Атаки по сторонним каналам на микроархитектуру, основанные на исполнении кода

Digital Security
2018

План

План

План

Дисклеймер

- я не бывший инженер Intel, ARM, AMD и др.

Я не знаю, **как на самом деле всё работает**, я просто брал информацию из открытых источников.

Дисклеймер

- я не бывший инженер Intel, ARM, AMD и др.
- я нуб в данном вопросе

На исследование **я потратил около 1,5 недели**, тема же требует много лет изучения.

Дисклеймер

- я не бывший инженер Intel, ARM, AMD и др.
- я нуб в данном вопросе
- вся информация абстрактна, деталей не будет

Доклад очень обширный, **невозможно всё охватить**, хотите деталей, могу дать источники.

Дисклеймер

- я не бывший инженер Intel, ARM, AMD и др.
- я нуб в данном вопросе
- вся информация абстрактна, деталей не будет
- я могу привирать

Возможно, я что-то не правильно понял, поэтому информация будет неверной, но **вы всё равно не узнаете где правда, а где ложь**.
Задавать вопросы сразу!

План

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**.

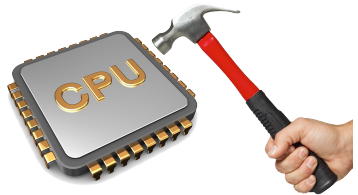
Самый примитивный пример атаки по сторонним каналам — **определение нажатых кнопок сейфа по звуку** при введении секретного кода.

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.

План

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

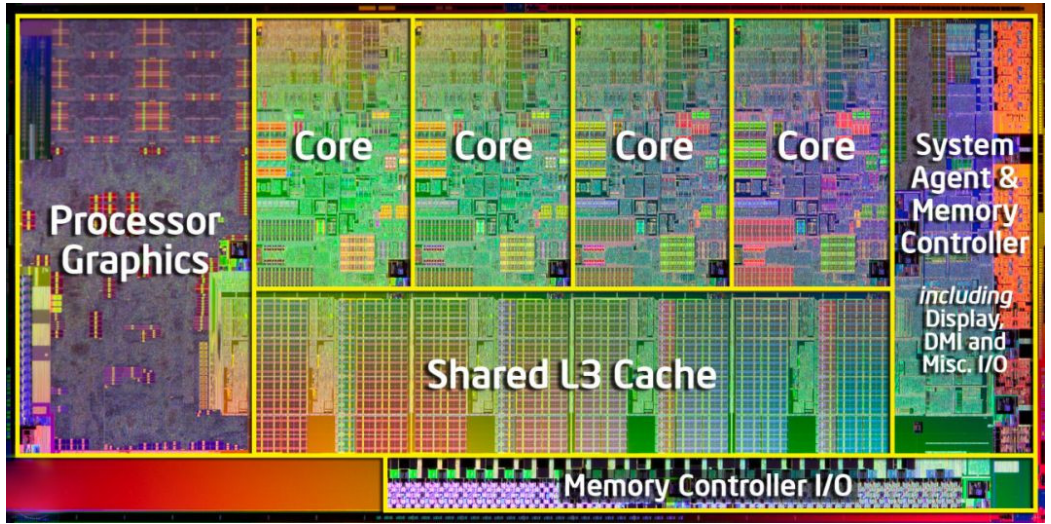
Атаки по сторонним каналам на микроархитектуру, основанные на использовании программного обеспечения, как правило, даже **не требуют физического доступа** к вычислительному устройству.
Также существуют атаки, **основанные на дефектах микроархитектуры**, например, ошибки, происходящие **во время оптимизации**.
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.

План

Теория
- Процессор
- Кэш–память
- DRAM
- Виртуальная память

План

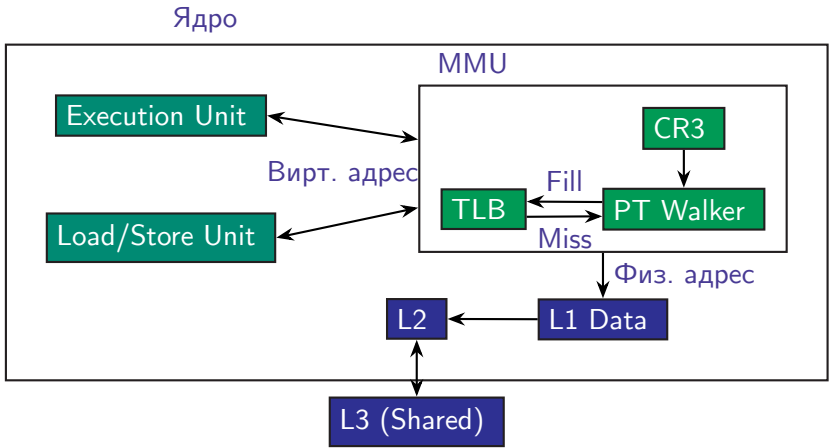
Процессор



Архитектура многоядерного процессора

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

Процессор



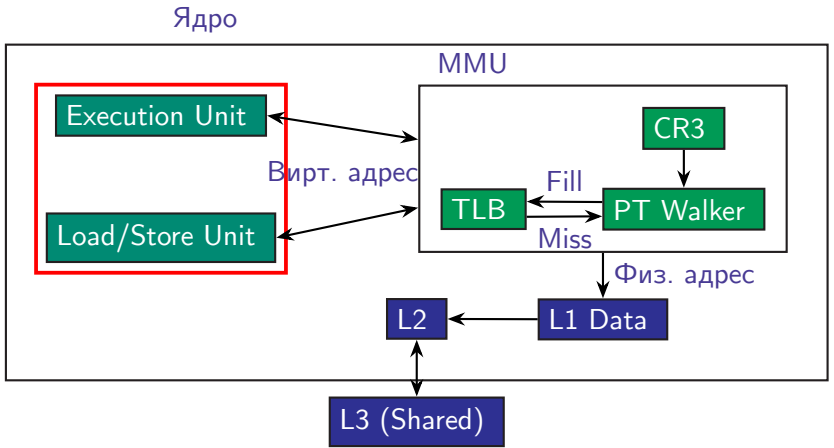
Абстрактная архитектура элементов ядра, работающих с данными

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 распараллеливанные 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**.

