen physikalischen Seitenkanalangriffen ist die Angreiferin gezwungen, eine örtliche Nähe zum System aufzuweisen.
Seitenkanalangriffe auf die Mikroarchitektur eines Prozessors sind von dieser Einschränkung befreit, da die Ausführung von Programmcode auf dem Zielsystem ausreicht. Seit der ersten Beschreibung von Seitenkanalangriffen auf die Mikroarchitektur vor über zehn Jahren [TSS+03], sind diese zu einem ernst zu nehmenden Sicherheitsrisiko herangewachsen [Ket18].
Auf der Mikroarchitekturebene konkurriert der Opferprozess mit dem Angriffsprozess um die Ressourcen des Prozessors, selbst wenn der Angriffsprozess mit keinen außerordentlichen Privilegien ausgestattet ist. Aufgrund dieser Konkurrenz kann die Angreiferin Informationen über den Opferprozess gewinnen, indem sie durch Messungen Varianzen beim Zugriff auf die Ressourcen feststellt.
Eine Reihe von Ressourcen hat sich durch das Leaken von Informationen als problematisch erwiesen, etwa die Einheiten der Sprungvorhersage oder der Return-Stack-Buffer. Das mit Abstand größte Problem stellen die vielfältigen Caches dar, welche eine Überwachung der Speicherzugriffe des Opferprozesses erlauben. Diese ermöglichen es, geheime Informationen wie kryptografische Schlüssel von RSA [YGH17a] oder AES [Ber05] aus dem Opferprozess zu extrahieren. Weitere Anwendungsmöglichkeiten sind Keylogger [GSM15] und Überwachungen der Webseitenaufrufe oder Nutzeraktivitäten [OKSK15].
Es ist trotz dieser Möglichkeiten wenig darüber bekannt, ob Cache-Angriffe praktisch relevant sind oder wie häufig sie eingesetzt werden. Einem breiten Angriffsvektor steht das im Großteil der Literatur angenommene Angriffsszenario entgegen, welches die Ausführung von nativem Code auf dem Opfersystem voraussetzt. Dieses Szenario hat seine Berechtigung unter anderem bei Angriffen auf virtuelle Maschinen, wo der Code der Angreiferin auf derselben Hardware wie der des Opferprozesses läuft (Cross-VM-Attack).
Auch Mehrbenutzersysteme, wie sie häufig in Unternehmen und Organisationen anzutreffen sind, fallen unter dieses Szenario. So ist es jedem Studierenden der Universität zu Lübeck jederzeit möglich, sich per SSH auf einen beliebigen Pool-Rechner einzuloggen,

unabhängig davon, ob der Rechner lokal in Benutzung ist.
In einem typischen Endbenutzerszenario ist es realitätsfern vorauszusetzen, dass vom Opfer nativer Angriffscode ausgeführt wird, denn den meisten Benutzern ist die einhergehende Gefahr der Ausführung von unbekanntem nativen Code bewusst.
In Folge dessen gibt es Bestrebungen, Cache-Angriffe auf Endbenutzergeräte zu ermöglichen [OKSK15,GPTY18,GRB+17], indem der Angriffscode in Websprachen wie Javascript übersetzt und folgend vom Browser des Opfers ausgeführt wird. Da der Browser faktisch bei jedem Webseitenbesuch fremden und unbekannten Code ausführt, jedoch keine Angriffsfläche bieten soll, sind die Möglichkeiten für die Angreiferin in Javascript und Co. entsprechend restriktiv. Auf Grund dieser Tatsachen sind Angriffe im Browser nur eingeschränkt oder aufwendiger auszuführen, wie sich im Verlauf dieser Arbeit zeigen wird.

1.1 Aufgabenstellung

In dieser Arbeit wird unter anderem der Angriff aus dem Paper [CAPGATBB18] auf Mozilla NSS portiert. Es handelt sich dabei um einen Prime-and-Probe-Angriff, der im Gegensatz zum Original-Paper über Javascript bzw. Webassembly durchgeführt wird. Dadurch erhöht sich die Zahl potenzieller Opfer, da der Angriff über eine Webseite oder Werbung auf dieser ausgerollt werden kann. Außerdem werden neue Leakages in der RSA-Primzahlgenerierung von Mozilla NSS und OpenPGP.js analysiert und ihre Einsatzmöglichkeiten bei einem praktischen Angriff überprüft.
Im Laufe der Arbeit werden folgende Fragen beantwortet:

1. Unter welchen Bedingungen sind im Oktober 2018 Cache-Angriffe aus dem Browser heraus möglich? Die Arbeiten [OKSK15, GPTY18, GRB+17] haben demonstriert, dass Cache-Angriffe im Browser möglich sind. Diese Arbeiten sind jedoch vor den Gegenmaßnahmen der Browserhersteller gegen Meltdown und Spectre veröffentlicht worden.

2. Welchen Einschränkungen unterliegen die Cache-Angriffe im Browser gegenüber Cache-Angriffen mit nativem Code?

3. Wie kann die Initialisierung des Angriffs, das heißt die notwendige Eviciton-Set-Suche, beschleunigt werden?

4. Gibt es im Bereich der RSA-Schlüsselgenerierung Leakages, die Informationen über den Schlüssel verraten und sind diese im Browser ausnutzbar?

Für die Beantwortung der ersten Frage spielt der Browser eine wichtige Rolle, da für Cache-Angriffe hochauflösende Timer im Nanosekundenbereich benötigt werden. In

Chrome ist das Erzeugen eines solchen Timers durch einen Counter-Thread standardmäßig möglich, wobei dies in Firefox zurzeit ausschließlich durch Veränderung der Browsereinstellungen möglich ist. Der angesprochene Counter-Thread ist während des gesamten Angriffs voll ausgelastet, womit ein virtueller Kern vollständig belegt ist. Da Entwickler hochauflösende Timer nachfragen, wird Mozilla ebenso wie Google in Zukunft versuchen, wieder hochauflösende Timer anzubieten, wobei der zurzeit nötige Timer-Thread entfallen wird. Dies wird passieren, sobald Maßnahmen gegen Meltdown und Spectre implementiert sind (siehe Abschnitt 5.6).

Beim Prime-and-Probe-Angriff steht die Browserimplementierung einer nativen Implementierung in nichts nach. Allerdings ist neben den Kosten für den Counter-Thread die zeitliche Auflösung des Angriffs um etwa 50% reduziert.

Der Opferprozess wird oft künstlich gebremst, da die Berechnungen auch beim Angriff mit nativem Code häufig zu schnell sind. Ebenso lassen sich im Browser Methoden umsetzen, um den Opferprozess zu verlangsamen. Aufgrund der fehlenden clflush-Instruktion, ist die damit verwendete Bremsmethode im Browser nicht umsetzbar. Anhand eines Angriffs auf die RSA-Schlüsselgenerierung wird gezeigt, dass die Portierung eines nativ laufenden Angriffs aufgrund dieser Einschränkungen nicht immer möglich ist.

Ein realistischer Angriff ist Voraussetzung für eine kurze Initialisierungsphase, daher wurden Optimierungen des in vielen Papern beschriebenen Eviction-Set-Suchalgorithmus [DKPT17,LYG+15,GPTY18] analysiert und die Ergebnisse anschließend mit der ursprünglichen Variante verglichen. Des Weiteren wurde eine neue noch unveröffentlichte Technik im Kontext der Eviction-Set-Suche im Browser erprobt. Diese nutzt spezifische Eigenschaften der Store-Queue in Intel-Prozessoren aus, womit die Suche gegenüber dem optimierten Standardalgorithmus um mehr als den Faktor 3 beschleunigt werden kann.

Die Schlüsselgenerierung von Mozilla NSS und OpenPGP.js wurde auf Leakages untersucht. Sowohl in der Primzahlgenerierung von Mozilla NSS als auch OpenPGP.js wurden Leakages entdeckt, welche die verwendeten Primzahlen in einem Gleichungssystem beschreiben. Es konnte noch nicht abschließend geklärt werden, ob diese Gleichungssysteme bei praktischen Instanzen wie RSA-2048 in annehmbarer Zeit gelöst werden können.

Des Weiteren wurde ein bekannter Cache-Angriff auf die RSA-Schlüsselgenerierung in OpenSSL nach Mozilla NSS portiert. Der ursprüngliche Angriff setzte die Ausführung nativen Codes voraus, wobei in dieser Arbeit eine Umsetzung im Browser geprüft wurde. Hierbei konnten mehrere Probleme identifiziert werden, welche Portierungen von nativen Cache-Angriffen hin zu Implementierungen im Browser erschweren. Diese werden in Kapitel 4 und 5 genauer beschrieben und beurteilt.

1.2 Verwandte Arbeiten

Hier soll ein Überblick bereits getätigter Forschungsarbeiten im Rahmen der für die Arbeit relevanten Themenbereiche Drive-by-Attacks und Angriffe auf die RSA Schlüsselgenerierung gegeben werden. Zudem soll eine Abgrenzung gegenüber verwandten, aber für diese Arbeit nicht relevanten Themenbereichen stattfinden.

1.2.1 Cache-Angriffe im Browser

Alle oben zitierten Arbeiten setzen die Ausführung von nativem Code auf dem Opfersystem voraus. Im Jahr 2015 wurde die erste Arbeit [OKSK15], welche sich mit Cache-Angriffen im Browser beschäftigt, veröffentlicht. Die Autoren konnten den Aufruf von verschiedenen bekannten Webseiten unterschiedlichen Cache-Zugriffsmustern zuordnen, um so das Surfverhalten von Nutzern zu überwachen. Gras et al. [GRB+17] konnten im Browser erfolgreich die Speicherverwürfelung ASLR aushebeln, wobei sich der Angriff aufgrund der niedrigen zeitlichen Auflösung nicht zur Schlüsselextraktion eignet.
Mit dem Rowhammer-Angriff [KDK+14] ist es möglich, Bits im Speicher ohne Schreibzugriffe zu verändern, um beispielsweise Sicherheitsvorkehrungen zu umgehen. Dieser wurde von Gruss et al. erfolgreich in Javascript implementiert [GMM16].
Code, der im Hinblick darauf designt wurde, unabhängig von den Eingaben dieselbe Laufzeit und denselben Codepfad zu besitzen, verhindert viele Leakages. Im Paper [GPTY18] wurde dargestellt, dass solcher Code durch die Ausführung im Browser angreifbar werden kann. Durch die Freiheiten des Javascript-Compilers ist es möglich, je nach Eingabe unterschiedliche Codepfade auszuführen und somit eine Leakage entstehen zu lassen.

1.2.2 Cache-Angriffe auf RSA

Viele Paper haben gezeigt [YGH17a,YF14,GPTY18], dass sich der RSA-Schlüssel oder Teile davon mittels Cache-Angriffen extrahieren lassen. Das Angriffsziel waren dabei Komponenten der Ver- und Entschlüsselung, beispielsweise die modulare Exponentiationsfunktion [YGH17a,GPTY18,GPTY18].
Weniger Aufmerksamkeit wurde der RSA-Schlüsselgenerierung geschenkt, da diese im Gegensatz zur Ver- und Entschlüsselung nur einmal ausgeführt wird und damit schwieriger anzugreifen ist. In der 2018 verfassten Arbeit [CAPGATBB18] wird ein Angriff auf die RSA-Schlüsselgenerierung in OpenSSL beschrieben. Hierfür wird die Funktion zum Finden des größten gemeinsamen Teilers angegriffen, welche die Teilerfremdheit der beiden erzeugten Primzahlen p und q zu dem Exponenten e überprüft.

1.3 Gliederung der Arbeit

Diese Arbeit untersucht aktuelle Fragestellungen bezüglich Cache-Angriffen, wobei praxisnahe Angriffe aus dem Browser heraus im Vordergrund stehen. Im Hinblick auf die Praxisrelevanz wird eine neuartige Eviction-Set-Suche evaluiert, welche die Initialisierungsphase eines Angriffs beschleunigt. Des Weiteren werden Leakages in der RSA-Schlüsselgenerierung von Mozilla NSS und OpenPGP.js untersucht sowie Probleme bei der Portierung nativer Angriffen erörtert.

In Kapitel 2 werden die nötigen technischen Grundlagen zum Verständnis der Arbeit vermittelt. So wird die in der Arbeit verwendete Angriffstechnik Prime-and-Probe vorgestellt und erläutert, warum der Angriff in aktuellen Prozessoren funktioniert. Des Weiteren werden die notwendigen Voraussetzungen erklärt wie beispielsweise die Verfügbarkeit hochpräziser Timer. Ebenso werden die vorteilhaften Merkmale eines Cache-Angriffs aus dem Browser beschrieben. Abschließend wird die Speicher-Disambiguierung erklärt, deren Verständnis für die beschleunigte Eviction-Set-Suche elementar ist.

Mit der realen Implementierung des Angriffs in Javascript beziehungsweise Webassembly beschäftigt sich Kapitel 3. Es wird beschrieben, wie die Voraussetzungen für einen Angriff im Webkontext trotz begrenzter Möglichkeiten geschaffen werden können. Weiterhin wird die Umsetzung des Eviction-Set-Suchalgorithmus im Webkontext dargelegt und dessen Leistung theoretisch analysiert. Zudem wird die Performance des Eviction-Set-Suchalgorithmus sowie Optimierungen desselben in der Praxis evaluiert. Ein spezifisches Verhalten von Intel-Prozessoren in Bezug auf die Speicher-Disambiguierung, welches neue Möglichkeiten in der Eviction-Set-Suche eröffnet, wird erläutert. Erstmals wird beschrieben, wie damit ein verbesserter Eviction-Set-Suchalgorthmius im Webkontext umsetzbar ist und wie dieser im Vergleich zur Standardversion performt. Als Beispiel wird ein verdeckter Kanal vom Browser zu einem nativ laufenden Programm aufgebaut.

Im 4. Kapitel werden verschiedene Leakages in der RSA-Schlüsselgenerierung von Mozilla NSS und OpenPGP.js analysiert. Dabei wird erörtert, wie die Ergebnisse einer automatisierten Leakage-Erkennung mit einbezogen werden können. Bei der Leakage in der RSA-Primzahlgenerierung von Mozilla NSS und OpenPGP.js ist offen, ob sich diese eignet, Teile der Primzahlen effizient zu rekonstruieren. Darüber hinaus wird die Portierung eines nativen Angriffs auf OpenSSL hin zu einem Angriff im Browser auf Mozilla NSS analysiert. Dabei auftretende Fallstricke, wie beispielsweise der fehlende clflush-Befehl im Webkontext, werden herausgearbeitet und mögliche Lösungen diskutiert.

Die Ergebnisse der Arbeit werden im 5. Kapitel bewertet. So wird beschrieben, welche zusätzlichen Auswirkungen die Hardware und Software des Opfers auf die Erfolgswahr-

scheinlichkeit eines Angriffs haben. Gegenmaßnahmen werden erläutert, indem die in diesem Kontext wichtige Rolle der Browserhersteller untersucht wird. Des Weiteren werden Vorteile der RSA-Schlüsselgenerierung in OpenPGP.js gegenüber Mozilla NSS dargestellt. Zudem wird die Performance von Prime-and-Probe-Implementierungen in Javascript und Webassembly verglichen.

Das letzte und 6. Kapitel wird die vorherigen Ergebnisse abschließend bewerten. Zusätzlich wird ein Ausblick gewährt, welche technischen Fortschritte und Erkenntnisse die Angriffsmöglichkeiten im Browser verbessern würden. Zudem wird ein Blick auf andere Endgeräte und deren Angreifbarkeit geworfen. Abschließend wird beschrieben, wie Angriffe verhindert werden können und welche Forschung in diesen Bereichen zukünftig notwendig ist.

2 Grundlagen

Im Folgenden sollen Verfahren und Techniken erläutert werden, welche für das Verständnis der späteren Kapitel essenziell sind. Dafür werden die relevanten Hardwarekomponenten und Vorausetzungen an die Caches erklärt. Zudem werden grundlegende Cache-Angriffe wie Prime-and-Probe sowie Flush-and-Reload vorgestellt. Weiterhin werden die mathematischen Grundlagen für die spätere Leakage-Analyse von RSA beschrieben.

2.1 Virtuelle Speicherverwaltung

Virtuelle Speicherverwaltung stellt eine Abstraktion für die vorhandenen physischen Speichermedien wie etwa den Hauptspeicher oder die Festplatte bereit [Tan06a]. Das Betriebssystem übersetzt virtuelle Adressen, welche von Prozessen genutzt werden, mit Hilfe der Hardware in physische Adressen. Jedem Prozess steht der gleiche virtuelle Adressraum zur Verfügung, wobei das Betriebssystem dafür Sorge trägt, dass für jeden Prozess die richtige Zuordnung von virtueller zu physischer Adresse sichergestellt wird. Ein Vorteil der virtuellen Speicherverwaltung ist eine erhöhte Sicherheit durch die Speicherisolierung aller Prozesse, da ein Prozess keinen Zugriff auf das Adressmapping eines anderen hat. So kann eine fehlerhafte Schreiboperation eines Prozesses keinen Fehler in anderen Prozessen verursachen, da gleiche virtuelle Adressen vom Betriebssystem auf unterschiedliche physische Adressen abgebildet werden.

2.2 Caches

Die Geschwindigkeitsentwicklung des Hauptspeichers konnte in den letzten Jahren nicht mit der des Prozessors Schritt halten [Car02]. Der Cache ist ein im Vergleich zum Hauptspeicher kleinerer, aber schnellerer Pufferspeicher, welcher im aktuellen Kontext häufig benötigte Daten vorhält. Diese Daten zeichnen sich auch häufig dadurch aus, dass sie räumlich nah beieinander liegen. Ohne Caches wäre ein Prozessor häufig gezwungen, auf Daten des langsamen Hauptspeichers zu warten, und würde in seiner Verarbeitungsgeschwindigkeit gebremst. Weiter liegt der Fokus der Arbeit auf der im Desktopbereich weit verbreiteten x86-Architektur. Deshalb werden mit Intel-Prozessoren der Core-Architektur bestückte Testrechner verwendet, da Intels Core-Architektur im x86-Desktopsegment zurzeit den höchsten Marktanteil besitzt [Pas18]. Aus diesem Grund werden Erklärungen im

Folgenden häufig mit Beispielen der Intel Core-Architektur veranschaulicht.

Ein Cache der Core-Architektur lagert nicht einzelne Bytes, sondern immer gleich 64 Bytes, Cache-Line genannt, auf einmal ein. Dabei werden die 64 Bytes beginnend ab der größten Adresse, welche zugleich kleiner als die Zieladresse und ein Vielfaches von 64 ist, angefragt. Angenommen 4 Bytes Daten an Adresse `0b10110111` werden angefordert, dann lagert der Cache die 64 Bytes beginnend mit der Adresse `0b10000000` ein. Heutige Arbeitsspeichermodule liefern 8 Bytes zeitgleich, wobei die CPU mit einem einzigen Befehl einen Burst von 8 Übertragungen initiieren kann, die das Lesen und Schreiben einer gesamten 64-Bytes-Cache-Line ermöglichen. Das Vorausladen von aktuell nicht benötigten Daten liegt in der Lokalitätseigenschaft typischer Programme begründet [Tan06b]. So besagt die zeitliche Lokalität, dass aktuell verwendete Daten mit hoher Wahrscheinlichkeit in naher Zukunft erneut benötigt werden. Räumlich Lokalität hingegen besagt, dass beim Zugriff einer bestimmten Speicheradresse benachbarte Speicheradressen in naher Zukunft mit hoher Wahrscheinlichkeit benötigt werden. Somit ist es von Vorteil, gleichzeitig 64 Bytes zu laden, da die zusätzlich geladenen Bytes mit hoher Wahrscheinlichkeit in den nächsten Zyklen ebenfalls benötigt werden. Der Performancenachteil, welcher durch ein Laden von gleichzeitig 64 Bytes entsteht, ist im Gegensatz dazu vernachlässigbar.

Ein Prozessorcache besteht üblicherweise aus mehreren Ebenen, Cache-Levels genannt, wobei die Core-Architektur etwa 3 Cache-Levels besitzt, welche absteigend größer und langsamer werden. Ein Intel i7-4770 besitzt pro physischen Kern beispielsweise einen 32 KiB-L1-Datencache mit einer Zugriffslatenz von 4 bis 5 Taktzyklen und einen 256 KiB-L2-Cache mit einer Latenz von 12 Taktzyklen [Int16b]. Im Unterschied zu den beiden ersten Cache-Levels teilen sich in der Core-Architektur alle Kerne den L3-Cache. Dies bedeutet aus Sicht der Angreiferin einen großen Vorteil, da jedes Programm den Zustand des L3-Caches beeinflusst, und zwar unabhängig davon, auf welchem Kern es ausgeführt wird. Dagegen muss die Angreiferin bei einem Angriff auf den L1- beziehungsweise L2-Cache sicherstellen, dass ihr Code und das angegriffene Programm auf demselben physischen Kern ausgeführt werden.

Die Ersetzungsstrategie legt fest, welcher Eintrag aus dem Cache verdrängt wird, sofern alle Einträge des zugehörigen Cache-Sets belegt sind. Intels Core-Prozessoren verdrängen typischerweise den Eintrag, welcher bezogen auf die letzte Zugriffszeit am ältesten ist, auch least-recently-used-Strategie (LRU) genannt. Ab der Ivy-Bridge-Generation passt Intel diese Strategie situationsbedingt an [Won13] und verwendet ebenfalls die bimodul insertion policy (BIP), welche häufig den zuletzt hinzugefügten Eintrag löscht. Dies kann von Vorteil sein, wenn das Working-Set des Programms die Größe des Caches übersteigt und die LRU-Strategie zu keinen Cache-Hits führen würde. Bei Ivy-Bridge Prozessoren stellten die Autoren des Papers fest, dass manche Cache-Sets die LRU-Strategie und man-

che die BIP-Strategie verfolgen. Es wird daher ein Echtzeitvergleich zwischen beiden Strategien vermutet, um herauszufinden, welche Strategie weniger Cache-Misses produziert. Bei den Nachfolger- Generationen Haswell und Broadwell stellten die Autoren fest, dass zwischen beiden Strategien mit hoher Frequenz gewechselt wird, wobei der Algorithmus hinter den Ersetzungsstrategien ebenfalls nicht öffentlich zugänglich ist.

2.2.1 Assoziativität

Sofern die Auswahl des Cache-Eintrags für die Daten einer bestimmten Hauptspeicheradresse keinerlei Beschränkungen unterliegt, wird von einem voll-assoziativen Cache gesprochen. Das andere Extrem wäre ein einfach-assoziativer Cache beziehungsweise eine direkte Abbildung, wobei die Adresse des Hauptspeichers, von der die Daten stammen, den zu wählenden Cache-Eintrag eindeutig festlegt. Der Mittelweg ist die n-Wege-Assoziativität, bei der die Daten nur an n Cache-Einträgen liegen können (siehe auch Abb. 2.2.1).

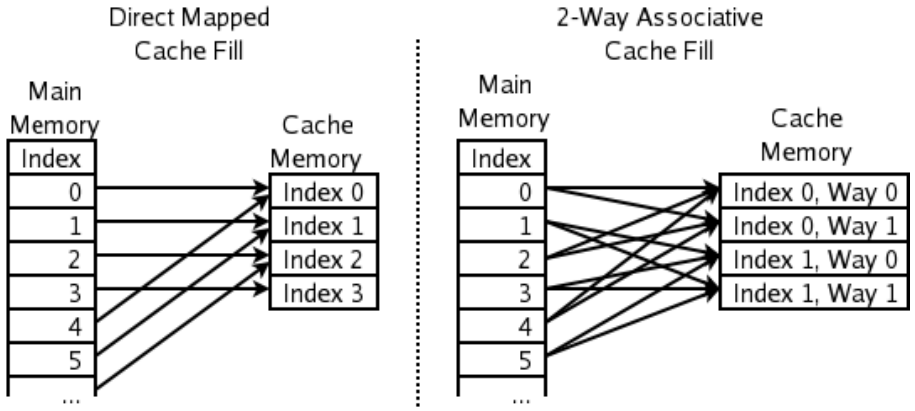

Abbildung 2.1: Links ist ein direkt abgebildeter Cache zu sehen, bei dem die Hauptspeicheradresse den Cache-Index vollumfänglich festlegt. Rechts ist ein 2-Wege assoziativer Cache zu sehen bei dem etwa die Hauptspeicheradresse 0 sowohl auf *Index 0, Way 0* als auch auf *Index 0, Way 1* im Cache abgebildet werden kann [Hel06].

Der L3-Cache eines Intel i7-4770 ist beispielsweise 8 MiB groß und verfügt daher über 131072 (2^{17}) Cache-Einträge in der Größe einer Cache-Line. Wäre dieser Cache nun voll-assoziativ, müsste er bei jeder Anfrage komplett durchsucht werden. Aus diesem Grund ist der Cache in Cache-Sets unterteilt, wobei Daten einer spezifischen Hauptspeicheradresse nur in genau ein Cache-Set eingelagert werden können. Der i7-4770 besitzt 8192 dieser Cache-Sets, womit sich eine Assoziativität von 16 ergibt, das heißt man teilt die Anzahl der Cache-Einträge durch Anzahl der Cache-Sets. Dies bedeutet, dass die Suche nach den Daten einer Hauptspeicheradresse im Cache auf 16 Einträge begrenzt ist. Des Weiteren wird der Cache in *Slices* unterteilt, wobei jede Slice einem Kern zugeordnet wird. Die

Slices sind mittels eines Ringbuses verbunden, so dass jeder Kern auf Daten jeder Slice zugreifen kann. Die Zugriffslatenz erhöht sich mit der Anzahl der benötigten Hops, jedoch ist diese Erhöhung im Bereich von etwa 10 ns pro Hop. Diese Latenzunterschiede sind daher zu gering, um die Erkennung zwischen Cache-Hit und Cache-Miss zu gefährden, welche bei ca. 60 ns liegen [OKSK15]. Bei i7-4770 sind jedem der 4 Slices 2048 Cache-Sets zugeordnet. Die Zuordnung der Cache-Sets innerhalb der Slices sind fest an die untersten 18 Bit der pysischen Adresse gekoppelt. Zu welchem Slice ein Adresse gemappt wird, wird durch eine nicht dokumentierte Hashfunktion bestimmt.

2.2.2 *Inclusive* und *exclusive*

Ein Cache wird als *inclusive* bezeichnet, falls alle Daten, die in einem niedrigen Cache-Level vorliegen, zusätzlich auch in den höheren Cache-Levels eingelagert sind. So besitzen die Caches aller Desktop-Versionen der Intel Core-Architektur diese inclusive-Eigenschaft (Stand Juni 2018). Die Caches der Desktop-Prozessoren des Konkurrenten AMD haben zum Beispiel die Zen-Architektur [Cut17a], während jene der aktuellen Skylake-X-Prozessoren [Cut17b] für Intels High-Performance-Plattform diese Eigenschaft hingegen nicht besitzen. Wegen des großen Marktanteils von Intel-CPUs kann festgehalten werden, dass der Großteil der sich im Einsatz befindlichen Prozessoren mit *Inclusive*-Caches ausgestattet ist.

2.3 Cache-Angriffe

Cache-Angriffe beschreiben eine generelle Klasse von Mikroarchitektur-Seitenkanalangriffen, welche den Cache verwenden, um Informationen abzugreifen, wobei der Cache als geteilte Ressource zwischen verschiedenen Prozessen fungiert. Durch diesen Angriff können sichere und unsichere Prozesse über den geteilten Cache trotz höher liegender Schutzmechanismen wie virtualisiertem Speicher oder Hypervisor-Systemen kommunizieren. Eine Angreiferin könnte ein Programm entwickeln, welches Informationen über den inneren Zustand eines anderen Prozesses sammelt, und so AES-Schlüssel [Ber05] sowie RSA-Schlüssel [Per05] abgreifen auch über die Grenzen von virtuellen Maschinen hinweg.

2.3.1 Flush-and-Reload

Diese Angriffstechnik wurde 2014 von Yuval Yarom und Katrina Falkner veröffentlicht [YF14]. Ausgang dieses Angriffs ist ein architekturspezifischer Flush-Befehl wie etwa der x86-Assemblerbefehl *clflush*, welcher eine Adresse entgegennimmt und die dazugehörige Cache-Line invalidiert. Dadurch müssen die Daten beim nächsten Zugriff aus dem

Hauptspeicher geladen werden. Dies ist die erste Phase des Angriffs und wird als Flush bezeichnet.
Nach dem Flush folgt die Wait-Phase, in welcher die Angreiferin eine bestimmte Zeitperiode wartet und anschließend die Zugriffszeit auf die geflushte Adressse misst.
Zu guter Letzt folgt die Reload-Phase, in welcher die Angreiferin auf die Adresse zugreift. Sofern das Opferprogramm in der Wait-Phase auf die Adresse zugegriffen hat, ist die Zugriffszeit gering, da die Daten bereits im Cache vorliegen. Im anderen Fall müssen die Daten erst aus dem Hauptspeicher geladen werden, womit eine messbar erhöhte Zugriffszeit einhergeht [YF14]. Eine Übersicht der eben beschriebenen Schritte als Pseudocode findet sich in Algorithmus 1.
Ein Nachteil dieser Angriffsmethode ist die Notwendigkeit, dass sich der Opferprozess Speicherseiten mit dem Prozess der Angreiferin teilen muss. Ein typisches Beispiel hierfür sind geteilte Bibliotheken, zum Beispiel .so Dateien unter Linux oder .dll Dateien unter Windows, welche etwa Kryptofunktionen bereitstellen. Des Weiteren ist der clflush-Befehl nicht in jeder Umgebung verfügbar. So ist etwa ein Angriff aus dem Browser heraus mit dieser Methode nicht möglich.

Pseudocode 1: Pseudo-Code für Flush-and-Reload

```
1 Function flushAndReload(address)
2     clflush(address)
3     wait()
4     timestampBefore <- getTimestamp()
5     readMem(address)
6     timestampAfter <- getTimestamp()
7     return timestampAfter - timestampBefore > threshold
```

2.3.2 Prime-and-Probe

Ein erfolgreicher Prime-and-Probe Angriff auf den L3-Cache wurde 2015 von Liu et al. veröffentlicht [LYG+15]. Er unterscheidet sich von Prime-and-Probe Angriffen auf niedrigere Cache-Level durch die aufwendigere Eviction-Set Suche.
Ein *Eviction-Set* sei eine Menge Adressen, welche einen Cache-Eintrag aus einem Cache verdrängen kann. D.h. ein Eviction-Set, welches einen Eintrag aus dem L3-Cache löscht, würde den gleichen Zweck wie der *clflush*-Assemblerbefehl im Flush-and-Reload-Angriff erfüllen. Um einen Eintrag aus dem Cache zu verdrängen, müssen mehrere Adressen der Daten aus dem Eviction-Set von der CPU auf das gleiche Cache-Set wie der zu verdrängende Eintrag abgebildet werden, so dass die Größe eines Eviction-Sets mindestens die Assoziativität des Caches erreichen muss.

Die Idee beim Prime-and-Probe Angriff besteht darin, in einer sich wiederholenden Abfolge zuerst den Cache zu primen, dann das Opferprogramm rechnen zu lassen und anschließend zu proben. In der Priming-Phase werden mittels der Eviction-Sets gezielt Cache-Sets vollständig mit den Daten aus dem Eviction-Set belegt. In der anschließenden Berechnungsphase werden einige Einträge aus den geprimten Cache-Sets vom Opferprogramm verdrängt. Abschließend berechnet die Angreiferin die Summe der Zugriffszeiten auf alle Einträge in einem Eviction-Set. Sofern das Opferprogramm in dem zum Eviction-Set korrelierenden Cache-Set Einträge verdrängt hat, kann die Angreiferin eine Abweichung nach oben in ihrer Messung feststellen, da die verdrängten Einträge eine erhöhte Zugriffszeit verursachen. Somit kann aus den Zugriffszeiten der Eviction-Sets auf die Speicherzugriffe des Opferprogramms geschlossen werden.
Die Eviction-Sets, die zur Durchführung eines Cache-Angriffs notwendig sind, lassen sich nicht immer leicht finden, da das Mapping der virtuellen auf die physischen Adressen in manchen Umgebungen nur eingeschränkt zugänglich ist. So ist aber in der Regel das Mapping der virtuellen und physischen Adressen bei den Adressbits der Speicher-Pages identisch, welche etwa unter Windows und Linux zur Zeit 4096 Bytes groß sind. Somit ist in diesem Fall garantiert, dass die untersten 12 virtuellen Adressbits mit den physischen Adressbits identisch sind.

