

Введение в атаки по сторонним каналам на микроархитектуру, основанные на исполнении кода

Abc Xyz
@dura_lex

DCG#7812
2018

План

Введение

Теория

Типы атак

Атаки, основанные на аппаратных дефектах

Meltdown & Spectre

Абстрактный пример эксплуатации

Заключение

План

Введение

- Атаки по сторонним каналам
- Атаки на микроархитектуру

Атаки по сторонним каналам



Пример цели для атаки по сторонним каналам

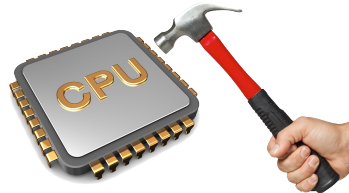
Идея атак по сторонним каналам **весьма стара**, ещё в 1980-х годах было о них известно. Но широкое распространение данный вид атак получил только после публикации **Пола Кохера в 1996 году**.

Самый примитивный пример атаки по сторонним каналам — **определение нажатых кнопок сейфа по звуку** при введении секретного кода.

Такого рода атаки обычно основываются на вычислении изменений в окружающей среде, например, изменения в **потреблении тока устройством, электромагнитном излучении, температуре, по издаваемым акустическим сигналам, по времени, затрачиваемому на выполнение тех или иных операций и другие**.

Атаки на микроархитектуру

```
code1a:  
  mov (X), %eax  
  mov (Y), %ebx  
  clflush (X)  
  clflush (Y)  
  jmp code1a
```



При эксплуатации аппаратных дефектов есть шанс нанести физические повреждения

Атаки по сторонним каналам на микроархитектуру, основанные на использовании программного обеспечения, как правило, даже **не требуют физического доступа** к вычислительному устройству.

Также существуют атаки, **основанные на дефектах микроархитектуры**, например, ошибки, происходящие **во время оптимизации**.

Атаки на микроархитектуру, которые используют аппаратные дефекты, **сложно воссоздать на практике**,. Примеров таких атак не много, но все они широко известны, это например, **Rowhammer атака**, которая, в случае успешно разработанного потока инструкций, может дестабилизировать работу процессора и даже нанести **неисправимые физические повреждения**, если атака будет проводиться в течении нескольких недель.

Все уязвимости можно найти, просто почитав главу оптимизаций в **спецификации процессора**, даже на Wiki есть раздел про оптимизацию работы CPU, в которой перечислены все элементы, в которых были найдены уязвимости.

- Процессор
- Кэш–память
- DRAM

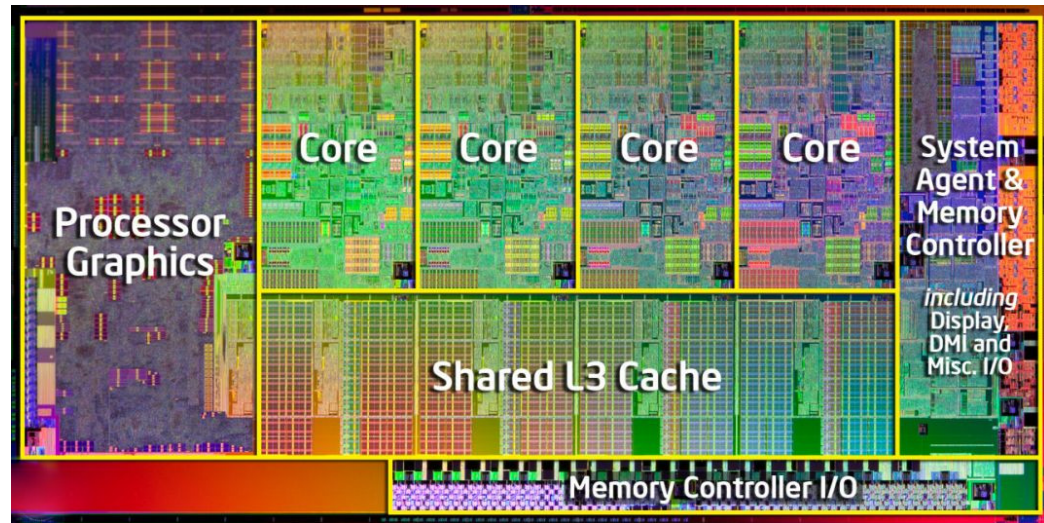
План

Теория

Процессор

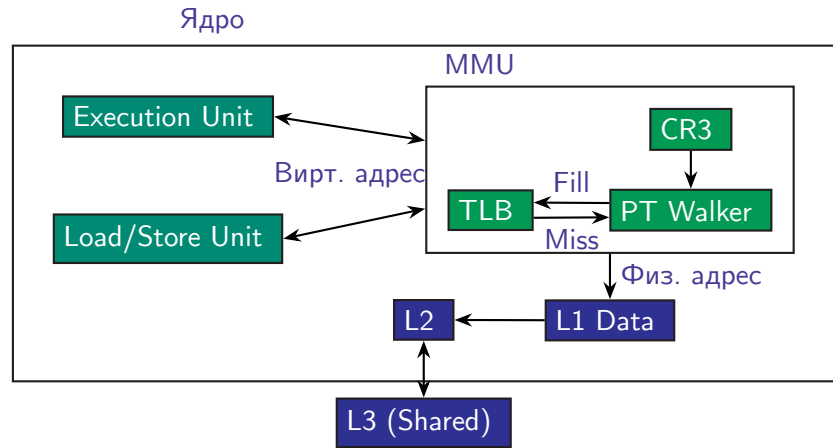
- Конвейеризация
- Оптимизатор потока инструкций
- Многоядерность

Процессор



Архитектура многоядерного процессора

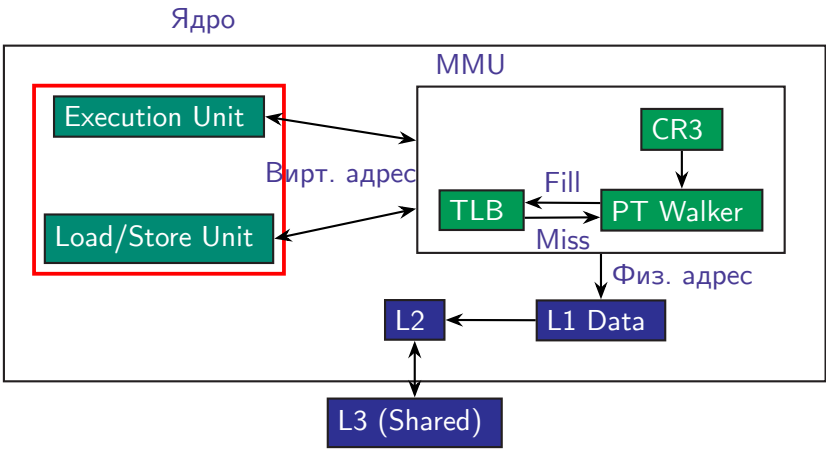
Современные процессоры состоят из множества крупных элементов: ядер, графического процессора, общего кэша, контроллера памяти и других.



Абстрактная архитектура элементов ядра, работающих с данными

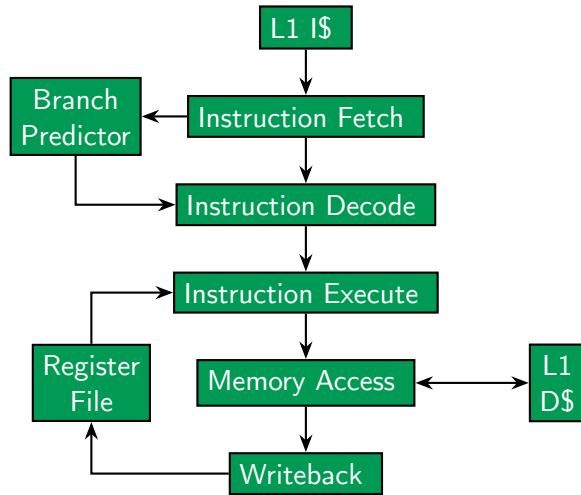
На рисунке 4 представлен общий план работы ядра с памятью и кэшем в Intel процессорах. Более подробно об алгоритмах работы будет рассказано ниже.

Современные процессоры представляют из себя **сильно распараллеливаемые машины**, которые оперируют данными на высоких скоростях. Размеры процессоров уменьшаются, уменьшается потребление памяти используемое для вычисления одной и той же операции, что позволяет **увеличивать тактовую частоту**. Однако, существуют и другие способы уменьшить время, затрачиваемое на выполнение инструкций — **различные оптимизации**, типы которых зависят от данных и состояния процессора. Рассмотрим некоторые **системы оптимизации, применяемые в ядрах и процессоре**.



Абстрактная архитектура элементов ядра, работающих с данными

Конвейеризация. По порядку



Элементы системы выполнения современного процессора (выполнение по порядку)

Конвейеризация — одна из главных причин высокой скорости работы процессора. В результате данного процесса **работа с инструкциями разделяется** на несколько этапов (рисунок 5, **выполнение инструкций по порядку**):

- **этап получения**, в результате которого код операции инструкции загружается в процессор;
- **этап декодирования**, в результате которого опкод декодируется во внутреннее представление процессора;
- **этап выполнения** — инструкция исполняется.

Конвейеризация. Не по порядку

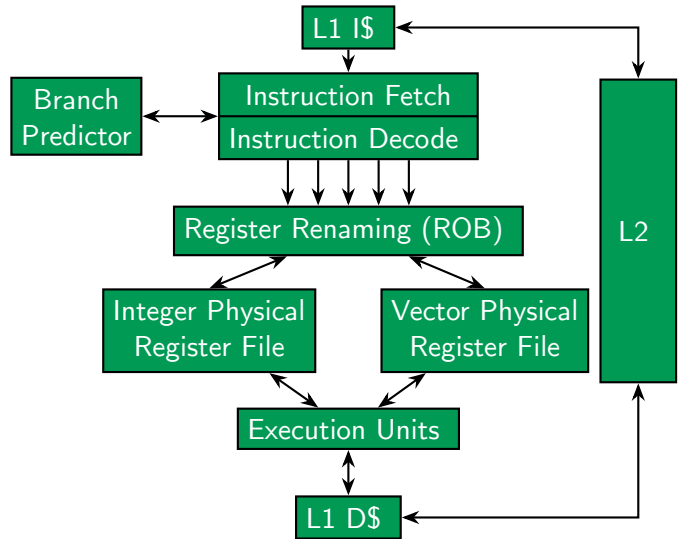
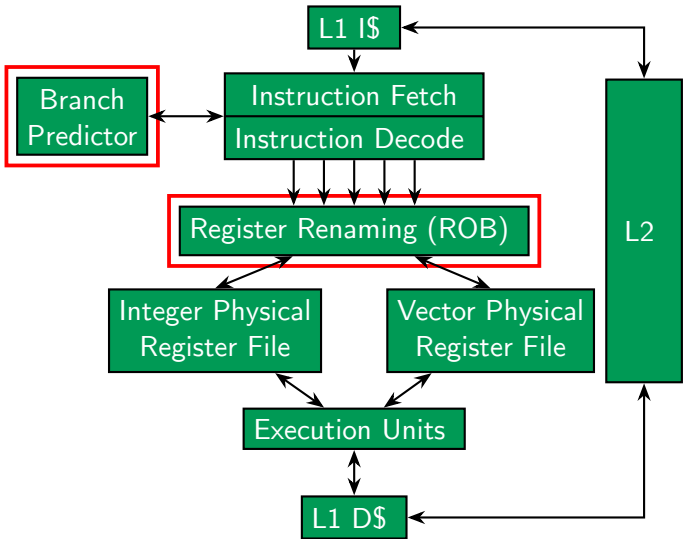


Рисунок 6, **выполнение инструкций не по порядку**.
Инструкции получают и декодируются по порядку во **front-end**.
ROB — Re-Order Buffer.