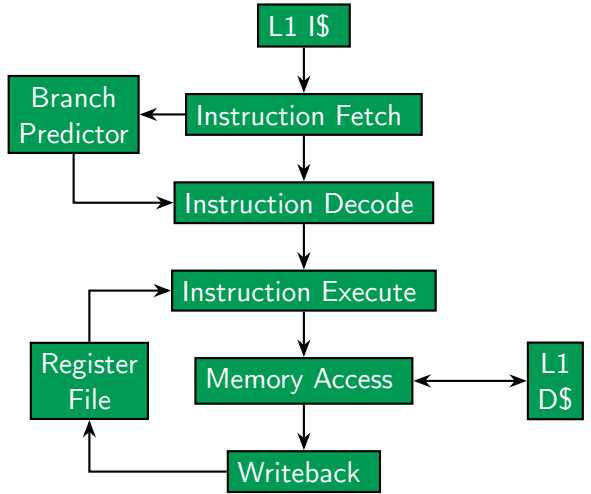
Процессор



Абстрактная архитектура элементов ядра, работающих с данными

Ниже **будет рассказано** о работе этих элементов.

Конвейеризация. По порядку

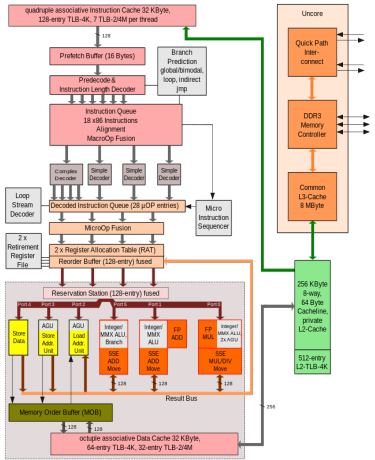


Элементы системы выполнения современного процессора (выполнение по порядку)

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.

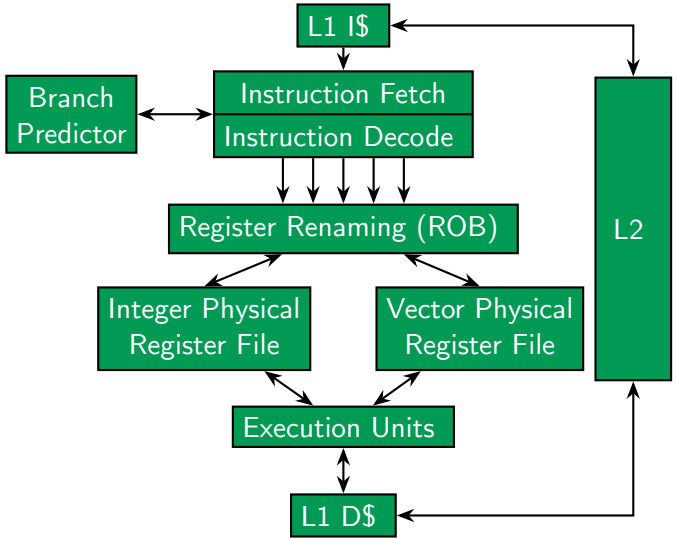
Конвейеризация. Не по порядку



Микроархитектура Intel Nehalem в 4-х ядерной реализации

На слайде представлено более полное описание микроархитектуры, но мы же **рассмотрим упрощённую версию**.

Конвейеризация. Не по порядку

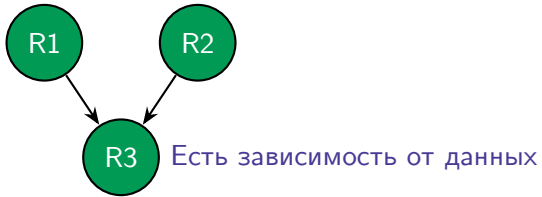


Элементы системы выполнения современного процессора (выполнение инструкций не по порядку)

Рисунок 7, **выполнение инструкций не по порядку**.
Инструкции получаются 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

R1 = LOAD A
R2 = LOAD B
R3 = R1 + R2
R1 = 1
R2 = 2
R3 = R1 + R2



Пример выполнения инструкций не по порядку

Представлен пример кода, который имеет и зависимые, и независимые от других данных участки. **На следующем слайде** будет представлено, как RoB обрабатывает такой код.

Конвейеризация. Не по порядку

R1 = LOAD A
R2 = LOAD B
R3 = R1 + R2
R1 = 1
R1 = 2
R3 = R1 + R2

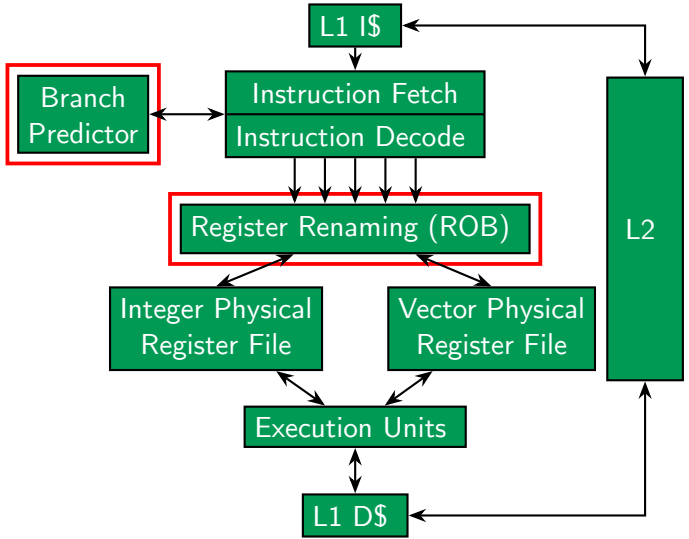
Порядок выполнения

№	Имя регистра	Инструкция	Зависимости	Готово?
1	P1 = R1	P1 = LOAD A	-	-
2	P2 = R2	P2 = LOAD B	-	-
3	P3 = R3	P3 = P1 + P2	1, 2	-
4	P4 = R1	P4 = 1	-	+
5	P5 = R2	P5 = 1	-	+
6	P6 = R3	P6 = P4 + P5	4, 5	-

Re-Order Buffer (ROB)

RoB **внутри себя производит трансляцию** в требуемый ему вид, переименовывает регистры, реорганизует опкоды.
Именно по этой причине код может исполняться внутри процессора не по порядку.