Pseudocode 2: Pseudo-Code für Prime-and-Probe

```
1 Function flushAndReload(address)
2     foreach address in evictionSet do
3         readMem(address)
4     wait()
5     timestampBefore <- getTimestamp()
6     foreach address in evictionSet do
7         readMem(address)
8     timestampAfter <- getTimestamp()
9     return timestampAfter - timestampBefore > threshold
```

2.3.3 Timer

Beide eben beschrieben Angriffsmethoden setzen voraus, dass die Angreiferin Laufzeitunterschiede zwischen dem Ladevorgang aus dem Cache und dem Hauptspeicher zuverlässig unterscheiden kann. Beim AIDA64-Benchmark mit einem aktuellen Intel i7-8700K ist die L3-Cache-Latenz im Mittel bei 12 ns, wobei gepaart mit DDR4-3200 CL16 RAM die Hauptspeicherlatenz im Mittel 49 ns beträgt [Hag17]. Somit ist eine Timer-Auflösung von unter 30 ns Voraussetzung für ein erfolgreichen Angriff. Anschaulich ist dies auch im

Diagramm 2.3.3 zu sehen. Die Autoren verwendeten ein i7-4960HQ und haben Zugriffszeitdifferenzen für Cache-Miss und Hit von etwa 70 ns festgestellt. Auf dem Testsystem mit einem Intel i7-4770 und DDR3-1600 RAM konnten mittels *rdtscp* durchschnittliche Zugriffswerte von 66 bei einem Hit und von 248 bei einen Miss Taktzyklen festgestellt werden.

Im Folgenden wird Zeit oft in Taktzyklen gemessen, wobei sich damit immer auf die *rdtscp*-Instruktion bezogen wird. Zu beachten ist hierbei, dass der im Testsystem verbaute i7-4770 je nach Kern und Gegebenheiten wie Last, Temperatur und so weiter von seinem 3,4 GHz Basistakt abweicht. Der Takt des *rdtscp*-Timers ist jedoch immer auf den gleichen Takt [Int16a], im Fall des Testsystems auf 3,4 GHz, fixiert. Somit ergibt die Differenz von 182 Taktzyklen zwischen einem Hit und Miss auf dem Testsystem einen Zeitunterschied von 54 ns.

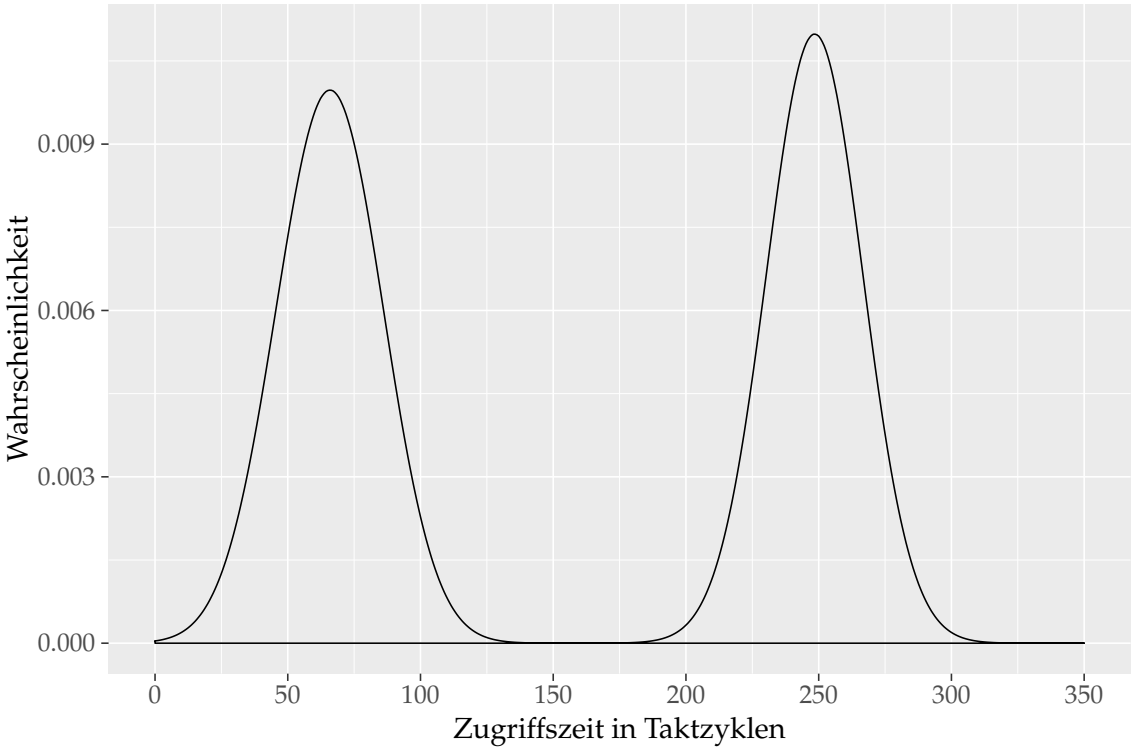

Abbildung 2.2: Wahrscheinlichkeitsverteilung der Zugriffszeiten für eine aus dem Cache verdrängte und dort vorliegende Variable.

2.4 RSA

RSA (Rivest–Shamir–Adleman) ist ein 1978 veröffentlichtes und weitverbreitetes Public-Key-Kryptosystem [RSA78]. Das grundlegende Prinzip hinter RSA ist es, große positive natürliche Zahlen e, d, n zu finden, so dass $(m^d)^e \equiv m \mod n$ gilt, aber schwer ein d zu finden ist, wenn nur e und n gegeben sind.
Der RSA-Algorihtmus besteht aus vier Schritten: Der Schlüsselgenerierung, der Schlüsselverteilung, der Verschlüsselung und Entschlüsselung.
Für die Schlüsselgenerierung werden zwei Primzahlen p und q benötigt, und anschließend wird $n = pq$ berechnet. Die Länge n wird auch als Schlüssellänge bezeichnet.
Eine heute gängige Schlüssellänge ist 2048 Bit, so dass die Bitlänge jeder Primzahl in etwa 1024 Bit beträgt. Um derart große Primzahlen in akzeptabler Zeit zu finden, werden probabilistische Primzahltests eingesetzt, welche mit einer hohen Wahrscheinlichkeit garantieren, dass die gefundenen Zahlen prim sind.
Des Weiteren wird $\lambda(n) = kgV(\lambda(p), \lambda(q)) = kgV(p-1, q-1)$ berechnet, wobei λ die Carmichael-Funktion beschreibt. Diese liefert zu jeder natürlichen Zahl n die kleinste Zahl $\lambda(n)$, sodass $a^m \equiv 1 \mod n$ für jedes a gilt.
Sei e eine natürliche Zahl mit $1 < e < \lambda(n)$ und teilerfremd zu $\lambda(n)$, d.h. es gilt $ggT(e, \lambda(n)) = 1$. Weiter sei $d \equiv e^{-1} \mod \lambda(n)$.
Im Rahmen dieser Arbeit liegt der Fokus darauf, während der Schlüsselgenerierung Informationen über die generierten Primzahlen p und q abzugreifen.

2.5 Chinesischer Restsatz

Beschrieben sei ein System linearer Kongruenzen [GAJ98]:

$$\begin{aligned} x &\equiv a_1 (\mod m_1) \\ x &\equiv a_2 (\mod m_2) \\ &\vdots \\ x &\equiv a_3 (\mod m_3) \end{aligned}$$

Für dieses System sollen alle x bestimmt werden, welche alle Gleichungen erfüllen. Wenn alle $m_1, m_2, ..., m_n$ paarweise teilerfremde natürliche Zahlen sein, so existiert für jedes ganzahlige Tupel $a_1, ..., a_n$ genau eine Lösung $x \in \{0, 1, ..., kgV(m_1, m_2, ..., m_n)\}$ [D.L18].
Sofern alle Moduli m_i Primzahlen sind vereinfacht sich $kgV(m_1, m_2, ..., m_n)$ zu $\prod_i m_i$.
Eine konkrete Lösung für x ist etwa mit Hilfe des erweiterten euklidischen Algorithmus berechenbar.

2.6 Javascript und Webassembly

Javascript ist eine interpretierte Hochsprache, welche zusammen mit HTML und CSS die wichtigsten Komponenten für das World Wide Web bildet. Interaktive Webseiten und Applikationen werden erst mit Hilfe von Javascript möglich, weshalb jeder moderne Browser eine Javascript-Engine mitbringt.
Webanwendungen mit Javascript sollen auf unterschiedlichsten Endgeräten vom Laptop über das Smartphone bis zur Heimkonsole laufen, so dass ein Vorhalten von kompilierten Binärdateien für jedes dieser Geräte schwierig ist. Dennoch soll die Webanwendung beim Anwender ohne lange Ladezeiten auskommen, womit die Kompilierung auf dem Anwendungsgerät als Lösung entfällt.
Unter anderem deshalb ist Javascript eine interpretierte Sprache, welche jedoch ein Geschwindigkeitsnachteil gegenüber vorkompilierten Sprachen wie C oder Java hat. Es wurden viele Anstrengungen unternommen, diesen designbedingten Performancenachteil auszugleichen. So erkennt etwa Googles Javascript-Engine V8 häufig genutzte, langsam laufende Codeteile und optimiert diese während der Laufzeit [Hab15].
WebAssembly ist ein W3C-Webstandard, welcher ein binäres Format für ausführbaren Code in Webseiten definiert. Dieser Standard wurde 2017 ins Leben gerufen, um den Performancenachteil von Javascript gegenüber nativen Code zu reduzieren, sowie eim kompakte Code-Repräsentation zu bieten, und wird von allen gängigen Webbrowsern unterstützt.
Des Weiteren ermöglicht WebAssembly die Entwicklungen von Webanwendungen in anderen Sprachen wie etwa C, dessen Code abschließend in ein binäres Format, Wasm-Modul genannt, kompiliert wird. Dieser ist aber weiterhin maschinenunabhängig, sodass nur ein Wasm-Modul vorgehalten werden muss. Diese Module können von Javascript als Bibliotheken geladen und verwendet werden.

2.7 Drive-by Attacks

Die große Herausforderung eines Angriff auf ein Endgerät ist es, den Schadcode auf dem Gerät des Opfers zur Ausführung zu bringen. Eine beliebte Methode ist das so genannte Social-Engineering, welches darauf abzielt, bei Anwendern bestimmte Reaktionen wie etwa das Öffnen eines E-Mail-Anhangs hervorzurufen. Allerdings muss hier das Opfer selbst aktiv mitwirken, um dem Angriff zum Erfolg zu verhelfen.
Im Gegensatz dazu steht der Drive-by-Attack [Sim16], bei dem das Opfer bereits durch den Besuch einer Website angegriffen wird. Zum einen ist es möglich, durch Lücken im Browser beliebigen Code auf dem Gerät des Opfers auszuführen. Zum anderen, wie in der vorliegenden Arbeit dargestellt, können die vorhandenen Mittel wie Scriptsprachen

für einen Angriff genutzt werden. Die Erfolgswahrscheinlichkeit eines Angriffs ist im letzteren Fall bedeutend größer, da bereits die von der Angreiferin geschalteten Werbeanzeigen auf häufig aufgerufenen Webseiten für einen umfangreichen Pool von Opfern sorgen. Hinzu kommnt, dass die meisten Opfer den Angriff nicht zu ihrem Ursprung zurückverfolgen können, da sie die Website mit der bösartigen Werbeanzeige regelmäßig besuchen. Somit ist das Schreiben von Drive-by-Angriffen zunächst aufwändiger und setzt in der Regel die Kenntnis unbekannter oder nicht gepatchter Sicherheitslücken voraus, verspricht dafür aber eine deutliche höhere Erfolgswahrscheinlichkeit als Angriffe bei denen das Opfer, etwa durch die aktive Ausführung von Code, mitwirken muss.

2.8 Speicher-Disambiguierung

Heutige Prozessoren, wie etwa die Intel-Core-Reihe, führen Store- und Load-Befehle Out-of-Order aus und stehen damit vor der Aufgabe, auf Datenabhängigkeiten zu reagieren [Hen14]. Die Techniken für eine Speicher-Disambiguierung erkennen echte Abhängigkeiten zwischen Speicheroperationen während der Ausführung und erlauben es der CPU, zu einem vorherigen Zustand zurückzukehren, sobald eine Abhängigkeit verletzt wurde.
Die Möglichkeit, Load- und Store-Befehle Out-of-Order auszuführen, sorgt für eine erhöhte Parallelität auf Instruktionsebene und eine damit verbesserte Single-Thread-Performance. Grafik 2.8 zeigt ein Beispiel für die Speicher-Disambiguierung.

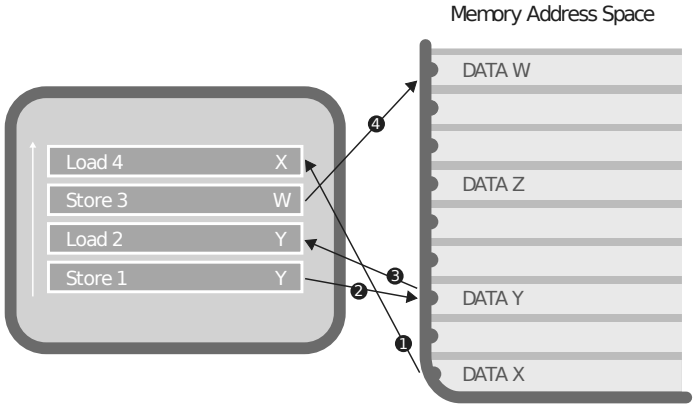

Abbildung 2.3: Beispiel für eine Speicher-Disambiguierung. Die Nummern in den Kreisen geben die chronologische und der weiße Pfeil am linken Rand die logische Ausführungsreihenfolge an. Load 2 kann nicht früher ausgeführt werden, da er von Store 1 abhängig ist. Load 4 hingegen ist von den anderen Operationen unabhängig und kann daher vor Store 1 und Store 3 ausgeführt werden. Durch diese vorgezogene Ausführung können Instruktionen, die den Wert von X benötigen, im Folgenden von einer geringen Zugriffslatenz profitieren. [Hel06].

Um diese Techniken umzusetzen, werden Store-Instruktionen bei der Ausführung nicht direkt an das Speichersystem, in diesem Fall den CPU-Cache, übermittelt. Stattdessen werden die Store-Instrukionen inklusive der Speicheradressen und Daten in einer Store-Queue gepuffert. Die Werte werden erst an das Speichersystem übergeben, wenn alle vorherigen Befehle im Code komplett abgeschlossen wurden. Dies vermeidet Write-after-Read(WAR)-Konflikte, bei denen ansonsten ein früheres LOAD einen inkorrekten Wert vom Speichersystem lesen würde, weil die Ausführung eines späteren STORES vorgezogen wurde. Auch Write-after-Write(WAW)-Konflikte werden dadurch gelöst, da keine früheren STORES ihre Werte nach späteren STORES in das Speichersystem übermitteln, wie es ohne STORE-Queue bei der Out-Of-Order-Ausführung der Fall wäre.
Des Weiteren ermöglicht die Store-Queue die spekulative Ausführung von bedingten Verzweigungen, deren Richtung zum Ausführungszeitpunkt noch nicht bekannt ist. Wenn ein Pfad falsch geraten wurde, muss der berechnete Pfad verworfen und alle STORES rückabgewickelt werden. Ohne die Store-Queue wäre dies nicht möglich, da in der Zwischenzeit die geschriebenen Werte von anderen Cores gelesen sein könnten und somit der Systemzustand korrumpiert wäre.
Jedoch schafft die Store-Queue auch ein neues Problem. Angenommen ein STORE wird ausgeführt und seine Adresse und seine Daten werden in der Store-Queue gepuffert. Kurz danach liest ein LOAD von derselben Adresse, auf die der STORE geschrieben hat. Würde der LOAD den Wert vom Speichersystem lesen, wäre der Wert veraltet, da der im Code vorher stehende STORE noch nicht übermittelt wurde.
Um diesem Problem zu begegnen, nutzten Prozessoren in der Store-Queue die store-to-load-forwarding-Technik. Diese veranlasst die Store-Queue, abgeschlossene STORES, die noch nicht an den Speicher übermittelt wurden, zu späteren LOADS weiterzuleiten. Bei der Ausführung eines LOADS wird die als assoziativer Speicher umgesetzte Store-Queue nach STORES auf derselben Adresse durchsucht, welche in der logischen Ausführungsreihenfolge vorher auftauchen. Sofern es einen Treffer gibt, wird der Wert des abgeschlossenen STORES aus der Store-Queue anstatt des veralteten Wertes aus dem Speichersystem verwendet.
Die verbesserten Eviction-Set Suche in Abschnitt 3.3 nutzt ein Verhalten der Speicher-Disambiguierung von Intel-Prozessoren aus.

2.9 Angegriffene Hardware und Software

Der Testrechner war ein Dell Precision T1700 mit einem Intel Core i7-4770, dessen 4 physischen Kerne beziehungsweise 8 virtuellen Kerne einen Grundtakt von 3,4GHz bieten. Der geteilte L3-Cache ist 8 MiB groß, besitzt eine Assoziativität von 16, eine Cache-Line-

Größe von 64 Byte und 8192 Cache-Sets, die in 4 Slices aufgeteilt sind. Weiterhin ist ein Intel C226 Chipsatz und 8GB DDR3-1600 verbaut. Als Betriebsystem wird Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-131-generic x86_64) verwendet. Die getesteten Browser sind Chromium 68.0.3440.106, Firefox 62 sowie Firefox Developer Edition 63.
Mozilla Network Security Services(NSS) [Moz18a] ist eine Menge von Bibliotheken, welche eine plattformübergreifende Entwicklung von sicheren Client- und Server-Anwendungen anstrebt. Dabei wird unter anderem TLS oder S/MIME implementiert. Mozilla NSS wird etwa im Firefox-Browser und der Mail-Anwendung Thunderbird eingesetzt. Der Quellcode ist unter der Mozilla Public License verfügbar und kann online etwa im Firefox-Repository [Moz18b] eingesehen werden.
OpenPGP ist ein Standard für das Signieren, Ver- und Entschlüsseln von Daten, welcher im RFC 4880 [JC07] definiert wird. OpenPGP.js [Ope18] ist eine Opensource-Implementierung des OpenPGP-Protokolls in Javascript, welche sich zum Ziel gesetzt hat, OpenPGP auf einer breiten Palette von Endgeräten zu ermöglichen.

3 Implementierung

Das folgende Kapitel beschreibt, mit Hilfe welcher Softwaretools der Cache-Angriff implementiert wird.
In dem in dieser Arbeit verwendeten praxisnahen Angriffsmodell reicht für den Start eines Angriffs der Besuch des Opfers auf einer vorher präparierten Website, auch Drive-By-Angriff genannt, aus. Dafür genügt bereits eine eingebundene JavaScript-Werbeanzeige, die von der Angreiferin kontrolliert wird. Gegenüber Angriffen, die ein Ausführen von nativem Code auf dem Endgerät des Opfers verlangen, ist mit diesen Voraussetzungen ein deutlich größerer Angriffsvektor gegeben. Da aufgrund des Angriffsmodells nur Webtechnologien verfügbar sind, liegt der komplette Angriffscode in JavaScript und Webassembly vor. Frühere Implementierungen von Cache-Angriffen im Browser [OKSK15] hatten noch keine Möglichkeit, Webassembly zu verwenden, weshalb deren kompletter Angriffscode in JavaScript geschrieben war. Webassembly ermöglicht hardwarenähere Programmierung und den Vorteil, dass der Code anders als in JavaScript nicht während der Laufzeit optimiert werden muss. Des Weiteren steht mit dem Emscripten-Compiler ein Tool bereit, welches die Übersetzung von C-Code in Webassembly anbietet. Somit kann ein bestehender Angriffscode in C, in diesem Fall von Mastik [Yar18], als Grundlage verwendet werden, und eine komplette fehleranfällige Neuimplementierung in JavaScript entfällt.

3.1 Timer in JavaScript

Der hier ausgeführte Cache-Angriff benötigt, wie im Grundlagenkapitel beschrieben, präzise Timer, welche eine Auflösung von unter 30 ns bereitstellen sollten. Dennoch könnte die Suche nach *Eviction-Sets* auch mit schlechteren Timerauflösungen bewerkstelligt werden, indem Operationen mehrfach ausgeführt werden und die Differenz der aufsummierten Zeiten zur Bewertung herangezogen wird. Im *Eviction-Set*-Algorithmus könnte etwa die Funktion $checkevict$ wie in Algorithmus 3 angepasst werden, wobei der Parameter $repeatIterations$ abhängig von der Timerauflösung gewählt wird. Es besteht jedoch das Problem, auch schwache Aktivitäten im Cache-Set während des eigentlichen Angriffs aufzudecken, da im Worst-Case nur ein Eintrag aus dem beobachteten Cache-Set verdrängt wird und somit lediglich die Zugriffszeit zwischen einem Hit und einem Miss ausschlaggebend ist. In diesem Fall könnte die Dauer mehrerer Prime-and-Probe-

Iterationen gesamtheitlich gemessen werden, und zwar unter der Vermutung, dass auf die für die Verdrängung verantwortliche Adresse über die Zeit mehrfach zugegriffen wird. Aus einem niedrig aufgelösten Timer folgt also eine geringere zeitliche Auflösung von Cache-Aktivitäten oder die Nichtregistrierung von schwachen Cache-Aktivitäten.

Pseudocode 3: Pseudo-Code für $checkevict$ im Fall von einer niedrig aufgelösten getTimestamp

```
1 Function checkevict(possibleEvictionSet, witness)
2     timestampBefore <- getTimestamp()
3     for i=1 to repeatIterations do
4         accessMemory(possibleEvictionSet)
5         accessMemory(witness)
6     timestampAfter <- getTimestamp()
7     return timestampAfter - timestampBefore > threshold
```

Der W3C hat die High-Resolution-Time-API spezifiziert, welche die Methode performance.now beinhaltet, die einen aktuellen Timestamp zurückgibt. Im Firefox hatte die Methode in früheren Versionen eine hinreichend genaue Auflösung im Nanosekundenbereich, wobei in Reaktion auf die Sicherheitslücken Meltdown und Spectre die Auflösung schrittweise auf 2 ms im aktuellen Firefox 60 abgesenkt wurde. Auch in den Browsern Edge und Chrome wurden im Zuge der Veröffentlichung von Meltdown und Spectre die Auflösung von perfomance.now() verringert. Darüber hinaus wird bei beiden Browsern auf den zurückgegebenen Timestamp ein Timerjitter addiert.
So bieten Edge und Chrome zurzeit (Stand Juni 2018) eine Auflösung von 20 µs + 20 µs Jitter respektive 100 µs + 100 µs Jitter. Das Paper "Fantastic Timers and where to find them" [SMGM17] beschreibt diverse andere Methoden, um mit Hilfe von JavaScript Timer zu generieren. Es werden etwa CSS-Animationen und Nachrichtenkanäle als mögliche Zeitgeber untersucht. Allerdings ist nur eine geeignete Methode dabei, da die Auflösung aller anderen mindestens im hohen einstelligen µs Bereich liegt und somit der Parameter $repeatIterations$ auf Werte von etwa 1000 gesetzt werden müsste, um zuverlässig *Eviction Sets* zu finden. Hierdurch würde die benötigte Ausführungszeit zum Finden der *Eviction Sets* auf ein Maß ansteigen, welches nicht mehr zum angenommenen Angriffsmodell passen würde.
Als einziges angemessenes Zeitmessungswerkzeug verbleibt der SharedArrayBuffer aus Javascript. Die Ausführung von Javascript geschieht in einem Thread, wobei die Möglichkeit besteht, so genannte Webworker zu starten, welche Code aus einem Skript in einem eigenen Thread ausführen. Der Speicherbereich eines Webworkers und des Mainthreads sind strikt getrennt, sodass Daten ursprünglich über Nachrichten ausgetauscht werden mussten. Hier setzt der SharedArrayBuffer an, welcher einen geteilten Speicherbereich

zwischen Mainthread und Webworker definiert.

Gemäß Code-Listing 3.1 wird im Mainthread zuerst ein SharedArrayBuffer von 4 Bytes angelegt. Anschließend wird der als Zeitgeber fungierende Webworker gestartet und ihm eine Referenz auf den eben angelegten SharedArrayBuffer übersandt.

```
1  var sharedArrayBuffer = new SharedArrayBuffer(4);
2  var counterWorker = new Webworker('counterWebworker.js');
3  counterWorker.postMessage(sharedArrayBuffer);
4  var sharedArrayBufferUin32Array = new Uint32Array(sharedArrayBuffer);
5
6  function measureTime(func){
7      var t1 = Atomics.load(sharedArrayBufferUin32Array[0]);
8      func();
9      var t2 = Atomics.load(sharedArrayBufferUin32Array[0]);
10     return t2 - t1;
11 }
```

Listing 3.1: main.js: Code für Zeitmessungen mittels counterWorker

Die Zählvariable soll hier eine Größe von 32 Bits haben, weshalb abschließend ein `Uint32Array` definiert wird, dessen Inhalt auf den SharedArrayBuffer referenziert. Ein Zählvariable kann nun durch Lesen des ersten Eintrags des Arrays erhalten werden. Problematisch ist jedoch, dass auf die Zählvariable sowohl lesend vom Mainthread als auch schreibend vom Webworker zugegriffen wird. Dadurch können die im Mainthread gelesenen Werte veraltet sein, da der SharedArrayBuffer noch nicht zwischen beiden Threads synchronisiert wurde. Abhilfe schafft hier die von Javascript bereitgestellte Atomics-Library, welche es ermöglicht, die Leseoperation atomar auszuführen.

Der Webworker iteriert nun in einem eigenen Thread die Zählvariable in einer Endlosschleife (siehe auch Pseudocode 3.2). Zuerst wird dazu dem Webworker via message die Referenz auf einen im Mainthread erstellten SharedArrayBuffer übergeben. Anschließend wird im Webworker ein `Uint32Array` angelegt, welches mit dem übergebenen SharedArrayBuffer verknüpft ist. Zum Schluss geht der Webworker in die Endlosschleife über, in welcher die Zählvariable *sharedArray[0]* durchgehend iteriert.

```
1  self.addEventListener('message', (m) => {
2    // Create an Uint32Array on top of the shared memory array
3    const sharedArray = new Uint32Array(m.data);
4    while{true}{
5      sharedArray[0]++;
6    }
7  });
```

Listing 3.2: counterWebworker.js: Iterieren der Zählvariable in einer Endlosschleife mittels Webworker

Das Iterieren einer Variable benötigt nur wenige Taktzyklen, wodurch der aktuelle Wert der Zählvariable als Zeitstempel interpretiert werden kann. Die Auflösung dieser Methode hängt also von der Geschwindigkeit der Iteration sowie der Speichersynchronisation des SharedArrayBuffers zwischen Mainthread und Webworker ab.
In Versuchen mit verschiedenen Webbrowsern und Hardwarekonfigurationen zeigte sich, dass die Auflösung mindestens im einstelligen Nanosekundenbreich liegt und somit ausreichend genau ist, um den Unterschied zwischen einem Cache-Miss und Hit festzustellen (siehe Tabelle 3.1).

Tabelle 3.1: Zeitauflösung des SharedArrayBuffer-Zählers mit verschiedenen Browsern auf Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-131-generic x86_64) mit einem i7-4770. Wertebereich der `Uint32` Zählvariable wird in der linken Spalte ausgeschöpft, in der rechten hingegen wird nur bis 2^{31} gezählt.

	Zählen bis 2^{32}	Zählen bis $2^{31}-1$
Chromium 68.0.3440.106	∼2,7ns	∼3ns
Google Chrome 69.0.3497.81	∼2,7ns	∼3ns
Firefox 63.0b4	∼5,1ns	∼2,2ns

Um die Auflösung zu bestimmen, wird die Javascriptfunktion performance.now zur Hilfe genommen (siehe Listing 3.3). Die Funktion `wait_edge` ruft zuerst performance.now für den Startwert auf und wartet anschließend in einer Endlosschleife, bis performance.now einen höheren Wert als den Startwert zurückgibt. Dieser höhere Wert wird ebenso wie der aktuelle Stand der Zählvariable gespeichert. Dann folgt ein erneuter Aufruf von `wait_edge`, wobei beim Zurückkehren wieder der letzte performance.now-Wert sowie der Wert der Zählvariable gespeichert wird. Für die performance.now-Funktion ist die Auflösung bekannt, sodass aus den Differenzen der performance.now-Werte und der Werte der Zählvariable eine Auflösung für den Timer errechnet werden kann. Wie oben beschrieben addiert Chrome auf den performance.now-Wert einen Timerjitter, und deshalb wurde diese Prozedur 20000 mal durchgeführt, um valide Mittelwerte zu erhalten.
Die Zählvariable besitzt den Datentyp `Uint32`, und in Javascript ergibt eine Iteration des Wertes $2^{32}-1$ wieder 0. Daher ist es naheliegend, den gesamten Wertebereich von 0 bis $2^{32}-1$ auszuschöpfen und bei Messwerten über 2^{31} anzunehmen, dass in diesem Zeitraum ein Overflow stattfand. Beim Testen mit Firefox zeigte sich jedoch, dass die Iteration der Zählvariable ab dem Wert 2^{31} signifikant langsamer wird, wobei dieses Phänomen mit Chrome nicht zu beobachten war. Als Workaround wurde eine Abfrage hinzugefügt, die bei einem Überschreiten von 2^{31} die Zählvariable auf 0 zurücksetzt. Die Ergebnisse zeigen, dass dieser Workaround die Auflösung in Chrome nur minimal verschlechtert, dafür aber in Firefox signifikant verbessert. Ohne diese Änderungen sorgt der Timer in Firefox für

Probleme bei der Zeitmessung, da ein Zeitintervall beim Überschreiten des Wertes 2^{31} wegen der verringerten Iterationsgeschwindigkeit als deutlich verkürzt wahrgenommen wird.

```
1  var start = wait_edge();
2  var start_count = Atomics.load(Module['sharedArrayCounter'], 0);
3  var end = wait_edge();
4  var end_count = Atomics.load(Module['sharedArrayCounter'], 0);
5  nsPerTick += (end - start) * 10^6 / (end_count - start_count);
6
7  function wait_edge() {
8    var next, last = performance.now();
9    while ((next = performance.now()) == last) {}
10   return next;
11 }
```

Listing 3.3: main.js: Code zur Bestimmung der Timerauflösung

Diese Methode geht allerdings mit dem Nachteil einher, dass der Webworker-Thread in der Messphase einen CPU-Kern komplett auslastet. Das heißt, dass im Angriffsszenario das Opferprogramm, der Javascript-Mainthread und der Webworker gleichzeitig rechnen, sodass mindestens 3 CPU-Kerne benötigt werden. Sofern sich der Webworker einen physischen Kern mit einen anderen aktiven Prozess teilt, können die gemessen Zeiten einer stärkeren Volatilität durch die erhöhte Iterationsdauer unterliegen. Folglich reduziert sich die Auflösung des Zeitgebers, wobei eine ausreichende Genauigkeit dennoch gegeben ist, da beide Prozesse in etwa die gleiche Rechenzeit zugesprochen bekommen.
Aufgrund der Option, den SharedArrayBuffer als Timer zweckzuentfremden, wurde dieser im Zuge der Veröffentlichung von Meltdown und Spectre in allen gängigen Webbrowsern deaktiviert [Wag18]. Jedoch planen die Hersteller, das Feature in Zukunft wieder zu aktivieren, sobald die Gefahr von Angriffen wie Meltdown und Spectre reduziert ist. Google ist der erste Hersteller, der in seinem Chrome-Browser mit Version 68 die SharedArrayBuffer wieder aktiviert hat [V818]. Als logische Konsequenz wurde angekündigt, in Zukunft erneut hochauflösende Timer bereitzustellen, da ein SharedArrayBuffer genau diese Möglichkeit schon jetzt bereitstellt [Rei18].
Aus diesen Gründen wird im Folgenden davon ausgegangen, dass das Opfer SharedArrayBuffer in seinem Webbrowser aktiviert hat.