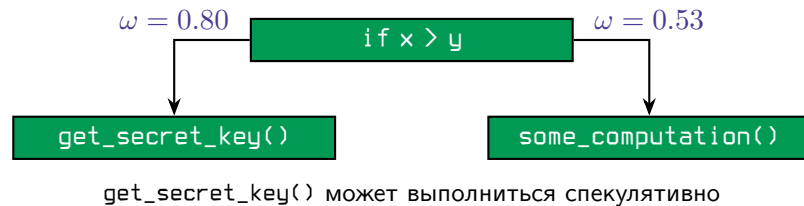
Именно поэтому процессор может выполнять несколько инструкций **одновременно**, при этом не обязательно в порядке их следования. Современные процессоры также могут параллельно выполнять одни и те же стадии для оптимизации вычислений.

Конвейеризация. Не по порядку



О работе Reorder Buffer **рассказано не будет**, существует множество его реализаций.
Ниже **будет рассказано** о работе этих элементов.

Оптимизатор потока инструкций

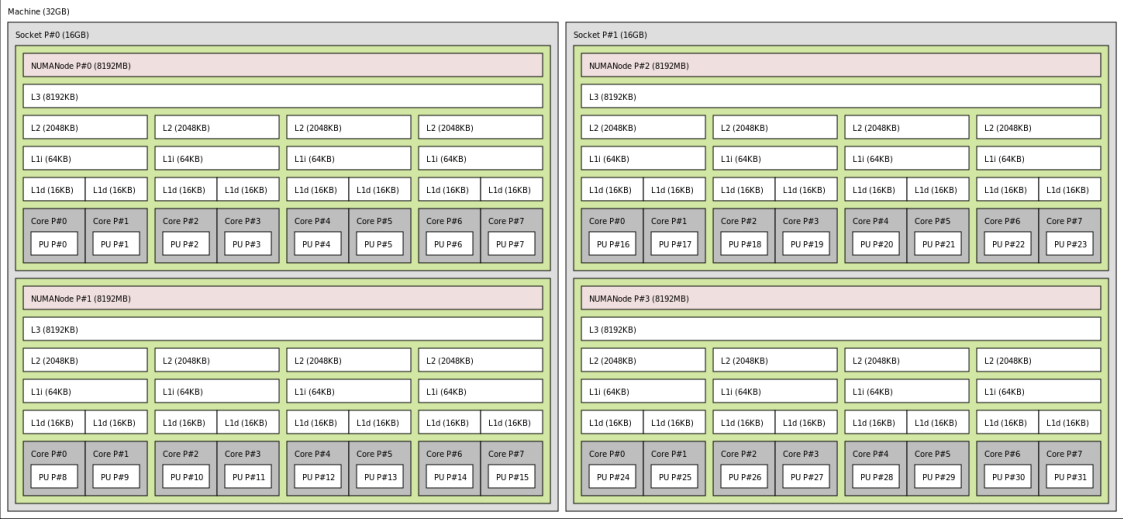


Ещё одна идея для повышения производительности процессора — **исполнение инструкций спекулятивно**. С помощью такой оптимизации процессор **угадывает возможный переход и исполняет его** прежде, чем он может выполняться на самом деле. Если угаданный путь был верным, то процессор просто берёт информацию, которую получил заранее, в противном случае **информация** о ходе спекулятивного выполнения просто **удаляется**.

Существуют:

- Статическое предсказание
- Динамическое предсказание
 - счётчик с насыщением
 - адаптивный двухуровневый предсказатель
 - локальный предсказатель перехода
 - глобальный предсказатель перехода
 - гибридный предсказатель перехода
 - предсказатель для цикла
 - предсказатель косвенных переходов
 - предсказатель инструкций возврата
 - предсказатель, основанный на машинном обучении

Многоядерность



Архитектура многоядерного процессора AMD Bulldozer

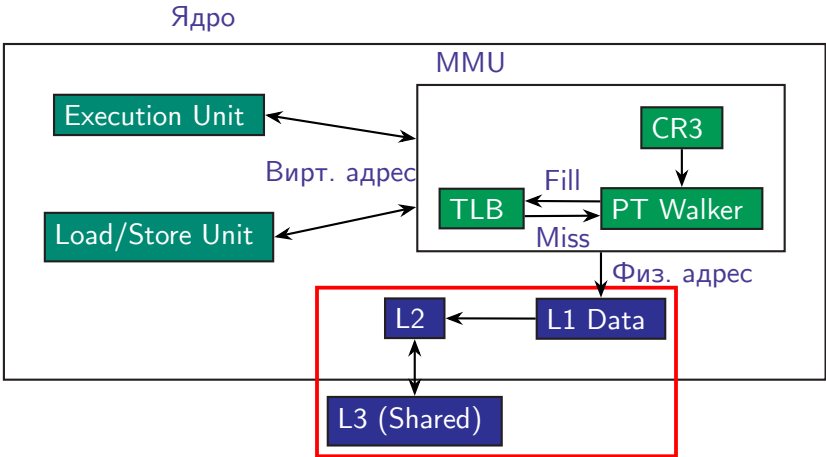
Вместо оптимизации скорости выполнения на единственном ядре, также существует возможность **увеличивать количество этих самых ядер**. Особенно часто много ядер установлено в процессорах, работающих на серверах, это позволяет выполнять многие независимые друг от друга задачи параллельно. Однако, если задачу невозможно распараллелить, то прироста в производительности, конечно же, не будет. В настоящее время, **почти любое устройство имеет несколько ядер**, в том числе IoT устройства и домашние компьютеры. К тому же, многие языки программирования позволяют без лишних сложностей писать приложения, которые будут исполняться на нескольких ядрах. **Каждое** из таких **ядер имеет свои приватные ресурсы**, например, регистры и конвейеры выполнения, а также **общие ресурсы**, например, основной доступ к памяти.

План

Теория

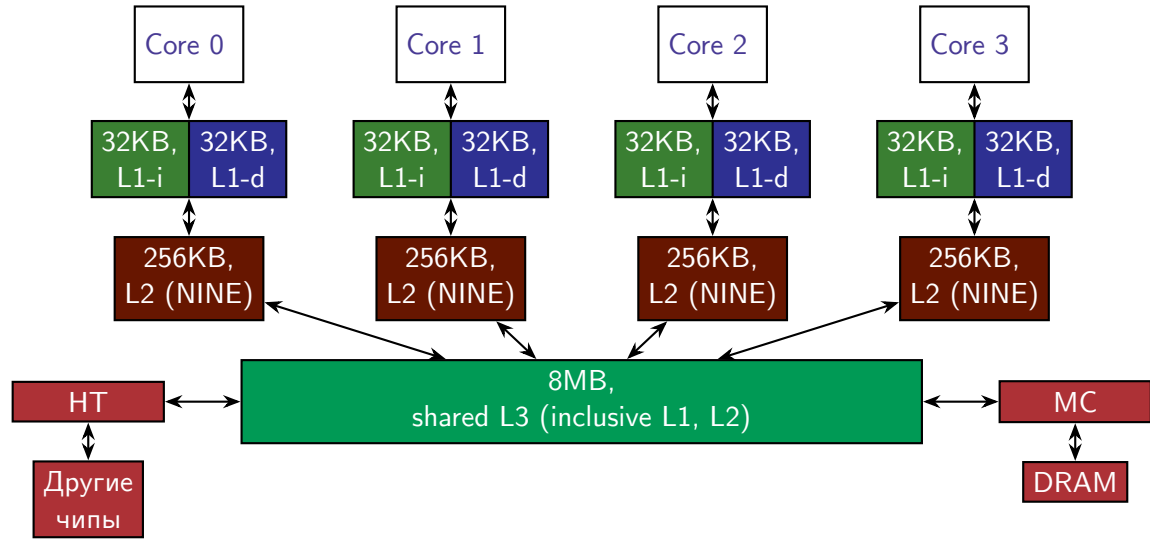
Кэш–память

- Типы кэш-памяти
- Наборно–ассоциативный кэш
- Правила вымещения из кэша
- Режимы адресации



Абстрактная архитектура элементов ядра, работающих с данными

Кэш-память

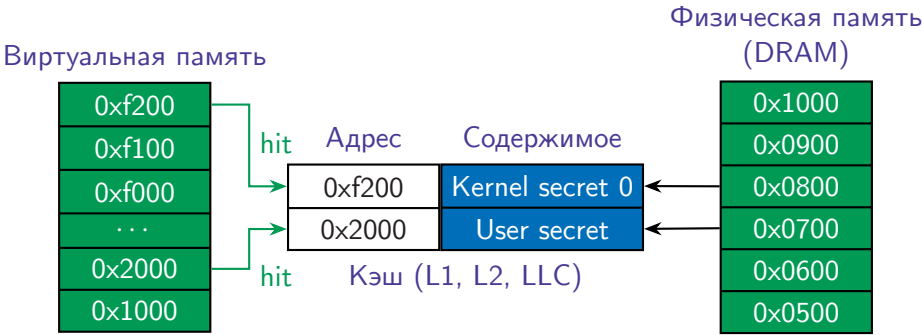


Архитектура процессора относительно кэшей

Современные процессоры имеют целую **иерархию кэшей** с различными размерами и скоростью обращения. Некоторые кэши приватные и работают в контексте **только одного процессора**, некоторые **общие**, их могут читать и писать все процессоры. Существует несколько **правил включения (инклюзивности)** кэша один в другой: правило **инклюзивности**, **эксклюзивности**, **NINE**. HT — HyperTransport; MC — Memory Controller.

Кэш-память

В общем случае, все доступы к памяти происходят через кэш. Если доступ к памяти происходит через кэш, то это называется **попаданием кэша** (*cache hit*).