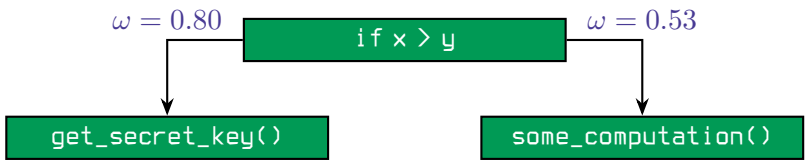
Конвейеризация. Не по порядку



Элементы системы выполнения современного процессора (выполнение инструкций не по порядку)

Рисунок 7, **выполнение инструкций не по порядку**.
Инструкции получаются 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.

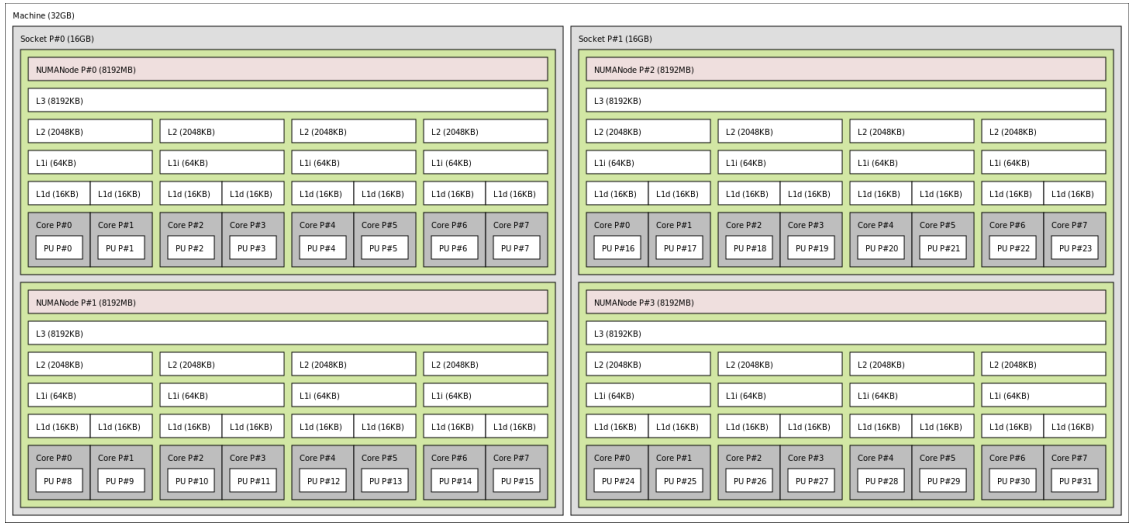
Оптимизатор потока инструкций



get_secret_key() может выполниться спекулятивно

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 выполниться 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**. Другая идея заключается в том, что процессор выполняет инструкции не по порядку: инструкции, которые зависят от каких-либо данных ожидают своего выполнения, пока инструкции, которые ни от чего не зависят выполняются.

Многоядерность

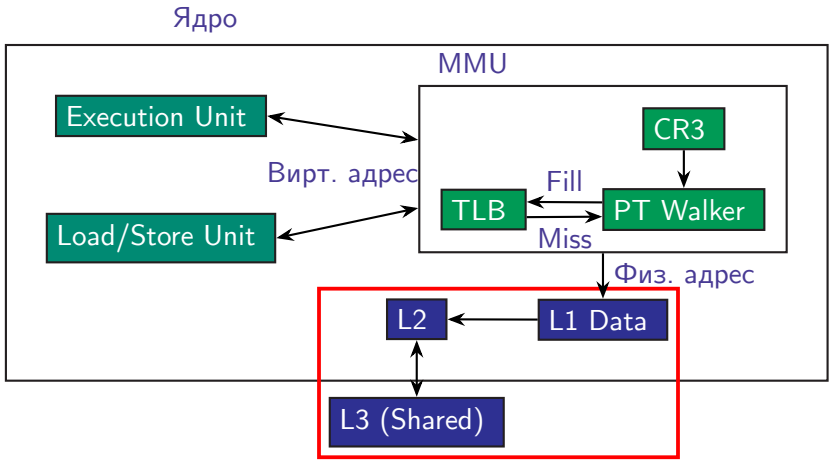


Архитектура многоядерного процессора AMD Bulldozer

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 исполнятся 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