3.2 Eviction-Set-Algorithmus in der Javascript-Umgebung

Der wichtigste Teil für einen Prime-and-Probe-Angriff ist die Fähigkeit, zuverlässig Eviction-Sets zu finden. Wie im Grundlagenkapitel beschrieben, führt die CPU das Cache-Mapping anhand der physischen Adressen durch. Webassembly emuliert eine

32-Bit-Umgebung, welche die internen Adressen in virtuelle Adressen des Hostprozesses, hier des Browsers, übersetzt. Webassembly-Code verwendet nur Adressen der emulierten Umgebung und hat keinerlei Zugriff auf das Mapping zu den virtuellen Adressen.
Somit sind die physischen Adressen in Webassembly durch gleich zwei Abstraktionsschichten geschützt. Jedoch lässt sich für das Finden der Eviction-Sets die Eigenschaft ausnutzen, dass im Betriebssystem 4-KiB-Pages existieren, sodass die letzten 12 Bits der virtuellen und physischen Adresse identisch sind. Des Weiteren alloziert Webassembly 4 KiB große Blöcke, die zur Übereinstimmung der 12 letzten Bits der Webassembly-Adresse mit der virtuellen und physischen führen.
Um aus dieser Eigenschaft Kapital zu schlagen, wird ein Array angelegt, der mindestens die Größe des L3-Caches in Webassembly hat. Im Folgenden soll der Intel i7-4770 mit 8 MiB großem L3-Cache erneut als Basis dienen. In diesem Array sind nun x Blöcke der Größe 4 KiB, deren letzte 12 Adressbits mit der physischen Adresse übereinstimmen. Im sogenannten Adresspool seien nun die Adressen des Arrays, bei denen die letzten 12 Bits gleich sind, also insgesamt x Stück.
Der i7-4770 besitzt 8192 Cache-Sets, die auf 4 Slices aufgeteilt sind, wobei für das Mapping der 2048 Cache-Sets innerhalb eines Slices nur die untersten 17 Bits der physischen Adresse relevant sind (siehe auch Abbildung 3.2. Dabei bestimmen die Bits 6 bis 17 eindeutig das Cache-Set und die Bits 0 bis 5 das Offset innerhalb der Cache-Line.

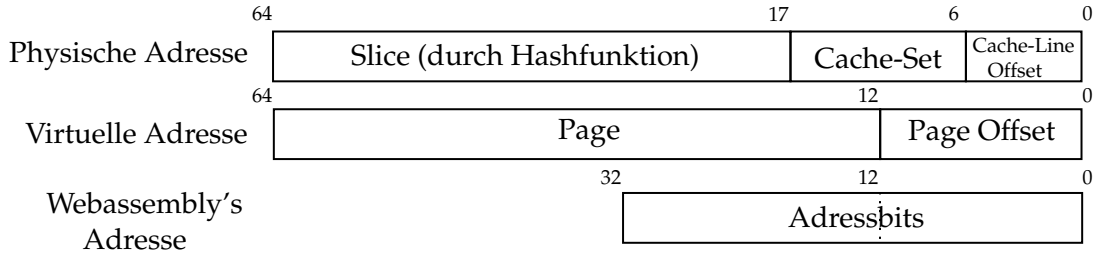

Abbildung 3.1: Physischer, virtueller und Webassembly's Adressraum auf dem Intel i7-4770 mit vier Slices und insgesamt 8192 Cache-Sets im Vergleich [GPTY18]. Die letzten 12 Bits einer Adresse stimmen in allen drei Adressräumen überein.

In welchem der 4 Slices die Daten landen, wird anhand der Adressbits 18 bis 63 bestimmt. Durch die Kenntnis der untersten 12 Bits der physischen Adresse sind gleichzeitig 6 Bits (6 bis 11) bekannt, welche für die Zuordnung zu den Cache-Sets verantwortlich sind.
Angenommen, im Pool sind ausschließlich Adressen, bei denen die letzten 12 Bits auf 0 gesetzt sind. Somit ist ein Abstand von 2^{12} für aufeinanderfolgende Adressen gegeben und jede Adresse lässt sich genau einem der 4-KiB-großen Blöcke zuordnen. Dann kann erwartet werden, dass im Mittel jede 128. Adresse auf das gleiche Cache-Set gemappt wird. Es gibt 8192 Möglichkeiten, eine Adresse einem Cache-Set zuzuordnen, also 13 Bits an Unsi-

cherheit. Durch die Kenntnis der untersten 12 Bits der Adresse sind davon 6 Bits bekannt, welche für eine eindeutige Zuordnung sorgen. Es bleiben noch 7 Bits an Unsicherheit, die durch Kenntnis der restlichen Adressbits beseitigt werden könnten.

3.2.1 Eviction-Set-Suchalgorithmus

Im Folgenden soll der Algorithmus beschrieben werden, der die Adressen im Pool verschiedenen Eviction-Sets zuordnet. Dieser Algorithmus ist in der Lage, Adressen Cache-Sets zuzuordnen, ohne Näheres über die CPU (L3-Cache Größe usw.) und die Adressbits 12 bis 63 zu wissen. Der Algorithmus basiert auf den von [GPTY18] und [DKPT17] beschriebenen Algorithmen zum Finden von Eviction-Sets, wobei einige Optimierungen und Modifikationen implementiert und getestet wurden.
Der Eviction-Set-Konstruktionsalgorithmus besteht aus drei Hauptphasen, der Expand-, Contract- und Collect-Phase. Zu Anfang wird zufällig eine Zeugenadresse aus dem Adresspool ausgewählt. Die Annahme in der Expand-Phase ist, dass eine bestimmte Teilmenge der Adressen aus dem Pool, genannt Candidate-Set, ein Eviction-Set für die Zeugenadresse bildet, sofern der Pool groß genug ist. Um ein Candidate-Set zu testen, wird zuerst auf die Zeugenadresse zugegriffen, um sicherzustellen, dass diese im Cache landet. Danach wird auf alle Adressen aus dem Candidate-Set zugegriffen und abschließend die Zugriffszeit auf die Zeugenadresse gemessen. Sofern das Candidate-Set ein Eviction-Set für die Zeugenadresse ist, werden die Daten der Zeugenadresse aus dem Cache verdrängt, womit bei einer erneuten Messung der Zeugenadresse eine erhöhte Zugriffszeit gemessen wird. Dieser Vorgang wird mehrmals wiederholt, um den Einfluss des Timer- und System-Rauschens zu vermindern.
In der Expand-Phase wird dem Addresspool iterativ eine zufällige Adresse entnommen, welche in dieser Iteration die Zeugenadresse ist (siehe auch Pseudocode 4). Nach jeder Iteration wird, wie eben beschrieben, getestet, ob das Candidate-Set ein Eviction-Set für die Zeugenadresse ist. Falls dies zutrifft, wird zur nächsten Phase übergegangen, andernfalls wird die Zeugenadresse dem Candidate-Set hinzugefügt und mit der nächsten Iteration fortgefahren.
Im Allgemeinen beinhaltet das Candidate-Set nach der Expand-Phase mehrere hundert Einträge, von denen eine Teilmenge der Größe 16 bereits ein Eviction-Set für die Zeugenadresse bilden würde. Der überwiegende Teil der Einträge gehört nicht zum gleichen Cache-Set wie die Zeugenadresse. Diese überflüssigen Einträge würden den Prime-and-Probe-Vorgang erheblich verlangsamen. Deshalb wird in der Contract-Phase versucht, das Candidate-Set auf die Größe 16 zu reduzieren (siehe auch Pseudocode 5). Hierzu wird ein Element aus dem Candidate-Set entfernt und dann erneut getestet, ob dieses reduzierte Candidate-Set noch ein Eviction-Set für die Zeugenadresse ist. Falls ja, wird dieses Ele-

Pseudocode 4: Pseudo-Code für Expand-Phase des Eviction-Set Algorithmus

```
1 Function Expand(evictionSet, addressPool)
2     while size(addressPool) > 0 do
3         witness = SelectRandomItem(addressPool)
4         if checkevict(evictionSet, witness) then
5             return witness
6         evictionSet.add(witness)
7     return failed
```

ment wieder dem Adresspool hinzugefügt, andernfalls verbleibt es im Candidate-Set, da es notwendiger Bestandteil des Eviction-Sets ist. Dieser Vorgang wird für jedes Element im Candidate-Set einmal durchgeführt, so dass im fehlerfreien Fall schlussendlich 16 Elemente im Candidate-Set verbleiben. Durch Messrauschen kann auch hier wieder ein Element fälschlicherweise als relevant für das Eviction-Set eingestuft werden, weshalb die Contract-Phase dreimal wiederholt wird. Die zweite und dritte Wiederholung sind weit weniger kostenintensiv, da das Candidate-Set nach einer Contract-Phase bereits um mehrere 100 Einträge bereinigt wurde.

Pseudocode 5: Pseudo-Code für Contract-Phase des Eviction-Set Algorithmus

```
1 Function Contract(evictionSet, addressPool, witness)
2     foreach candidate in evictionSet do
3         evictionSet.remove(candidate)
4         if checkevict(evictionSet, witness) then
5             addressPool.add(candidate)
6         else
7             evictionSet.add(candidate)
```

Wenn im Anschluss sofort wieder eine neue Zeugenadresse aus dem Pool gewählt wird, könnte eine Adresse gewählt werden, welche auf dasselbe Cache-Set wie die vorherige Zeugenadresse abgebildet wird. Deshalb folgt im Anschluss an eine erfolgreiche Contract-Phase die Collect-Phase. In dieser werden alle Adressen aus dem Pool entfernt, welche ebenfalls von dem in der Contract-Phase gefundenen Eviction-Set aus dem Cache verdrängt wurden (siehe auch Pseudocode 6). Durch diesen Schritt wird vermieden, dass die spätere Menge von Eviction-Sets dahingehend überprüft werden müsste, ob Eviction-Sets paarweise dasselbe zugrundeliegende Cache-Set besitzen. Zudem beschleunigt die Collect-Phase die nächsten Iterationen, da weniger Adressen im Pool vorhanden sind. Hierzu wird eine Adresse aus dem Pool durch einen Zugriff in den Cache geladen und anschließend auf alle Einträge im Eviction-Set zugegriffen. Daraufhin wird die Zugriffs-

zeit auf die Adresse gemessen und bei einer erhöhten Zeit aus dem Pool entfernt, da dann die Adresse auf dasselbe Cache-Set wie die Einträge des Eviction-Set beziehungsweise die letzte Zeugenadresse abgebildet wird.

Pseudocode 6: Pseudo-Code für Collect-Phase des Eviction-Set Algorithmus

```
1 Function Collect(evictionSet, addressPool)
2     foreach candidate in addressPool do
3         if checkevict(evictionSet, candidate) then
4             addressPool.delete(candidate)
```

Abschließend wird in der Expand-Phase das Candidate-Set soweit vergrößert, bis es ein Eviction-Set bildet Danach wird es in der Contract-Phase auf die Größe 16 verkleinert, und anschließend werden in der Collect-Phase alle auf dasselbe Cache-Set abgebildeten Adressen aus dem Pool entfernt (siehe auch Pseudocode 7). Das gefundene Eviction-Set wird gespeichert und der Vorgang solange wiederholt, bis der Pool der Adressen erschöpft ist oder aufgrund von Fehlern in einer Phase mehrmals kein Eviction-Set gefunden wurde.

Pseudocode 7: Pseudo-Code für Eviction-Set Algorithmus

```
1  Function EvictionSetFinder(addressPool)
2      evictionSets ← empty
3      while size(addressPool) > 0 do
4          evictionSet ← empty
5          witness ← expand(evictionSet, addressPool)
6          if witness != failed then
7              contract(evictionSet, addressPool, witness)
8              collect(evictionSet, addressPool)
9              evictionSets.add(evictionSet)
10     return evictionSets
```

3.2.2 Wahl der Adresspoolgröße

Um die Anzahl der Blöcke beziehungsweise die Arraygröße in Webassembly sinnvoll zu bestimmen, kann zuerst die vereinfachte Annahme getroffen werden, dass die physischen Adressbits 12 bis 63 der 4-KiB-Blöcke zufällig gewählt sind. Die unbekannte Cache-Mapping-Funktion nimmt nun die zufälligen Bits 12 bis 63 und die auf 0 gesetzten Bits 0 bis 11 entgegen und gibt eines von 128 möglichen Cache-Sets zurück. Ziel ist es, mit einer Poolgröße x und einer hohen Wahrscheinlichkeit für jedes Cache-Set 17 Zuordnungen beziehungsweise ein Eviction-Set zu finden. Für ein Eviction-Set werden wegen der

L3-Cache-Assoziativität 16 Adressen gebraucht, wobei die weitere Adresse für den Anfangskandidaten benötigt wird, welcher zum selben Cache-Set zugeordnet werden muss. Gesucht ist somit die Wahrscheinlichkeit, bei einer Poolgröße x mindestens 17 Zuordnungen zu einem fixen Cache-Set cs bei 128 Möglichkeiten zu finden. Hierfür eignet sich das Urnenmodell für Ziehungen mit Zurücklegen ohne Berücksichtigung der Reihenfolge, wobei die x Adressen im Pool die Ziehungen und die Cache-Sets die 128 verschiedenfarbigen Kugeln repräsentieren. Sei $P(count(cs) >= 17)$ die Wahrscheinlichkeit dafür, dass in der gezogenen Folge mindestens 17 mal das Cache-Set cs auftaucht, unter der Bedingung, dass die Poolgröße $\#add = x$. Leichter ist es in diesem Fall, die Gegenwahrscheinlichkeit zu berechnen, die mit

$$\begin{aligned} P(count(cs) < 17 | \#add = x) &= \left(\sum_{i=0}^{16} P(count(cs) = i | \#add = x) \right) \\ &= \left(\sum_{i=0}^{16} \binom{x}{i} \frac{127}{128}^{x-i} \cdot \frac{1}{128}^{i} \right) \end{aligned}$$

beschrieben ist.
Die Wahrscheinlichkeit des $P(count(cs) >= 17)$ ist etwa bedeutend im ersten Szenario. Angenommen, die Angreiferin hat es auf eine bestimmte Adresse abgesehen und möchte ein korrelierendes Eviction-Set finden. Wie wahrscheinlich ist ein erfolgreicher Angriff beziehungsweise wie hoch ist die Wahrscheinlichkeit, ein korrelierendes Eviction-Set zu finden? Das Diagramm 3.2.2 zeigt die Erfolgswahrscheinlichkeiten für verschiedene x-Werte.
Im zweiten Szenario möchte die Angreiferin in einem komplexen Angriff eine Vielzahl von Adressen in unterschiedlichen Cache-Sets überwachen. Sie interessiert sich nun dafür, wie wahrscheinlich es ist, alle 8192 Cache-Sets zu finden und somit auch die für sie relevanten.
Die Wahrscheinlichkeit, für alle 128 möglichen Cache-Sets jeweils 17 Zuordnungen und damit gleichbedeutend alle 8192 möglichen Eviction Sets konstruieren zu können, ist approximativ mit

$$P(count(cs) >= 17 | \#add = x)^{128} = (1 - P(count(cs) < 17 | \#add = x))^{128}$$

beschrieben. Die Ereignisse sind nicht unabhängig, weshalb diese Vereinfachung zu Ungenauigkeiten führt. Diese sind aber im zur Veranschaulichung gezeigten Bereich unter 0,3 Prozentpunkten und damit visuell nicht identifizierbar.
Die durchgezogene Linie im Diagramm 3.2.2 gibt die Erfolgswahrscheinlichkeiten für verschiedene x-Werte im zweiten Szenario an.

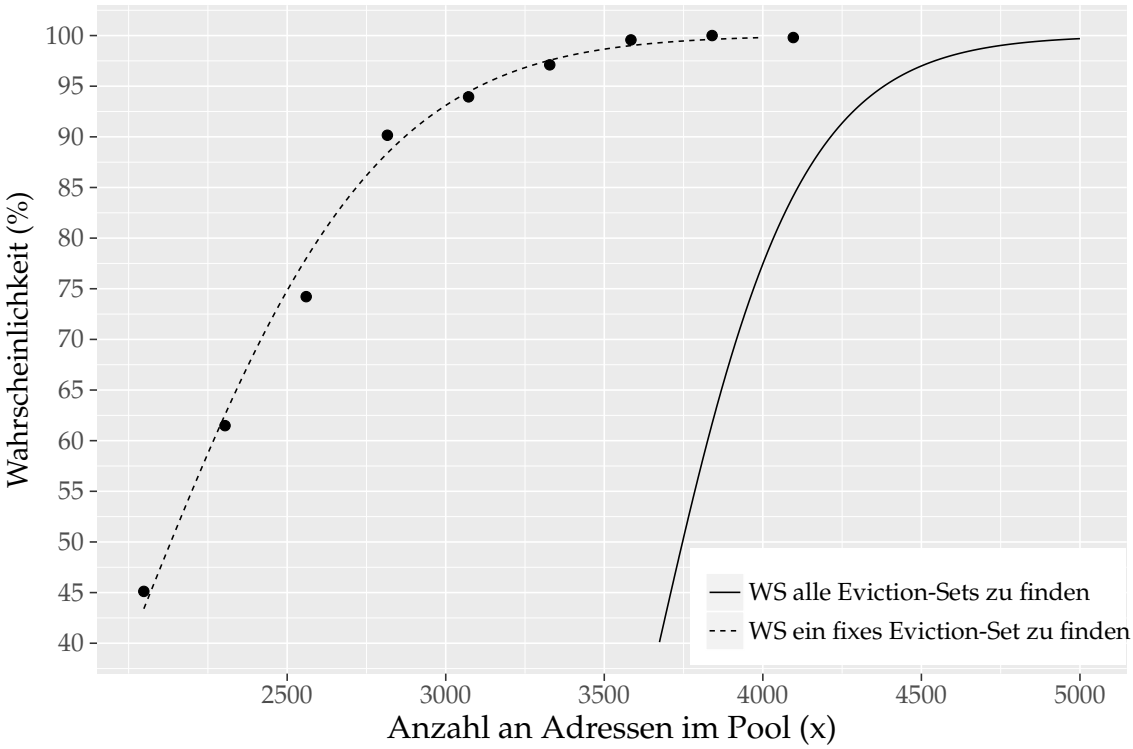

Abbildung 3.2: Die gestrichelte Kurve (Y-Achse) veranschaulicht die Wahrscheinlichkeit, ein fixes Eviction-Set mit einer bestimmten Anzahl an Pooladressen (X-Achse) bei einer Assoziativität von 16 zu finden. Die einzelnen Punkte stellen Werte dar, die mit dem Eviction-Set-Suchalgorithmus in der Praxis ermittelt wurden. Die durchgezogene Linie stellt die Wahrscheinlichkeit dar, alle 8192 Eviction-Sets zu finden.

Häufig wird in Benchmarks angegeben, wie viele Eviction-Sets überhaupt gefunden wurden. Diese Fragestellung ist identisch mit dem ersten Szenario. Wenn ein fixes Eviction-Set mit einer Wahrscheinlichkeit von x Prozent gefunden wird und die Zuordnung wie hier zufällig ist, werden insgesamt im Mittel $x/100 \cdot 128$ Eviction-Sets gefunden.

Ein weiteres wesentliches Kriterium ist die Dauer der Eviction-Set-Suche, Tabelle 3.2 gibt hier Performancewerte für verschiedene Poolgrößen an.

Um Abweichungen zu minimieren wurde jede Poolgröße 60 mal getestet. Angegeben sind die Median-Werte, da Ausreißer existierten deren Laufzeit über 5 Minuten beträgt. In der Praxis würde bei einer Poolgröße von 4096 und einer Laufzeit von über 100 Sekunden die Suche abgebrochen und erneut gestartet werden, um unverhältnismäßig lange Suchzeiten zu vermeiden. Es fällt auf, dass die Poolgröße 4096 ähnlich schnell wie die kleineren Poolgrößen ist. Auffällig ist der hohe Anteil der Expand-Phase an der Berechnungszeit bei der Poolgröße 3072 gegenüber 4096, welcher für den Gleichstand in der Gesamtzeit sorgt. Grund dafür ist, dass mit einer Poolgröße von 3072 im Mittel nur 94% der Eviction-Sets

Tabelle 3.2: Performancewerte des Eviction-Set-Suchalgorithmus, wobei pro Poolgröße 60 Läufe durchgeführt wurden. Angegeben sind die Median-Werte, um den Einfluss langer sporadisch auftretender Läufe zu senken. Alle Werte beziehen sich auf einen Lauf. Spalte „Dauer“ beschreibt die Gesamtlaufzeit der Eviction-Set-Suche. Spalte „Fehlgeschlagene Contract-Phasen“ beschreibt wie viele Iterationen aufgrund von Fehlern in der Contract-Phase abgebrochen wurden. Die letzten drei Spalten beschreiben den Anteil, den die Expand-, Contract- und Collect-Phasen an der Gesamtlaufzeit haben.

Poolgröße	Dauer	Fehlgeschlagene Contract-Phasen	Expand Anteil	Contract Anteil	Collect Anteil
3072	45s	191	25%	72%	2.9%
3584	49s	127	20%	76%	3.1%
4096	46s	392	14%	84%	3.0%
4608	54s	256	12%	84%	3.8%

gefunden werden und somit die Adressen, welche keinem Eviction-Set zugeordnet werden können, bis zum Ende im Adresspool verbleiben. Somit ist die Anzahl der Adressen im Pool gegen Ende der Suche größer, womit die Berechnungszeit der Expand-Phasen im Vergleich zu einer initialen Poolgröße von 4096 länger dauert.

Die Contract-Phase nutzt den größten Anteil der Berechnungszeit, da es Abschnitte in den Läufen gibt, in denen das Eviction-Set in der Contract-Phase mehrmals hintereinander nicht ausreichend minimiert werden kann. Dieses Problem existiert auch bei der nativen Implementierung, so wird die Contract-Phase in dem Toolkit Mastik standardmäßig dreimal hintereinander ausgeführt, um das Eviction-Set ausreichend zu reduzieren. Daher ist diese Problematik, zusammen mit dem Lösungsansatz die Contract-Phase mehrmals auszuführen, dafür verantwortlich, dass die Contract-Phase einen hohen Anteil an der Gesamtlaufzeit hat.

Die Dauer der Collect-Phase wird signifikant von der Poolgröße beeinflusst, da sie am Ende einer Iteration alle restlichen Adressen des Pools darauf testen muss, ob diese auf das Cache-Set gemappt werden, für welches in der Iteration ein Eviction-Set gefunden wurde. Aus diesem Grund nimmt die absolute Zeit für den Anteil der Collect-Phase mit der Poolgröße annähernd quadratisch zu, jedoch ist der Anteil bei diesen Poolgrößen mit unter 3s bzw. unter 4% der Gesamtlaufzeit von wenig Bedeutung.

Zusammenfassend stellt eine Poolgröße von 4096 den besten Kompromiss dar, insofern, dass damit einerseits mit über 80% Wahrscheinlichkeit alle Eviction-Sets gefunden werden (siehe Abbildung 3.2.2) und andererseits eine schnelle Laufzeit gewährleistet wird.

3.2.3 Mögliche Optimierungen der Phasen

Ein Eviction-Set muss mindestens die Größe der Cache-Assoziativität besitzen, weshalb es naheliegend ist, in der Expand-Phase nicht mit einem leeren Candidate-Set, sondern mit einem Candidate-Set der Größe der Assoziativität zu starten und so in jeder Expand-Phase Überprüfungen in der Größenordnung Assoziativität minus 1 einzusparen.
Weiterhin ist es nicht optimal, bei jeder Iteration dem Candidate-Set nur eine Adresse hinzuzufügen, da insbesondere bei einem kleinen Candidate-Set eine geringe Wahrscheinlichkeit besteht, dass sich durch eine zusätzliche Adresse dieses zu einem Eviction-Set entwickelt. Die Idee ist folglich, bei einem noch kleinen Candidate-Set in jeder Iteration möglichst viele Adressen aus dem Pool hinzuzufügen und mit zunehmender Größe des Candidate-Sets die Anzahl, der in jeder Iteration hinzukommenden Adressen zu verringern. Dies muss jedoch in Abhängigkeit der schon gefundenen Eviction-Sets geschehen, da in den späteren Iterationen, ein durch die Expand-Phase entstandenes Eviction-Set deutlich kleiner als in den ersten Iterationen ist.

Tabelle 3.3: Performancewerte für Optimierungen des Eviction-Set-Suchalgorithmus, wobei jede Zeile durch 60 Läufe entstanden ist. Angegeben sind die Median-Werte, um den Einfluss langer sporadisch auftretender Läufe zu senken. Alle Tests wurden mit einer Poolgröße von 4096 durchgeführt. Die restlichen Spalten sind analog zu Tabelle 3.2. Die erste Zeile beschreibt den Ansatz die in der Contract-Phase aussortierten Einträge in dem Candidate-Set für die nächste Expand-Phase zu sammeln. Zeile 2 zeigt die Performance wenn die Contract-Phase vorzeitig aufgrund zu wenig entfernter Einträge aus dem Eviction-Set abgebrochen wird.

Beschreibung	Dauer	Fehlgeschlagene Contract-Phasen	Expand Anteil	Contract Anteil	Collect Anteil
Contract-Einträge wiederverwenden	46s	276	12%	85%	2.6%
Contract-Phase vorzeitiger Abbruch	45s	762	21%	74%	3.8%
Ansätze aus Zeile 1 und 2 kombiniert	35s	306	15%	81%	3.2%

Die Idee der Expand-Phase, durch gleichzeitiges Hinzufügen von mehreren Adressen die Laufzeit zu verkürzen, lässt sich mit folgenden Ansatz noch verbessern. Der Standardalgorithmus fügt die in der Contract-Phase aussortierten Adressen wieder dem Adresspool hinzu, worauf die neue Iteration mit einem leeren Candidate-Set gestartet wird. Ein anderer Ansatz wäre es, die in der Contract-Phase aussortierten Adressen direkt als neues Candidate-Set zu verwenden, womit ein Großteil der Laufzeit der nächsten Expand-Phase entfällt. Die Benchmarks mit einer Adresspoolgröße von 4096 zeigen, dass dieser Ansatz

die Eviction-Set-Suche nicht beschleunigen kann (siehe Zeile 1 von Tabelle 3.3). Er senkt zwar den Anteil der Expand-Phase, dafür schlägt die Contract-Phase jedoch mit größeren Eviction-Sets fehl und nimmt damit die in der Expand-Phase gewonnene Berechnungszeit in Anspruch, sodass letztendlich kein Zeitvorteil entsteht. Es hat sich gezeigt, dass es besser ist alle Adressen des Candidate-Sets zurück in den Adresspool zu transferieren, sofern eine Contract-Phase fehlschlägt. Das bedeutet, Teile des Eviction-Sets oder die in der Contract-Phase bis dahin aussortierten Adressen in die nächste Expand-Phase mitzunehmen, hat sich als nachteilig erwiesen.
Wie oben erwähnt, können in einem Durchlauf der Contract-Phase meist nicht alle unnötigen Eintrage des Eviction-Sets identifiziert werden, sodass diese dreimal ausgeführt wird. Bei Beobachtungen der Contract-Phase mit Poolgrößen von etwa 4000 hat sich gezeigt, dass, sofern am Ende der drei Contract-Phasen das Eviction-Set auf die gewünschte Größe von 16 reduziert werden konnte, die Größe des Eviction-Sets nach dem ersten Lauf der Contract-Phase kleiner als 900 und nach dem zweiten Lauf kleiner als 100 war. Diese Beobachtung kann genutzt werden um die Iteration abzubrechen, wenn nach dem ersten Durchlauf der Contract-Phase noch mehr als 900, beziehungsweise nach dem zweiten Durchlauf noch mehr als 100 Einträge im Eviction-Set vorhanden sind (siehe Zeile 2 von Tabelle 3.3).
Die Ergebnisse zeigen, dass der Berechnungsanteil der Contract-Phase gesenkt werden kann. Im Gegenzug wird jedoch die Anzahl der fehlgeschlagenen Contract-Phasen erhöht. Somit steigt auch die Anzahl der Expand-Phasen, beziehungsweise deren Berechnungsanteil, sodass auch dieser Ansatz in der Summe keine Verbesserung erfährt.
Wenn die Ergebnisse der beiden eben betrachteten Ansätze verglichen werden, so zeigt sich, dass einer den Expand-Anteil verringert und den Contract-Anteil erhöht (Zeile 1) und der andere den Expand Anteil erhöht und den Contract-Anteil verringert (Zeile 2). Wie Zeile 3 der Tabelle 3.3 zeigt, ergänzen sich die beiden Ansätze, womit eine Senkung der Dauer von 46s auf 35s erzielt werden kann.

3.2.4 Details der realen Implementierung

Ein false-positive bedeutet, dass ein Candidate-Set kein Eviction-Set für eine Zeugenadresse ist, aber fälschlicherweise als solches erkannt wird. Wie weiter oben beschrieben, wird insbesondere der Test, ob ein Set ein Eviction-Set für eine bestimmte Adresse darstellt, bei positivem Ergebnis mehrfach wiederholt. Hierdurch sollen false-positive Fehler ausgeschlossen werden. Problematisch ist dies vor allem in der Expand-Phase, da das Candidate-Set eine Größe von mehreren hundert Einträgen annimmt und nach jeder Iteration gegen die Zeugenadresse getestet wird. Eine einzige erhöhte Zugriffszeitmessung würde das Candidate-Set fälschlicherweise als Eviction-Set für die Zeugenadresse ein-

ordnen. Es wurde festgestellt, dass erhöhte Zugriffszeiten mehrmals hintereinander auftreten können. Deshalb wird eine erhöhte Messung in der Expand-Phase 20 mal erneut überprüft. Die Überprüfung bricht ab, sobald eine der Messungen eine widersprüchliche Aussage zulässt. Trotz der hohen Anzahl von 20 Wiederholungen sind die Kosten hierfür gering, da im fehlerfreien Optimalfall nur zusätzlich 20 Prüfungen anfallen und in den meisten Fehlerfällen nur einzelne zusätzliche Überprüfungen durchgeführt werden. Demgegenüber steht der Vorteil, nicht fälschlicherweise in die Contract-Phase zu wechseln und dort erst spät zu bemerken, dass das Candidate-Set kein Eviction-Set ist.

Ein false-negative bedeutet, dass ein Candidate-Set ein Eviction-Set für eine Zeugenadresse ist, jedoch nicht als solche erkannt wird. Dieser Fehler kann nur auftreten, wenn der Counter-Thread unterbrochen oder gestört wird, sodass dieser die Zählvariable gar nicht oder sehr langsam erhöht. Dieser Fehler ist in der Expand-Phase unerheblich, da ausschließlich zusätzliche Einträge hinzugefügt werden, welche aber in der Contract-Phase wieder in den Adresspool verschoben werden. In der Contract-Phase würden bei einem false-negative Einträge erhalten bleiben, die für das Eviction-Set nicht notwendig sind. Somit würde die Contract-Phase fehlschlagen, da das Eviction-Set nicht auf eine Größe von 16 reduziert werden kann. Ein false-negative in der Collect-Phase würde Adressen, die zum Eviction-Set gehören, nicht aussortieren, sondern im Pool belassen. Wenn ein false-negative mindestens 17 mal in der Collect-Phase auftritt, sind ausreichend Adressen für ein weiteres Eviction-Set desselben Cache-Sets im Pool vorhanden. Das heißt, in der nächsten Iteration könnte ein Eviction-Set für dasselbe Cache-Set gebildet werden. Da ein false-negative allerdings selten auftritt, ist dieses Szenario zu vernachlässigen.