Пример взаимодействия с кэшем

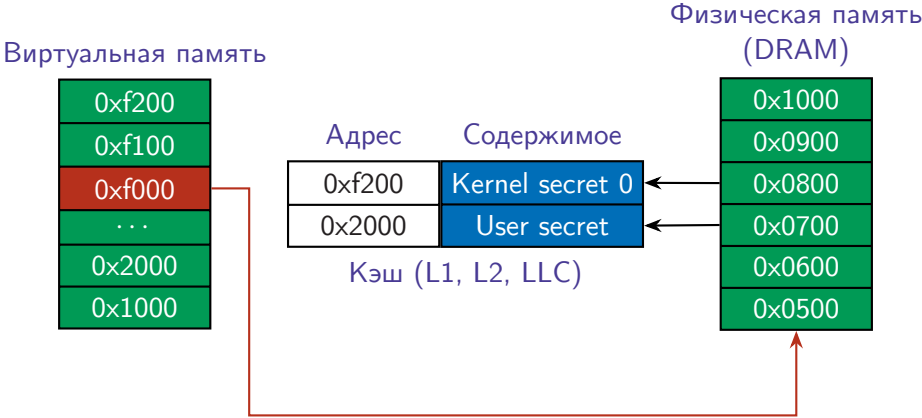
Кэш-память

В противном случае происходит **промах кэша** (*cache miss*).

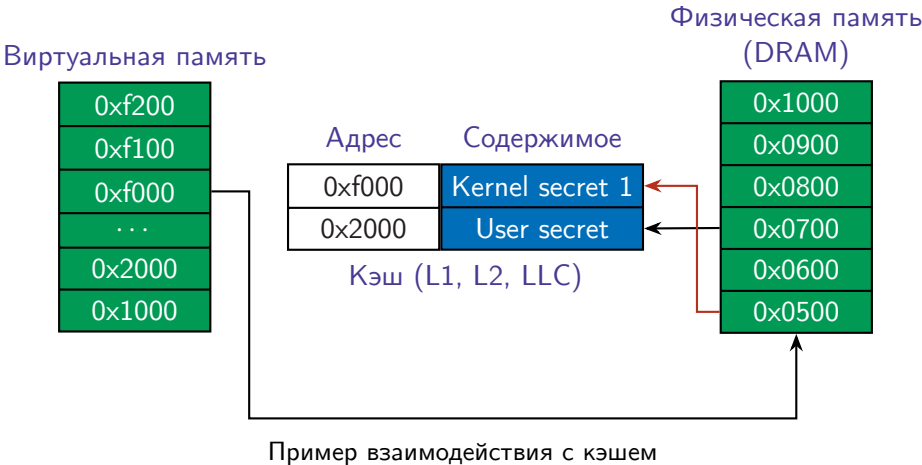


Кэш-память

Данные берутся из медленной памяти.



Пример взаимодействия с кэшем

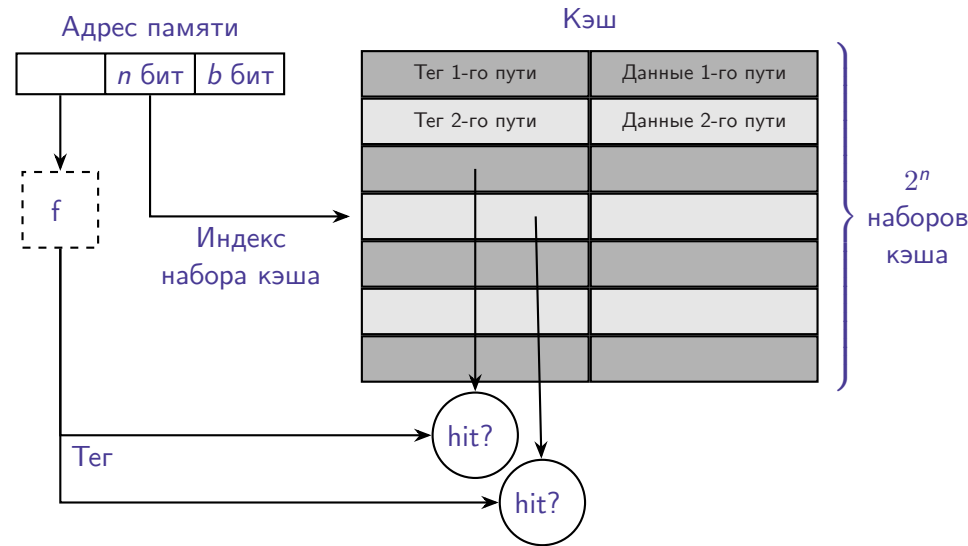


Типы кэш-памяти

- ▶ кэш с прямым отображением (direct mapped cache)
- ▶ полностью ассоциативный кэш (fully associative cache)
- ▶ наборно-ассоциативный кэш (2/4/8/12-way set associative cache)

1. Главная **проблема** такого вида кэша — это то, что кэш может **хранить единственную** линию кэша из всех **конгруэнтных**. Следовательно, если процессору требуется работать с двумя или более конгруэнтными линиями кэша, то такого рода кэш будет совершать **множество промахов**.
2. Такие кэши становятся более **дорогими с увеличением путей**. Поэтому они обычно содержат небольшое количество путей, например, в современных процессорах используются **буферы ассоциативной трансляции (translation-lookaside buffers TLB)** с 64 путями.

Наборно-ассоциативный кэш



Компромиссом между этими двумя видами кэша оказывается **кэш с наборами**, а не с линиями кэша. Данные кэши широко используются в современных процессорах, где их называют ***m*-путейные (или *m*-входовые) кэши с ассоциативным набором**. Рисунок отображает абстрактную модель 2-путейного кэша данного вида. Кэш делится на 2^n набора. **Индекс набора** в кэше определяется средними ***n*** битами адреса. Каждый набор имеет ***m* путей** для возможности хранения местоположения ***m* конгруэнтных адресов**. Наборы кэша могут быть также представлены в виде крошечного полностью ассоциативного кэша с ***m*** путями для набора конгруэнтных адресов. Поэтому **тег** снова используется для определения какой путь кэша содержит определённый адрес.

Правила вымещения из кэша

- ▶ FIFO
- ▶ LIFO
- ▶ least recently used, LRU
- ▶ time aware least recently used, TLRU
- ▶ most recently used, MRU
- ▶ pseudo-LRU, PLRU
- ▶ random replacement, RR
- ▶ segment LRU, SLRU
- ▶ least frequently used, LFU
- ▶ least frequent recently used, LFRU
- ▶ LFU with dynamic aging, LFUDA
- ▶ low inter-reference recency set, LIRS
- ▶ adaptive replacement cache, ARC
- ▶ clock with adaptive replacement, CAR
- ▶ multi queue, MQ
- ▶ и другие.

Количество путей или линий в кэше ограничено, а конгруэнтных адресов, которые требуется хранить, — **достаточно много**, требуется производить замены данных в кэше на новые, полученные из главной памяти.

Производители процессоров хранят детали этих правил в **секрете**, так как данные правила очень сильно влияют на скорость работы процессора в целом.

Самое широкое распространение получили правила вытеснения «**вытеснение давно неиспользуемых**» (**least-recently used, LRU**).

Процессоры **ARM** обычно используют правила **случайного вымещения**, так как такие правила просто реализовать на аппаратных средствах, и в ходе своей работы они потребляют мало энергии, а также показывают себя высокопроизводительными.

Режимы адресации

- ▶ Virtually indexed, virtually tagged (VIVT)
- ▶ Physically indexed, virtually tagged (PIVT)
- ▶ Virtually indexed, physically tagged (VIPT)
- ▶ Physically indexed, physically tagged (PIPT)

Кэши могут использовать как **виртуальные адреса**, так и **физические для вычисления индекса кэша и тега**. На практике используется три способа вычисления данных.

Позволяет использовать **тег из физического адреса**, при этом небольшая задержка, так как для поиска в первую очередь и чаще всего требуется определить номер набора, который задан виртуальным адресом.

- уникальный тег — **возможность применять разделяемые данные**
- всё происходит быстро, потому что **трансляция** адреса происходит **параллельно** поиску **индекса кэша**

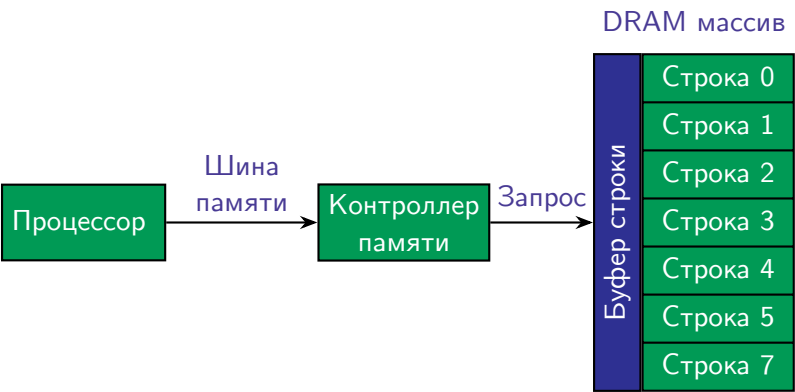
План

Теория

DRAM

- Алгоритм работы
- Физическое строение

Алгоритм работы



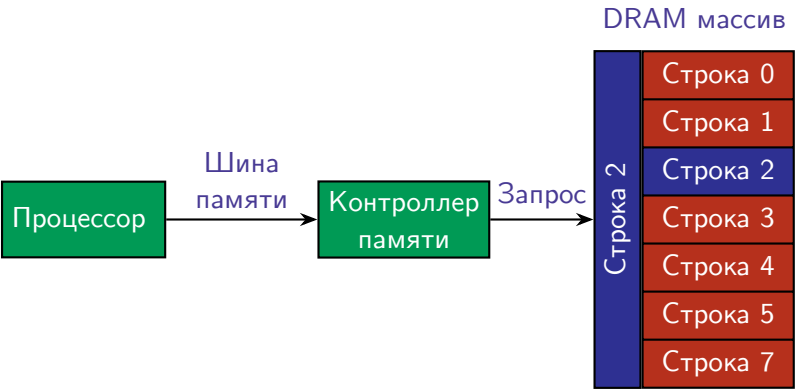
Простая компьютерная система с единственным DRAM массивом

DRAM (dynamic random-access memory, динамическая память с произвольным доступом).

DRAM имеет большую задержку в сравнении с кэш-памятью. Причина большой задержки не только в том, что ячейки DRAM имеют меньшую тактовую частоту, но и в том, как DRAM организован и подключён к процессору. Современные процессоры используют чипы-контроллеры памяти, которые позволяют передавать/получать данные в/с DRAM.

DRAM содержит: **строки (row)** и **колонки (columns)** (обычно 1024).