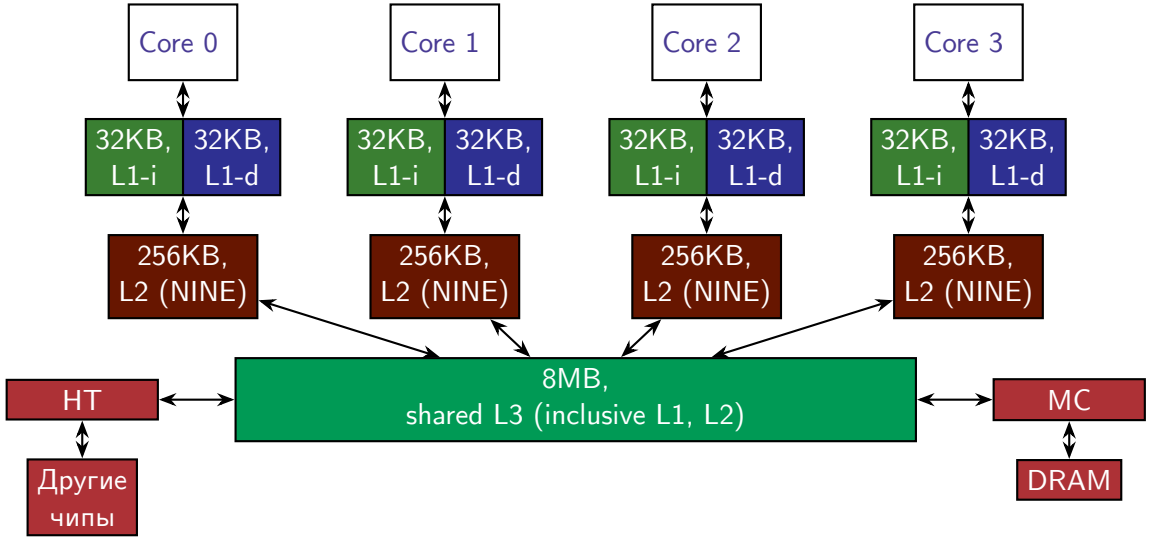
Кэш–память



Абстрактная архитектура элементов ядра, работающих с данными

Ниже **будет рассказано** о работе этих элементов.

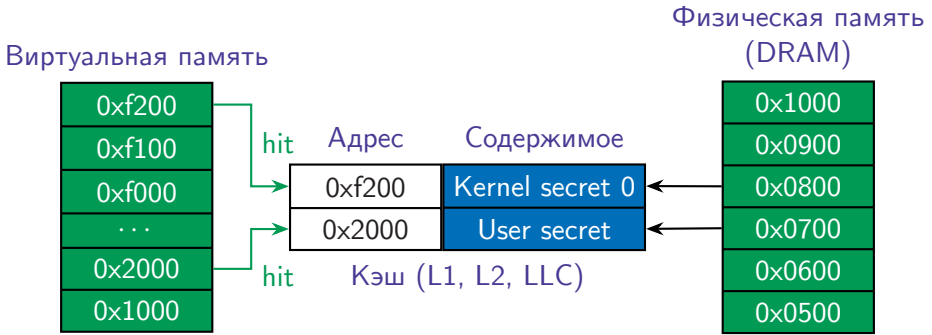
Кэш–память



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

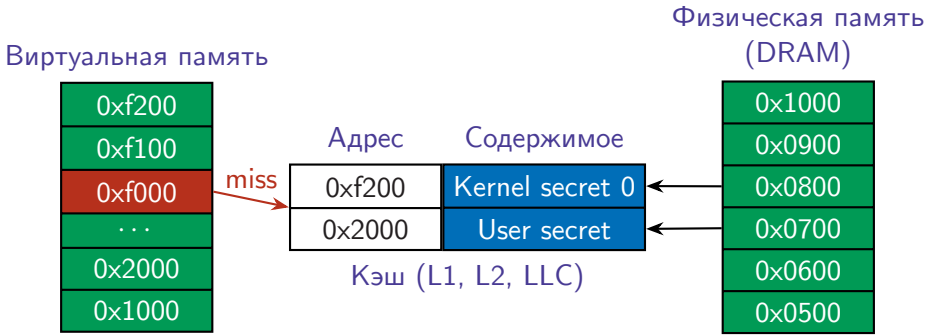


Пример взаимодействия с кэшем

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

Кэш–память

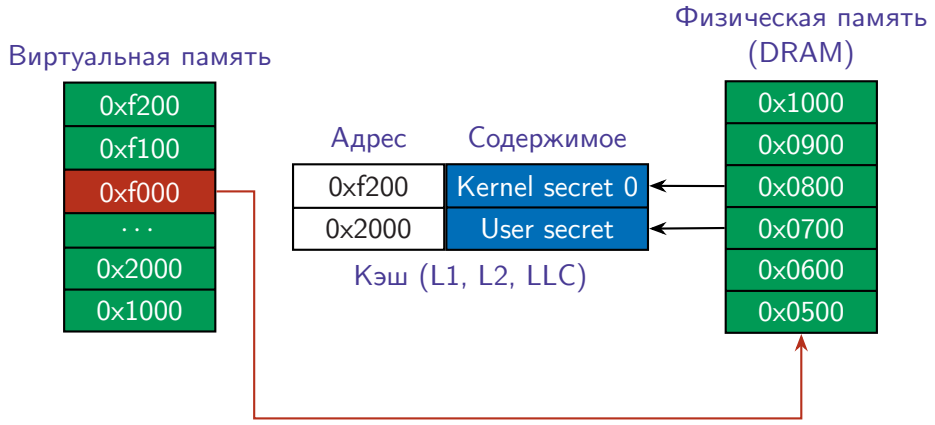
В противном случае происходит **промах кэша** (*cache miss*).



Пример взаимодействия с кэшем

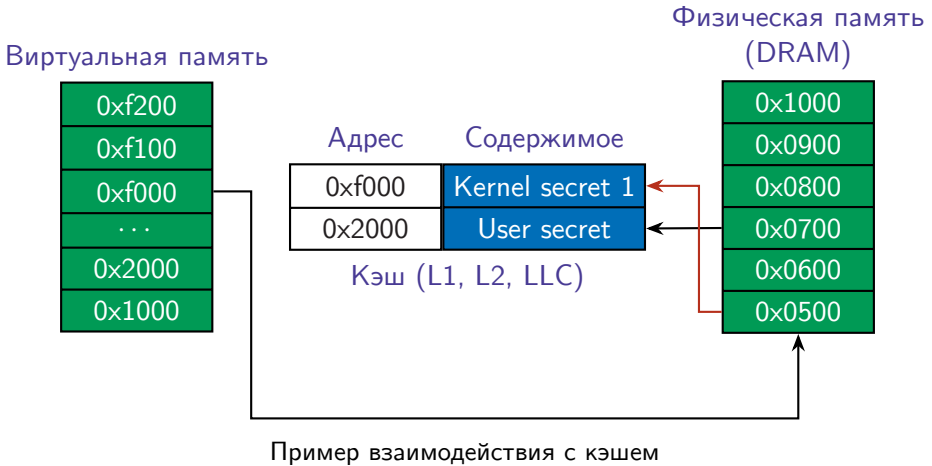
Кэш–память

Данные берутся из медленной памяти.