Auch die Messung, ob ein Set für eine bestimmte Adresse ein Eviction-Set ist, wird mehrfach wiederholt. Die Autoren von Mastik schlagen eine 16-fache Wiederholung des Tests vor und bilden den Median über die Zugriffswerte. Hiermit bestimmen sie, ob das Set ein Eviction-Set ist. Eigene Tests haben ergeben, dass ein höherer Wert keine Vorteile bringt, sondern ausschließlich die Laufzeit erhöht. Ein niedrigerer Wert verursacht mehr Fehler und damit abgebrochene Phasen, sodass sich dieser ebenfalls negativ auf die Laufzeit auswirkt.

Ein Problem in der Praxis ist der Hardware-Prefetcher, welcher Daten in den Cache lädt, bevor sie benötigt werden. Angenommen es soll eine Prime-and-Probe-Operation ausgeführt werden, und auf alle Einträge des Eviction-Sets wird mittels einer For-Schleife zugegriffen. Wenn die Einträge des Eviction-Sets in einem Array liegen, könnte der Hardware-Prefetcher die Daten bereits vor dem Messvorgang laden und somit kurze Zugriffszeiten unabhängig von den Cache-Aktivitäten erzeugen. Um diesem Verhalten entgegenzuwirken, wird Pointer-Chasing eingesetzt (siehe auch Pseudocode 8).

Hierbei werden die Einträge des Eviction-Sets analog zu einer verketteten Liste gespei-

Pseudocode 8: Pseudo-Code für Pointer-Chasing-Methode

```
1 Function AccessTimeEvictionSet(pointerToAddress)
2     pointerToAddressFirst ← pointerToAddress
3     timestampBefore ← getTimeStamp()
4     while pointerToAddressFirst != pointerToAddress do
5         pointerToAddress ← readValue(pointerToAddress)
6     return getTimeStamp() - timestampBefore
```

chert, wodurch der Prozessor immer zuerst die Daten eines Eintrags lesen muss, bevor er die Adresse für den nächsten Eintrag kennt.

3.3 Verbesserte Eviction-Set-Suche

Die Forscher Saad Islam und Ahmad Moghimi des Worcester Polytechnic Institute sind auf ein noch nicht veröffentlichtes Verhalten bei Intel-Prozessoren gestoßen, das das Auffinden von Adressen ermöglicht, deren 20 letzten physischen Bits identisch sind, ohne auf Techniken wie Huge-Pages zugreifen zu müssen. Solche Adressen seien nachfolgend *colliding-addresses* genannt. Die Idee hierbei ist, den Store-Buffer in einer Schleife mit einer Vielzahl von Schreibbefehlen zu fluten und direkt danach eine Zeitmessung für das Lesen einer Adresse x auszuführen.
Wenn einer der zuletzt hinzugefügten Befehle auf eine Adresse schreibt, deren letzte 20 physischen Adressbits identisch mit denen der Read-Adresse x sind, dann lässt sich ein Ausschlag bei der Leseoperation für x feststellen.
Pseudocode 9 beschreibt das Suchverfahren, um colliding-addresses zu finden.

Pseudocode 9: Pseudo-C-Code für das Finden von colliding-addresses

```
1  Function FindCollidingAddresses()
2      evictionBuffer = calloc(1, PAGE_SIZE * PAGE_COUNT)
3      for p from WINDOW_SIZE to PAGE_COUNT-1 do
4          total = 0
5          foreach r in [0 .. Rounds] do
6              for i from WINDOW_SIZE-1 to 0 do
7                  evictionBuffer[(p-i) * PAGE_SIZE] = 0
8              timeStamp = rdtscp()
9              read evictionBuffer[0]
10             total += rdtscp() - timeStamp
11         measurementBuffer[p] = total / ROUNDS
```

Zuerst wird ein großer Speicherbereich alloziert , der ein Vielfaches der Page-Größe von 4

KiB hat. Wie im Abschnitt 3.2.1 wird hier mittels 4-KiB-Speicherblöcken gesucht, da so die letzten 12 Bits der Adresse mit Sicherheit identisch sind. Dies beschleunigt die Suche um den Faktor 4096, da dann im Mittel etwa eine von 2^8 statt eine von 2^{20} Adressen eine colliding address ist. Im Pseudocode werden colliding-addresses zu *evictionBuffer*[0] gesucht. Hierzu werden iterativ (Zeile 3) Speicherblöcke aus dem Pool auf diese Eigenschaft hin geprüft. Wie oben beschrieben muss der Store-Buffer mit Schreibbefehlen geflutet werden, wobei *WINDOW_SIZE* deren Anzahl angibt. Zeile 6 und 7 sorgen für die Schreibbefehle auf die Speicherblöcke $p + WINDOW\_SIZE - 1$ bis p des Pools, sodass der Schreibzugriff auf den aktuell zu testenden Block p als letztes erfolgt. Die Zeilen 8 bis 10 messen die Zeit für einen Lesezugriff auf die Adresse *evictionBuffer*[0] beziehungsweise Speicherblock 0.

Der Testvorgang für einen Speicherblock p wird mehrfach wiederholt (Zeile 5) und die Zugriffszeit auf *evictionBuffer*[0] gemittelt (Zeile 11), um Ausreißer bei der Zugriffszeit durch Timer- oder Systemrauschen auszuschließen.

Grafik 11 zeigt die typische Verteilung der Zugriffszeiten auf *evictionBuffer*[0], stellt also *measurementBuffer* visuell dar.

Eigene Tests haben ergeben, dass mit *WINDOW_SIZE=60* colliding-addresses zuverlässig durch Peaks bei der Zeitmessung der Leseoperation für x hervorstechen. Ein höherer Wert für *WINDOW_SIZE* bringt keine Vorteile bei der Identifizierung der colliding addressses, erhöht allerdings die Laufzeit der Suche. Andersherum verringern kleinere Werte für *WINDOW_SIZE* die Suchlaufzeit, sorgen aber auch für kleinere Peaks der colliding-addresses bei der Zeitmessung, sodass eine Identifikation schwieriger wird.

Angenommen es wird ein Pool mit x colliding-addresses gefunden. Da die letzten 20 Bits aller Adressen gleich sind, gibt es auf dem verwendeten Testsystem nur 4 mögliche Cache-Sets, denen die Adressen zugeordnet werden können. Die Anzahl der möglichen Cache-Sets ist gleichbedeutend mit der Anzahl der Slices, da die unteren 20 Bits das Cache-Set innerhalb eines Slices komplett bestimmen.

Wenn nun x groß genug gewählt ist, können aus dem colliding-address-Pool 4 Eviction-Sets gebildet werden, wobei die untersten 12 Bits aller Adressen im Pool auf 0 gesetzt sind. Daher können aus jedem der 4 Eviction-Sets wiederum 63 neue Eviction-Sets gebildet werden, indem die Bits 6 bis 11 der bestehenden Eviction-Sets nach und nach inkrementiert werden.

Somit muss insgesamt 32 mal (32*4*64=8192) ein colliding-address-Pool aufgebaut werden, um alle Eviction-Sets zu bestimmen.

Die Ergebnisse in Tabelle 3.4 zeigen eine deutliche Verbesserung der Laufzeit gegenüber dem Standard-Algorithmus.

Der optimierte Standard-Algorithmus (siehe Tabelle 3.3) ist mehr als 3 mal so langsam wie der neue Algorithmus. Aufgrund von Ausreißern sind in der Tabelle 3.3 Median-Werte

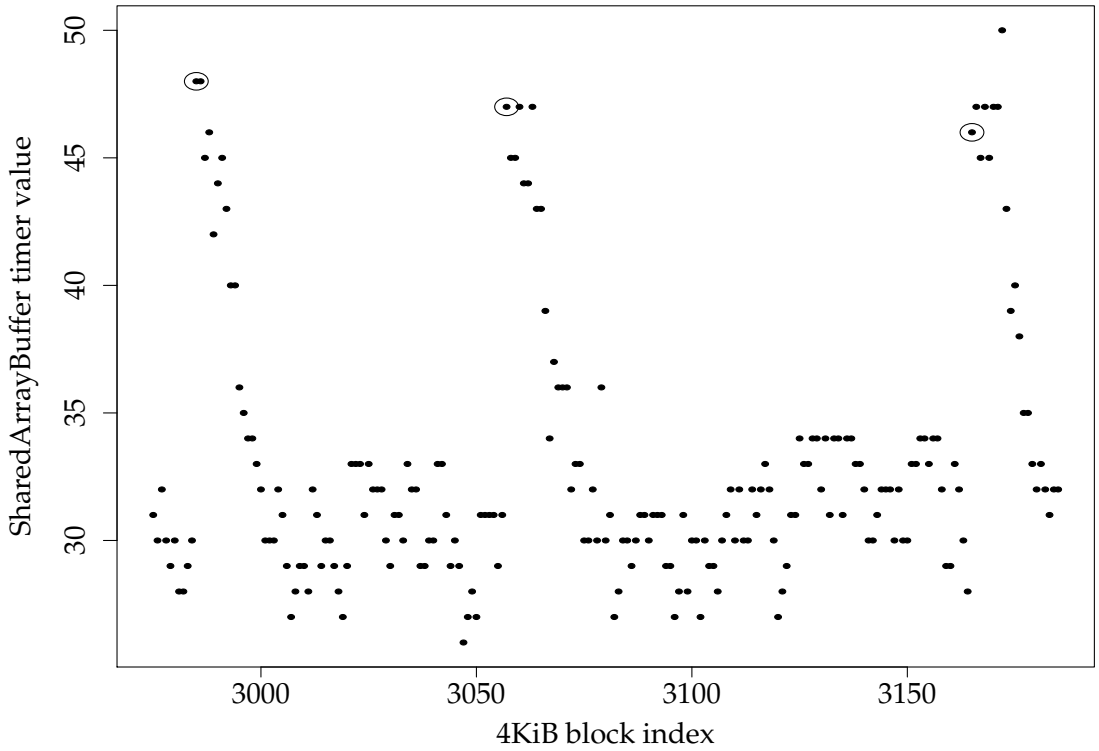

Abbildung 3.3: Erkennung von colliding-addresses unter Javascript. Punkte geben die Latenz für eine Leseoperation auf einer fixen Adresse an, nachdem Schreiboperationen auf jeden der 64 Blöcke im aktuellen Fenster durchgeführt wurden. Die drei umkreisten Punkte stellen vom Algorithmus identifizierte colliding-addresses dar. Es wurde verifiziert, dass diese drei Adressen Teil eines späteren Eviction-Sets geworden sind und somit die Identifikation seitens des Algorithmus korrekt war.

angegeben, wohingegen der neue Algorithmus nicht mit Ausreißern zu kämpfen hat. So liegt die durchschnittliche Dauer des neuen Algorithmus in 60 Läufen bei 12 Sekunden, wobei der längste Lauf 37 Sekunden dauerte. Der optimierte Standard-Algorithmus aus Abschnitt 3.2.3 benötigte im Durchschnitt 74 Sekunden und der längste von 60 ausgeführten Läufen dauerte 275 Sekunden.

Dieser Performancevorteil ergibt sich dadurch, dass beim optimierten Standardalgorithmus 128 Eviction-Sets in einem Pool von 4096 Adressen gefunden werden müssen, wohingegen bei der neuen Methode das 32-malige Finden von 4 Eviction-Sets bei einer Poolgröße von etwa 100 ausreicht.

Wie sich in Tabelle 3.4 zeigt, führt eine $WINDOW\_SIZE < 60$ zu häufigen Fehlern bei der colliding-address-Suche, sodass eine anschließende Eviction-Set-Suche selten erfolgreich

Tabelle 3.4: Performancewerte des neuen Eviction-Set-Suchalgorithmus, wobei pro Parametersatz (die ersten drei Spalten) 60 Läufe durchgeführt wurden. Angegeben sind die Mittelwerte, wobei sich alle Werte auf einen Lauf beziehen. Die Spalte „Initiale Poolgröße“ gibt die Poolgröße der colliding-addresses an, ab der eine Eviction-Set-Suche gestartet wird. Die Spalten „*WINDOW_SIZE*“ und „*Rounds*“ sind die Parameter aus Pseudocode 9. Die Spalten „Anteil colliding-addr Suche“ und „Anteil Eviction-Set Suche“ beschreiben den Anteil, welchen die beiden Phasen an der Gesamtlaufzeit haben. Die letzte Spalte beschreibt wie viele der 60 Läufe erfolgreich alle 8192 Eviction-Sets bilden konnten.

Initiale Poolgröße	*WINDOW SIZE*	*Rounds*	Mittlere Dauer	Anteil colliding-addr Suche	Anteil Eviction-Set Suche	Anteil erfolgreicher Läufe
115	60	10	10s	54%	46%	67%
115	60	15	37s	24%	76%	98%
115	55	20	12s	42%	58%	32%
105	60	20	61s	22%	78%	88%
110	60	20	11s	69%	31%	80%
115	60	20	12s	75%	25%	100%

verläuft. Dies ist unabhängig von der Wahl des Parameters *Rounds*.

Ein ähnliches Verhalten zeigt der Parameter *Rounds*, für den sich in dem Test der Optimalwert 20 herausgestellt hat. Geringe Werte führen zu mehr Fehlern in der colliding-address-Suche und verlängern somit die Gesamtlaufzeit oder verringern die Anzahl der erfolgreichen Läufe.

Die Tabelle zeigt außerdem, dass die Größe eines colliding-address-Pools von 115 am besten performt. Häufig sind weniger Adressen ausreichend, um daraus 4 Eviction-Sets zu bilden, jedoch zieht eine gescheiterte Eviction-Set-Suche eine erneute Suche mit vergrößertem Pool nach sich, wobei die Kosten dafür im Verhältnis zur Suche zusätzlicher colliding-addresses hoch sind. Deshalb ist es günstiger, bereits zu Beginn mehr colliding-addresses zu suchen, um den Fall einer gescheiterten Eviction-Set-Suche zu vermeiden.

Somit stellt sich eine initiale Poolgröße von 115, eine *WINDOW_SIZE* von 60 und ein Wert von 20 für den Parameter *Rounds* als guter Kompromiss heraus, da so das Finden aller 8192 Eviction-Sets mit hoher Wahrscheinlichkeit gewährleistet ist und die Laufzeit gegenüber verschärften Parametern im Mittel höchstens um 2 Sekunden ansteigt.

Um eine weitere Beschleunigung zu erzielen, könnte die Suche nach den colliding addresses und die eigentliche Suche nach den Eviction-Sets im colliding-address-Pool parallelisiert werden. Wie in der letzten Zeile der Tabelle 3.4 zu sehen ist, ist die Suche nach den colliding-addresses langsamer als die Suche nach den Eviction-Sets.

Bei der Parallelisierung könnten die Parameter für den langsameren der Teile sicherer

gewählt werden, sodass beide Teile des Algorithmus in etwa die gleiche Geschwindigkeit aufweisen. Somit würde nicht nur die Performance ansteigen, der Algorithmus wird auch weniger fehleranfällig. Die gesamte Eviction-Set-Suche ist in Webassembly implementiert. Da Webassembly zurzeit jedoch keine Multithreading-Unterstützung [Web18a] bietet, lässt sich dieser Ansatz nicht testen. An der Multithreadunterstützung in Webassembly wird aber gearbeitet, das heißt der verwendete Code könnte in Zukunft leicht um diesen Parallelisierungsansatz erweitert werden.

3.4 Verdeckter Kanal

Die maximale Sendegeschwindigkeit eines Kanals ist durch die Rate, mit welcher der Sender ein beliebiges Cache-Set primen kann, begrenzt. Damit der Empfänger ein zufälliges Rauschen von einem Priming unterscheiden kann, sollte der Sender mehrere Einträge aus dem zu primenden Cache-Set verdrängen, wobei im Optimalfall die Anzahl der zugegriffenen Speicheradressen der Assoziativität des Caches entspricht. Hiermit wird die Wahrscheinlichkeit erhöht, dass sich die vom Empfänger im Probe-Schritt gemessene Zugriffszeit signifikant von Fällen unterscheidet, in denen zufällig einzelne Einträge aus dem überwachten Cache-Set verdrängt werden. Im Folgenden sollen verschiedene Methoden des Primens eines Cache-Sets verglichen werden, indem entweder die Anzahl der zugegriffen Speicheradressen oder die Zugriffsmethode verändert werden. Wenn etwa die Zahl der zugegriffenen Speicheraddressen verringert wird, sind auf der einen Seite mehr Timeslots in einem Zeitabschnitt möglich, und die Chance sinkt, dass benachbarte Timeslots zusätzlich beeinflusst werden. Auf der anderen Seite sind die messbaren Ausschläge der Zugriffszeiten verringert, wodurch ein bewusst geprimtes Cache-Set schwieriger von einem Messrauschen oder von zufälligen Zugriffen unterschieden werden kann. Sende- und Empfangsseite können durchaus abweichende Parameter verwenden, wenn wie etwa im vorliegenden Fall der Empfänger langsamer als der Sender arbeitet. Um die Timeslots anzugleichen, könnte der Empfänger die Dauer einer Priming-Operation durch die Senkung der Anzahl der zugegriffenen Speicheraddressen verringern und der Empfänger andersherum die Dauer für eine Priming-Operation erhöhen.

Auf dem Testrechner benötigt ein in C geschriebenes Sendeprogramm für eine Million Prime-Vorgänge mit 16 Adressen und der Single-Pointer-Chasing-Methode (siehe Algorithmus 8) etwa 323 Millionen Taktzyklen. Im Optimalfall kann im Timeslot x ein durch den Sender erfolgter Prime-Vorgang als 1 und ein nicht erfolgter Prime-Vorgang als 0 interpretiert werden. Bei einem typischen All-Core-Turbo-Takt von 3,4 Ghz des i7-4770 ergibt sich so eine maximale Senderate von 10,5 Mbit/s. Diese Rate wird jedoch vom Empfänger beschränkt, welcher zusätzlich noch eine Zeitmessung durchführen muss.

Der Worst-Case ist hier eine in Webassembly geschriebene Empfangsroutine, da dort eine Zeitmessung kostenintensiver ist. In Chromium 66 können eine Million Messungen eines Cache-Sets in etwa 200 ms durchgeführt werden. Im Mittel dauert eine Messung also 0,2 µs, womit eine Empfangsrate von maximal 5 MBit/s realisiert werden kann.
Im Folgenden soll die maximal mögliche Senderate unter optimalen Bedingungen ermittelt werden. Hierfür wird im Voraus ein Cache-Set ausgewählt, auf dem im Idle-Zustand des Systems ein geringes Rauschen herrscht. Um die Synchronisation des Senders und Empfängers aufrechtzuerhalten, wird nach 10 gesendeten Bits ein Synchronisationsblock eingefügt, welcher durch sb-Prime-Vorgänge auf der Senderseite erzeugt wird. Eine 1 wird durch s-Prime-Vorgänge repräsentiert und eine 0 durch das Unterlassen der Prime-Vorgänge. Um die einzelnen Bits auseinanderzuhalten, wird zwischen jedem gesendeten Bit eine Pause von p-Taktzyklen eingelegt. Grafik 3.4 zeigt die Übertragung eines Bitstrings von einem C-Programm zum Javascript/Webassembly-Empfänger. Zwischen dem Senden eines Bits wurde eine Pause von $p = 1500$ Taktzyklen eingelegt. Die Größe des Synchronisationsblocks ist auf $sb = 10$ gesetzt. Mit den in der Grafik verwendeten Parametern ergibt sich eine ungefähre Netto-Datenrate von 4,6 KB/s.

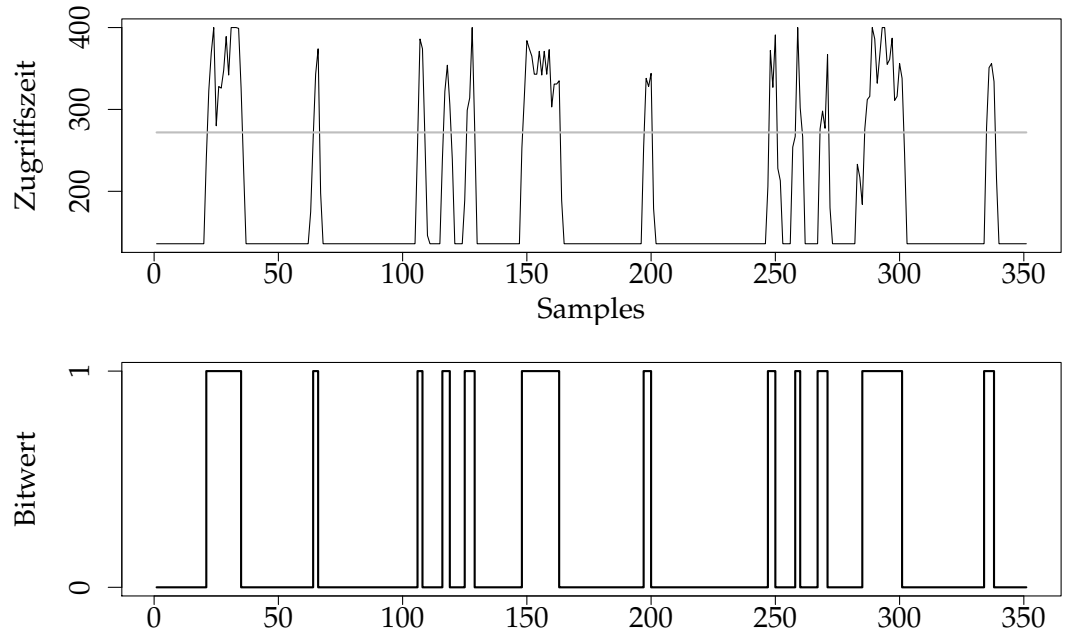

Abbildung 3.4: Verdeckter Kanal mit einem Javascript/Webassembly-Empfänger und einem C-Programm als Sender. Dargestellt ist das zweimalige Empfangen des Bitstrings „0010001110“ im Bereich von 38 bis 147 und von 165 bis 282. Der obere Plot spiegelt die Zeitmessung mittels Prime-and-Probe wieder, wobei die graue durchgezogene Linie $2 \cdot Median(y)$ ist. Zugriffswerte oberhalb der Linie werden als 1 und unterhalb als 0 interpretiert. Der untere Plot zeigt dies daraus erkannte Muster. Größere Blöcke von Einsen wie etwa im Bereich 21 bis 37 dienen der Synchronisation zwischen Sender und Empfänger. Die Nullen im Bitstring werden über den Abstand zwischen Einsen beziehungsweise dem Synchronisationsblock ermittelt.

4 Identifikation von Angriffszielen

4.1 RSA Key Generierung

Frühere Arbeiten zu Cache-Angriffen hatten es vor allem auf die Verschlüsselungs- und Entschlüsselungsroutinen der RSA-Implementierungen abgesehen [YF14, YGH17a]. Deshalb wird bei deren Implementierung die Möglichkeit von Seitenkanalangriffen in Betracht gezogen und verstärkt auf Algorithmen gesetzt, deren Ausführungszeit (constant-time) und Kontrollfluss (constant-execution-flow) unabhängig von den Eingaben ist. Einen anderen bislang weniger untersuchten Einstiegspunkt für die Schlüsselextraktion bieten die Routinen zur Schlüsselgenerierung, welche in den letzten Monaten etwa in [CAPGATBB18] näher untersucht wurden.

4.1.1 Primzahlgenerierung in Mozilla NSS

Im Folgenden soll die Primzahlgenerierung von Mozilla NSS genauer beleuchtet werden. Um einen neuen Schlüssel zu erzeugen, werden zuerst zwei Primzahlen generiert. Der Code hierfür befindet sich im Unterordner lib/freebl. Der Einstiegspunkt für die Primezahlgenerierung ist die Funktion generate_prime in der Datei rsa.c 10. Dort wird zuerst eine ungerade Zufallszahl *random_num* entsprechend der Bitlänge der gewünschten Primzahl erzeugt, deren zwei höherwertigsten Bits auf 1 gesetzt sind. Anschließend wird versucht, auf Basis von *random_num* eine Primzahl zu erzeugen und diese im Erfolgsfall zurückgegeben. Falls dies fehlschlägt, wird der gesamte Vorgang bis zu zehnmal wiederholt, bevor die Primzahlgenerierung endgültig abgebrochen wird.

Pseudocode 10: Pseudo-Code für generate_prime in Mozilla NSS

```
1 Function generate_prime(primeLen)
2     for 1 to 10 do
3         random_num <- RNG_GenerateGlobalRandomBytes(primeLen)
4         random_num[0] |= 0xC0 /* set two high-order bits */
5         random_num[primeLen - 1] |= 0x01 /* set low-order bit */
6         successful <- mpp_make_prime(random_num, &prime)
7         if successful then
8             return prime
9     return no prime found
```

Die Zahl *random_num* wird an die Funktion mpp_make_prime in der Datei mpprime.c übergeben 12. Global definiert ist das Array *primes_tab*, das alle Primzahlen von 3 bis 2^{16} beinhaltet. Dieses Array wird nicht dynamisch beim Start erzeugt. Stattdessen stehen alle benötigten Primzahlen fix im Quellcode. Das Array *primes_tab* wird im nachfolgenden Siebvorgang benötigt, um mit wenig Rechenleistung eine Reihe von Primzahlkandidaten auszuschließen. Dieser Siebvorgang wird in der Funktion mpp_sieve, die ebenfalls in mpprime.c zu finden ist definiert (siehe auch Pseudocode 11). Zahlen der Form $\textit{random\_num} + 2 \cdot i$, mit $i \in [1 \,..\, \textit{SIEVE\_SIZE}]$ sind mögliche Primzahlkandidaten. Die später verwendeten Fermat- und Miller-Rabin-Tests sind relativ aufwendig, weswegen ein Siebvorgang analog zum Sieb des Eratosthenes vorangestellt wird. Im Siebvorgang wird über alle Primzahlen im *primes_tab*-Array iteriert und jeweils der Rest *rem* von $\textit{random\_num} \mod \textit{small\_prime}$ errechnet.

Durch den Aufruf von *mp_mod_d* im Pseudocode in Zeile 4 wird angedeutet, dass diese Modulo-Operation nicht in einen einzigen Maschinenbefehl übersetzt werden kann, da die Zahl random_num nicht in ein Register passt. Anders hingegen ist dies bei der Modulo-Operation in Zeile 10, da dort sowohl i als auch 2 in ein Register passen.

Nun müssen zwei Fälle unterschieden werden: Im ersten Fall sei *random_num* durch *small_prime* teilbar, das heißt $\textit{random\_num} \mod \textit{small\_prime} = 0$. Dann ist *random_num* zusammengesetzt, sowie auch alle $\textit{random\_num} + k \cdot \textit{small\_prime}$ mit $k \in \mathbb{N}$. Das *sieve*-Array dient später dazu, die Primzahlkandidaten der Form $\textit{random\_num} + 2 \cdot i$ mit $i \in [1 \,..\, \textit{SIEVE\_SIZE}]$ zu erzeugen, wobei Kandidat i nicht weiter betrachtet wird, wenn *sieve*[i] auf 1 gesetzt ist.

Im ersten Fall muss also $\textit{sieve}[0] = 1$ gesetzt werden, da $\textit{random\_num} \mod \textit{small\_prime} = 0$. Dies wird in der ersten Iteration der For-Schleife in Zeile 9 umgesetzt. Außerdem muss $\textit{sieve}[j] = 1$ gesetzt werden, falls $j = k \cdot \textit{small\_prime}/2$ gilt, da wie oben festgestellt $\textit{random\_num} + k \cdot \textit{small\_prime}$ beziehungsweise $\textit{random\_num} + 2 \cdot j$ zusammengesetzt ist. Das Setzen von $\textit{sieve}[j] = 1$ wird in den restlichen Iterationen der For-Schleife umgesetzt, wobei Einträge im *sieve*-Array nur gesetzt werden, wenn $k \cdot \textit{small\_prime}$ gerade ist. Ungerade $k \cdot \textit{small\_prime}$ sind uninteressant, da *random_num* ebenfalls ungerade ist und somit $\textit{random\_num} + k \cdot \textit{small\_prime}$ immer mindestens durch 2 teilbar ist.

Im zweiten Fall sei *random_num* nicht durch *small_prime* teilbar, das heißt $\textit{rem} \neq 0$. Dann ist die Zahl $\textit{random\_num} + (\textit{small\_prime} - \textit{rem})$ durch *small_prime* teilbar, da

$$
\begin{aligned}
&(\textit{random\_num} && +(\textit{small\_prime} - \textit{rem})) \mod \textit{small\_prime}\\
=&\textit{random\_num} \mod \textit{small\_prime} && +(\textit{small\_prime} - \textit{rem}) \mod \textit{small\_prime}\\
=&(\textit{rem} + (\textit{small\_prime} - \textit{rem})) = 0
\end{aligned}
$$

Deshalb wird in der ersten Iteration der For-Schleife in Zeile 9 $sieve[(small\_prime - rem)/2] = 1$ gesetzt, unter der Bedingung das $(small\_prime - rem)/2$ gerade ist. Zudem sind alle $random\_num + (small\_prime - rem) + k \cdot small\_prime$ durch *small_prime* teilbar, womit analog zu Fall 1 alle $sieve[j] = 1$ gesetzt werden, falls $j = ((small\_prime - rem) + k \cdot small\_prime)/2$ ist.

Pseudocode 11: Pseudo-Code für mpp_sieve in Mozilla NSS

```
1  Function mpp_sieve(random_num, primes_tab, size)
2      for ix = 0; ix < nPrimes; ix++ do
3          small_prime <- primes_tab[ix]
4          rem <- mp_mod_d(random_num, small_prime)
5          if rem == 0 then
6              offset = 0
7          else
8              offset = small_prime - rem
9          for i = offset; i < nSieve * 2; i += prime do
10             if i mod 2 == 0 then
11                 sieve[i / 2] = 1
```

Nachdem der Siebvorgang abgeschlossen ist, wird das *sieve*-Array in der Foreach-Schleife in Zeile 6 der Funktion *mpp_make_prime* verwendet. Wie eben beschrieben, können Kandidaten der Form $random\_num + 2 \cdot i$ ausgeschlossen werden, wenn $sieve[i] = 1$ gesetzt ist. Die restlichen Kandidaten werden mittels des Fermat- und Miller-Rabin-Tests auf Primzahleigenschaft hin überprüft. Werden auch diese Tests bestanden, wird die Zahl als Primzahl eingestuft und zurückgegeben.

4.1.2 Mögliche Leakage in Mozilla NSS

Das Ziel ist es, die Zahl *random_num* oder Teile davon zu rekonstruieren. Der Siebvorgang sticht hier besonders hervor, da dort unabhängig von *random_num* genau 6541 - das ist Anzahl der Primzahlen von 3 bis 2^{16} - Modulo-Operationen mit vorher bekannten Primzahlen auf *random_num* ausgeführt werden.
Typische RSA-Schlüssel besitzen heute eine Länge von 2048 Bit, sodass *random_num* etwa eine Länge von 1024 Bit besitzt. Die Berechnung von $random\_num \mod small\_prime$ dauert dann im Schnitt 4800 Taktzyklen. Somit ist bekannt, dass auf das *primes_tab*-Array nur etwa jede 4800 Taktzyklen einmal zugegriffen wird.
Das *primes_tab*-Array ist vom Datentyp mp_digit, der bei einer Linux x64-Kompilierung gleichbedeutend mit einem 8 Byte Integer ist. Somit hat das *primes_tab*-Array eine Gesamtgröße von $8 \cdot 6541 = 52328$ Bytes und liegt in $\lceil 52328/64 \rceil = 818$ verschiedenen Cache-Lines

Pseudocode 12: Pseudo-Code für mpp_make_prime in Mozilla NSS

```
1  primes_tab <- [3, 5, 7, ..., 65521] // 6541 primes from 3 to 2^16
2  SIEVE_SIZE <- 32 * 1024
3  Function mpp_make_prime(random_num)
4      num_miller_rabin_tests <- get_num_tests(bitlen(random_num)) // augmented by
         FIPS-186 requirements, Table C.2 and C.3
5      sieve <- mpp_sieve(random_num, prime_tab, SIEVE_SIZE)
6      foreach i in SIEVE_SIZE do
7          if sieve[i] then
8              /*number is composite*/
9              continue
10         probablePrime <- random_num + 2*i
11         if !mpp_fermat(probablePrime, 2) then
12             //Fermat test with 2 continue
13         if !mpp_pprime(probablePrime, num_tests) then
14             //Miller Rabin test continue
15         return probablePrime
```

beziehungsweise $\lceil 52328/4096 \rceil = 13$ Pages.