Алгоритм работы

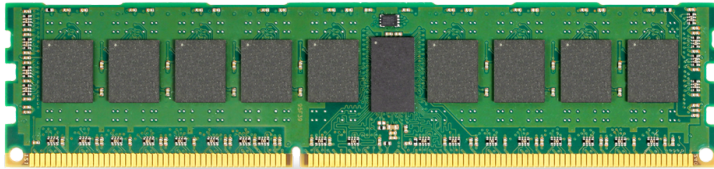


Простая компьютерная система с единственным DRAM массивом

Строка может быть **открытой** и **закрытой**. Если какая-либо строка открыта, то она вся сохраняется в **буфер строки (row buffer)**. Если текущая открытая строка содержит необходимые данные, то контроллер памяти просто берёт их из буфера строки. Эта ситуация очень похожа на кэш попадание и называется **попадание строки**. Если текущая открытая строка не содержит нужных данных, то это называется **промах строки**. **Контроллер памяти** в таком случае сначала **закрывает строку**, т. е. **записывает буфер строки обратно в DRAM**, а затем **открывает нужную строку** и считывает данные из буфера строки. Также как и промахи кэша, промахи строки вызывают повышение задержки.

Физическое строение

DIMM

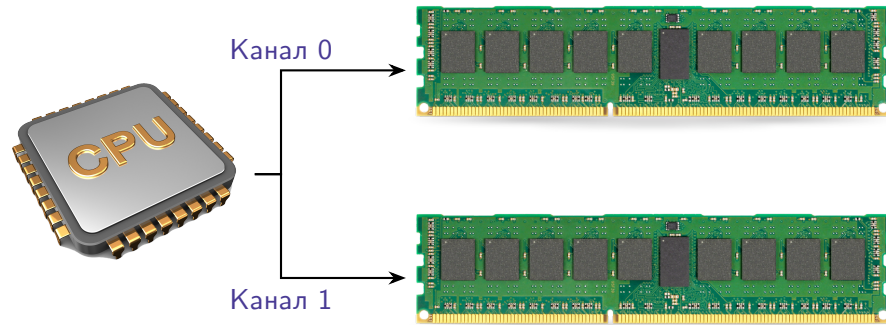


Архитектура DRAM

Для повышения производительности работы с DRAM были использованы те же методы, что и в случае с кэшем. Современные компьютерные системы организуют DRAM в виде **каналов**, **DIMM (Dual Inline Memory Modules)**, **рангов** и **банков**.

Количество памяти увеличивается в разы при перемножении количества банков на количество **DIMM модулей**.

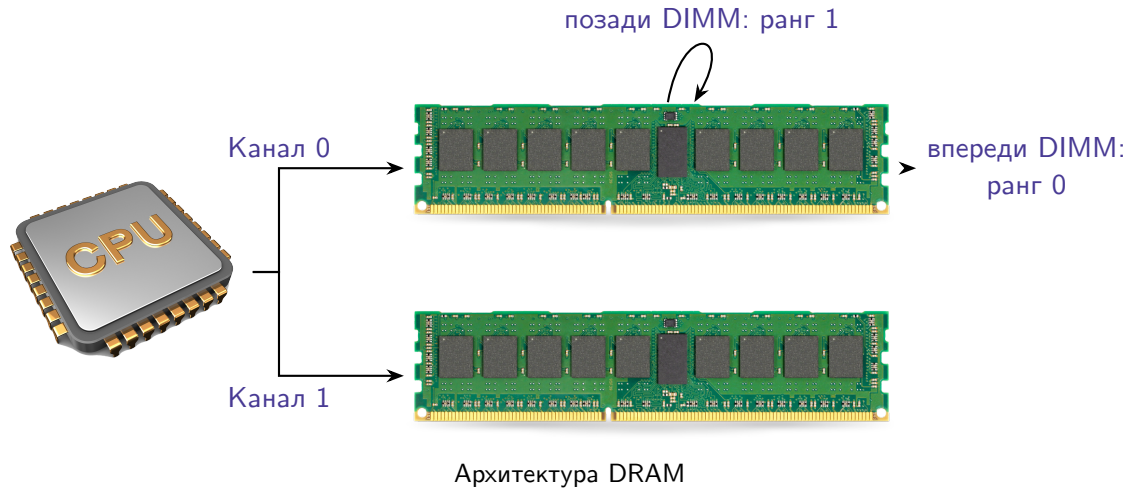
Физическое строение



Архитектура DRAM

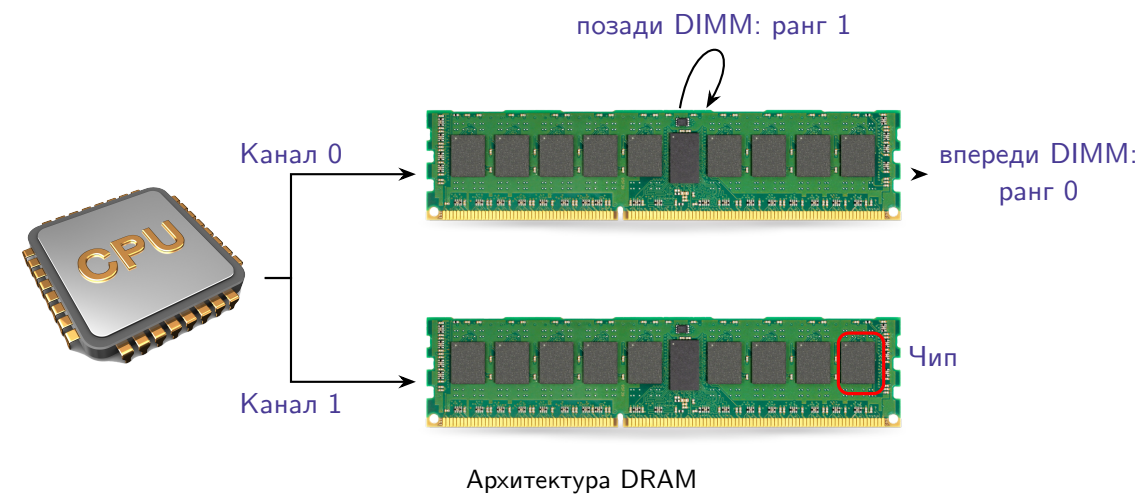
Для **увеличения ширины потока данных** и увеличения количества параллельных потоков современные компьютерные системы используют **несколько каналов**. Каждый канал управляется независимо и параллельно через DRAM шину.

Физическое строение

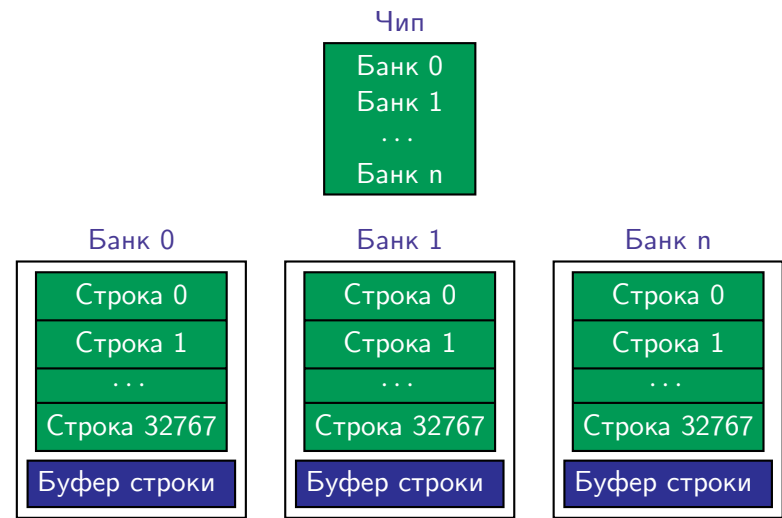


Современные модели DRAM имеют обычно от **1 до 4 рангов**, количество которых увеличивается при перемножении с количеством банков. Такого рода параллелизм позволяет **уменьшить промах строки**.

Физическое строение



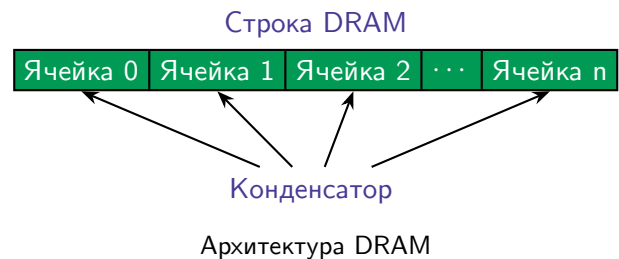
Физическое строение



Архитектура DRAM

Каждый из этих банков имеет своё собственное состояние и может иметь **независимо от других банков** открытую строку. Современная DDR3 DRAM память имеет 8 банков, а DDR4 DRAM 16 банков (на ранг).

Физическое строение



В случае если два адреса **отображаются на пространство одного и того же** DIMM модуля, ранга и банка, то эти адреса физически **расположены рядом друг с другом** в DRAM. В таких случаях два адреса оказываются в банке с одним и тем же номером. В случае, если адреса отображаются на пространство банков с одним и тем же номером, но разных рангов или DIMM'ов, то они не расположены физически рядом.

Также как и с кэшем, существуют функции, которые производят отображение физического адреса на конкретные канал, DIMM, ранг и банк. Как работают эти функции, публично известно только у AMD, Intel не публиковала никакой информации. Однако, относительно недавно (2015, 2016) был произведён реверс-инжиниринг данных функций. Знание того, **как производится отображение физического адреса на физические структуры** даёт нам новый вектор атак — **атаки по сторонним каналам на DRAM**.

План

Типы атак

- Атаки на кэш

- Атаки на предсказатель переходов

- Атаки на буфер ассоциативной трансляции

- Атаки, основанные на срабатывании исключительных ситуаций

- Атаки на DRAM

- Скрытые каналы

План

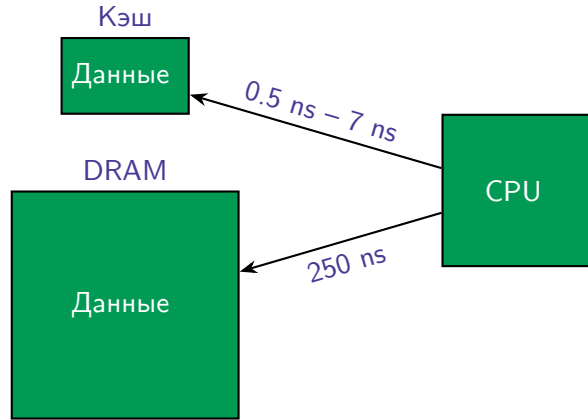
Типы атак

Атаки на кэш

Flush + Reload

Другие типы атак на кэш

Атаки на кэш



Кэш — это не только полезно, но и опасно

Главное **предназначение кэш-памяти** — **нивелировать задержки** при работе с медленной главной физической памятью. При работе с данными через кэш задержки значительно уменьшаются. Kocher в 1996 году описал возможность использования разницы во времени при обращении к данным для совершения атаки, которая стала называться **атака на кэш по времени**.