Пример взаимодействия с кэшем

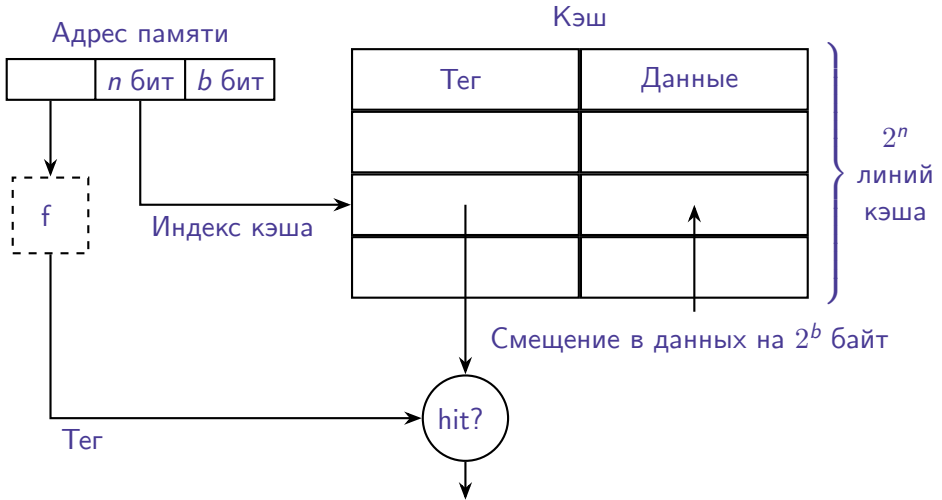
Кэш–память



Пример взаимодействия с кэшем

В кэш записываются новые данные.

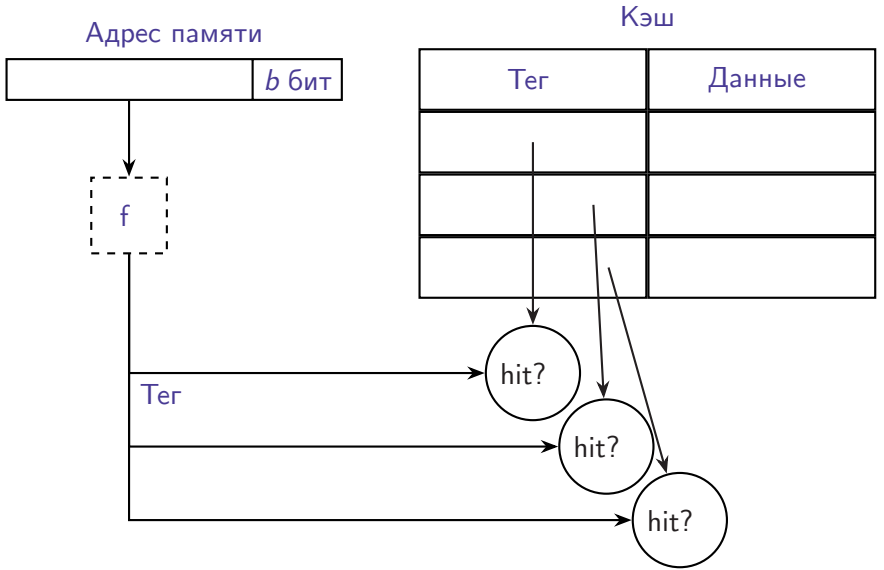
Кэш с прямым отображением



Кэш состоит из 2^n **линий кэша**, каждая линия содержит **тег** и 2^b байт **ассоциированных данных**. Тег вычисляется из соответствующего адреса памяти, который добавляется в эту кэш линию. **Тег используется** в дальнейшем для того, чтобы определять **присутствие** того или иного адреса в линии кэша. Последние b бит адреса **используются** в качестве **смещения для данных** в линии кэша. Современные процессоры имеют длину линии кэша в *64 байта*, т. е. b = 6. **Средние** n бит адреса памяти **используются в качестве *индекса кэша***, который говорит о номере линии кэша, в котором содержатся данные.

Размер кэша определяет, как много бит будет использовано, т. е. как много индексов будет использовано. **Адреса с теми же *n* битами** являются **конгруэнтными**, так как они отображают те же линии кэша. 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.

Полностью ассоциативный кэш

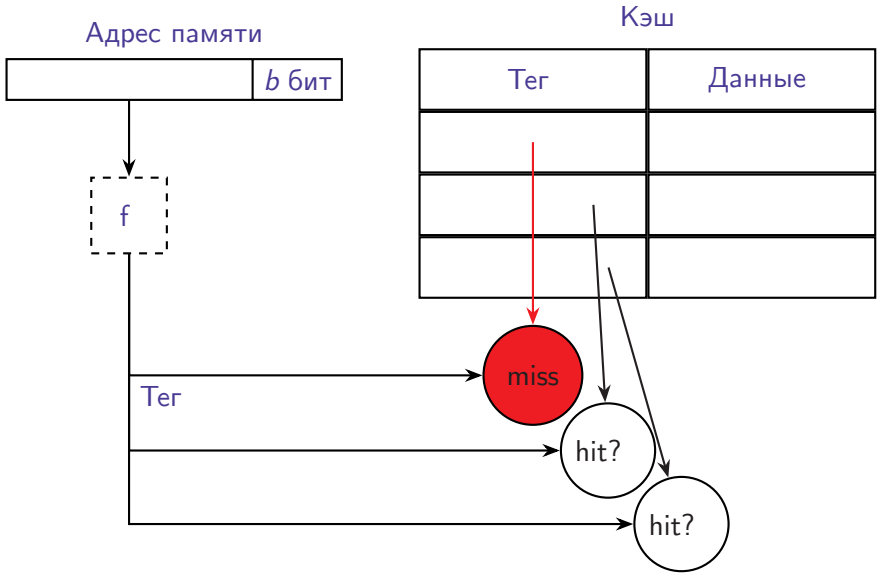


Проблема конгруэнтности **решается в полностью ассоциативном кэше**.