Angenommen, die zu *primes_tab*[0-7] gehörige Cache-Line wird mittels Prime-and-Probe überwacht, dann können nach Start des Siebvorgangs 8 Zugriffe mit jeweils etwa 4800 Taktzyklen (mittlere Dauer einer Iteration von *mpp_sieve* bei RSA 2048) Abstand gemessen werden. Nach dem 8. Zugriff wird auf die Cache-Line *primes_tab*[8-15] gewechselt, um dort die nächsten 8 Zugriffe zu messen und so weiter.

Um die notwendigen Cache-Sets zu finden, muss nur jeweils ein Cache-Set pro 4-KiB Page identifiziert werden, da wie bereits oben erwähnt die Adressen innerhalb einer Page identisch sind. Das Zugriffsmuster auf jeder Cache-Line des *primes_tab*-Array entspricht immer 8 Zugriffen mit jeweils etwa 4800 Taktzyklen Pause. Außerdem liegt das *primes_tab*-Array immer an derselben Adresse im Speicher. Die Angreiferin kann selber Schlüsselgenerierungen anstoßen, sodass es ihr möglich ist, die passenden Eviction-Sets für das *primes_tab*-Array zu finden.

Der Schreibzugriff auf das *sieve*-Array in der For-Schleife (siehe 11 Zeile 9-11) ist mit einer Laufzeit von etwa 100 Taktzyklen zu schnell, um einzelne Zugriffe unterscheiden zu können. Das *sieve*-Array ist ein Byte Array der Größe 32768 Bytes und liegt somit in 512 Cache-Lines. Bei den ersten Iterationen der For-Schleife mit kleinen Primzahlen wie der 3 wird noch auf alle 512 Cache-Lines zugegriffen. Sobald die Primzahlen aber den Wert 64 überschreiten, werden Cache-Sets ausgelassen.

Zur Veranschaulichung hier ein Beispiel:

Angenommen *random_num* $\mod 67 = 0$ und die For-Schleife der Zeile 9-11 ist in der 42.

Iteration, das heißt $i = 2814$ ist gerade. Dann wird gemäß Zeile 11 *sieve*[1407] $= 1$ gesetzt, hiermit auf die Cache-Line für *sieve*[1344-1407] zugegriffen und abschließend $i = 2881$ gesetzt.
In der 43. Iteration wird auf keine Cache-Line von *sieve* zugegriffen, da $i = 2881$ ungerade ist.
Die 44. Iteration arbeitet mit $i = 2948$ und setzt somit den Wert *sieve*[1474] $= 1$. Die zugehörige Cache-Line ist nun *sieve*[1472-1535], womit keinerlei Zugriff auf die Cache-Line *sieve*[1408-1471] erfolgt.
Im Gegensatz zum *primes_tab*-Array wird das *sieve*-Array dynamisch in der Funktion erzeugt, also kann die Angreiferin dies nicht durch vorherige Schlüsselgenerierungen lokalisieren. Ihr ist aber bekannt, dass, solange *small_prime* < 64 (17 Primzahlen) ist, auf alle Cache-Lines des *sieve*-Arrays zugegriffen wird.
Durch die Änderung der Bits 6 bis 11 der Adressen eines bestehenden Eviction-Sets in derselben Weise entsteht ein neues gültiges Eviction-Set, welches im Folgenden als benachbartes Eviction-Set bezeichnet wird (siehe auch Abbildung 3.2). Angenommen die Bits 6 bis 11 der Adressen eines Eviction-Sets sind alle 0. Nun wird Bit 6 aller Adressen auf 1 gesetzt, womit die Adressen ein neues benachbartes Eviction-Set bilden.
Das *sieve*-Array ist in 512 Cache-Lines aufgeteilt, auf die wie oben beschrieben nacheinander zugegriffen wird. Durch die Berechnung von *random_num* $\bmod$ *small_prime* entsteht zwischen den Zugriffen auf das *sieve*-Array eine Pause von etwa 4800 Taktzyklen. Dagegen ist die eigentliche Aktivität auf dem *sieve*-Array (Zeile 9-11) mit unter 100 Taktzyklen pro Iteration vergleichsweise kurz. Daher muss die Angreiferin nach dem Zugriff auf das *primes_tab*-Array eine Reihe von Eviction-Sets primen, warten, bis auf das *primes_tab*-Array erneut zugegriffen wird, und dann proben.
Sofern ein Eviction-Set in dieser Phase Aktivität gezeigt hat, werden in der nächsten Iteration seine benachbarten Sets überwacht. Denn wenn das Eviction-Set eine Cache-Line des *sieve*-Array verdrängt hat, so müssen auch die benachbarten Eviction-Sets in der nächsten Runde Aktivität auf ihrer zugehörigen Cache-Line zeigen. So können die relevanten Eviction-Sets schnell eingegrenzt werden, da initial maximal 128 Eviction-Sets in Frage kommen. Die restlichen 8064 Eviction-Sets sind zu diesen 128 benachbart, müssen also nicht initial überwacht werden.
Cache-Lines im *sieve*-Array, auf die nicht zugegriffen wird, können Informationen preisgeben, da das i der For-Schleife in Abhängigkeit der Berechnung *random_num* $\bmod$ *small_prime* definiert wird.
Dies soll an folgendem Beispiel verdeutlicht werden:
Angenommen es wurde ein Zugriff auf die Cache-Line *sieve*[64-127], aber nicht auf die Cache-Line *sieve*[0-63] gemessen und *small_prime* ist für die aktuelle Iteration bekannt.

Festzuhalten ist, dass $i \in \{128, ..., 254\}$ und gerade sein muss, sodass ein Schreibzugriff auf *sieve*[64-127] in Zeile 11 ausgeführt wird. Weiter kann nicht *random_num* $\bmod$ *small_prime* $= 0$ gelten, da ansonsten ein Zugriff auf die Cache-Line *sieve*[0-63] messbar wäre. Es wird nun zwischen verschiedenen Werten für die Variable offset (siehe Zeile 6 und 8 in mpp_sieve) unterschieden.

1. Fall: Sei *offset* $\in [128 \,..\, 254]$ und gerade, also *offset* $\bmod 2 = 0$.

Dann ist *rem* $=$ *small_prime* $-$ *offset* und durch die Kenntnis von *small_prime* und *offset* $\in [128 \,..\, 254]$ gerade kann *rem* auf 64 mögliche Werte eingegrenzt werden.

2. Fall: Sei *offset* ungerade, also *offset* $\bmod 2 = 1$ und

$$\textit{offset\_ex} = \textit{offset} + \textit{small\_prime} \in [128 \,..\, 254]$$

sowie *offset_ex* $\bmod 2 = 0$.

Da $i =$ *offset* und *offset* ungerade erfolgt in der ersten Iteration der For-Schleife kein Zugriff auf das *sieve*-Array. Wenn der Zugriff also in der zweiten Iteration erfolgt, ist $i =$ *offset* $+$ *small_prime* $=$ *offset_ex*.

Da *offset* $=$ *offset_ex* $-$ *small_prime* ist

$$\textit{rem} = \textit{small\_prime} - (\textit{offset\_ex} - \textit{small\_prime}) = 2 \cdot \textit{small\_prime} - \textit{offset\_ex}$$

und durch die Kenntnis von *small_prime* und *offset_ex* $\in [128 \,..\, 254]$ gerade kann *rem* auf 64 mögliche Werte eingegrenzt werden.

Es ist unbekannt, ob *rem* gerade oder ungerade ist, weshalb *rem* nicht weiter als auf 128 mögliche Werte eingegrenzt werden kann.

Mit dieser Leakage ergeben sich Kongruenzen bezüglich des Wertes *random_num* der folgenden Form:

$$\textit{random\_num} \mod \textit{small\_prime} \equiv a \text{ mit } a \in A \tag{4.1}$$

a kann durch diese Leakage nicht exakt bestimmt werden, deshalb ist A_i die Menge mit den möglichen Werten, welche a annehmen kann. Im oberen Beispiel gilt etwa $|A_i| = 128$ und $\{128, 130, 132, ..., 254\} \subsetneq A_i$, wobei die anderen 64 Einträge von A_i durch *small_prime* definiert werden.

Mit dieser Beobachtung kann für *random_num* folgendes Gleichungssystem definiert wer-

den:

$$\begin{aligned} \textit{random\_num} \mod \textit{small\_prime}_i &\equiv a_i && \text{mit } a_i \in A_i \\ \textit{random\_num} \mod \prod_i \textit{small\_prime}_i &< 2^{bits} - 1 \\ \textit{random\_num} \mod \prod_i \textit{small\_prime}_i &> 2^{bits-1} \end{aligned}$$

A_i ist hier analog zu oben die Menge der möglichen Werte für $\textit{random\_num} \mod \textit{small\_prime}_i$, die durch die Leakage bekannt werden. Um den Lösungsraum dieses Gleichungssystems einzuschränken, wird die Tatsache genutzt, dass $2^{bits-1} < \textit{random\_num} < 2^{bits} - 1$.

Angenommen jedes a_i wäre eindeutig, das heißt $|A_i| = 1$, dann existiert gemäß dem chinesischen Restsatz 2.5 genau eine Lösung im Bereich von 0 bis $\prod_i \textit{small\_prime\_i}$. Weiter sei $bitlen(\textit{small\_prime}_i) > 10 \; \forall i$, dann würden $bitlen(\textit{random\_num})/10$ Gleichungen ausreichen, um *random_num* eindeutig zu bestimmen. Jede Gleichung bringt also in etwa 10 Bit an Information über *random_num*.

Im Fall der oben beschrieben Leakage bringt eine Gleichung aber 7 Bit oder 128 mögliche Werte als Unsicherheit mit. Angenommen es existieren 1000 Gleichungen mit $bitlen(\textit{small\_prime}_i) > 10 \; \forall i$, sodass es 2^{7000} mögliche Lösungen gibt und diese im Bereich 0 und 2^{10000} liegen. Da $2^{bits-1} < \textit{random\_num} < 2^{bits} - 1$ gilt und die möglichen Lösungen über den gesamten Lösungsraum verstreut sind, ist die Wahrscheinlichkeit hoch, eine eindeutige Lösung für *random_num* zu finden.

Sofern erlaubt ist, dass die Moduli $\textit{small\_prime}_i$ beliebige Werte annehmen könnten, ist das Problem beweisbar NP-schwer [D.W18]. Eine wesentliche Voraussetzung für die Beweisidee sind hier die beliebigen Werte für die Moduli, sodass eine Adaption der Beweisidee an das ursprüngliche Problem nicht möglich scheint.

Auch wenn das Problem NP-schwer sein sollte, schließt dies nicht automatisch eine schnelle Lösung für bestimme Instanzen des Problems aus. So lässt sich das Problem etwa als Integer Programming Problem (ILP) formulieren: Ein Constraint ist die Einschränkung von *random_num* in den Bereich von 2^{bits-1} bis $2^{bits} - 1$. Und jede Modulo-Gleichung $x \equiv a \mod m$ wird zu $x + f \cdot m = a$ übersetzt.

Typische ILP-Solver wie etwa LP-Solve [lps18] arbeiten nicht mit großen Ganzzahlen über 2^{64}, sodass diese als Lösung wegfallen. In Mathematica [Mat18] lässt sich das Problem beschreiben, jedoch wurde der Versuch, kleine Probleminstanzen mit $\textit{random\_num} \approx 2^{100}$ zu lösen, nach mehreren Stunden abgebrochen. Durch das Laufzeitverhalten lässt sich vermuten, dass Mathematica den langsamen und trivialen Ansatz verfolgt, alle Lösungsmöglichkeiten durchzuprobieren.

Aufgrund des Fehlens eines Beweises für die NP-Schwere oder eines effizienten Lösungs-

algorithmus' konnte die Problemkomplexität nicht abschließend geklärt werden.
Die Fallunterscheidung in der Funktion mpp_sieve in den Zeilen 5 bis 8 würde die Information liefern, ob *random_num* $\mod$ *small_prime* $= 0$ ist. Unabhängig von *rem* wird aber stets der gesamte Assemblercode für die Zeilen 5 bis 8 geladen (siehe 4.1), da die Vergleichsoperation an Adresse 806A4 und der Code für die folgende For-Schleife an der Adresse 806CA immer benötigt wird. Der Abstand zwischen den Adressen 806A4 und 806CA ist kleiner als die Größe einer 64 Byte Cache-Line, das heißt die vorherige Aussage ist losgelöst von den Adressen des Codes in der Binary. Somit ist ein Prime-and-Probe Angriff auf den entsprechenden Codeteil nicht möglich, da hierbei der Assemblercode für die Zeile 5 bis 8 in zwei Cache-Lines fällt und bei beiden immer eine erhöhte Zugriffszeit gemessen wird.

```
1   806A4:         cmp    [rem], 0
2   806AA:         jnz    else
3   806AC:         mov    [offset], 0
4   806B4:         jmp    next_code
5   806B6: else:   mov    rax, [rem]
6   806BB:         mov    rcx, [small_prime]
7   806C0:         sub    rcx, rax
8   806C3:         mov    rax, rcx
9   806C6:         mov    [offset], eax
10  806CA: //Code für Zeile 9
```

Listing 4.1: Assemblercode für die Zeilen 5 bis 8 der Funktion mpp_sieve, welche Teile des Codes der RSA-Primzahlgenerierung für Mozilla NSS ist.

4.1.3 Primzahlgenerierung in OpenPGP.js

Die Primzahlgenerierung von OpenPGP.js ist in der Datei prime.js beschrieben und startet mit der Funktion randomProbablePrime.Zuerst wird eine Zufallszahl entsprechend der gewünschten Bitlänge generiert. In OpenPGP.js wird kein Siebverfahren wie in Mozilla NSS angewandt, sondern es wird sichergestellt, dass ausschließlich nicht durch 2,3 und 5 teilbare Zahlen als Primzahlkandidaten weitergehend geprüft werden. Dazu wird eine Zahl $n \mod 30$ berechnet und der Rest betrachtet. Ist dieser weder durch 2,3 oder 5 teilbar, so ist auch n nicht durch 2,3 oder 5 teilbar. Andernfalls wird die kleinstmögliche Zahl $k \in \mathbb{N}$ auf n addiert, sodass $n + k \mod 30$ nicht durch 2,3 oder 5 teilbar ist. Diese kleinstmögliche Zahl ist $adds[n \mod 30]$, das heißt beispielsweise ist $125 \mod 30] = 2$ addiert gleich 127, die kleinstmögliche Zahl größer 125, welche nicht durch 2,3 und 5 teilbar ist.
Mit dieser Methode wird in der Zeile ein Kandidat erzeugt, der nicht durch 2,3 oder 5 teilbar ist und in der Funktion isProbablePrime tiefergehend geprüft wird. Zuerst wird die

Teilerfremdheit von *random_num*-1 zu dem Exponenten e überprüft (näheres dazu in 4.2). Danach wird ein einfacher Divisionstest mit allen Primzahlen zwischen 7 und 5000 durchgeführt, wohingegen dieser in Mozilla NSS bereits durch den Siebvorgang abgedeckt ist. Abschließend wird wie in Mozilla NSS ein Fermat- und Miller-Rabin-Test durchgeführt.

Pseudocode 13: Pseudo-Code für Primzahlgenerierung in OpenPGP.js

```
1  num_miller_rabin_tests <- get_num_tests(bitlen(random_num))
2  small_primes <- [7, 11, 13, 17, ..., 4999] //665 primes from 7 to 5000
3  Function randomProbablePrime(bits, e)
4      adds <- [1, 6, 5, 4, 3, 2, 1, 4, 3, 2, 1, 2, 1, 4, 3, 2, 1, 2, 1, 4, 3, 2, 1, 6, 5, 4, 3, 2, 1, 2]
5      random_num <- random.getRandomBN
6      i <- random_num mod 30
7      repeat
8          random_num <- random_num + adds[i]
9          i <- (i + adds[i]) mod 30
10         // If reached the maximum, go back to the minimum.
11         if bitlen(random_num) > bits then
12             random_num <- (random_num mod 2^bits) + 2^(bits-1)
13             i <- random_num mod 30
14     until !isProbablePrime(random_num,e,num_miller_rabin_tests)
15 Function isProbablePrime(random_num, e, num_miller_rabin_tests)
16     if gcd(random_num-1, e) ≠ 1 then
17         return false
18     foreach small_prime in small_primes do
19         if random_num mod small_prime = 0 then
20             return false
21     if !fermat(random_num, 2) then
22         return false
23     if !millerRabin(random_num, k) then
24         return false
25     return true
```

4.1.4 Mögliche Leakage in OpenPGP.js

Auch hier ist der Siebvorgang ein möglicher Codeabschnitt, um Informationen über *random_num* zu gewinnen. Ansatzpunkt ist wie im letzten Abschnitt die Überwachung des *small_primes*-Arrays, wobei Chrome auf dem verwendeten Testsystem 8 Byte für jeden Eintrag verwendet. Javascript legt die Größe des Datentyps für das *small_primes*-Array nicht fest, weshalb diese Aussage nicht auf andere Systemkonfigurationen zutreffen muss. Der Wert *random_num* wird nach jedem Abbruch der Funktion *isProbablePrime* um einen bestimmten Wert erhöht. In den folgenden Absätzen sei *random_num*$_k$ der Wert *ran-*

dom_num in dem k.-Durchlauf der Schleife (Zeilen 7 bis 14).

Es werden also pro Cache-Line, welche 8 Einträge des *small_primes*-Array abdeckt, bis zu 8 Zugriffe im Abstand von der Dauer einer Modulo-Operation, hier im Mittel etwa 1600 Taktzyklen, erwartet. In der Cache-Line mit weniger als 8 Zugriffen gilt für einen der $small\_prime_i$-Werte $random\_num_k \mod small\_prime_i = 0$, wobei dies zu einem sofortigen Abbruch (siehe Zeile 20) führt. Folglich haben wir die Information $random\_num_k \mod small\_prime_i = 0$, aber auch $random\_num_k \mod small\_prime_j \neq 0 \; \forall j < i$. Um sicherzugehen, dass die Foreach-Schleife in Zeile 18 vorzeitig verlassen wurde, kann zusätzlich der Code der Zeilen 7 bis 14 überwacht werden, welcher nach einem Abbruch geladen wird.

Da sich *random_num* in jeder Schleifen-Iteration ändert, müssen die aus der *isProbablePrime*-Funktion für $random\_num_k$ gewonnen (Un)gleichungen angepasst werden.

Die Differenz von $random\_num_k$ und $random\_num_{k+1}$ ist mit dem maximalen Wert im *adds*-Array, also 6 gleichzusetzen. Somit könnten aus den Beobachtungen Gleichungen folgender Form aufgestellt werden:

$$(random\_num + b) \mod small\_prime_j \not\equiv 0 \text{ mit } b \in [k \,..\, 6k] \text{ und } j < i$$
$$(random\_num + b) \mod small\_prime_i \equiv 0 \text{ mit } b \in [k \,..\, 6k]$$

Es steht die Frage im Raum, wie viele Gleichungen für die vollständige Rekonstruktion einer Primzahl benötigt werden. Für den Informationsgehalt der zweiten Gleichung ist die mittlere Größe beziehungsweise die mittlere Bitlänge von $small\_prime_i$ ausschlaggebend. Eigene Tests haben ergeben, dass die mittlere Bitlänge von $small\_prime_i$ bei den Primzahllängen 1024,2048 und 4096 etwa 5,3 Bit beträgt. Weiterhin haben etwa 75% der getesteten Zahlen eine der Zahlen in *small_prime* als Teiler, womit bei ca. 75% der getesteten Zahlen die zweite Gleichung entsteht. Die restlichen 25% werden durch den Fermat- und Miller-Rabin-Test aussortiert.

Angenommen $small\_prime_i$ hat im Mittel eine Bitlänge von l, dann liefert die zweite Gleichung l Bit an Information über $(random\_num + b)$. Allerdings erhöht sich durch b die Anzahl der möglichen Lösungen mit jeder Gleichung um den Faktor 6. Analog zu den Betrachtungen im Abschnitt 4.1.2 kann jedoch die Zusatzinformation $2^{bits-1} < random\_num < 2^{bits}$ herangezogen werden, um den Lösungsraum einzuschränken. Somit gibt es bei t-Gleichungen 6^t mögliche Lösungen, welche sich auf das Intervall $[0 \,..\, 2^{l \cdot t}]$ verteilen, wobei nur Lösungen im Intervall $[2^{bits-1} \,..\, 2^{bits} - 1]$ valide sind. Wenn $t > \lceil bits/10 \rceil$ gilt, dann ist $6^t / 2^{l \cdot t} \cdot 2^{bits}$ eine Abschätzung, wie viele der möglichen Lösungen im Mittel im validen Intervall $[2^{bits-1} \,..\, 2^{bits} - 1]$ zu finden sind.

Es bleibt zu klären, wie viele der Gleichungen des 2. Typs erwartet werden können, oder anders ausgedrückt, wie viele Zahlen im Mittel getestet werden müssen, bis eine davon die Primzahltests besteht. Für eine Abschätzung der zu erwartenden Anzahl sei angenommen, dass die Funktion ausschließlich Primzahlen zurückgibt, sodass der Primzahlsatz herangezogen werden kann, um die Anzahl der Primzahlen abzuschätzen. Ist beispielsweise eine 1024 Bit Primzahl gewünscht, so wird aus dem Intervall $[2^{1023} \,..\, 2^{1024} - 1]$ eine der

$$\pi(2^{1024} - 1) - \pi(2^{1023}) = \frac{(2^{1024} - 1)}{\ln(2^{1024} - 1)} - \frac{2^{1023}}{ln(2^{1023})} \approx 2^{1013,53} \tag{4.2}$$

möglichen Primzahlen gewählt. Somit ist im Intervall $[2^{1023} \,..\, 2^{1024} - 1]$ etwa eine von $2^{1023}/2^{1013,53} \approx\approx 710$ Zahlen eine Primzahl. Da durch 2,3 und 5 teilbare Zahlen nicht getestet werden, entfallen 22 von 30 aller natürlicher Zahlen als Testkandidaten, wie anhand des *adds*-Arrays zu sehen ist. Folglich ist bei einer Bitlänge von 1024 jede $710 \cdot 8/30 \approx 189$ getestete Zahl eine Primzahl, sodass ausgehend von einer zufälligen Zahl im Intervall $[2^{1023} \,..\, 2^{1024} - 1]$ im Mittel etwa 95 Iterationen zum Fund einer Primzahl ausreichen.
Die im Mittel benötigte Anzahl skaliert linear mit der Bitlänge, die Übersichtstabelle 4.1 gibt den Erwartungswert für verschiedene Bitlängen an.

Tabelle 4.1: Veranschaulicht den linearen Zusammenhang zwischen dem Erwartungswert für die Anzahl der Iterationen in Funktion randomProbablePrime (siehe Pseudocode 13) bei verschiedenen Primzahlbitlängen.

Bitlänge Primzahl	E(Anzahl Iterationen)
1024	95
1536	142
2048	189
4096	379

Um die Unsicherheit der obigen Kongruenzen zu verringern, soll im Folgenden das *adds*-Array näher analysiert werden. Dieses benötigt im Chrome 8 Byte pro Dateneintrag und ist somit über 4 Cache-Lines mit folgenden Werten verstreut:
Cache-Line 1: 1, 6, 5, 4, 3, 2, 1, 4
Cache-Line 2: 3, 2, 1, 2, 1, 4, 3, 2
Cache-Line 3: 1, 2, 1, 4, 3, 2, 1, 6
Cache-Line 4: 5, 4, 3, 2, 1, 2
Wenn etwa bekannt ist, dass der *adds*-Array Zugriff in Zeile 8 der Schleife eine Aktivität in Cache-Line 2 ausgelöst hat, so war $adds[i] \in \{1, 2, 3, 4\}$, sodass die Möglichkeiten 5 und 6 ausgeschlossen werden können. Somit kann der Offset-Wert b bei Kenntnis von Eviction-

Sets zu den 4 Cache-Lines eingrenzt werden.

Problematisch ist jedoch die Definition des *adds*-Arrays innerhalb der Funktion *randomProbablePrime*, da bei Versuchen mit Chrome das Array mit jedem Funktionsaufruf an einer anderen Stelle im Speicher stand. Das *adds*-Array während der Laufzeit der *randomProbablePrime*-Funktion zu finden, ist schwierig, da nur ein Zugriff in jeder der über 2000 Taktzyklen dauernden Schleifeniteration erfolgt. Alle Eviction-Sets, die eine Aktivität in diesem Zeitraum zeigen, müssten ebenso wie ihr direkt benachbartes Eviction-Set über mehrere Iterationen überwacht werden. Wenn ein Eviciton-Set, das initial gefunden wurde, in einer der Folgeiterationen keine Aktivität mehr misst, dann muss das direkt benachbarte Eviciton-Set Aktivität messen.

Erschwerend ist außerdem, dass die Schleife in den Zeilen 7 bis 14 nur begrenzt oft ausgeführt wird (siehe Tabelle 4.1) und somit die Zugriffe auf das *adds*-Array nach oben begrenzt sind.

Wie bereits erwähnt, optimieren moderne Browser häufig verwendete Codeteile während der Laufzeit. Wenn die Angreiferin also häufig hintereinander eine Schlüsselgenerierung mit kleinstmöglicher Bitlänge anstößt, könnte dies den Browser dazu veranlassen, die Funktion *randomProbablePrime* zu optimieren. Folglich könnte das *adds*-Array dauerhaft außerhalb der Funktion vorgehalten werden, anstatt es bei jedem Funktionsaufruf dynamisch zu erzeugen. Dann könnte die Angreiferin die zu den 4 Cache-Lines gehörigen Eviction-Sets bereits im Voraus suchen. Mit der verwendeten Systemkonfiguration konnte ein solches Verhalten in Chrome nicht erzeugt werden.

Zusammenfassend lassen sich Gleichungen wie im Abschnitt 4.1.2 aufstellen, wobei dieselben Fragen offen sind.

4.2 Zusätzliche Schlüsselprüfungen in Mozilla NSS

Eine wichtige Eigenschaft ist, dass beide Primzahlen des Schlüssels teilerfremd zum Exponenten e sind. In Mozilla NSS werden zuerst die Schlüsselparameter p, q, n, d, e bestimmt und anschließend in der Funktion RSA_PrivateKeyCheck auf Gültigkeit überprüft (siehe Pseudo-Code 14).

Relevant für diese Arbeit sind im Wesentlichen die Zeilen 4 und 5, in denen die Teilerfremdheit von e zu $p-1$ und $q-1$, d.h. $gcd(e, p-1) = 1$ und $gcd(e, q-1) = 1$ geprüft wird. Aus Performancegründen wird der Exponent e, anders als ursprünglich im RSA-Algorithmus beschrieben, auf den Wert 65537 fixiert.

Pseudocode 14: Pseudo-Code für RSA_PrivateKeyCheck aus rsa.c

```
1  Function RSA_PrivateKeyCheck(key)
2      assert(p ≠ q)
3      assert(n == p * q)
4      assert(gcd(e, p-1) == 1)
5      assert(gcd(e, q-1) == 1)
6      assert(d*e == 1 mod p-1)
7      assert(d*e == 1 mod q-1)
8      assert(d_p == d mod p-1)
9      assert(d_q == d mod q-1)
10     assert(q * q^-1 == 1 mod p)
```

4.2.1 Theoretische Leakage-Analyse

Interessant ist die Funktion *mp_gcd*, welche den größten gemeinsamen Teiler nach dem binären Verfahren von Josef Strein [Ste67] berechnet. Dieser Algorithmus (siehe Pseudocode 15) verwendet zum Berechnen des ggT ausschließlich Rechts-Shift-Operationen (Teilen durch 2) und Subtraktionen, wodurch dieser besonders für die in diesem Kontext verwendeten großen Zahlen interessant ist. Die Zeilen 1 bis 6 der Funktion *mp_gcd* können in diesem Fall ignoriert werden, da der Exponent e wie oben beschrieben immer 65537 und damit ungerade ist. Bedeutender hingegen ist die $while$-Schleife in den Zeilen 11 bis 17, welche abhängig von den Eingaben Fallunterscheidungen durchführt.

Die im folgenden beschriebene Leakage geht auf die Veröffentlichung [CAPGATBB18] zurück, in welcher die Autoren die binäre GCD-Implementation in OpenSSL untersucht haben.

Der Kontrollfluss der *mp_gcd*-Funktion ist stark abhängig von den Eingaben und sorgt dafür, dass zwischen zwei Aufrufen der Subtraktionsfunktion (Zeile 17) unterschiedlich viele Rechts-Shift-Operation (Zeile 13) stattfinden. Gemäß der der hierzu vorhandenen Literatur [ASSS17] wird der Kontrollfluss des Algorithmus mit 2 Symbolen beschrieben. Dabei „L“ steht für eine Rechts-Shift-Operation und „S“ für eine Subtraktion.

Eine kompaktere Repräsentation verwendet die Variable Z_i für die Anzahl der Rechts-Shift-Operationen in Iteration i und Variable X_i für den binären Wert des Konditionals in Zeile 14. Es gilt $X_i =$ true wenn $t > 0$ und $X_i =$ false wenn $t \leq 0$.

Bei der folgenden Analyse wird ausgenutzt, dass der Wert $e = 65535$ fix und bekannt, sowie deutlich kleiner als p beziehungsweise q bei heute gängigen 1024 Bit Primzahlen ist. Die Analyse soll sich im Folgenden exemplarisch auf p beschränken. Diese große Differenz zwischen e und p sorgt dafür, dass die Bedingung in Zeile 14 für fast alle Iterationen mit $X_i = f$ vorhergesagt werden kann. Denn beim Aufruf der Funktion *mp_gcd* gilt $u = e, v = p - 1$ und da $u = e = 65535$ ungerade ist, gilt $t = -p + 1$. Beispielhaft wird

Pseudocode 15: Pseudo-Code für mp_gcd nach Josef Stein

```
1  Function mp_gcd(u,v)
2      k ← 0
3      while iseven(u) & iseven(v) do
4          u ← u/2
5          v ← v/2
6          k++
7      if isodd(u) then
8          t ← -v
9      else
10         t ← v
11     while t ≠ 0 do
12         while iseven(t) do
13             t ← t/2
14         if t > 0 then
15             u ← t
16         else v ← -t
17         t ← u - v
18     return u*2^k
```

nun die erste Iteration der Schleife in Zeile 11 bis 17 betrachtet. Durch die Shift-Operation der Schleife in Zeile 12 und 13 wird $t = (-p+1)/2^{Z_1}$ gesetzt. Es gilt $t < 0$, daher wird $v = -t = (p-1)/2^{Z_1}$ gesetzt. Zum Abschluss der Iteration setzt die Subtraktion in Zeile 17 $t = u - v = e - (p-1)/2^{Z_1}$.