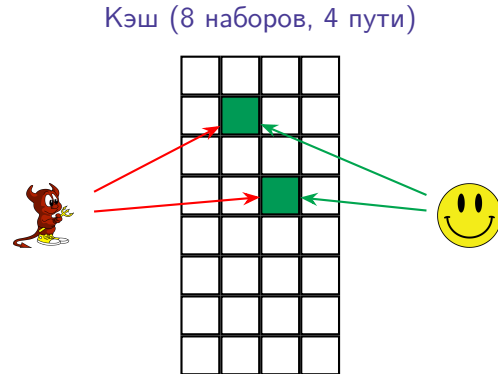
Flush + Reload

1. Отобразить бинарный файл (например, разделяемый объект) в своё адресное пространство
2. Сбросить содержимое кэш-линии (код или данные)
3. Передать управление программе-жертве
4. Определить какие линии кэша были загружены программой-жертвой снова

Данная атака считается **наиболее эффективной** атакой на кэш. Целью данной атаки является не просто набор кэша, а **отдельная линия кэша**, более того, у атакующего существует возможность проверить закеширована ли та или иная область памяти. Атака Flush + Reload выполняется в три фазы.

Flush + Reload

Отобразить бинарный файл (например, разделяемый объект) в своё адресное пространство

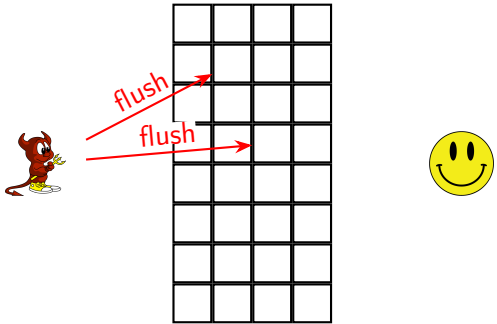


Flush + Reload атака работает при условии **общей памяти**, ярким примером может служить общая библиотека, которую использует и атакующий и программа-жертва. По этой причине, в случае, если нет такого элемента, как общая память между атакующим и жертвой, придётся использовать схему атаки Prime + Probe.

Flush + Reload

Сбросить содержимое кэш-линии (код или данные)

Кэш (8 наборов, 4 пути)



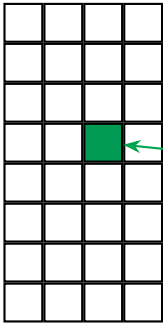
Во время первой фазы контролируемый участок памяти удаляется из структуры кэша (удаляется линия кэша с помощью инструкции **clflush** в случае с Intel).

Flush + Reload

Во второй фазе программа-шпион находится в режиме ожидания, **давая жертве время воспользоваться** участком памяти.

Передать управление программе-жертве

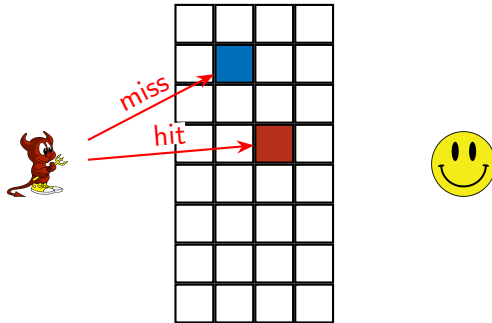
Кэш (8 наборов, 4 пути)



Flush + Reload

Определить какие линии кэша были загружены программой-жертвой снова

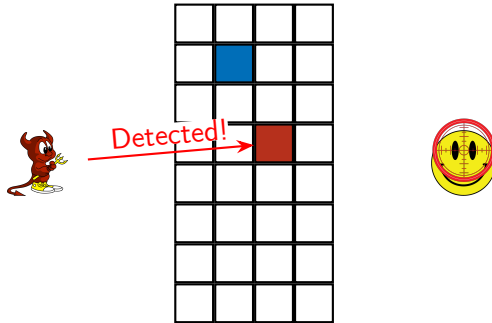
Кэш (8 наборов, 4 пути)



Во время третьей фазы программа-шпион **перезагружает** участок памяти и **замеряет время** загрузки. Если во время второй фазы жертва **воспользовалась** участком памяти, то этот участок будет доступен из кэша и операция перезагрузки **пройдёт быстро**. С другой стороны, если линия кэша **осталась неиспользованной**, понадобится время на загрузку участка и операция перезагрузки пройдёт **значительно дольше**.

Flush + Reload

Кэш (8 наборов, 4 пути)



Такие методы атаки, как Flush + Reload и Flush + Flush (описан ниже), используют **непривилегированную** x86-инструкцию сброса `clflush` для удаления строки данных из кеш-памяти. Однако, за исключением процессоров ARMv8-A, **ARM-платформы не имеют непривилегированных инструкций сброса кеша**, и поэтому в 2016 году был предложен **косвенный метод вытеснения кеша**, с использованием эффекта Rowhammer.

Другие типы атак на кэш

- ▶ Evict + Time
- ▶ Prime + Probe
- ▶ Prime + Abort
- ▶ Flush + Flush
- ▶ Evict + Reload
- ▶ AnC (ASLR \oplus Cache)
- ▶ и др.

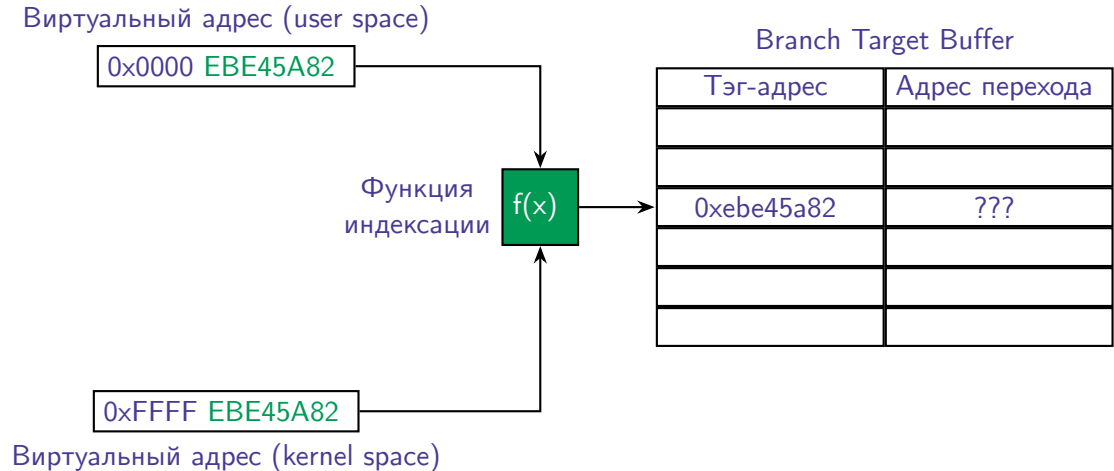
Существуют другие атаки на кэш, которые применяются в различных ситуациях, позволяют проводить атаку в стелс-режиме, направлены на различные уровни кэшей и т. д.

План

Типы атак

Атаки на предсказатель переходов

Атаки на предсказатель переходов



Тег вычисляется, основываясь на последних байтах виртуального адреса

Буфер адресов перехода (branch target buffer, BTB) кэширует информацию о ранее выбранных переходах выполнения для быстрого угадывания будущих. Кэш использует индексацию на **базе виртуального адресного пространства**, таким образом **атакующему не обязательно знать физический адрес** для выполнения атаки. **Атакующий заполняет буфер адресов перехода** путём выполнения последовательности различных переходов. Если жертва-программа будет выполнять ту ветвь, которой не было в кэше, то она добавится туда, вытеснив тем самым существующую запись. **Атакующий может вычислить** какая ветвь была вытолкнута из буфера по сравнительно **большому времени выполнения** этой ветви. Пример атаки позволяет взломать KASLR из пользовательского процесса. Основывается на возникающих **коллизиях** в кэше branch target buffer. По времени выполнения своего кода атакующий имеет возможность **вычислить адрес перехода в ядерном пространстве** (значение берётся из BTB), тем самым вычислить смещение, полученное в результате KASLR. Возникает два типа коллизий:

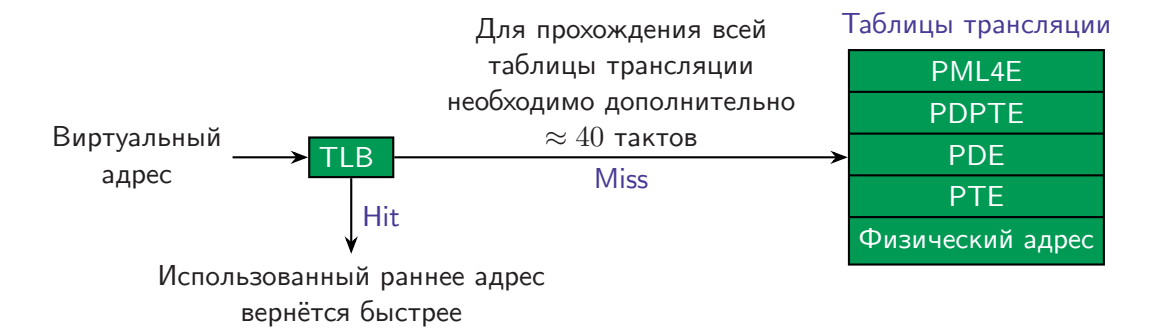
1. cross domain collisions — user и kernel space;
2. same domain collisions — разные user-space процессы, позволяет

План

Типы атак

Атаки на буфер ассоциативной трансляции

Атаки на буфер ассоциативной трансляции



Translation lookaside buffer (TLB) используется как для ускорения трансляции виртуальных адресов ядерного пространства, так и пользовательского!

Трансляция адресов должна происходить очень быстро. С использованием таблиц трансляций, расположенных в памяти, данная операция быстро выполняться не может. По этой причине был введён кеш для трансляции адресов, который помогает уменьшить задержку при процессе трансляции — **буфер ассоциативной трансляции (translation lookaside buffer, TLB)**.

Атака впервые была представлена Ralf Hund в 2013 году. Попытка чтения или записи памяти, к которой нет доступа по причине того, что данный участок памяти **используется ядром, занимает меньше времени**, если бы память не была размечена вовсе, т. к. используемые адреса памяти попадают в кэш независимо от уровня привилегий. Это позволяет узнать, какие адреса используются, и более того, узнать какие адреса **используются той или иной частью ядра**, т. е. данный вид атаки позволяет обойти технику рандомизации памяти в ядерном пространстве (kernel address-space-layout randomization, **KASLR**).