Такой вид кэша не содержит индексов и каких-либо линий кэша. Вместо этого он хранит множество **путей кэша**, которые в свою очередь содержат данные. **Тег** теперь **используется для определения существования адреса в кэше** и какой именно путь кэша содержит ассоциированные данные.

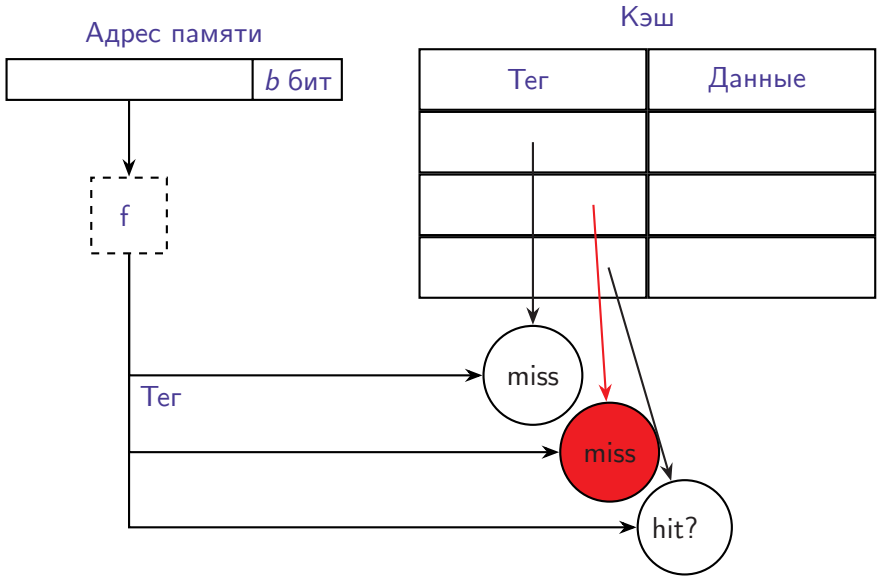
Пример на следующих слайдах!

Полностью ассоциативный кэш



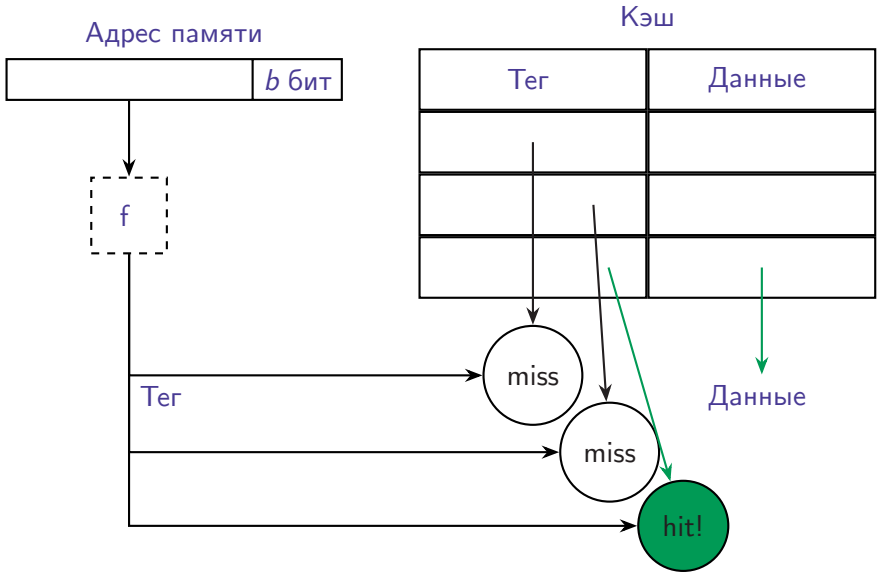
Такие кэши становятся более **дорогими с увеличением путей**. Поэтому они обычно содержат небольшое количество путей, например, в современных процессорах используются **буферы ассоциативной трансляции (translation-lookaside buffers TLB)** с 64 путями.

Полностью ассоциативный кэш



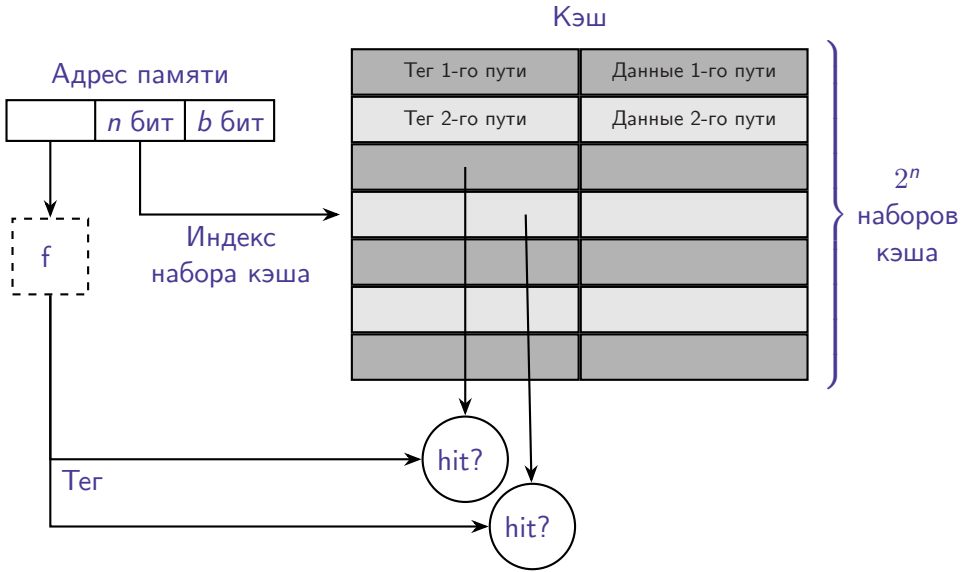
Такие кэши становятся более **дорогими с увеличением путей**. Поэтому они обычно содержат небольшое количество путей, например, в современных процессорах используются **буферы ассоциативной трансляции (translation-lookaside buffers TLB)** с 64 путями.