Somit gilt $t > 0$ solange, bis t ungefähr auf die Bitlänge 17 geschrumpft ist, also $u = e$ gilt. Wenn die Subtraktionen von e außer Acht gelassen werden, wird t etwa $\mathrm{ld}(p) - \mathrm{ld}(e)$ mal geteilt, bevor der obige Fall eintritt.

Gemäß der Definition der Z_i verliert t in jeder Iteration Z_i Bits. Die Anzahl der Iterationen k die in der Schleife der Zeilen 11 bis 17 nötig sind, bis t dieselbe Bitlänge wie u erreicht, ist das kleinste k welches folgende Gleichung erfüllt:

$$n = \sum_{i}^{k} = Z_i >= \mathrm{ld}(p) - \mathrm{ld}(e) \tag{4.3}$$

Es stellt sich also die Frage, wie viele Bits in diesem Szenario rekonstruiert werden können. Angenommen die Angreiferin erhält alle Z_i bis zur Iteration k, in der das erste Mal $t < u = e$, das heißt $X_i = f\ \forall i < k$ gilt. Sei t_i der Wert von t am Beginn der Iteration (Zeile

11) $i < t$, wobei $u_i = u_1 = e$ für alle Iterationen $i < t$ bleibt. Es gilt

$$t_1 = -p + 1,\ t_{i+1} = \frac{t_i - e}{2^{Z_{i+1}}} \tag{4.4}$$

Die Gleichung $u - v \equiv 0 \mod 2$ gilt für alle Iterationen, da u und $v = -t$ am Ende der Iteration jeweils ungerade sind. Aufgelöst gilt für $e - t_k$:

$$e - t_k = e - \cfrac{-p+1}{e - \cfrac{2^{Z_1}}{e - \cfrac{2^{Z_2}}{e - \cfrac{2^{Z}_{3}}{\ddots \\ e - \cfrac{}{2^{Z_k}}}}}} \equiv 0 \mod 2 \tag{4.5}$$

Um Bits von p zu erhalten, wird der obige Kettenbruch nach p aufgelöst,

$$p = -e(2^Z_1 + 2^{Z_1+Z_2} + ... + 2^n) + 1 \mod 2^{n+1} \tag{4.6}$$

wobei n die Anzahl der rekonstruierten Bits nach Gleichung 4.3 angibt. Zusammenfassend könnten etwa $bitlen(p) - bitlen(e) = bitlen(p) - 17$ Bits rekonstruiert werden. Mittels des Coppersmith-Angriffs [Cop96] benötigt die Angreiferin jedoch nur die Hälfte der Bits der Primzahlen p oder q, um den gesamten Schlüssel zu rekonstruieren. Dies ist im Hinblick auf die Dauer der Shift- und Subtraktions-Operationen interessant. Denn diese werden mit der Zeit immer schneller, da die t_i mit zunehmenden i kleiner werden und somit der Rechenaufwand sinkt. Dies erschwert die richtige Bestimmung der hinteren Z_i, wobei diese mit dem obigen Algorithmus nicht alle gebraucht werden.

4.2.2 Praktische Leakage-Analyse

Wie im vorherigen Abschnitt beschrieben, muss die Angreiferin die Werte Z_i bestimmen. Hierzu bietet es sich an, die Codefragmente zu überwachen, welche die Shift- beziehungsweise Subtraktionsfunktionen ausführen.
In Mozilla NSS wird die Shift-Operation durch die Funktion *s_mp_div_2* umgesetzt. Diese ist wiederum nur ein Wrapper und ruft intern die *s_mp_div_2d*-Funktion mit dem Parameter 1 auf, welches dem Teilen durch 2 entspricht (siehe auch Abbildung 4.1). Die *s_mp_div_2d*-Funktion setzt die Division mittels der *s_mp_rshd*-Funktion (Rechtsshift) um. Am Ende der *s_mp_div_2d*-Funktion erfolgt ein Funktionsaufruf an *s_mp_clamp*, um die entstandenen führenden Nullen zu entfernen. Welche Adressen die Funktionen im As-

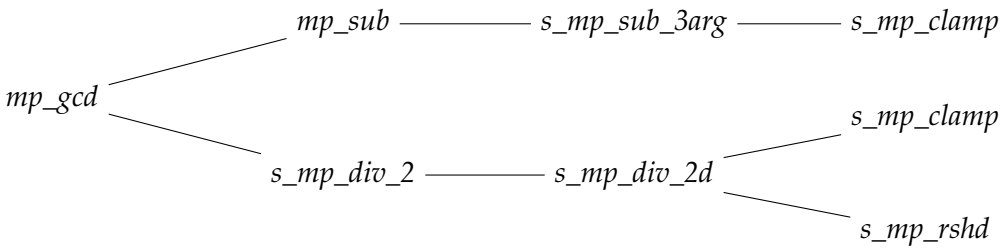

Abbildung 4.1: Zeigt die Funktionen welche beim Ausführen der Shift- und Subtraktions-Operation in Mozilla NSS aufgerufen werden.

semblercode haben, ist in Tabelle 4.2 ersichtlich.

Tabelle 4.2: Größe, Start- und Endadresse der für die Shift-Operation relevanten Funktionen (fett hervorgehoben) sowie deren direkten Nachbarn im Assemblercode.

Funktion	Startadresse	Endadresse	Differenz (Größe der Funktion)
s_mp_alloc	$b7d0_{16}$	$b7f4_{16}$	25_{16}
s_mp_free	$b7f5_{16}$	$b815_{16}$	21_{16}
s_mp_clamp	$b816_{16}$	$b85e_{16}$	49_{16}
s_mp_exch	$b85f_{16}$	$b8ce_{16}$	70_{16}
⋮	⋮	⋮	⋮
s_mp_mul_2d	$b9c1_{16}$	$bb76_{16}$	$1b6_{16}$
s_mp_rshd	$bb77_{16}$	$bc5f_{16}$	$e9_{16}$
s_mp_div_2	$bc60_{16}$	$bc7e_{16}$	$1f_{16}$
s_mp_mul_2	$bc7f_{16}$	$bd58_{16}$	da_{16}
⋮	⋮	⋮	⋮
s_mp_mod_2d	$bd59_{16}$	$be16_{16}$	be_{16}
s_mp_div_2d	$be17_{16}$	$bf1d_{16}$	107_{16}
s_mp_norm	$bf1e_{16}$	$bfde_{16}$	$c1_{16}$

Eine Änderung innerhalb der Funktionen, welche die Größe beeinflusst, ist bei diesen grundlegenden Funktionen unwahrscheinlich. Bei Änderungen im Code außerhalb dieser Funktionen werden sich alle Adressen um einen bestimmten Offset ändern. Deshalb sind in der Tabelle die Größe der Funktionen im Assemblercode und deren direkten Nachbarn angegeben, um abzuschätzen, welche weiteren Funktionen überwacht werden. Die im weiteren Verlauf vorgenommene Bewertung der Funktionen hängt also auch von deren Lage im Assemblercode ab und kann bei Änderungen der Adressen variieren.
Es bestehen mehrere Möglichkeiten, die Ausführung der Shift-Operation zu erkennen, wobei sich manche im Callstack befindliche Funktionen weniger eignen, da sie auch an anderer Stelle aufgerufen werden. So wird etwa die Funktion *s_mp_clamp* auch im Call-

stack der Subtraktions-Operation verwendet, sodass deren Überwachung für die Shift-Operation falsche Ergebnisse liefern würde.
Die Funktion *s_mp_div_2* ist ein guter Kandidat für die Überwachung der Shift-Operation, da die benachbarten Funktionen keine Schnittmenge mit den aufgerufenen Funktion in der Subtraktionoperation haben. Die Funktion liegt sowohl innerhalb der Cache-Line von 940 bis 97f als auch innerhalb der Cache-Line 980 bis 9af.
Angenommen es wird die Cache-Line von 980 bis 9af gewählt, dann wird auch Aktivität gemessen, sofern ausschließlich die Funktion *s_mp_mul_2* aufgerufen wird.
Die Überwachung der Cache-Line von 940 bis 97f dagegen schlägt auch dann an, wenn die Funktion *s_mp_rshd* aufgerufen wird. Dies passiert etwa im Zuge der Shift-Operation in weniger als 100 Taktzyklen nach dem Aufruf von *s_mp_div_2*. Sollte die Prime-and-Probe-Dauer weniger als 100 Taktzyklen betragen, würde die Gefahr bestehen, fälschlicherweise zwei anstatt einer Shift-Operation zu messen. Nach den in dieser Arbeit durchgeführten Tests scheint eine solche Dauer aber unrealistisch niedrig. Daher eignet sich die Cache-Line von 940 bis 97f besser, da lediglich Funktionen überwacht werden, die Teil der Shift-Operation sind.
Die Subtraktions-Operation wird durch die Funktion *mp_sub* umgesetzt, welche prüft, welche Vorzeichen die Operanden besitzen, und in Abhängigkeit vom Ergebnis weitere Funktion aufruft. Es ist bekannt, dass bei der Subtraktion in Zeile 16 sowohl $u = e$ als auch $v = -t$ positiv sind, womit die Operanden die gleichen Vorzeichen besitzen und die Funktion *s_mp_sub_3arg* aufgerufen wird. Diese verwendet am Ende wieder die Funktion *s_mp_clamp*, um bei der Subtraktion entstandene führende Nullen zu entfernen. Welche Adressen die Funktionen im Assemblercode haben, ist in Tabelle 4.3 ersichtlich.

Tabelle 4.3: Größe, Start- und Endadresse der für die Subtraktions-Operation relevanten Funktionen (fett hervorgehoben) sowie deren direkten Nachbarn im Assemblercode.

Funktion	Startadresse	Endadresse	Differenz (Größe der Funktion)
mp_add	8377_{16}	$846f_{16}$	$f8_{16}$
mp_sub	8470_{16}	$85c3_{16}$	153_{16}
mp_mul	$85c4_{16}$	$894b_{16}$	387_{16}
⋮	⋮	⋮	⋮
s_mp_sub	$c914_{16}$	$ca6a_{16}$	157_{16}
s_mp_sub_3arg	$ca6b_{16}$	$cc01_{16}$	197_{16}
s_mp_mul	$cc02_{16}$	$cc2a_{16}$	29_{16}

Geeignet für eine Überwachung der Subtraktionsfunktion ist die Funktion *mp_sub*, wobei hierfür etwa die Cache-Line von 780 bis 7bf in Frage kommt.

Am Beginn des Angriffs steht die Angreiferin vor der Aufgabe, passende Eviction-Sets zu den für die Überwachung ausgewählten Cache-Lines zu suchen. Hierzu wird sie die Schlüsselgenerierung manuell anstoßen und nach bestimmten Aktivitätsmustern in den überwachten Cache-Sets suchen.
Sobald eine beliebige Cache-Line einer 4-KiB-Page einem Eviction-Set zugeordnet werden kann, sind auch alle anderen Zuordnungen dieser 4-KiB-Page bekannt. Daher genügt es, die Cache-Line von b500 bis b53f einem Eviction-Set zuzuordnen, um ein Eviction-Set für die Cache-Line von b940 bis b97f zu finden, welche für die Überwachung der Shift-Operation genutzt werden kann.
Um die Werte Z_i zu bestimmen, muss die Prime-and-Probe mindestens die gleiche Geschwindigkeit wie die Shift- beziehungsweise Subtraktionsfunktion erreichen. In der Tabelle 4.4 ist die durchschnittliche Anzahl der Zyklen für beide Funktion bei verschiedenen Schlüssellängen angegeben. Zum Vergleich wurde die Dauer der gesamten *mp_gcd*-Funktion hinzugefügt. Die Dauer der Shift- sowie Subtraktionoperation hängt linear von der Schlüssellänge ab, wobei dies für die Dauer der gesamten *mp_gcd*-Funktion nicht der Fall ist.

Tabelle 4.4: Darstellung des linearen Zusammenhangs der Taktzyklendauer einer Shift- und Subtraktionsfunktion zu der Bitlänge. Als Vergleich die Dauer der gesamten GCD-Funktion ohne linearen Zusammenhang zu der Bitlänge.

Bitlänge	Shift	Subtraktion	*mp_gcd*
2048	340	477	424K
4096	577	718	1,2M
8192	9597	1223	3,73M

Theoretisch bestünde die Möglichkeit, allein durch die Zeitdifferenz zwischen zwei Subtraktionsfunktionen die Anzahl der Shift-Operationen zu bestimmen. Sobald eine Auflösung von unter 330 Taktzyklen bereit steht, können Lücken in der Aktivität als Subtraktions-Operation interpretiert werden. Andersherum wird die Abschätzung ungenau, wenn nur eine Auflösung im Bereich der Subtraktions-Operationsdauer bereitsteht, sodass beispielsweise die Unterscheidung von einer oder zwei zwischenzeitlichen Shift-Operationen fehleranfällig wird.
Die Prime-and-Probe Implementation in Javascript/Webassembly erwies sich als zu langsam, um aufeinanderfolgende Shift-Operationen zu unterscheiden. Wie in den Benchmarks in Kapitel 2 beschrieben, dauert eine Prime-and-Probe Operation bei keiner Aktivität auf dem Cache-Set etwa 200ns, was ca. 700 Taktzyklen entspricht. Im Falle eines aktiven Cache-Set steigt die Dauer jedoch auf ca. 500ns (1700 Taktzyklen) an. Selbst für eine Schlüssellänge von 8192 Bit ist dieser Wert noch zu hoch.

4.3 Bremsen der Ausführungsgeschwindigkeit

Auch in anderen Fällen ist die Geschwindigkeit der überwachten Funktion zu schnell, um mittels der Prime-and-Probe-Funktion überwacht zu werden. Hierfür gibt es mehrere Vorschläge, die Funktion künstlich auszubremsen, welche in der Literatur [WXW12, VZRS15, IES17] unter dem Begriff performance-degradation-attacks beschrieben werden. Ein trivialer Angriff ist etwa die Überwachung selber, da diese mit der Prime-Phase ein Cache-Set komplett füllt, sodass Daten des angegriffenes Programms verdrängt und so aus dem langsamen Hauptspeicher geladen werden müssen. Der Effekt ist auch in umgekehrter Richtung, also von dem Opferprogramm auf das Angriffsprogramm vorhanden. Die Differenz der Prime-and-Probe-Operationsdauer bei keiner gegenüber viel Aktivität auf dem Cache-Set von über 200 ns verdeutlicht dies (siehe auch Abschnitt 5.7.1).

In anderen Kontexten wird häufig der $clflush$-Befehl verwendet, um schnell und zuverlässig Daten aus dem Cache zu verdrängen. Ein Äquivalent existiert jedoch weder in Javascript noch in Webassembly.

Des Weiteren kann die Ausführung von Code im Browser nicht auf bestimmte CPU-Kerne fixiert werden. Hiermit sind Angriffe, die einen Thread mit hoher Rechenlast auf dem gleichen physischen Kern wie das Opferprogramm laufen lassen, nicht ohne Weiteres möglich.

Der oben beschriebene triviale Angriff kann optimiert werden, um die Ausführungsgeschwindigkeit weiter zu bremsen. Dafür wird die Prime-and-Probe-Operation ohne Zeitmessung für eine bestimmte Zeit in einer Endlosschleife, nachfolgend Prime-Spam genannt, ausgeführt.

Wie oben beschrieben, erstreckt sich der relevante Code für die Shift- und Subtraktions-Operation über mehrere Funktionen und Cache-Lines. Jeder dieser Cache-Lines ist ein potenzielles Ziel für die Performancereduktion. Da ein Prime-Spam auf eine Cache-Line einen kompletten physischen Kern auslastet, sollte das Ziel sorgfältig gewählt sein.

Um die Auswirkung vom Prime-Spam auf unterschiedliche Cache-Lines zu bewerten, wird die *mp_gcd*-Funktion mit den Parametern $e = 65535$ und $p - 1$ ausgeführt. p ist eine zufällige aber über den gesamten Benchmarkzeitraum feste Pseudoprimzahl, wobei RSA-2048 als Parameter gewählt wurde. Tabelle 4.5 betrachtet alle Cache-Lines, welche Code für die Shift-Operation beinhalten, wobei die Funktion *s_mp_clamp* wie oben erwähnt auch in der Subtraktionsoperation verwendet wird. Funktionen sind nicht auf Cache-Line-Grenzen aligned , weshalb die erste und letzte Cache-Line eines Funktionscodes auch Code für andere Funktionen beinhaltet.

Die Referenzwerte stammen aus der Tabelle 4.4, wobei diese Werte unter demselben Parameter $p - 1$ entstanden sind. Am besten geeignet sind die beiden Cache-Lines der

Tabelle 4.5: RSA-2048 performance-degradation-attack mittels Prime-Spam auf Cache-Lines, welche Code der Shift-Operation beinhalten. 1. Spalte: ID für die Cache-Line für übersichtlichere Bezeichnung, 2. Spalte: Adressbereich der durch den Angriff verdrängten Cache-Line, 3. Spalte: Funktionen deren Code innerhalb des Adressbereichs der Cache-Line liegt, Spalten „Shift“, „Sub“ und „*mp_gcd*“: Mittlere Dauer der Shift-, Subtraktions-, beziehungsweise *mp_gcd*-Operation in Taktzyklen bei einem aktiven Angriff auf die in Spalte 2 spezifizierte Cache-Line. Die erste Zeile gibt die Referenzzeiten ohne aktiven Angriff an. Die horizontalen Linien innerhalb der Tabelle zeigen Sprünge zwischen Adressen der Cache-Lines an.

ID	Cache-Line	Funktion(en) innerhalb der Cache-Line	Shift	Sub	mp_gcd
	Referenzzeiten ohne Prime-Spam		340	477	424k
$V1_1$	$b800_{16}$ - $b83f_{16}$	s_mp_free, s_mp_clamp	541	608	549k
$V1_2$	$b840_{16}$ - $b87f_{16}$	s_mp_clamp, s_mp_exch	538	605	553k
$V2_1$	$bb40_{16}$ - $bb7f_{16}$	$s\_mp\_mul\_2d, s\_mp\_rshd$	450	514	479k
$V2_2$	$bb80_{16}$ - $bbbf_{16}$	*s_mp_rshd*	451	542	486k
$V2_3$	$bbc0_{16}$ - $bbff_{16}$	*s_mp_rshd*	486	563	581k
$V2_4$	$bc00_{16}$ - $bc3f_{16}$	*s_mp_rshd*	520	563	597k
$V2_53_1$	$bc40_{16}$ - $bc7f_{16}$	$s\_mp\_rshd, s\_mp\_div\_2, s\_mp\_mul\_2$	480	503	474k
$V4_1$	$be00_{16}$ - $be3f_{16}$	$s\_mp\_mod\_2d, s\_mp\_div\_2d$	511	560	544k
$V4_2$	$be40_{16}$ - $be7f_{16}$	*s_mp_div_2d*	503	549	536k
$V4_3$	$be80_{16}$ - $bebf_{16}$	*s_mp_div_2d*	539	557	551k
$V4_4$	$bec0_{16}$ - $beff_{16}$	*s_mp_div_2d*	537	547	552k
$V4_5$	$bf00_{16}$ - $bf3f_{16}$	$s\_mp\_div\_2d, s\_mp\_norm$	470	506	485k

s_mp_clamp-Funktion, da sie nicht nur Spitze in der Verlangsamung der Shift-Operation sind, sondern zusätzlich noch die Subtraktionsoperation besser als alle anderen bremsen.

Tabelle 4.6 betrachtet alle Cache-Lines, welche Code für die Subtraktionsoperation beinhalten, wobei die Zeilen 2 und 3 der Tabelle 4.5 auch hier relevant sind. Anders als bei der Shift-Operation bremsen bei der Subtraktions-Operation die Cache-Lines der *mp_sub* signifikant besser als die Cache-Lines der *s_mp_clamp*-Funktion.

Zusammenfassend kann die Shift- beziehungsweise Subtraktionsoperation durch einen Angriff maximal von 340 auf 541 ($V1_1$) Taktzyklen beziehungsweise von 477 auf 766 ($S1_7$) gebremst werden, welches einer Erhöhung der Ausführungszeit von ∼59% respektive ∼61% entspricht. Tabelle 4.7 zeigt, dass bei RSA-4096 und RSA-8192 prozentual und sogar bei den absoluten Werten schwächere Effekte zu beobachten sind. Beim Vergleich wurden ausschließlich die Cache-Lines für die Shift- beziehungsweise Subtraktionsoperation getestet, welche den stärksten Bremseffekt bei RSA-2048 aufwiesen. So wird die Shift-Operation unter RSA-4096 von 543 auf 700 ($V1_1$) Taktzyklen und unter RSA-8192 von 982 auf 1125 ($S1_7$) Taktzyklen gebremst, welches die Ausführungszeit um ∼29% beziehungsweise ∼15% erhöht.

Tabelle 4.6: RSA-2048 performance-degradation-attack mittels Prime-Spam auf Cache-Lines, welche Code der Subtraktions-Operation beinhalten. Spalteninhalte analog zu Tabelle 4.5

ID	Cache-Line	Funktion	Shift	Sub	mp_gcd
	Referenzzeiten ohne Prime-Spam		340	477	424k
$S1_1$	8440_{16}-$847f_{16}$	mp_add, mp_sub	357	643	487k
$S1_2$	8480_{16}-$84bf_{16}$	*mp_sub*	359	684	538k
$S1_3$	$84c0_{16}$-$84ff_{16}$	*mp_sub*	362	681	531k
$S1_4$	8500_{16}-$853f_{16}$	*mp_sub*	357	700	538k
$S1_5$	8540_{16}-$857f_{16}$	*mp_sub*	362	683	534k
$S1_6$	8580_{16}-$85cf_{16}$	*mp_sub*	358	732	534k
$S1_7$	$85c0_{16}$-$85ff_{16}$	mp_sub, mp_mul	359	766	531k
$S2_1$	$ca40_{16}$-$ca7f_{16}$	$s\_mp\_sub, s\_mp\_sub\_3arg$	357	643	490k
$S2_2$	$ca80_{16}$-$cabf_{16}$	*s_mp_sub_3arg*	359	644	494k
$S2_3$	$cac0_{16}$-$caff_{16}$	*s_mp_sub_3arg*	357	675	534k
$S2_4$	$cb00_{16}$-$cb3f_{16}$	*s_mp_sub_3arg*	358	669	536k
$S2_5$	$cb40_{16}$-$cb7f_{16}$	*s_mp_sub_3arg*	357	663	533k
$S2_6$	$cb80_{16}$-$cbbf_{16}$	*s_mp_sub_3arg*	359	676	546k
$S2_7$	$cbc0_{16}$-$cbff_{16}$	*s_mp_sub_3arg*	356	663	535k
$S2_8$	$cc00_{16}$-$cc3f_{16}$	$s\_mp\_sub\_3arg, s\_mp\_mul$	352	571	483k

Die beiden Cache-Lines $V1_1$ und $V1_2$ der *s_mp_clamp*-Funktion sind dabei zu bevorzugen, da sie beide Operationen gleichzeitig gut bis sehr gut bremsen. Des Weiteren sollte der Fokus auf das Bremsen der Shiftoperation gelegt werden, da diese zum einen schneller und die Unterscheidung einzelner Shiftoperationen für die Rekonstruktion der Primzahl wesentlich ist. Zum anderen Bremsen viele der Cache-Lines mit Shift-Operationscode auch die Subtraktionsfunktion messbar, wobei dieses Phänomen andersherum nicht beobachtet werden kann.

In den obigen Tabellen wurde jeweils nur eine Cache-Line gebremst. Tabelle 4.8 zeigt, lässt sich der Bremseffekt verbessern, wenn statt einer mehrere Cache-Lines gebremst werden.

Da nur ein Thread die Bremsung ausführt, werden die Cache-Lines nacheinander mittels Prime-Spam gebremst, sodass jede Cache-Line die gleichen Anteile an der Berechnungszeit des Bremsthreads erhält. Auffällig ist, dass die Cache-Lines möglichst nicht die gleiche Funktionen abdecken sollten. Beispielsweise bringt das Bremsen von $V1_1$ und $V1_2$ gegenüber dem alleinigen Bremsen von $V1_1$ oder $V1_2$ keinerlei Vorteile.

Echte Vorteile im Vergleich zum Bremsen einer Cache-Line ergeben sich sobald die Cache-Lines verschiedene Funktionen abdecken. So decken $V1_1$,$V1_2$,$V2_4$ drei verschiedene Funktionen ab die im Laufe der Shift-Operation benötigt werden und erzielen damit eine um 25% beziehungsweise 136 Taktzyklen erhöhte Ausführungszeit gegenüber des bes-

Tabelle 4.7: Performance-degradation-attack mittels Prime-Spam mit verschiedenen RSA-Bitlängen. Als Cache-Lines wurden diejenigen verwendet, welche sich am effektivsten in RSA-2048 erwiesen haben. Sonstige Spalteninhalte analog zu Tabellen 4.5 und 4.6

RSA-Bitlänge	ID	Cache-Line	Funktion	Shift	Subt	mp_gcd
2048		Referenzzeiten ohne Prime-Spam		340	477	424k
2048	$V1_1$	$b800_{16}$ - $b83f_{16}$	s_mp_free, s_mp_clamp	541	608	549k
2048	$S1_7$	$85c0_{16}$-$85ff_{16}$	mp_sub, mp_mul	359	766	531k
4096		Referenzzeiten ohne Prime-Spam		543	723	1,19M
4096	$V1_1$	$b800_{16}$ - $b83f_{16}$	s_mp_free, s_mp_clamp	700	770	1,64M
4096	$S1_7$	$85c0_{16}$-$85ff_{16}$	mp_sub, mp_mul	609	1057	1,51M
8192		Referenzzeiten ohne Prime-Spam		982	1223	3,71M
8192	$V1_1$	$b800_{16}$ - $b83f_{16}$	s_mp_free, s_mp_clamp	1125	1553	4,83M
8192	$S1_7$	$85c0_{16}$-$85ff_{16}$	mp_sub, mp_mul	1046	1679	4,64M

ten Ergebnisses mit nur einer Cache-Line. Insgesamt kann als die mittlere Zeit für eine Shift-Operation um 99% von 340 auf 677 Taktzyklen erhöht werden.
Trotz der Effekte der Bremsung ist die Ausführungsgeschwindigkeit der Shift- beziehungsweise Subtraktionsoperation in allen Fällen zu hoch, da eine Prime-and-Probe-Operation $\sim$1700 Taktzyklen benötigt. Ein weiteres Problem dieser Methode ist, dass sie nicht alle Shift- beziehungsweise Subtraktions-Operation gleichmäßig stark bremsen, wie folgendes Beispiel erläutert: Eine Operationsfolge innerhalb der *mp_gcd*-Funktion startet mit einer Subtraktions-Operation, folgend von x Shift-Operationen und abschließend wieder einer Subtraktions-Operation. Vor dem Beginn der ersten Shift-Operation wurden durch den Prim-Spam-Thread Teile des Codes aus dem Cache verdrängt. Deshalb ist die erste Shift-Operation stark gebremst, da sie Codeteile aus dem Hauptspeicher laden muss. Die Prime-Spam-Funktion ist nur aber zu langsam um während der Berechnungsphase der Shift-Operation erneut Teile des Codes aus dem Cache zu verdrängen. Deshalb kann die Zweite Shift-Operation in der Folge den Code aus dem Cache nutzen und ist so deutlich schneller in der Ausführung.
Ab wann ein Angriff möglich wäre, wird in Abschnitt 5.4 analysiert.

Tabelle 4.8: RSA-2048 Performance-Degradation-Attack mittels Prime-Spam auf mehrere Cache-Lines. Für die Auflösung der Cache-Line IDs siehe Tabelle 4.5. Die 2. Spalte gibt der Übersicht halber nur die Funktionen innerhalb der Cache-Lines (1. Spalte) an, welche bei der Berechnung der Shift-Operation relevant sind. So wird etwa die Funktion *s_mp_free* in der ersten Zeile nicht aufgeführt, obwohl die Cache-Line $V1_1$ Teile des Codes von *s_mp_free* enthält. Aus Platzgründen wurden die Funktionsnamen gekürzt, wobei für den vollen Funktionsnamen $*$ mit *s_mp_* zu substituieren ist.

Cache-Line IDs	Relevante Funktion(en) innerhalb der Cache-Line	Shift	Sub	mp_gcd
$V1_1$,$V1_2$	$*clamp$	542	557	528k
$V4_3$,$V4_4$	$*div\_2d$	533	533	572k
$V2_4$,$V2_5 3_1$	$*rshd$, $*div\_2$	526	563	551k
$V1_1$,$V2_4$	$*clamp$, $*rshd$	643	549	548k
$V1_1$,$V1_2$,$V2_4$	$*clamp$, $*rshd$	677	535	573k
$V1_2$,$V2_4$,$V4_3$	$*clamp$, $*rshd$, $*div\_2d$	601	565	541k
$V1_1$,$V2_4$,$V4_3$	$*clamp$, $*rshd$, $*div\_2d$	623	560	544k
$V1_1$,$V1_2$,$V2_4$,$V4_3$	$*clamp$, $*rshd$, $*div\_2d$	649	557	551k
$V1_1$,$V1_2$,$V2_4$,$V4_3$,$V4_4$	$*clamp$, $*rshd$, $*div\_2d$	669	563	563k
$V1_1$,$V1_2$,$V2_4$,$V4_1$-$V4_4$	$*clamp$, $*rshd$, $*div\_2d$	665	575	574k
$V1_1$,$V1_2$,$V2_4$-$V4_4$	$*clamp$, $*rshd$, $*div\_2$, $*div\_2d$	622	574	562k
$V1_1$-$V4_5$	$*clamp$, $*rshd$, $*div\_2$, $*div\_2d$	622	574	562k

5 Diskussion

In diesem Kapitel sollen die Ergebnisse der durchgeführten und beschriebenen Angriffe eingeordnet und sich daraus ergebende Fragestellungen diskutiert werden.

5.1 Anzahl der benötigten Kerne

Ein Cache-Angriff im Browser ist wie in den letzten Kapiteln gezeigt möglich, sofern präzise Timer vorhanden sind. Da präzise Timer zurzeit mittels eines Shared-Array-Buffer erzeugt werden müssen, benötigt die Angreiferin mindestens drei virtuelle Kerne, um einen Angriff auszuführen. Einen Kern für die Iteration der Timer-Variable, einen für den Prime-and-Probe- Angriffscode und einen, auf dem das Opferprogramm läuft.
Theoretisch reichen deshalb bereits Prozessoren mit zwei physischen und vier virtuellen Kernen, wie sie noch vielfach von Intel im Umlauf sind. Bei zwei physischen Kernen wird der Timer-Thread jedoch zeitweise exklusiv auf einem physischen Kern rechnen und sich zeitweise einen Kern mit einem der anderen Threads teilen. Wenn ein anderer Thread auf dem gleichen Kern rechnet, halbiert sich in etwa die Iterationsgeschwindigkeit der Timer-Variable. Da das Scheduling von Linux die Threads nicht auf Kerne fixiert, unterliegen Zugriffswerte aufgrund der unterschiedlichen Iterationsgeschwindigkeit des Timers starken Schwankungen. Problematisch ist dies beispielsweise in der Expand-Phase, da dann wegen der falschen Interpretation der Timer-Werte nicht erkannt wird, dass ein Candidate-Set bereits ein Eviction-Set für die Zeugenadresse ist.
Somit ist es essenziell zu erkennen, wann der Timer-Thread einen Kern für sich alleine beansprucht. Es könnte zusätzlich zur Laufzeit wiederholt - nach einer festen Zeitspanne - die Timergenauigkeit überprüft werden. Den Wert der Funktion *performance.now* als konstanten Zeitgeber heranzuziehen, wie im Kapitel 2 geschehen, wäre aufgrund der geringen Auflösung (Chrome etwa 0,1ms) sehr zeitaufwendig. Als Alternative bietet sich etwa das Messen der Laufzeit einer Prime-and-Probe Operation auf einem wenig aktiven Cache-Set als Referenzwert an. Hierbei ist zu beachten, dass die Prüfung im Thread des Angriffscodes läuft und dieser sich eventuell ebenfalls einen physischen Kern teilen muss. Ein weiteres Problem ist die Wahl der Zeitspanne, da eine kleine viel Overhead erzeugen und eine große zu langsam reagieren würde, sodass in der Zwischenzeit bereits Timer-Werte falsch interpretiert würden.
Wie erwähnt steht der Thread des Angriffscodes vor dem gleichen Problem, sodass sich

die Dauer einer Prime-and-Probe Iteration im schlimmsten Fall verdoppeln könnte. Folglich sollte ein Angriff auch mit halber Geschwindigkeit der Prime-and-Probe Operation erfolgreich sein, da ansonsten Cache-Aktivitäten in bestimmten Zeitabschnitten nicht registriert werden.
Andersherum wird es im Laufe des Angriffs passieren, dass sich Opferprogramm und Timer einen physischen Kern teilen. Sofern das Problem der halbierten Timerauflösung bewältigt wird, kann sich die Auflösung des Prime-and-Probe-Angriffs effektiv verdoppeln, da auch das Opferprogramm mit halber Geschwindigkeit läuft. Die Zuordnung der Prozesse zu den virtuellen Kernen im Browser kann nicht beeinflusst werden und erfolgt somit aus Sicht der Angreiferin willkürlich. Deshalb werden nur zufällige Zeitabschnitte besser aufgelöst. Allerdings sollten auch die Kosten gegen gerechnet werden, die für die Erkennung der veränderten Timerauflösung entstehen.
Aufgrund der beschrieben Probleme und der Auswirkungen der Abmilderungen ebendieser ist es besser, auf mindestens drei physische Kerne zurückzugreifen. Sobald die Browserhersteller die Auflösung von *performance.now* wieder in den Nanosekundenbereich erhöhen, wären auch zwei physische Kerne ausreichend.
Wenn das angegriffene Endgerät n virtuelle Kerne besitzt, können also $n - 3$ für die Verlangsamung des Opferprogramms eingesetzt werden.