Атаки, основанные на срабатывании исключительных ситуаций

Атаки, основанные на срабатывании исключительных ситуаций

- ▶ прерывание планировщика
- ▶ инструкции прерывания
- ▶ ошибка отсутствия страницы в памяти
- ▶ поведенческие изменения (например, возврат кода ошибки)

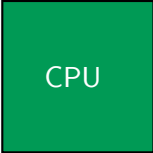
Данного рода атаки получают необходимую информацию из исключительных ситуаций, которые происходят при работе процессора. Обычными для процессора исключительными ситуациями являются: прерывание планировщика, прерывания инструкции, ошибка страницы памяти, а также поведенческие изменения, например, инструкции предоставляют пользователю код ошибки.

Во время возникновения исключительных ситуаций можно получить информацию о работе процессора либо **напрямую** (основываясь на поведении самого процессора), либо **косвенно (через измерения времени, при возникновении исключительных ситуаций)**.

Одним из ярких примеров атак подобного рода является **атака на систему дедупликации памяти**.

Алгоритм работы DRAM

Ещё раз о том, как работает **row buffer**.



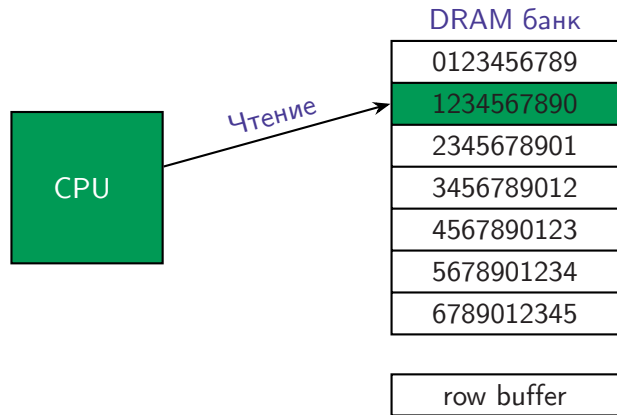
DRAM банк

0123456789
1234567890
2345678901
3456789012
4567890123
5678901234
6789012345

row buffer

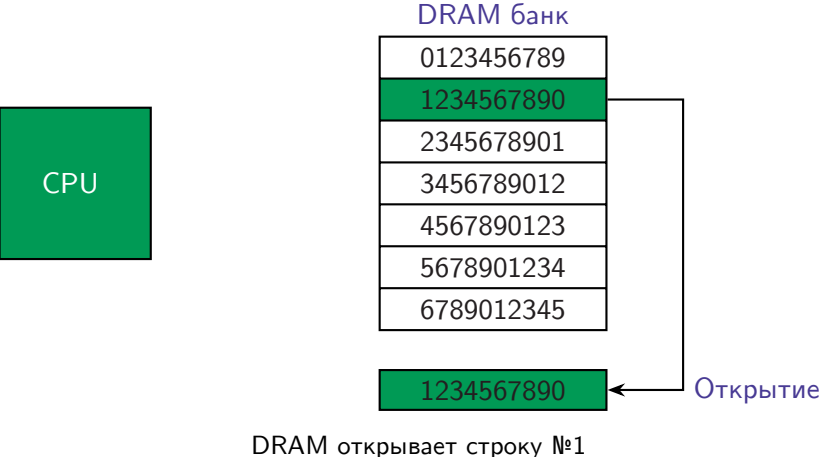
Работа DRAM (ещё раз)

Алгоритм работы DRAM

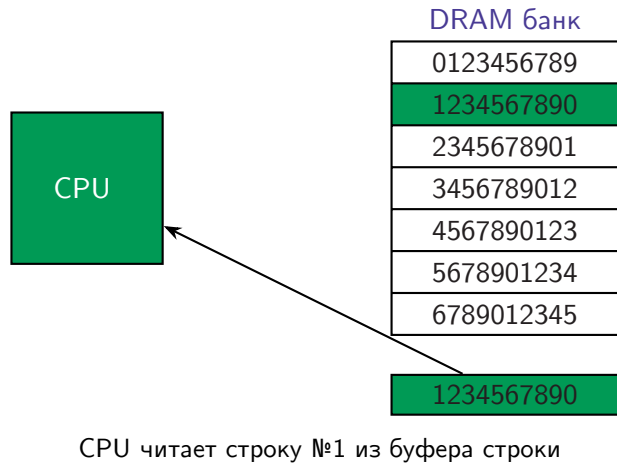


CPU запрашивает на чтение строку №1

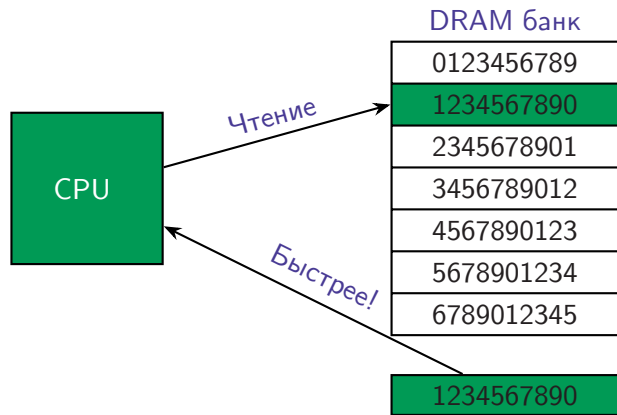
Алгоритм работы DRAM



Алгоритм работы DRAM



Алгоритм работы DRAM



CPU снова запрашивает на чтение строку №1, которая уже есть в буфере строки, чтение происходит быстрее

При попадании строки чтение данных происходит быстрее, при промахе строки — медленнее, что похоже на поведение при работе с кэшем.

Типы атак на DRAM

- ▶ DRAMA
- ▶ Row hit (Flush + Reload)
- ▶ Row miss (Prime + Probe)
- ▶ и др.

Существуют атаки непосредственно на DRAM (DRAM addressing, **DRAMA**).

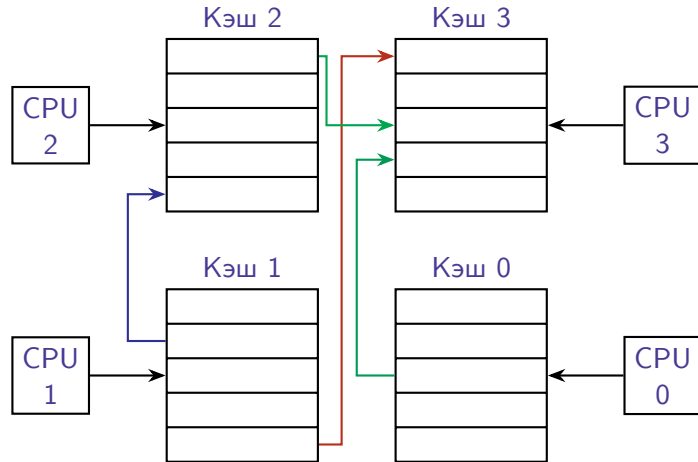
Атака при **промахе строки** похожа на атаку на кэш **Prime + Probe**, атака при **попадании строки** сравнима с атакой **Flush + Reload**. Оба типа атак работают и при **отсутствии разделяемой памяти**. DRAMA эксплуатирует буфер строки DRAM, как будто это **кэш с прямым отображением используемый банком**.

Во время подготовки атаки DRAMA применяли методы реверс-инжиниринга, основанные на подобном поведении DRAM, для того, чтобы **выявить местоположение и архитектуру банков**.

План

Типы атак
Скрытые каналы

Скрытые каналы



Скрытые каналы между процессорами

Современные облачные системы часто имеют **несколько процессоров**, установленных в материнскую плату с **мультисокетом**. Кэши процессоров содержатся в когерентном состоянии с помощью **межпроцессорных протоколов, обеспечивающих когерентность**. Однако, это обеспечивает эффект **разделяемой кэш линии**.

Кеш-атаки позволяют реализовать **высокопроизводительные межъядерные и межпроцессорные скрытые кеш-каналы** на современных смартфонах, используя Flush+Reload, Evict+Reload или Flush+Flush. Скрытый канал позволяет двум **непривилегированным приложениям** взаимодействовать друг с другом **без использования** каких-либо системных **механизмов передачи данных**. Благодаря этому можно вырваться из песочницы и обойти систему «ограниченных разрешений». В частности, на Android злоумышленник может использовать одно приложение, которое имеет **доступ к личным контактам** владельца устройства, для **отправки данных по скрытому каналу** другому приложению, имеющему доступ к **интернет**.

Скрытые каналы

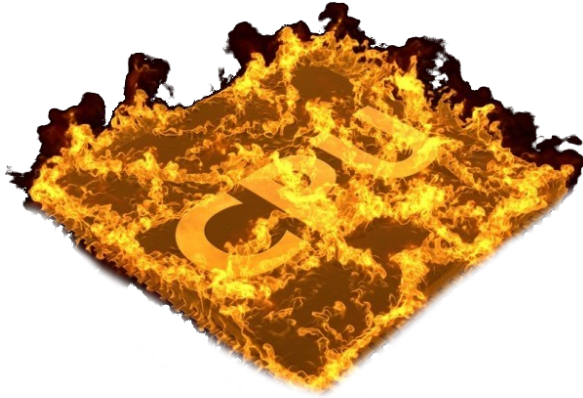
- ▶ Атаки на кэш (использование разделяемой библиотеки)
- ▶ Row miss атака (DRAM)
- ▶ Тепловой канал
- ▶ Радио канал (без специализированного аппаратного обеспечения)

Wu в 2012 и в 2014 годах обнаружил **разницу во времени при доступе к памяти**, возникающую из-за **задержки в шине памяти**, что позволяет наладить скрытый канал передачи данных между **близко расположенными виртуальными машинами**. В облаке Amazon EC2 был налажен канал скоростью 13.5 КБ/с при 0.75 % ошибок. В 2016 Inci также обнаружил задержку в шине памяти, позволяющую наладить канал в облаках Microsoft Azure.

В 2017 году была представлена первая в своём роде реализация **скрытого канала, работающего по протоколу SSH**, с относительно высокой пропускной способностью (45 Кбит/с); эта реализация обеспечивает отказоустойчивые коммуникации между двумя виртуальными машинами даже в условиях экстремальной зашумлённости кеша.

Атаки, основанные на аппаратных дефектах
Rowhammer

Атаки, основанные на аппаратных дефектах



Обычно предполагается, что безопасность системы и программная безопасность опирается на безопасность аппаратную и на то, что в аппаратном средстве нет ошибок. Однако, это не так, и **аппаратные средства не идеальны**, особенно часто дефекты встречаются в случаях, когда работа производится **за границами спецификации**. Уникальность атак, основанных на дефектах микроархитектуры в том, что они используют эффекты, вызванные микроархитектурными элементами или операциями, которые реализованы на микроархитектурном уровне. В атаках, которые основаны на использовании программного обеспечения все микроархитектурные эффекты и операции вызываются из программного обеспечения.