Полностью ассоциативный кэш



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 **кеш 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.

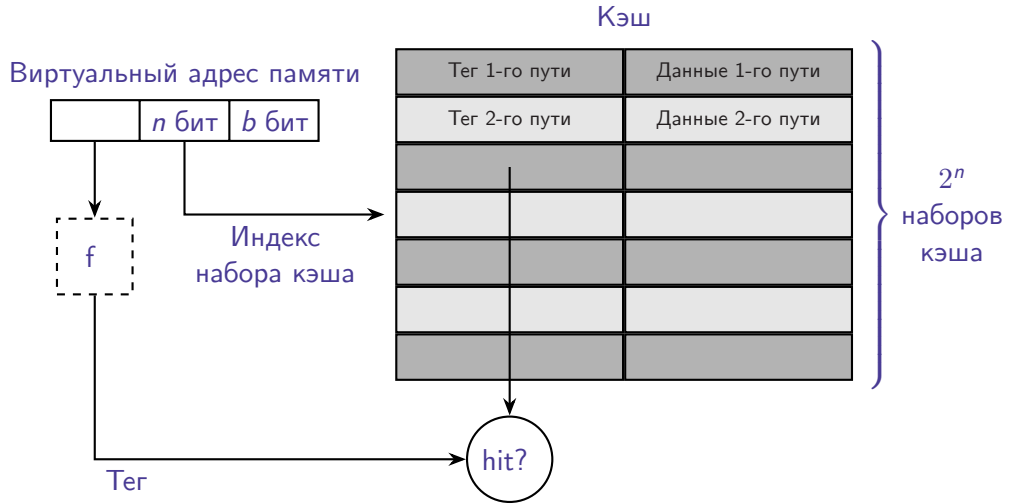
Количество путей или линий в кэше ограничено, а **конгруэнтных адресов**, которые требуется хранить, — **достаточно много**, требуется производить замены данных в кэше на новые, полученные из главной памяти.

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. VIVT



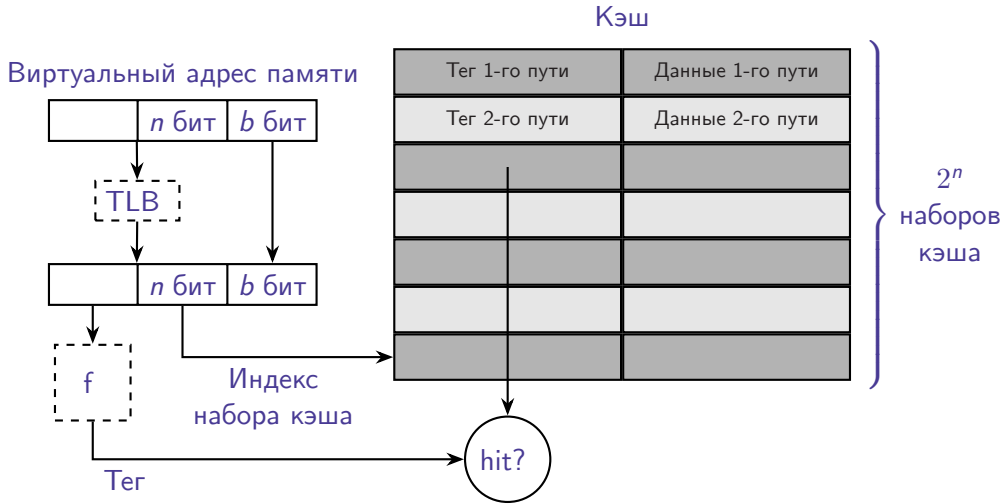
Виртуальная индексация виртуальное тагетирование

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

- Виртуальный тег не уникален при переключении контекста — **данные не могут быть разделяемыми**
- Трансляция адреса не происходит — **быстрая скорость работы**

Виртуальная индексация виртуальное тагетирование (virtually-indexed virtually-tagged VIVT). Используется для маленьких данных, с которыми производятся быстрые операции, в **ARM процессорах** используются в качестве **кэша инструкций**.

Режимы адресации. PIPT



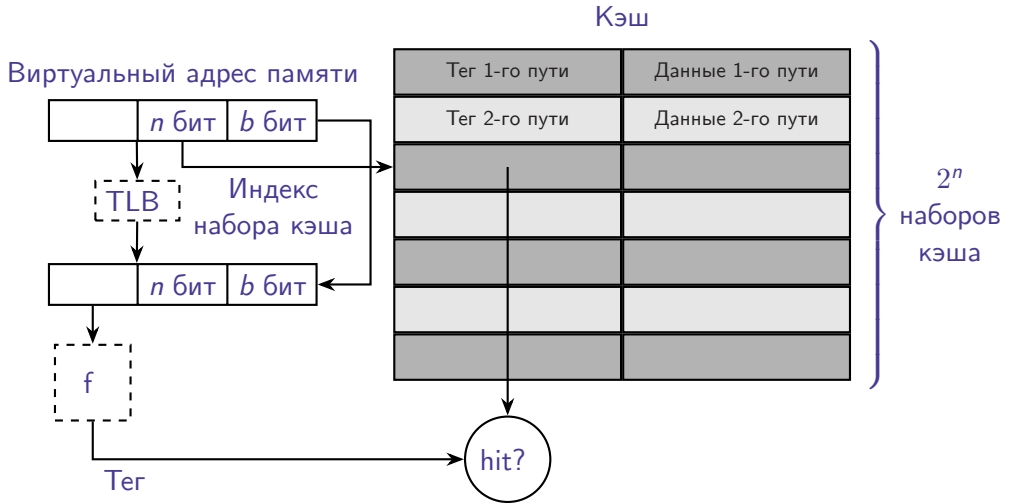
Физическая индексация физическое тагетирование

Физическая индексация физическое тагетирование (physically-indexed physically-tagged PIPT). Используется **тег и индекс из физического адреса**.