5.2 Multithread Prime-Spam

Nach den Überlegungen des vorherigen Kapitels sind $n - 1 - k$ Kerne für die Bremsung nutzbar, wobei $k \geq 1$ die Anzahl der Kerne für die eigentliche Überwachung ist. Abschnitt 4.2.2 hat gezeigt, dass ein Thread die Shift- beziehungsweise Subtraktions-Operation nicht ausreichend abbremst. Daher soll im Folgenden analysiert werden, ob das Vorhandensein mehrerer Threads Vorteile beim Bremsen bringen. Der Testrechner hat 4 physische und 8 virtuelle Kerne, und zur besseren Einschätzung des Effekts wird ausschließlich gebremst und nicht gemessen, das heißt es können bis zu 6 Bremsthreads auf die virtuellen Kerne verteilt werden. Zu beachten ist das Problem der Threadaufteilung, wenn mindestens 3 und höchstens 5 Bremsthreads verwendet werden. In diesem Fall werden Threads zeitweise exklusiv auf einem physischen Kern laufen und etwa doppelt so performant sein wie Threads, die sich ihren Kern mit einem anderem Thread teilen müssen. Da die Zuordnung der Threads zu den Kernen im Browser nicht verändert werden kann, ist unklar, welche Threads zu welchem Zeitpunkt alleine laufen.
Daher werden die Laufzeitmessungen über mehrere Sekunden durchgeführt und anschließend gemittelt, damit der Effekt die Ergebnisse nicht verfälscht.
Um den Effekt, der allein durch die Last der zusätzlichen Threads entsteht, einschätzen

zu können, wird ein Test mit Bremsthreads durchgeführt, die beliebige Codezeilen bremsen, welche nicht Teil der Shift- oder Subtraktions-Operation sind. Tabelle 5.1 zeigt, dass ab 4 Bremsthreads ein signifikanter Performancerückgang messbar ist, welcher bei den folgenden Ergebnissen mit einberechnet werden muss.

Tabelle 5.1: Zeigt die Ausführungsgeschwindigkeit bei unterschiedlichen Anzahlen an Bremsthreads, welche nicht die Shift- oder Subtraktions-Funktion bremsen. Spalten „Shift“, „Sub“ und „*mp_gcd*“: Mittlere Dauer der Shift-, Subtraktions-, beziehungsweise *mp_gcd*-Operation in Taktzyklen.

Anzahl der Bremsthreads	Shift	Sub	mp_gcd
0	340	475	430k
1	347	477	443k
2	361	516	467k
3	373	505	467k
4	435	558	546k
5	448	574	556k
6	474	601	598k

In den folgenden Tests findet aufgrund der Vielzahl von Möglichkeiten, die Cache-Lines auf die Threads zu verteilen, eine Beschränkung auf die Shift-Operation statt, welche für das Rekonstruieren der Primzahlen wesentlich ist. Für die Tests wurden die Cache-Lines verwendet, welche in Abschnitt 4.2.2 (siehe Tabelle 4.5 und 4.8) am besten bremsten. Weiterhin werden nur ausgewählte Ergebnisse gezeigt, die die mit dem besten gemessenen Bremsverhalten einschließen. Tabelle 5.2 zeigt die Ergenisse mit zwei Prime-Spam-Threads.
Bei einem Bremsthread wurde im Abschnitt 4.3 festgestellt, dass es besser ist, verschiedene Cache-Lines zu bremsen. Dieser Vorteil tritt bei mehreren Bremsthreads nicht mehr auf, sodass es besser ist, pro Thread nur eine Cache-Line zu bremsen.
Tabelle 5.2 zeigt die Ergenisse mit mehr als zwei Prime-Spam-Threads, wobei hier die in Tabelle 5.1 beschriebenen Effekte auftreten.
Wenn diese Effekte miteinbezogen werden, sind mehr als zwei Bremsthreads nicht nennenswert besser als lediglich zwei. Zudem werden zusätzliche Bremsthreads auch die Messung der Zugriffszeiten verlangsamen, da physische Kerne nicht in ausreichender Zahl vorhanden sind.
Im Ergebnis können zwei Bremsthreads gegenüber einem die Shift-Operation auf 959 statt 677 Taktzyklen bremsen, welches einer Verbesserung von 42% entspricht. Insgesamt kann die Shift-Operation um 182% respektive 619 Taktzyklen gebremst werden. Auf dem Testrechner konnte keine weitere Verlangsamung durch drei Bremsthreads festgestellt wer-

Tabelle 5.2: Zeigt das Bremsverhalten mit zwei Prime-Spam-Threads. Die & in der 1. Spalte trennen die Threads, so werden etwa zwei Threads, bei denen der 1. Thread die Cache-Lines $V1_1$ sowie $V2_4$ und der 2.Thread die Cache-Lines $V1_1$ sowie $V4_3$ bremsen, als $V1_1$,$V2_4$ & $V1_1$,$V4_3$ beschrieben. Spalten „Shift“, „Sub“ und „mp_gcd“: Mittlere Dauer der Shift-, Subtraktions-, beziehungsweise mp_gcd-Operation in Taktzyklen. Für eine Beschreibung der Cache-Line IDs siehe Tabelle 4.5

Aufteilung der Cache-Lines auf die Threads	Shift	Sub	mp_gcd
$V1_1$,$V1_2$,$V2_4$ & $V1_1$,$V1_2$,$V2_4$	730	532	610k
$V1_1$,$V1_2$,$V2_4$ & $V2_53_1$,$V4_3$	795	559	602k
$V1_1$,$V2_4$,$V4_3$ & $V1_2$,$V2_3$,$V4_4$	563	561	530k
$V1_1$ & $V2_4$,$V4_3$	880	673	716k
$V1_1$ & $V1_1$	623	722	665k
$V1_1$,$V2_4$ & $V1_1$,$V4_3$	764	574	600k
$V1_1$,$V2_4$ & $V1_2$,$V4_3$	745	584	590k
$V1_1$ & $V1_2$	569	496	611k
$V4_3$ & $V4_4$	529	519	630k
$V1_1$ & $V4_3$	942	510	797k
$V1_2$ & $V4_3$	928	548	788k
$V1_1$ & $V4_4$	672	521	731k
$V1_1$ & $V2_4$	959	619	807k
$V1_1$ & $V2_53_1$	752	533	667k

den, wobei auf einem Prozessor mit mehr physischen Kernen damit gerechnet werden kann, dass mehr als zwei Threads signifikante Verbesserungen bringen.

5.3 Memory-Locking

Als Memory-Locking soll im Folgenden ein Ansatz bezeichnet werden, welcher die Ausführungsgeschwindigkeit von Prozessen verlangsamt, in dem der Zugriff auf den Hauptspeicher wiederholt kurzzeitig blockiert wird. Beschrieben und verwendet wurde diese Technik in [WXW12, VZRS15, IES17]. Die grundlegende Idee besteht darin, dass die Blockierung des Speicher-Busses erzwungen wird, indem man atomar auf eine Variable schreibt, die in zwei Cache-Lines liegt.
Atomare Operationen verursachen systemweite Blockierungen in den Speicherregionen, auf denen sie arbeiten, wobei diese Blockierungen nicht immer lokal begrenzt sind. Die ersten Generationen von x86-Prozessoren blockierten bei einer atomaren Operation noch den gesamten Speicherbus. Dies sorgt jedoch für Performanceeinbußen, da ein Befehl alle Speicherzugriffe, die etwa aufgrund von Out-of-Order-Execution parallel ausgeführt werden könnten, blockiert. Des Weiteren skaliert dieser Ansatz schlecht mit Mehrkernprozessoren, da andere Kerne während der atomaren Operation auf Speicherzugriffe verzichten

Tabelle 5.3: Zeigt das Bremsverhalten bei mehr als zwei Prime-Spam-Threads. Beschreibung analog zu Tabelle 5.2. Zu beachten ist hier, dass die hohe Anzahl der Threads an sich zu der Verlangsamung der Ausführung beiträgt (siehe Tabelle 5.1).

Aufteilung der Cache-Lines auf die Threads	Shift	Sub	mp_gcd
$V1_1$ & $V2_4$ & $V4_1$	1017	652	903k
$V1_1$ & $V2_4$ & $V4_3$	900	618	927k
$V1_1$ & $V1_2$ & $V2_4$	717	553	709k
$V1_1$ & $V1_2$ & $V2_4$ & $V4_3$	913	755	944k
$V1_1$ & $V1_2$ & $V2_4$ & $V4_3$ & $V4_4$	1026	809	1106k
$V1_1$ & $V1_2$ & $V2_4$ & $V4_1$ & $V4_3$ & $V4_4$	1109	915	1354k

müssten.

Häufig arbeiten atomare Operationen auf Speicherbereichen, die in eine Cache-Line passen. Diese Eigenschaft nutzen x86-Prozessoren ab dem Pentium Pro aus, indem sie nur die zugehörige Cache-Line und nicht mehr den gesamten Speicherbus sperren. Auf diesen Systemen kann es dennoch zur vorübergehenden Sperrung des gesamten Speicherbusses kommen, etwa wenn der Speicher für die atomare Operation nicht aligned ist und sich über zwei Cache-Lines spannt.

x86-Prozessoren ab dem Intel Nehalem und AMD K8/K10 verwenden im Gegensatz zu den vorherigen Generationen keinen gemeinsamen Speicherbus mehr. Stattdessen ist der Speicher aufgeteilt und jedem Kern ein Bereich als lokaler Speicher zugeordnet, auch Non-Uniform Memory Access (NUMA) genannt. Die Kerne besitzen direkten Zugriff auf ihren Speicherbereich, wobei auf die Bereiche der anderen Kerne über einen gemeinsamen Adressraum zugegriffen werden kann. Wenn die Daten einer atomaren Operation innerhalb einer Cache-Line liegen, wird hier analog zu den vorherigen Generationen nur diese Cache-Line gesperrt. Bei einer atomaren Operation, deren Daten sich über zwei Cache-Lines spannen, stimmen sich aufgrund des fehlenden gemeinsamen Speicherbusses alle Prozessoren ab, um ihre aktuell in Ausführung befindlichen Speichertransaktionen zu flushen [WXW12]. Dies entspricht einer Simulation für das Sperren des kompletten Speicherbusses.

Ein solches Verhalten kann nun bewusst provoziert werden, indem atomare Operationen auf Daten, die zwischen zwei Cache-Lines liegen, ausgeführt werden. Pseudocode 16 zeigt exemplarisch die Umsetzung, wobei die dort verwendete atomare Operation einen Pointer auf einem 4-Byte Datentyp erwartet. Der Pseudo-Code zeigt, wie eine Adresse gefunden wird, etwa wenn die Adresse am Ende einer Cache-Line liegt und so die 4 Bytes, auf denen die atomare Operation arbeitet, in zwei Cache-Lines liegen. Wenn eine solche Adresse gefunden wurde, wird die atomare Operation auf dieser Cache-Line endlos wie-

derholt.

Pseudocode 16: Pseudocode für einen Memory-Locking Angriff

```
1 Function Find_Slow_Addresses(size)
2     byte_arr <- malloc(size)
3     for i = 0; i < size - 4; i++ do
4         start <- timeStamp()
5         atomicOperation(get_ptr(byte_arr[i]))
6         if timeStamp() - start > threshold then
7             print("Found slow Operation at"+ i)
8             while true do
9                 atomicOperation(get_ptr(byte_arr[i]))
```

Dadurch werden die Speicherzugriffe aller Programme gebremst, und zwar in der Hoffnung, dass sich das Opferprogramm stärker als das Angriffsprogramm verlangsamt.
Der Angriff wird zuerst in C getestet, da hier leicht eine atomare Operation mit x-86 Instruktionen wie $xchg$ oder $fadd$ erzeugt werden kann. Ein erster Test mit ausschließlich einem oder zwei laufenden Memory-Locking-Threads offenbart, dass die Ausführung der Shift-Operation und Subtraktions-Funktion nicht nennenswert verlangsamt wird (siehe Tabelle 5.4).

Tabelle 5.4: Performance-degradation-attack mittels Prime-Spam und Memory-Locking. 1. Spalte: Anzahl der Threads, die für das Memory-Locking eingesetzt wurden, 2. Spalte: Cache-Lines in der Shift-Operation, die mittels Prime-Spam gebremst wurden (siehe 4.5 für ID-Zuordnung), die Spalten „Shift“, „Sub“ und „mp_gcd“: Mittlere Dauer der Shift-, Subtraktions-, beziehungsweise mp_gcd-Operation in Taktzyklen.

ML-Threads	Cache-Line IDs	Shift	Sub	mp_gcd
0	-	341	480	432k
1	-	370	510	469k
2	-	361	506	455k
0	$V1_1$	539	600	566k
1	$V1_1$	894	527	594k
2	$V1_1$	628	527	526k
0	$V1_1$ & $V2_4$	964	587	865k
1	$V1_1$ & $V2_4$	1337	596	743k
2	$V1_1$ & $V2_4$	1262	575	701k
1	$V1_1$ & $V2_4$ & $V4_3$	1312	680	803k

Das durch die Memory-Locking-Threads alleine keine Verlangsamung eintritt, kann damit erklärt werden, dass der Code für die Shift- und Subtraktions-Operationen komplett in den Cache passt und so Speicherzugriffe vermieden werden können.

Interessanter ist daher die Kombination mit einem Prime-Spam-Angriff (siehe Abschnitt 4.2.2 und 5.2), welcher Codeteile der Shift- und Subtraktions-Operation aus dem L3-Cache verdrängt, sodass diese bei der Ausführung wiederholt aus dem Hauptspeicher geladen werden müssen. So verlangsamt die Kombination eines Memory-Locking-Threads mit einem Prime-Spam-Thread auf die Cache-Line $V1_1$ die Shift-Operation auf 894 Taktzyklen (siehe Tabelle 5.4). Wenn der Memory-Locking-Thread zusammen mit zwei Prime-Spam-Threads auf die Cache-Lines $V1_1$ und $V2_4$ (eine der besten Kombinationen nach Abschnitt 5.2) genutzt wird, verlangsamt sich die Shift-Operation im Mittel auf 1337 Taktzyklen. Dies ist besser als alle Ergebnisse, die mit einem Memory-Locking-Thread und drei Prime-Spam-Threads sowie mit ausschließlich drei Prime-Spam-Threads (siehe Abschnitt 5.2) erzielt werden konnten. Zusätzlich ist zu bemerken, dass durch den Memory-Locking-Angriff die Ausführungszeit der Shift-Operation sehr stark schwankt. So sind bei im Mittel 1337 Taktzyklen einige Operationen mit $\sim$3000 Taktzyklen zu beobachten. Diese entstehen, wenn Codeteile der Shift-Operation gerade durch den Prime-Spam-Thread aus dem Cache verdrängt wurden und der Ladevorgang des Codes aus dem Hauptspeicher mit der atomaren Operation zusammenfällt.

Somit kann festgehalten werden, dass eine Kombination von Prime-Spam und Memory-Locking im Mittel besser bremst als eine der beiden Techniken allein. Jedoch hat diese Technik negative Auswirkungen auf die Prime-and-Probe-Operation, welche in der Grafik 9 visualisiert wird. Unten sind die Zugriffszeiten eines nicht aktiven Cache-Sets ohne Memory-Locking-Angriff und oben mit aktivem Memory-Locking-Angriff zu sehen. Mit aktivem Memory-Locking-Angriff zeigen sich in regelmäßigen Abständen hohe Zugriffszeiten, die durch die notwendigen Speicherzugriffe während der Prime-and-Probe-Operation entstehen.

Das Muster der erhöhten Zugriffszeiten ist zu unregelmäßig, um es mit einfachen Mitteln zu filtern. Des Weiteren sind erhöhte Zugriffszeiten, die durch das Opferprogramm entstehen, ebenfalls im Bereich von 250 bis 350 Zeiteinheiten. Es kann aber bei Zugriffszeiten von über 400 davon ausgegangen werden, dass bei diesem Sample auch das Opferprogramm für die erhöhte Zugriffszeit verantwortlich ist, da der Memory-Locking-Angriff allein keine Zugriffszeiten von über 400 verursacht. Problematisch sind allerdings Zugriffszeiten von etwa 300, bei denen nicht mehr unterschieden werden kann, ob der Ausschlag vom Memory-Locking-Angriff oder von Zugriffen des Opferprogramms verursacht wurde.

Die Frequenz der Störungen kann verringert werden, indem der Memory-Locking-Angriff verlangsamt wird, etwa durch das Hinzufügen einer Sleep-Operation in der Schleife der Zeile 8, wobei im Umkehrschluss allerdings auch die Speicherzugriffe weniger häufig gebremst werden.

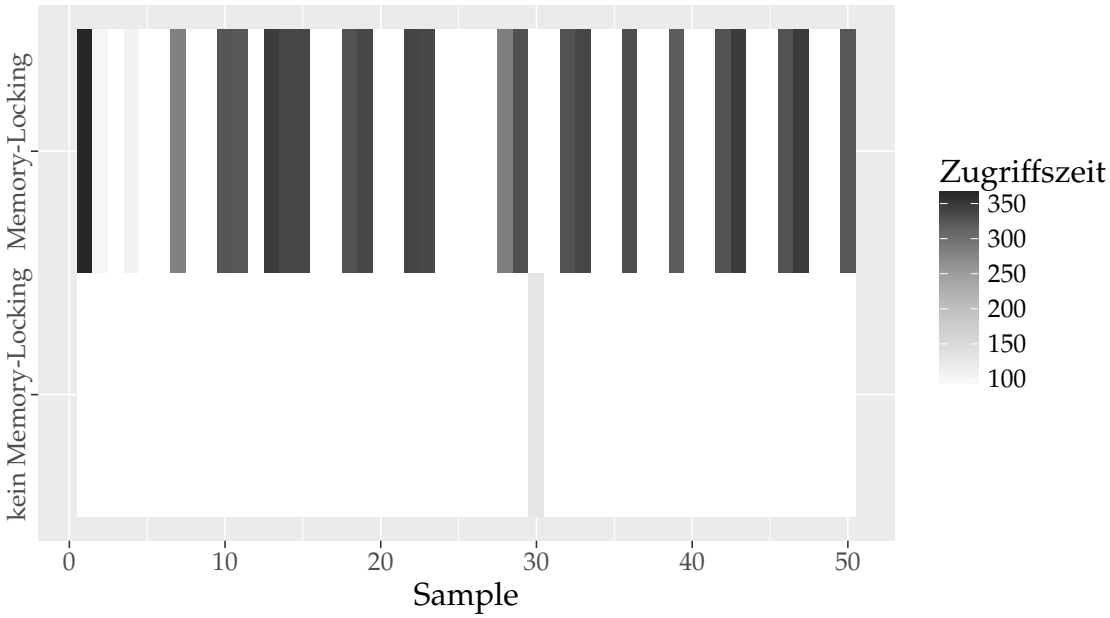

Abbildung 5.1: 50 Messungen der Zugriffszeit mittels Prime-and-Probe auf einem Cache-Set ohne Aktivität. Der obere Teil der Abbildung ist mit, der untere ohne Memory-Locking-Angriff entstanden. Im oberen Teil ist zu sehen, wie der Memory-Locking-Angriff Speicherzugriffe verzögert.

Zusammengefassend kann gesagt werden, dass der Memory-Locking-Angriff die Speicherzugriffe in regelmäßigen Abständen stark verlangsamt, jedoch für einen Prime-and-Probe-Angriff ungeeignet scheint. Ursächlich dafür ist der Einfluss auf den Messvorgang, welcher in gleichen Abständen stark verlangsamt wird, und so hohe Zugriffszeiten in vielen Fällen nicht eindeutig zugeordnet werden können. Zudem verlangsamt der Angriff die Shift-Operation nur, wenn deren Code aus dem Cache verdrängt wurde. Dies wird hier durch die Prime-Spam-Methode forciert, welche ebenfalls durch den Memory-Locking-Angriff gebremst wird.

5.4 Benchmarks mit künstlicher Bremsung

Abschnitt 5.2 und 5.3 haben gezeigt, dass Prim-Spam und Memory-Locking auch mit mehreren Threads nicht ausreichen, um einen Teil der Primzahlen zu rekonstruieren.
Im Folgenden soll daher untersucht werden, wie viel mehr gebremst werden muss, um eine Rekonstruktion der Primzahlen zu ermöglichen. Dazu wird nach jeder Shift- und Subtraktions-Operation eine künstliche Pause von x-Taktzyklen eingefügt.
Diese Methode sorgt im Gegensatz zu den Bremsverfahren mittels Prim-Spam und Memory-Locking für gleichmäßig erhöhte Ausführungszeiten der Shift-Operation. In

der Praxis können die angegebenen Ausführungszeiten daher als untere Schranke für einen erfolgreichen Angriff angenommen werden. In Grafik 5.4 wird die Überwachung einer Folge von Shift-Operationen bei verschiedenen Pausenzeiten visualisiert.

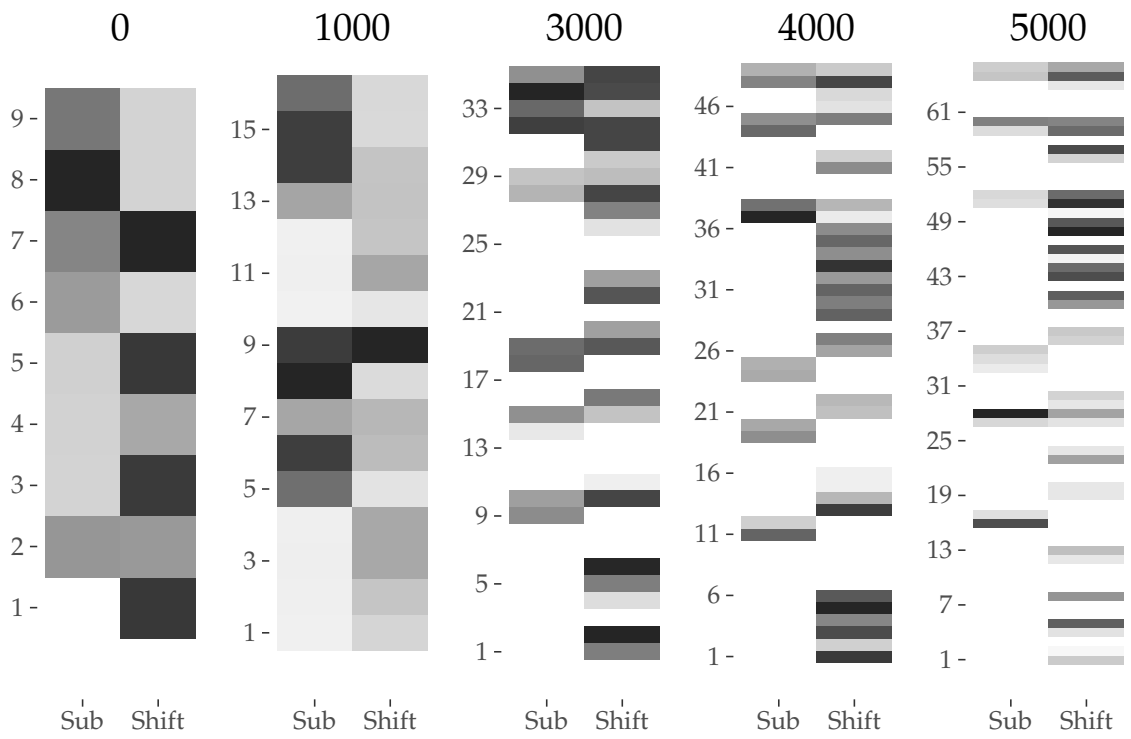

Abbildung 5.2: Zu sehen ist die Überwachung der Shift-Folge $Z_1 = 4, Z_2 = 2, Z_3 = 1, Z_4 = 5, Z_6 = 2, Z_7 = 1$ mit Z_i als Anzahl der Rechts-Shift-Operationen in Iteration i (siehe auch Abschnitt 4.2.1). Die Zahlen in der Kopfzeile geben die Pausen in Taktzyklen an, welche nach den Shift- und Subtraktions-Operationen durchgeführt wurden. Für den Messvorgang wurde ein Thread verwendet. Für die Messungen für die Prime-and-Probe-Methode wurde „Wasm Split“ genutzt, bei der zwei Pointer-Ketten mit jeweils acht Einträgen anstatt einer mit 16 verwendet werden (siehe Abschnitt 5.7.2).

Ganz links ist die Messung ohne Bremsung zu sehen, wobei keine einzelnen Shift-Operationen ausgemacht werden können. Auch lange Shift-Operationsfolgen wie $Z_1 = 4$ sind schwer identifizierbar, da sie zusammen mit anderen Shift-Operationen in einem Sample abgebildet werden.

Mit 1000 Taktzyklen Bremszeit sind hingegen lange Operationsfolgen von kurzen gut unterscheidbar, wie Sample 1-4 ($Z_1 = 4$) und Sample 10-13 ($Z_4 = 5$) gegenüber Sample 5-9 ($Z_2 = 2, Z_3 = 1$) zeigen. Diese künstliche Bremsung mit 1000 Taktzyklen ist besser, als die ungebremste Version mit allen erdenklichen Varianten, welche in dieser Arbeit vorgestellt werden. Das schließt eine Messung mit 2 Threads statt einem, zusätzliche Prime-

Spam-Bremsthreads, einem zusätzlichen Memory-Locking-Angriff und eine verbesserte Prime-and-Probe-Operation mit ein.
Ab 3000 Taktzyklen Bremszeit können einzelne Shift-Operationen ausgemacht werden, wobei insbesondere der Abstand zwischen den Subtraktions-Operationen viel über die Anzahl der dazwischen liegenden Shift-Operationen verrät. Jedoch ist dies fehleranfällig, da etwa der Unterschied in den Shift-Operationen bei $Z_1 = 4$ in Sample 1 bis 9 und $Z_4 = 5$ in Sample 20 bis 27 nicht auszumachen ist.
Bei einer Bremszeit von 5000 Taktzyklen können die einzelnen Shift-Operationen gut unterschieden werden, wobei bei langen Shift-Folgen die Länge nicht immer exakt bestimmt werden kann, wie $Z_4 = 5$ in Sample 35 bis 51 veranschaulicht.
Die Ergebnisse zeigen, dass eine mittlere Shift-Operationsdauer von etwa 3000 bis 4000 Taktzyklen notwendig ist, um einzelne Shift-Operationen voneinander trennen zu können und so eine Bestimmung der Z_i für eine Rekonstruktion der Primzahlen zu ermöglichen. Durch die Verwendung eines weiteren Threads für die Messung kann die benötigte Zeit etwa halbiert werden, jedoch steht dann auf dem Testsystem kein physischer Kern mehr zum Bremsen zur Verfügung (siehe Abschnitt 5.1).

5.5 Speicherallokation

Wenn der Speicher, etwa nach dem Booten, wenig fragmentiert ist und große zusammenhängende physische Speicherbereiche frei sind, scheint Ubuntu 16.04.5 LTS das $evictionBuffer$-Array bevorzugt in diese Bereiche zu mappen. Angenommen der komplette evictionBuffer wird in einen zusammenhängeden physischen Speicherbereich gemappt. Dann ist zu erwarten, dass nach jeweils 2^8 Speicherblöcken eine neue colliding-address gefunden wird, da 2^8 Speicherblöcke genau 2^{20} Bytes belegen und bei einer colliding-address die letzten 20 Bits identisch sind.
Die ersten Versuche hierzu fanden statt, nachdem der Testrechner bereits über 30 Minuten in Betrieb war und viele Tests und Programme ausgeführt worden waren. Wenn man unter diesen Bedingungen den Abstand der ersten 10 Speicherblöcke, welche eine colliding-address zum Speicherblock 0 bilden, zueinander anschaut, zeigen sich Abstandsfolgen wie

$$654, 938, 1438, 224, 372, 543, 464, 84, 38, 134$$
$$170, 236, 573, 66, 452, 124, 206, 228, 1026, 62$$

Diese schwanken auch nach näherer Betrachtung rein willkürlich um den Erwartungswert 256.

Interessant werden die Folgen, wenn der Testrechner neu gestartet wird und der physische Adressbereich somit wenig fragmentiert ist. Dann können Abstandsfolgen wie

$$256, 256, 256, 353, 256, 256, 258, 254, 256, 188$$
$$256, 256, 256, 256, 256, 256, 271, 256, 256, 256$$

beobachtet werden. Bei den gegebenen Beispielen sind vereinzelt Abstände ungleich 256 zu beobachten, die scheinbar nicht aus einem Fehler bei der colliding-address Erkennung resultieren.
Denn angenommen, die Abweichungen stammen aus vereinzelten Fehlern bei der Erkennung, dann sollten Abstandsfolgen wie 256, 353, 159, 256 entstehen mit $353+159 = 2*256$. Die beobachteten Folgen jedoch weisen nach der vereinzelten Abweichung häufig wieder den Abstand 256 auf, sodass die Abweichung kein Fehler in der Erkennung zu sein scheint.
Ein weiteres Indiz dafür liefert die erfolgreiche Verifikation, dass der Speicherblock mit dem abweichenden Abstand später Teil eines Eviction-Sets wird.
Die längste ununterbrochene Folge mit dem optimalen Abstand 256 war bei allen Tests immer auf den Wert 10 beschränkt, wobei etwa 60-80% aller Abstände den optimalen Wert 256 aufweisen. Somit scheint die Größe zusammenhängender physischer Speicherbereiche aus dem Autor nicht bekannten Gründen begrenzt zu sein.
Wenn der Speicher wenig fragmentiert ist, kann diese Eigenschaft ausgenutzt werden um die Suche nach colliding-addresses zu beschleunigen. Dies würde den neuen Eviction-Set-Suchalgorithmus weiter verbessern, da bei optimaler Parameterwahl der Anteil der colliding-address-Suche 75% der gesamten Berechnungszeit beträgt (siehe letzte Zeile der Tabelle 3.4). So würde die Suche nach einer colliding-address zuerst für Blöcke im Abstand von 256 ausprobiert werden und im Fehlerfall alle dazwischen liegenden Blöcke untersucht.