Аппаратные дефекты можно эксплуатировать с помощью исполнения кода

Атаки, основанные на аппаратных дефектах

Rowhammer

Необходимые примитивы Rowhammer

Разновидности Rowhammer

Необходимые примитивы Rowhammer

- ▶ быстрый некэшируемый доступ к памяти
- ▶ определение местонахождения уязвимых строк DRAM
- ▶ знание функций адресации физической памяти

Быстрый некэшируемый доступ к памяти. Данный примитив не так-то легко получить, т. к. **контроллер памяти** на CPU может **недостаточно быстро обрабатывать запросы** на чтение памяти. Нерасторопность контроллера, как правило нивелируется загрузкой данных в кэши. **Использование кэшей также необходимо предотвратить** для того, чтобы было постоянное обращение к DRAM.

Определение местонахождения уязвимых строк. Кроме того, требуется, чтобы жертва использовала нужные атакующему строки DRAM для хранения важной информации.

Знание функций адресации физической памяти. Требуется для определения методов трансляции виртуальных адресов в физические и трансляции физических адресов на аппаратное средство.

Разновидности Rowhammer

- ▶ Flip Feng Shui — целенаправленный Rowhammer
- ▶ Throwhammer — удалённая атака
- ▶ Nethammer — улучшенная удалённая атака
- ▶ Drammer, RAMpage — атака на ARM
- ▶ Glitch — улучшенная атака на ARM

Очень много НО.

- дедупликация памяти, **разделяемая память**.
- удалённый прямой доступ к памяти (**remote direct memory access, RDMA**)
- **Intel CAT**, **некэшируемая память**, инструкции очистки кэша в сетевых драйверах.
- ARM — медленная запись в память, инструкция очистки кэша — привилегированная, **Android ION аллокатор памяти**.
- **GPU на мобильных устройствах**, т. к. CPU и GPU используют одну оперативную память, используется WebGL.

План

Meltdown & Spectre

- Variant 3
- Variant 1
- Variant 2
- Variant 4
- Производные и не только

Meltdown & Spectre

Variant 3

- Необходимые условия
- Предотвращение

Variant 3

CVE-2017-5754: спекулятивное чтение недоступных данных



Meltdown

Необходимые условия

- ▶ Конвейерное выполнение кода не по порядку
- ▶ Трансляция адресов памяти с привилегированными данными
- ▶ На некоторых процессорах данные также должны лежать в кэше данных L1
- ▶ Точные счётчики времени
- ▶ и др.

- На самом деле не все архитектуры используют выполнение не по порядку в том виде, которое требуется для реализации meltdown.
- Должна существовать возможность обращения к привилегированным данным по виртуальному адресу памяти.
- Всё зависит от того, как происходит работа с кэшем.
- Для реализации атаки на кэш.

Предотвращение

При смене контекста происходит проседание производительности ввиду того, что полностью сбрасывается TLB, изменяется CR3 регистр.

Изоляция адресного пространства ядра — kernel page-table isolation, KPTI (KEISER — Kernel Address Isolation to have Side-channels Efficiently Removed)

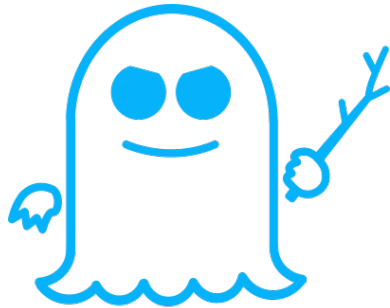
Meltdown & Spectre

Variant 1

- Необходимые условия
- Предотвращение

Variant 1

CVE-2017-5753: обход проверки границ



SPECTRE

Необходимые условия

- ▶ Спекулятивное выполнение кода
- ▶ Присутствие необходимых гаджетов в коде
- ▶ Обход других систем защиты в коде
- ▶ Точные счётчики времени
- ▶ и др.

- Ожидаемое поведение при спекулятивном выполнении кода.
- Составляется особая ROP цепочка для эксплуатации, которая **помещается в спекулятивное окно**.
- Обход **ASLR** другими атаками.
- Для реализации атаки на кэш.

Предотвращение

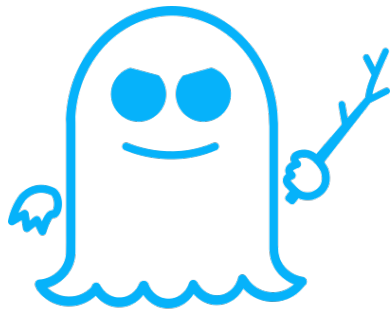
- ▶ отключение спекулятивного выполнения
- ▶ ограничение доступа к высокоточным таймерам
- ▶ привилегированная очистка кэша
- ▶ полное изолирование важных данных
- ▶ вставка инструкций для остановки спекулятивного выполнения

- Как отключить? Большое проседание в производительности!
- сделали свои таймеры
- другие методы очистки
- spectre работает и на безопасных анклавах
- большое проседание по производительности + всё перекомпилировать

Meltdown & Spectre
Variant 2
Предотвращение

Variant 2

CVE-2017-5715: тренировка предсказателя переходов



SPECTRE

Предотвращение

Всё медленно!

retpoline — замена всех косвенных переходов, дополнение инструкций возврата, паузы перед непрямыми вызовами функций

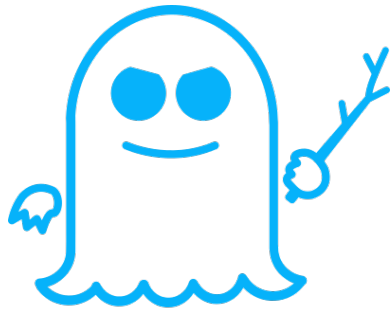
Частично такое же, как и в случае с Variant 1

- ▶ отключение предсказателя переходов
- ▶ очистка буфера предсказателя переходов при переключении контекста
- ▶ вставка «барьеров» (Indirect Branch Restrict Speculation, Indirect Branch Predictor Barrier и др.)
- ▶ retpoline — «оборачивание» косвенных переходов
- ▶ oo7 — умное «оборачивание» косвенных переходов

Meltdown & Spectre
Variant 4
Предотвращение

Variant 4

CVE-2018-3639: спекулятивное выполнение чтения памяти после сохранения её в регистр



SPECTRE

Частично такое же, как и в случае с Variant 2

- ▶ отключение реорганизации операций чтения и записи
- ▶ SafeSpec

Meltdown & Spectre

Производные и не только

Производные и не только

Spectre-NG

- ▶ MeltdownPrime & SpectrePrime
- ▶ SgxPectre
- ▶ SMM Speculative Execution Attacks
- ▶ BranchScope
- ▶ LazyFP
- ▶ ...

OpenBSD — LazyFP, отключение Hyper-Threading (TLBleed).

- **другой способ атаки на кэш** + задействование двух ядер CPU — утечка данных работы **протокола согласования содержимого кэша для разных ядер CPU** (Invalidation-Based Coherence Protocol)
- позволяет обойти средства изоляции кода и данных, предоставляемые технологией **Intel SGX** (Software Guard Extensions)
- SMM — **режим системного управления** — запускается специальная программа в привилегированном режиме.
- **variant 2** + вместо BTB — **направления ветвления для спекулятивного перехода (directional branch predictor)** и манипулирует содержимым **таблицы с историей шаблонов переходов (PHT, Pattern History Table)**.
- **variant 3a** — «ленивое» режим **переключения контекста FPU**, при котором реальное восстановление состояния регистров производится **не сразу** после переключения контекста, а только при выполнении первой инструкции

Производные и не только

- ▶ TLBleed
- ▶ Spectre 1.1, 1.2 (Speculative Buffer Overflows)
- ▶ SpectreRSB
- ▶ NetSpectre
- ▶ ...

TotalMeltdown?

- ML атака на TLB, с помощью Hyper-Threading технологии (Intel, AMD).
- Spectre 1 + Buffer Overflows, спекулятивная запись.
- Засорение RSB (return stack buffer), спекулятивное выполнение после return.
- Производится **поиск leak и transmit гаджетов**, низкая производительность (1-3 байта за 3-8 часов).

Абстрактный пример эксплуатации

Требуется для атаки с помощью техники спекулятивного выполнения:

1. Примитив спекулятивного выполнения

А так ли всё легко и просто? Представим, что мы хотим совершить атаку.

Рассмотрим на примере Spectre v2 (variant 2).

Процессор **должен поддерживать** возможность спекулятивного выполнения.

Требуется **найти место** в коде, где может происходить спекулятивное выполнение по желанию атакующего.

Абстрактный пример эксплуатации

Требуется для атаки с помощью техники спекулятивного выполнения:

1. Примитив спекулятивного выполнения

- ▶ Обход проверки границ
 - ▶ Тренировка предсказателя переходов
 - ▶ Чтение памяти после сохранения её в регистр
 - ▶ Отложенная исключительная ситуация
 - ▶ Засорение таблиц с историей шаблонов переходов
 - ▶ Засорение Return Stack Buffer
 - ▶ Спекулятивная запись (buffer overflow)
- } Микроархитектура — ?

На данный момент существует несколько техник, позволяющих производить спекулятивное выполнение по желанию атакующего.
Для того, чтобы использовать те или иные техники **требуется досконально знать микроархитектуру процессора**.

Абстрактный пример эксплуатации

Вид ПП

Алгоритм работы ПП

Характерные условия работы ПП

Фундамент башни
атаки на основе спекулятивного выполнения

Выберем тренировку предсказателя переходов. Мы столкнёмся:

- требуется знать **вид предсказателя переходов**
- требуется знать **алгоритм работы предсказателя переходов**
- для **разных процессоров — разные условия**, например, в i7 два буфера предсказателя переходов.

В whitepaper и в PoC даны **примеры для конкретных процессоров**.

Абстрактный пример эксплуатации

Требуется найти гаджеты, которые позволят создать достаточно **длительное по времени выполнения окно** для спекулятивного выполнения.

Требуется для атаки с помощью техники спекулятивного выполнения:

1. Примитив спекулятивного выполнения
2. Гаджеты для создания «окна» спекулятивного выполнения

Абстрактный пример эксплуатации

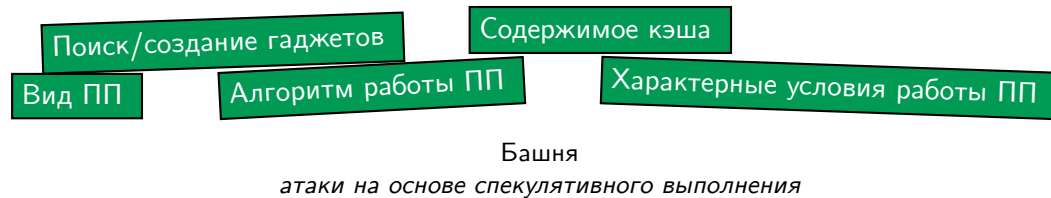
Не везде есть такие цепочки в окружении. В некоторых случаях требуется знать, какие данные сейчас в кэше.