5.6 Warum Google präzise Timer im Browser reaktiviert

Wie erwähnt hat Google bereits Anfang August 2018 mit der Chrome Version 68 [V818] die SharedArrayBuffer standardmäßig wieder aktiviert. Mozilla hingegen hat auch in der aktuellen Firefox Developer Edition 63.0b8 (Stand Ende September 2018) die SharedArrayBuffer in der Standardeinstellung deaktiviert.
Google begründet seinen Schritt mit der Einführung der Site Isolation [Chr18], welche jeden Renderprozess auf Dokumente einer Webseite beschränkt. Somit ist der Inhalt jeder Webseite in einem eigenen Prozess gekapselt, und Chrome kann auf die Sicherheitsmaßnahmen des Betriebssystem zurückgreifen. Chrome besaß bereits vor der Version 68 eine

Multiprozessarchitektur, in der Tabs auf unterschiedliche Prozesse aufgeteilt wurden. Jedoch war es vorher möglich, dass eine vertrauenswürdige Webseite und ein darin enthaltender Iframe mit Inhalten einer bösartigen Webseite im selben Renderprozess landeten. In diesem Fall wäre ein Spectre-Angriff möglich, in welchem Daten wie Cookies oder Passwörter von der vertrauenswürdigen Seite abgeschöpft werden könnten. Der Unterschied lässt sich in der Praxis leicht anhand einer typischen mit Drittanbieterwerbung versehenen Webseite wie Spiegel online nachvollziehen. Chromium 67.0.3396.0 startet einen zusätzlichen Prozess, Chromium 68.0.3440.0 hingegen 6 weitere Prozesse, um Spiegel Online mit all seiner Werbung in einem neuen Tab darzustellen.
Mozilla arbeitet ebenfalls an Site-Isolation, wird aber vermutlich noch Monate brauchen, um dieses Feature in die Release-Version von Firefox zu integrieren [Fir18]. Die aktuelle Firefox Developer Edition 63.0b8 teilt in der Standardeinstellung alle Webseiteninhalte auf vier Content-Prozesse auf. Somit kann - wie oben beschrieben - mittels eines Spectre-Angriffs auf alle Webseitendaten des Content-Prozesses zugegriffen werden, in dem die Webseite mit dem Angriffscode liegt. Eigene Tests haben ergeben, dass Firefox die Tabs nach dem Round-Robin-Verfahren auf die Content-Prozesse aufteilt, womit eine erfolgreiche Angreiferin die Webseitendaten von einem Viertel der offenen Tabs abgreifen könnte. Aus diesem Grund wird Mozilla in Firefox vermutlich erst nach der Einführung der Site Isolation Änderungen bezüglich der Verfügbarkeit hochpräsizer Timer vornehmen.
Auch Spectre

5.7 Prime-and-Probe Optimierungen

In diesem Abschnitt sollen mögliche Optimierungen der Prime-and-Probe-Operation vorgestellt und evaluiert werden. Dazu wird einerseits die Implementierung der Prime-and-Probe-Operation in Webassembly mit der in Javascript verglichen und andererseits wird der Code an sich optimiert.

5.7.1 Javascript vs. Webassembly

Wie in Kapitel 3 beschrieben, ist die Prime-and-Probe-Operation in Webassembly geschrieben. Der Ablauf einer solchen Prime-and-Probe-Operation wird in Pseudocode 17 beschrieben.
Zurzeit ist Webassembly nicht multithreadingfähig [Web18a], das heißt, aus dem Webassembly-Kontext kann nicht direkt auf die Zählvariable des SharedArrayBuffers zugegriffen werden. In Folge dessen muss der Wert der Zählvariable über Javascript abgerufen werden. Um Javascript-Code aus Webassembly aufzurufen, existieren mehrere Möglichkeiten [Cal18], wobei sich die Implementierung einer C API in JavaScript am performan-

Pseudocode 17: Pseudocode für die Prime-and-Probe-Operation in Webassembly

```
1 Function Prime_And_Probe(ptr_eviction_set)
2     start <- get_counter_value_via_javascript()
3     first_entry <- ptr_eviction_set
4     repeat
5         ptr_eviction_set <- *ptr_eviction_set
6     until ptr_eviction_set ≠ first_entry
7     return get_counter_value_via_javascript() - start
```

testen herausgestellt hat. Alternativ kann auch die gesamte Prime-and-Probe-Operation in Javascript implementiert werden, wobei der Wert der Zählvariable ohne Umwege abgefragt werden kann.

Um die Performance der beiden Implementierungen zu vergleichen, wird die mittlere Dauer einer Prime-and-Probe-Operation durch die Ausführung von 10 Millionen Operationen approximiert, wobei dies einem Messvorgang von wenigen Sekunden entspricht. So dauert eine Prime-and-Probe-Operation in Webassembly bei keiner Aktivität auf dem Cache-Set etwa 190 ns, wobei diese Zeit auf 440 ns ansteigt, wenn die Cache-Line $V2_53_1$ (siehe Tabelle 4.5) während der Ausführung der mp_gcd-Funktion überwacht wird.

In der Javascript-Implementierung verringert sich die Zeit bei keiner Aktivität auf dem Cache-Set um etwa 20 ns auf etwa 170 ns, wobei die benötigte Zeit bei einem aktiven Cache-Set um etwa 40 ns auf 400 ns gegenüber der Webassembly-Implementierung sinkt (siehe auch Abbildung 7).

Somit lassen sich mit der Umsetzung der Prime-and-Probe-Operation in Javascript $\sim 10\%$ mehr Prime-and-Probe-Operationen in der gleichen Zeit realisieren. Die ersten ca. 2000 gemessenen Zugriffswerte in Javascript neigen zu starkem Rauschen, so sind auch bei keiner Aktivität auf dem Cache-Set reproduzierbar mehr als hundert mal hintereinander Zugriffszeiten von über 400 Zeiteinheiten zu beobachten. Eine mögliche Begründung für diese Beobachtung wäre, dass der Browser in dieser Phase versucht den Javascript-Code zu optimieren [Hab15], da er wegen der häufigen wiederholten Ausführung als wichtig erkannt wurde. Dieses anfängliche Rauschen der Javascript-Implementierung ist in der Praxis jedoch vernachlässigbar, da die Messphasen in der Regel deutlich länger als 2000 Prime-and-Probe-Operationen andauern und somit die anfänglichen verrauschten Werte verworfen werden können.

Zusammenfassend betrachtet, bringt die Implementierung in Javascript einen Performance-Vorteil von ca. 10% ohne in der Praxis relevante Nachteile zu bieten. Da die Umsetzung von Multithreading in Webassembly geplant ist [Web18a], kann bei dieser Implementierung in Zukunft mit einem Performancegewinn gerechnet werden, da dann die Javascript-Funktionsaufrufe bezüglich der Zählvariable entfallen.

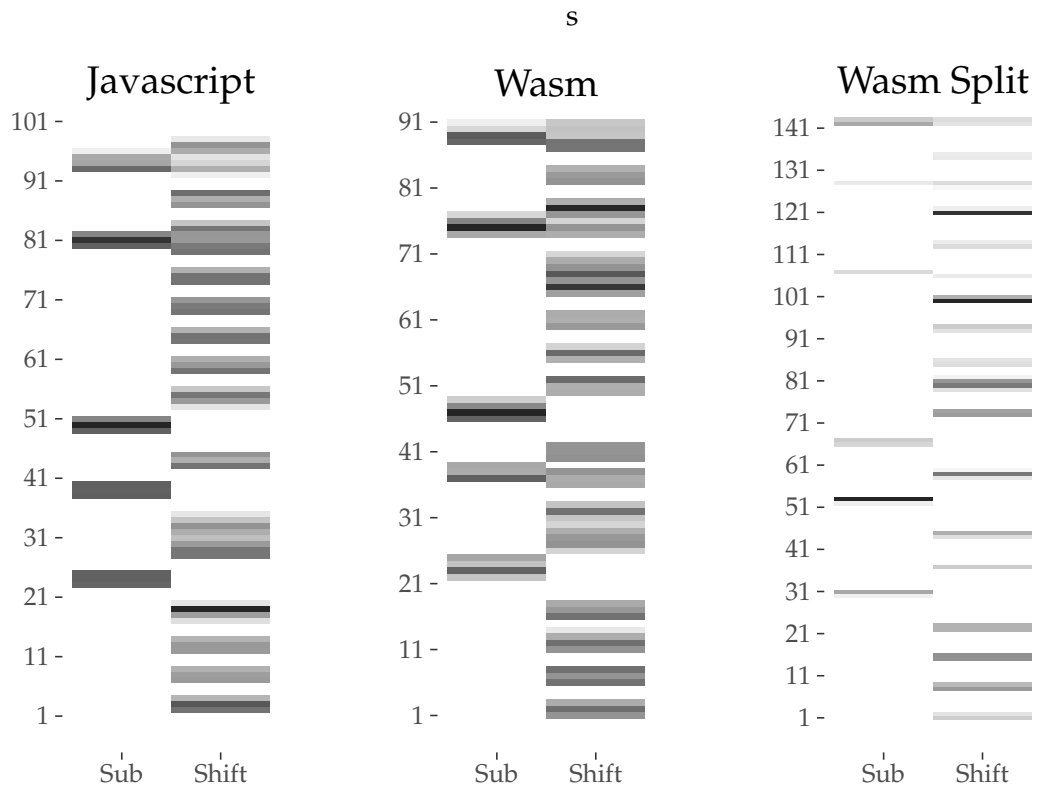

Abbildung 5.3: Zu sehen ist die Überwachung der Shift-Folge $Z_1 = 4, Z_2 = 2, Z_3 = 1, Z_4 = 5, Z_6 = 2, Z_7 = 1$ mit Z_i als Anzahl der Rechts-Shift-Operationen in Iteration i analog zu Abbildung 5.4. Es wurde eine Pause von 10000 Taktzyklen zwischen den Shift- und Subtraktions-Operationen analog zu Abschnitt 5.4 eingefügt. „Javascript“ und „Wasm“ bilden die Performance der Javascript- beziehungsweise Webassembly-Implementierung ab (siehe Abschnitt 5.7.1). „Wasm Split“ nutzt zwei Pointer-Ketten mit jeweils acht Einträgen anstatt einer mit 16 (siehe Abschnitt 5.7.2).

5.7.2 Verschiedene Varianten von Prime-and-Probe

Die Standardimplementierung für die Prime-and-Probe-Operation verwendet die Point-Chasing-Methode (siehe Pseudocode 8). In dieser wird einer Kette von 16 Einträgen gefolgt, wobei der Prozessor immer erst die Adresse des vorherigen Eintrags bestimmen muss, um die des nächsten Eintrags laden zu können. Um eine Prime-and-Probe Operation zu beschleunigen wären zwei Ketten mit jeweils acht Einträgen, beziehungsweise vier Ketten mit jeweils vier Einträgen denkbar. Dadurch würde eine Parallelisierung auf Instruktionsebene möglich und der Prozessor könnte weiterhin keine Folgeadressen innerhalb der Ketten im Voraus laden.

Anhand der unterschiedlichen Sampleanzahl für die gleiche Shift-Folge in Abbildung 7 ist erkennbar, dass sich durch die Aufspaltung in zwei Ketten mit jeweils acht Einträgen ein

Performancevorteil von etwa 50% ergibt. Mit einer Aufteilung auf vier Ketten mit jeweils vier Einträgen konnten hingegen keine erfolgreichen Messungen durchgeführt werden.
Die Aufteilung in zwei Ketten kann auch bei der Prime-Spam-Bremsmethode eingesetzt werden, um dort die Bremsleistung zu erhöhen. Wenn die Cache-Lines $V1_1$ und $V2_4$ (siehe Tabelle 4.5) mittels Prime-Spam in einem Thread und der Prime-and-Probe-Operation mit einer Kette gebremst wurden, konnte die Shift-Operation von ca. 350 auf 420 Taktzyklen verlangsamt werden. Unter den gleichen Bedingungen, aber einer Prime-and-Probe-Operation mit zwei Ketten, konnte die Shift-Operation auf etwa 565 Taktzyklen gebremst werden. Damit kann bei der Prime-Spam-Methode ein Performance-Vorteil von ca. 35% erzielt werden. Angemerkt sei, dass der Code um die Bremsfunktionen in diesen Tests nicht optimiert war, weswegen die absoluten Werte nicht mit den Bremsergebnissen aus Abschnitt 4.3 und 5.2 verglichen werden können.
Hiermit steigert die Aufteilung in zwei Ketten die Performance der Prime-and-Probe-Operation merklich und ist der Implementierung mit einer Kette vorzuziehen.

6 Schlussbetrachtung

In diesem Kapitel werden die wesentlichen Ergebnisse dieser Arbeit zusammengefasst und die gewonnenen Erkenntnisse überprüft. Abschließend wird ein Ausblick in die Zukunft der Cache-Angriffe im Allgemeinen sowie Browserangriffe im Speziellen gewährt.

6.1 Bewertung

Die Ergebnisse zeigen, dass L3-Cache-Angriffe im Browser trotz des verstärkten Problembewusstseins im Zuge von Meltdown und Spectre wieder möglich sind. Im Vergleich zur nativen Variante ist jedoch die Ausführung des Angriffs langsamer und weiterhin sind die Möglichkeiten zum Bremsen des Opferprozesses beschränkt.
Um dem zu begegnen, können mehrere Angriffsinstanzen gestartet werden. Zum einen kann die Überwachung aufgeteilt werden, wobei die Synchronisation mit den Zeitstempeln möglich ist. Zum anderen sind mehrere Bremsinstanzen für eine verbesserte Reduktion der Performance nutzbar. Hierdurch steigt jedoch die Systemauslastung an und damit die Gefahr, dass der Angriff vom Opfer entdeckt wird.
Der Angriff nutzt die spezifische Eigenschaft der Intel-Architektur aus, die den L3-Cache zwischen den Kernen teilt und inklusiv ist, sowie LRU oder eine vergleichbare Eviction-Policy verwendet. Auf Prozessoren des Konkurrenten AMD ist er beispielsweise nicht lauffähig.
Ein großer Vorteil des Angriffs ist, dass er wenig Spuren nach der Ausführung hinterlässt. Trotz der Möglichkeit des Angriffs aus dem Browser heraus ist dieser in der Praxis aufwendig, da er speziell auf den Code einer Softwareversion abgestimmt ist, und diese zum Zeitpunkt des Angriffs laufen und die gewünschten Berechnungen ausführen muss. Zudem ist ein Erfolg nicht garantiert, der native Angriff auf die RSA-Keygenerierung [CAPGATBB18] weist beispielsweise nur eine Erfolgsquote von etwa 28% auf.
Diesen Einschränkungen unterliegt auch die Leakage mit den Unklarheiten in Bezug auf die praktische Anwendung bei der RSA-Primzahlgenerierung (siehe Abschnitt 4.1.2 und 4.1.4), wobei etwa folgendes Szenario denkbar ist: Die E-Mail-Zertifikate der neuen Studierenden der Universität zu Lübeck werden zum Beginn des Wintersemesters in einem Einführungskurs lokal erzeugt. Eine Angreiferin könnte versuchen, die Website zu manipulieren, welche die Zertifikatsgenerierung beschreibt. Da der Zeitraum, in dem die Zertifikatsgenerierung stattfindet, bekannt sowie eingegrenzt ist und es viele potenzielle

Opfer gibt, könnte sich der Angriff hier auszahlen.

6.2 Ausblick

Die fehlenden Möglichkeiten, Code ausreichend zu bremsen, verhindern zurzeit eine erfolgreiche Portierung der RSA-Schlüsselgenerierungs-Leakage (siehe Abschnitt 4.2). Eine offene Frage ist daher, ob im Webkontext bessere Optionen zum Bremsen von Prozessen im Allgemeinen und spezifischen Codeteilen im Speziellen existieren.
Da in Zukunft vermutlich kein eigener Count-Thread mehr nötig ist (siehe Abschnitt 5.6), sinkt der Ressourcenbedarf des Angriffs um einen Thread. So kann ein Angriff entweder mit einem Thread weniger auskommen oder der frei gewordene Thread wird zum Bremsen des Codes verwendet.
Zudem könnte der neue Eviction-Set-Suchalgorithmus weiter beschleunigt werden, indem die Suche nach den colliding-addresses und die Eviction-Set-Suche im Pool der colliding-addresses nebenläufig abgearbeitet werden. Javascript kann Multithreading durch Webworker umsetzen, wobei eine komplette Reimplementierung der Eviction-Set-Suche in Javascript nötig wäre. Weiterhin müsste evaluiert werden, ob diese Nachteile gegenüber der Implementierung in Webassembly mit sich bringt. Andererseits wird zurzeit an der Umsetzung von Multithreading in Webassembly gearbeitet [Web18a], womit der vorhandene Code in Zukunft um den Parallelisierungsansatz erweitert werden könnte.
Wenn der Angriffsprozess auf demselben physischen Kern wie der Opferprozess läuft, ergeben sich weitergehende Angriffsmöglichkeiten. So kann beispielsweise eine erhöhte räumliche Auflösung ermöglicht werden, welche über die übliche 64 Byte Cache-Line-Auflösung hinausgeht [MES17]. Ein weiterer im Juli 2018 veröffentlichter Angriff [BG18] nutzt konkurrierende Zugriffe auf den Translation-Lookaside-Buffer (TLB) aus, welcher zwischen zwei Hyperthreads eines selben physischen Kerns geteilt ist. Im Zuge der Veröffentlichung wurde im Betriebssystem OpenBSD Hyperthreading auf allen Intel-Prozessoren standardmäßig deaktiviert [Ket18].
Wie schon in vorherigen Abschnitten erwähnt wurde, kann im Browser nicht ohne Weiteres ein Kern festgelegt werden, auf dem der Angriffscode ausgeführt wird. Als mögliche Lösung könnte die Angreiferin $n-1$ Instanzen ihres Angriffs starten, wobei n der Anzahl der virtuellen Kerne entspricht. Somit würde durchgehend eine Instanz auf demselben physischen Kern wie dem des Opferprogramms laufen. Allerdings ist der ausführende Kern nicht während der gesamten Ausführungszeit fix, sodass verschiedene Instanzen zu unterschiedlichen Zeiten den Angriff ausführen müssten. Somit ist die Auslotung eines solchen Angriffs und die Suche nach Möglichkeiten, die Wahl des ausführenden Kerns zu

beeinflussen, lohnenswert, da eine neue Klasse von nativen Angriffen im Browser realisierbar würde.
Im Hinblick auf die Ersetzung des Desktop-Computers oder Notebooks durch Smartphone und Tablet ist eine nähere Untersuchung der Mikroarchitekturen dieser Geräte interessant. Da diese Geräte dieselben Webtechnologien unterstützen, welche auch in dieser Arbeit zum Einsatz kamen, sind die Voraussetzungen vergleichbar. Problematisch ist jedoch die zu Intel-Prozessoren unterschiedliche Struktur und Eviction-Policy der Caches [GRLZ+17]. Außerdem bietet der Markt mobiler Geräte im Vergleich zum Desktop- und Notebook-Bereich zahlreiche Prozessorenhersteller mit signifikantem Marktanteil, welche unterschiedliche Mikroarchitekturen verwenden. Zwar sind die meisten Prozessoren ARM-kompatibel, jedoch weisen die Performanceunterschiede zwischen den Prozessoren [Gee18] auf deutlich abweichende Designs hin.

6.2.1 Gegenmaßnahmen

Die Veröffentlichung von Meltdown und Spectre Anfang 2018 hat ein neues Bewusstsein für die Relevanz von Cache-Angriffen geschaffen. In der Folge haben die Browserhersteller alle öffentlich bekannten Optionen für hochauflösende Timer unterbunden. Google ist der erste Hersteller, welcher seit Chrome Version 68 in der Standardeinstellung wieder hochauflösende Timer erlaubt. Dieser Schritt wird von Google mit der Einführung der Page-Isolation-Technik [Chr18] begründet, die eine Aufspaltung von verschiedenen Webseiten in unterschiedliche Prozesse sicherstellt und somit hinreichend vor den von Meltdown und Spectre aufgezeigten Gefahren schützen soll (siehe auch Abschnitt 5.6). Gegen Angriffe auf den geteilten L3-Cache, welche in dieser Arbeit diskutiert wurden, hilft diese Technik allerdings nicht. Somit werden wieder Angriffe ermöglicht, die es nötig machen, dass Softwareentwickler ihre Implementierungen gegen diese Angriffe schützen.
Um den Code gegen Angriffe zu schützen, müssen Algorithmen mit konstanter Ausführungszeit (constant-time) und ohne eingabeabhängige Verzweigungen und Speicherzugriffe implementiert werden. Methoden für die Entwicklung solcher Algorithmen wurden bereits vorgestellt [BLS12]. Jedoch ist die korrekte Anwendung schwierig, wie etwa der Angriff auf die constant-time RSA-Implementierung von OpenSSL zeigt [YGH17b]. Im Webkontext kommt hinzu, dass der Javascript-Compiler viele Freiheiten besitzt und somit neue Leakages schaffen kann [GPTY18]. Das Ausführungsverhalten von Javascript-basierten Kryptoimplementierungen wie OpenPGP.js kann sich also je nach Browser ändern und so zu unvorhergesehenen Leakages führen. Daher ist eine Schnittstelle wie die WebCrypto API [Web18b] zu bevorzugen, da diese Kryptofunktionen nativ im Browser, der unter gleichbleibenden Bedingungen kompiliert wurde, ausführt. Alternativ könnte an einem deterministischen Ausführungsverhalten in Javascript geforscht werden, wobei

auch Spracherweiterungen denkbar wären.
Aber auch Anwender können Maßnahmen ergreifen, um sich vor Angriffen im Browser zu schützen. Wenn genaue Zeitquellen nicht benötigt werden, können ebenfalls speziell gehärtete Browser wie Fuzzyfox [KS16] Verwendung finden, welche die Auflösung der möglichen Zeitquellen im Browser herabsetzen [SMGM17]. Zudem steigt bei vielen Systemen die Lüfterdrehzehl hörbar an, wenn eine erhöhte Systemlast vorliegt. Der hier vorgestellte Prime-and-Probe-Angriff lastet bei einem Angriff mindestens zwei Kerne komplett aus und ist auf dem verwendeten Desktop-Testsystem akustisch detektierbar. Da der Angriff Javascript benötigt und in einem praktischen Szenario vorzugsweise über Werbeanzeigen ausgeführt wird, stellen die ohnehin empfehlenswerten Ad- und Javascriptblocker eine effektive Gegenmaßnahme dar.
Das eigentliche Problem, der geteilte und inklusive L3-Cache mit LRU oder einer vergleichbaren Eviction-Policy, wird in Intel-Prozessoren noch Jahre Bestand haben. Zurzeit nutzen im Desktopbereich nur die High-End-Prozessoren in Form von Skylake-X einen nicht inklusiven L3-Cache [Cut18]. Im Mainstream werden nicht exklusive L3-Caches eventuell mit der neuen Architektur Ice Lake eingeführt, welche frühestens Ende 2019 erscheinen wird [Gü18]. Dies legt zumindest die angekündigte Verdoppelung des L2-Caches pro Kern bei gleichbleibender L3-Cache-Größe nahe. Die Prozessoren des Konkurrenten AMD hingegen sind aufgrund der exklusiven Caches [Cut17a] von diesen Angriffen nicht betroffen.
Cache-Angriffe im Browser sind allgegenwärtig und ihnen sollte von Softwareseite mit den genannten Gegenmaßnahmen begegnet werden, da Websprachen der Angreiferin die komplette Kontrolle über den Code überlassen und hardwareseitige Lösungen erst in einigen Jahren flächendeckend verfügbar sein werden.

Literaturverzeichnis

[ASSS17] Alejandro Cabrera Aldaya, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. Spa vulnerabilities of the binary extended euclidean algorithm. *Journal of Cryptographic Engineering*, 7(4):273–285, Nov 2017.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[BG18] Herbert Bos Cristiano Giuffrida Ben Gras, Kaveh Razavi. Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks. 2018.

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, pages 159–176, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Cal18] Interacting with code. https://kripken.github.io/emscripten-site/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html, 2018. Accessed: 2018-10-01.

[CAPGATBB18] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on rsa key generation. Apr 2018.

[Car02] Carlos Carvalho. The gap between processor and memory speeds. https://pdfs.semanticscholar.org/6ebe/c8701893a6770eb0e19a0d4a732852c86256.pdf, 2002.

[Chr18] Site isolation. https://www.chromium.org/Home/chromium-security/site-isolation, 2018. Accessed: 2018-09-24.

[Cop96] Don Coppersmith. Finding a small root of a univariate modular equation. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 155–165, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[Cut17a] Ian Cutress. The amd zen and ryzen 7 review: A deep dive on 1800x, 1700x and 1700. https://www.anandtech.com/show/11170/

the-amd-zen-and-ryzen-7-review-a-deep-dive-on-1800x-1700x-and-1700/ 9, 2017. Accessed: 2018-06-14.

[Cut17b] Ian Cutress. Intel announces skylake-x: Bringing 18-core hcc silicon to consumers for $1999. https://www.anandtech.com/show/11464/intel-announces-skylakex-bringing-18core-hcc-silicon-to-consumers-for-1999/3, 2017. Accessed: 2018-06-14.

[Cut18] Ian Cutress. Announcement three: Skylake-x's new l3 cache architecture. https://www.anandtech.com/show/11464/intel-announces-skylakex-bringing-18core-hcc-silicon-to-consumers-for-1999/3, 2018. Accessed: 2018-09-24.

[DKPT17] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+abort: A timer-free high-precision l3 cache attack using intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 51–67, Vancouver, BC, 2017. USENIX Association.

[D.L18] D.Lazard. Chinese remainder theorem. https://en.wikipedia.org/wiki/Chinese_remainder_theorem, 2018. Accessed: 2018-09-14.

[D.W18] User D.W. "fuzzy" chinese remainder theorem np-hard? https://cs.stackexchange.com/questions/94227/fuzzy-chinese-remainder-theorem-np-hard, 2018. Accessed: 2018-09-14.

[Fir18] Mozilla still working on firefox's site isolation security revamp. https://nakedsecurity.sophos.com/2018/08/01/mozilla-still-working-on-firefoxs-site-isolation-security-revamp, 2018. Accessed: 2018-09-27.

[GAJ98] Josephine M. Jones Gareth A. Jones. Elementary number theory. chapter Congruences, pages 37–63. Springer, 1998.

[Gee18] Mobile benchmarks. http://browser.geekbench.com/mobile-benchmarks, 2018. Accessed: 2018-09-25.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321, Cham, 2016. Springer International Publishing.

[GPTY18] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. Cryptology ePrint Archive, Report 2018/119, 2018. https://eprint.iacr.org/2018/119.

[GRB+17] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *NDSS*, 2017.

[GRLZ+17] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. Autolock: Why cache attacks on ARM are harder than you think. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1075–1091, Vancouver, BC, 2017. USENIX Association.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., 2015. USENIX Association.

[Gü18] Michael Günsch. Intel: Die ersten 10-nm-cpus bringt 2019 der weihnachtsmann. https://www.computerbase.de/2018-07/intel-10-nm-cpu-ice-lake-release-2019, 2018. Accessed: 2018-09-24.

[Hab15] Michael Hablich. Digging into the turbofan jit. https://v8project.blogspot.com/2015/07/digging-into-turbofan-jit.html, 2015. Accessed: 2018-08-29.

[Hag17] Hilbert Hagedoorn. Intel core i7 8700k processor review - performance - ddr4 system memory. https://www.guru3d.com/articles-pages/intel-core-i7-8700k-processor-review,17.html, 2017. Accessed: 2018-08-21.

[Hel06] Hellisp. Cache,associative-fill-both.png. https://commons.wikimedia.org/wiki/File:Cache,associative-fill-both.png, 2006. Accessed: 2018-08-21.

[Hen14] Henry. Openpgp message format. http://blog.stuffedcow.net/2014/01/x86-memory-disambiguation, 2014. Accessed: 2018-09-21.

[IES17] Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Hit by the bus: Qos degradation attack on android. In *Proceedings of the 2017 ACM on*

Asia Conference on Computer and Communications Security, ASIA CCS '17, pages 716–727, New York, NY, USA, 2017. ACM.

[Int16a] Intel® 64 and ia-32 architectures developer's manual: Vol. 3b. chapter Time-Stamp Counter, pages 152–154. 2016.

[Int16b] Intel. Intel® 64 and ia-32 architectures optimization reference manual. https://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2016. Accessed: 2018-06-14.

[JC07] H. Finney D. Shaw R. Thayer J. Callas, L. Donnerhacke. Openpgp message format. https://tools.ietf.org/html/rfc4880, 2007. Accessed: 2018-09-16.

[KDK+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *SIGARCH Comput. Archit. News*, 42(3):361–372, June 2014.

[Ket18] Mark Kettenis. Disable hyper-threading by default. https://www.mail-archive.com/source-changes@openbsd.org/msg99141.html, 2018. Accessed: 2018-09-23.

[KS16] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 463–480, Berkeley, CA, USA, 2016. USENIX Association.

[lps18] lpsolve. https://sourceforge.net/projects/lpsolve, 2018. Accessed: 2018-09-26.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 605–622, Washington, DC, USA, 2015. IEEE Computer Society.

[Mat18] Wolfram mathematica. https://www.wolfram.com/mathematica, 2018. Accessed: 2018-09-26.

[MES17] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *CoRR*, abs/1711.08002, 2017.

[Moz18a] Network security services. https://developer.mozilla.org/de/docs/Mozilla/Projects/NSS, 2018. Accessed: 2018-09-25.

[Moz18b] Mozilla. Mozilla dxr. https://dxr.mozilla.org/mozilla-central/source/security/nss, 2018. Accessed: 2018-08-21.

[OKSK15] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1406–1418, New York, NY, USA, 2015. ACM.

[Ope18] Openpgp.js. https://openpgpjs.org, 2018. Accessed: 2018-09-16.

[Pas18] PassMark. Amd vs intel market share. https://www.cpubenchmark.net/market_share.html, 2018. Accessed: 2018-06-16.

[Per05] Colin Percival. Cache missing for fun and profit. 2005.

[Rei18] Charlie Reis. Mitigating spectre with site isolation in chrome. https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html, 2018. Accessed: 2018-08-29.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[Sim16] Mike Simmonds. Feature. *Computer Fraud & Security*, 2016(10):19–20, 2016.

[SMGM17] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security*, pages 247–267, Cham, 2017. Springer International Publishing.

[Ste67] J Stein. Computational problems associated with racah algebra. 1, Feb 1967.

[Tan06a] Andrew S. Tanenbaum. Structured computer organization. chapter Virtual Memory, pages 428–452. Pearson, 2006.

[Tan06b] Andrew S. Tanenbaum. Structured computer organization. chapter Cache Memory, pages 293–298. Pearson, 2006.

[TSS+03] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of des implemented on computers with cache. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, pages 62–76, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[V818] V8. Chrome 68 on windows, linux, mac and chromeos supports sharedarraybuffers again. https://twitter.com/v8js/status/1024922907582099456, 2018. Accessed: 2018-08-29.

[VZRS15] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 913–928, Berkeley, CA, USA, 2015. USENIX Association.

[Wag18] Luke Wagner. Mitigations landing for new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack, 2018. Accessed: 2018-08-29.

[Web18a] Features to add after the mvp. https://webassembly.org/docs/future-features, 2018. Accessed: 2018-10-01.

[Web18b] Web crypto api. https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API, 2018. Accessed: 2018-09-25.

[Won13] Henry Wong. Intel ivy bridge cache replacement policy. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement, Jan 2013. Accessed: 2018-06-16.

[WXW12] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[Yar18] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. https://cs.adelaide.edu.au/~yval/Mastik, 2018. Accessed: 2018-10-02.

[YF14] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, 2014. USENIX Association.

[YGH17a] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, Jun 2017.

[YGH17b] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, Jun 2017.