Требуется для атаки с помощью техники спекулятивного выполнения:

1. Примитив спекулятивного выполнения
2. Гаджеты для создания «окна» спекулятивного выполнения

- ▶ Загрузка некешированных данных
- ▶ Цепочка из зависимых загрузок данных
- ▶ Цепочка из зависимых целочисленных операций в АЛУ

Абстрактный пример эксплуатации



Выберем гаджеты для загрузки некешированных данных. Мы столкнёмся:

- в случае JIT — создание гаджетов, в других случаях гаджеты следует искать,
- что хранится в кэше на данный момент.

В whitepaper и в PoC даны **примеры для конкретных процессоров**.

Абстрактный пример эксплуатации

Требуется найти гаджеты, которые позволят **считать необходимую закрытую информацию** в ходе спекулятивного выполнения.

Требуется для атаки с помощью техники спекулятивного выполнения:

1. Примитив спекулятивного выполнения
2. Гаджеты для создания «окна» спекулятивного выполнения
3. Гаджеты обнародования информации

Абстрактный пример эксплуатации

Требуется для атаки с помощью техники спекулятивного выполнения:

1. Примитив спекулятивного выполнения
2. Гаджеты для создания «окна» спекулятивного выполнения
3. Гаджеты обнародования информации

- ▶ ASLR
- ▶ CFI
- ▶ SMAP
- ▶ DEP/NX
- ▶ retpoline
- ▶ И т. д.

Для применения необходимых гаджетов требуется обойти некоторые системы защиты.

Абстрактный пример эксплуатации

В whitepaper и в PoC все защиты отключены.



Вавилонская башня
атаки на основе спекулятивного выполнения

Абстрактный пример эксплуатации

Требуется возможность **считать полученную** через сторонний канал информацию или **удостовериться**, что она была считана.

Требуется для атаки с помощью техники спекулятивного выполнения:

1. Примитив спекулятивного выполнения
2. Гаджеты для создания «окна» спекулятивного выполнения
3. Гаджеты обнародования информации
4. **Примитив обнародования информации**

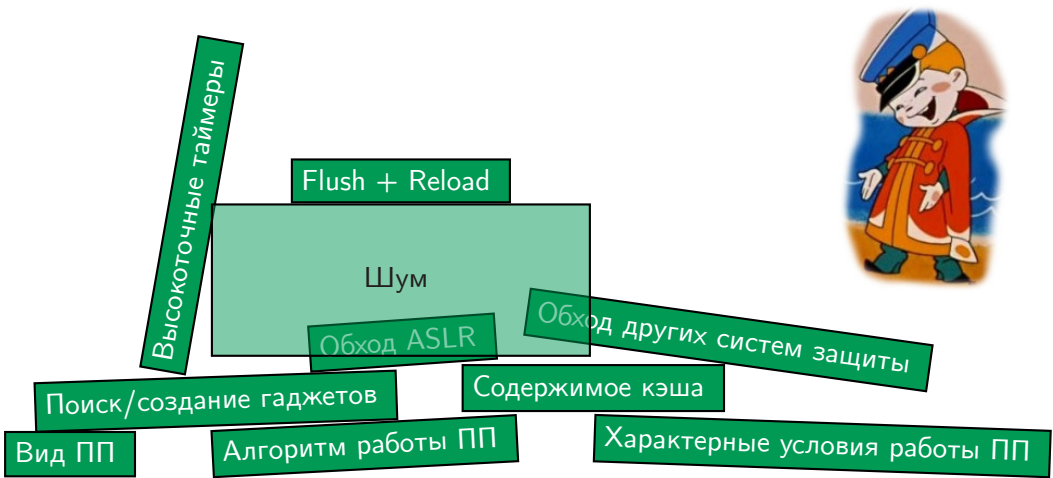
Абстрактный пример эксплуатации

Требуется для атаки с помощью техники спекулятивного выполнения:

1. Примитив спекулятивного выполнения
2. Гаджеты для создания «окна» спекулятивного выполнения
3. Гаджеты обнародования информации
4. **Примитив обнародования информации**

- ▶ Устройство кэша
- ▶ Правила вымещения из кэша
- ▶ Эксклюзивность и инклюзивность
- ▶ Тип атаки
- ▶ Зашумлённость
- ▶ Счётчики
- ▶ И т. д.

- Требуется знать, какой тип кэша используется.
- Требуется знать, какие данные сейчас хранятся в кэше, как выталкивать.
- На какой кэш будет направлена атака.
- Возможность проведения той или иной атаки.
- Возможность многократного повторения атаки.
- Для измерения времени требуются высокоточные счётчики.



Вавилонская башня
атаки на основе спекулятивного выполнения

Заключение

- ▶ атаки на микроархитектуру становятся **популярными**

Всё больше исследований проводится в этой области, всё больше обычных обывателей интересуются данной проблемой. Уязвимости в ПО **всё сложнее эксплуатировать, переходим к железу.**

Уязвимости находят, **прочитав и разобравшись в спецификации архитектуры**, процессора. Обратную разработку производят с помощью базовых атак по сторонним каналам.

Заключение

- ▶ атаки на микроархитектуру становятся **популярными**
- ▶ требуется **много ресурсов** для разработки эксплоита

Не смотря на все многочисленные плюсы (для атакующего), **разработка эксплоита** для широкого спектра программного и аппаратного ПО — **весьма затруднительна**.

Заключение

- ▶ атаки на микроархитектуру становятся **популярными**
- ▶ требуется **много ресурсов** для разработки эксплоита
- ▶ атаки на микроархитектуру могут быть **автоматизированы**

Представлено множество работ и инструментов, позволяющих провести атаку практически на любую популярную архитектуру. **Создаются эксплоит-паки**, содержащие атаки на микроархитектуру.

Заключение

- ▶ атаки на микроархитектуру становятся **популярными**
- ▶ требуется **много ресурсов** для разработки эксплоита
- ▶ атаки на микроархитектуру могут быть **автоматизированы**
- ▶ множество атак ещё **не опубликовано/найдено**

Описание атак, использующих спекулятивное выполнение, ещё **не до конца опубликованы**.

Множество возможных **изъянов микроархитектуры не найдены**.

Заключение

- ▶ атаки на микроархитектуру становятся **популярными**
- ▶ требуется **много ресурсов** для разработки эксплоита
- ▶ атаки на микроархитектуру могут быть **автоматизированы**
- ▶ множество атак ещё **не опубликовано/найдено**
- ▶ создание контрмер — **не тривиальный процесс**

Для исправления сложившейся ситуации требуются **фундаментальные изменения** в ходе работы процессора.

Исправления, **разработанные на уровне ОС**, требуют **детального изучения уязвимости** и алгоритма противодействия, к тому же **не всегда возможно предотвратить** эксплуатацию на уровне ОС, а если и удаётся, то в большинстве случаев приносят **в жертву процессы оптимизации**.


Источники I


 [Daniel Gruss](#)
Software-based Microarchitectural Attacks.

 [Moritz Lipp, Daniel Gruss](#)
ARMageddon: Cache Attacks on Mobile Devices.


 [D. Page](#)
MASCAB: a Micro-Architectural Side-Channel Attack Bibliography.


 [Pessl P., Gruss D. and others](#)
DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks.


 [Bos H., Fratantonio Y. and others](#)
Drammer: Determenistic Rowhammer Attacks on Mobile Platforms.


 [Microsoft](#)
Mitigating speculative execution side channel hardware vulnerabilities.


Источники II

 [Google Project Zero](#)
Reading privileged memory with a side-channel.

 [Daniel Gruss, Moritz Lipp](#)
KASLR is Dead: Long Live KASLR.

 [Daniel Gruss, Clémentine Maurice and others](#)
Flush+Flush: A Fast and Stealthy Cache Attack.

 [Fangfei Liu, Yuval Yarom and others](#)
Last-Level Cache Side-Channel Attacks are Practical.

 [Caroline Trippel, Daniel Lustig, Margaret Martonosi](#)
MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols.

Источники III



[Michael Schwarz, Clémentine Maurice, Daniel Gruss, Stefan Mangard](#)

Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript.



[Moritz Lipp, Misiker Tadesse Aga and others](#)

Nethammer: Inducing Rowhammer Faults through Network Requests.



[Andrei Tatar, Radhesh Krishnan and others](#)

Throwhammer: Rowhammer Attacks over the Network and Defenses.



[Giovanni Camurati, Sebastian Poeplau and others](#)

Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers.



[Julian Stecklina, Thomas Prescher](#)

LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels.



[Mordechai Guri, Assaf Kachlon and others](#)

GSMem: Data Exfiltration from Air-Gapped Computers over GSM Frequencies.

Источники IV

 [Dean Sullivan, Orlando Arias, Travis Meade, Yier Jin](#)

Microarchitectural Minefields: 4K-Aliasing Covert Channel and Multi-Tenant Detection in IaaS Clouds.

 [B. Gras, K. Razavi, E. Bosman, H. Bos, C. Giuffrida](#)

ASLR on the Line: Practical Cache Attacks on the MMU.

 [van Schaik, S. Giuffrida, C. Bos, H. Razavi, K.](#)

Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think.

 [Daniel Gruss, Anders Fogh and others](#)

Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR.

 [Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh and others](#)

Spectre Returns! Speculation Attacks using the Return Stack Buffer.

Источники V

 [Giorgi Maisuradze, Christian Rossow](#)

ret2spec: Speculative Execution Using Return Stack Buffers.

 [Guoxing Chen, Sanchuan Chen and others](#)

SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution.

 [Moritz Lipp, Michael Schwarz and others](#)

Meltdown.

 [Paul Kocher, Daniel Genkin and others](#)

Spectre Attacks: Exploiting Speculative Execution.

 [ARM Whitepaper](#)

Cache Speculation Side-channels.

 [Michael Schwarz, Martin Schwarzl, Moritz Lipp, Daniel Gruss](#)


NetSpectre: Read Arbitrary Memory over Network.


Источники VI


 [Sophia D'Antoine](#)
Out-of-Order Execution and Its Applications.

 [Vladimir Kiriansky, Carl Waldspurger](#)
Speculative Buffer Overflows: Attacks and Defenses.

 [Gras, B. Razavi, K. Bos, H. Giuffrida, C.](#)
Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks.

 [Craig Disselkoen, David Kohlbrenner, Leo Porter, Dean Tullsen](#)
Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX.

 [Moritz Lipp, Michael Schwarz](#)
Meltdown & Spectre Side-channels considered hARMful.

 [Jon Masters](#)
Exploiting modern microarchitectures: Meltdown, Spectre, and other attacks.