- Тег будет уникальным даже при смене контекста — **разделяемая память будет реально разделяемой**
- Смена контекста — **большие задержки**

Используется для кэшей данных и инструкций, задержка по большей части уменьшается посредством использования кэшей в системе трансляции адресов (TLB).

Режимы адресации. VIPT



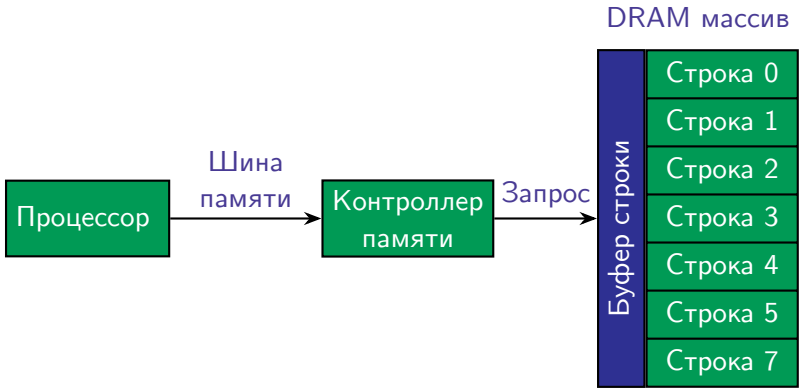
Виртуальная индексация физическое тагетирование

Виртуальная индексация физическое тагетирование (virtually-indexed physically-tagged VIPT).
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

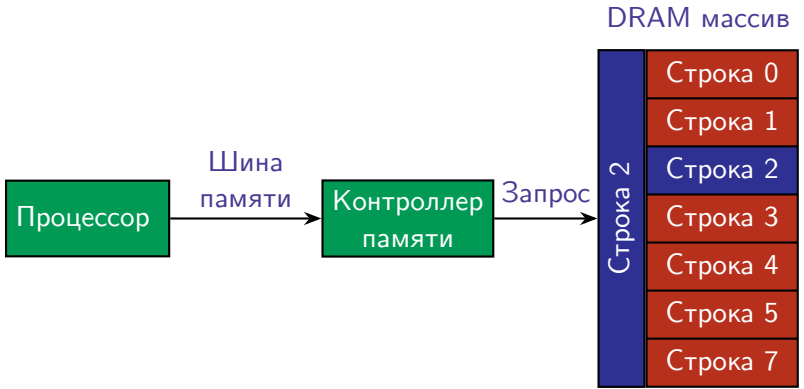
Алгоритм работы



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

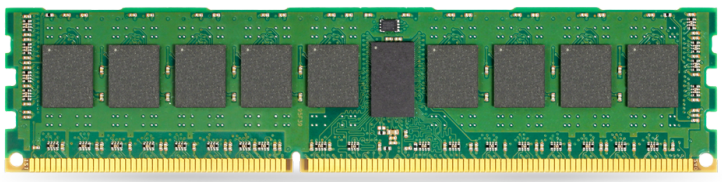


Простая компьютерная система с единственным DRAM массивом

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

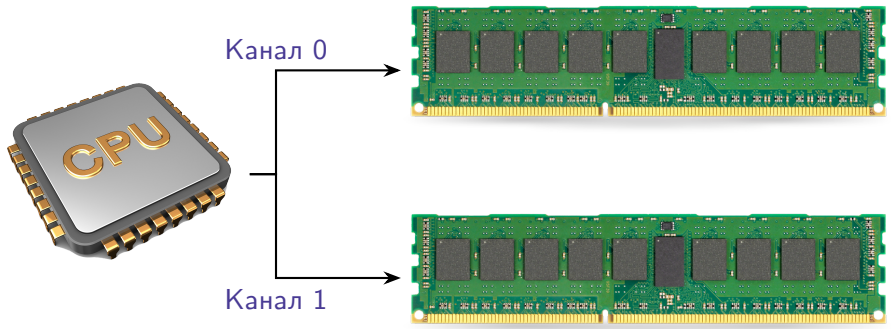
DIMM



Архитектура DRAM

Для повышения производительности работы с DRAM были использованы те же методы, что и в случае с кэшем. Современные компьютерные системы организовывают 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**.

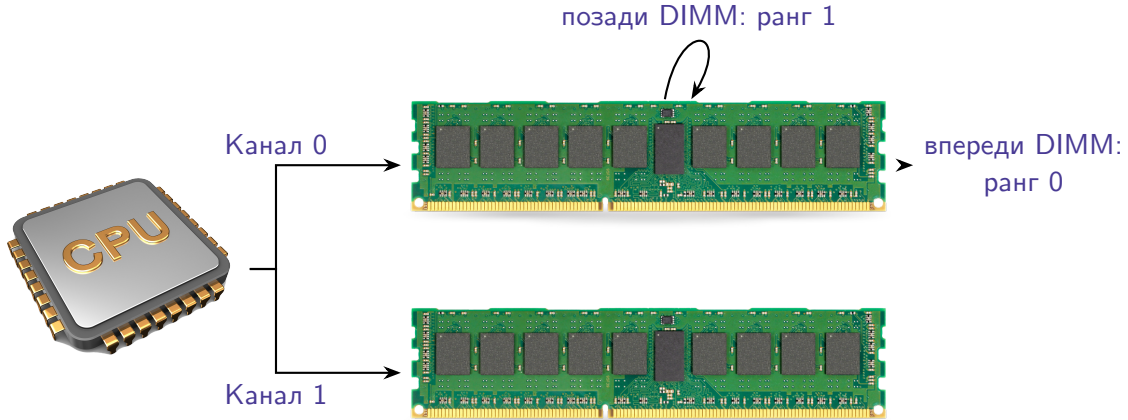
Физическое строение



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.

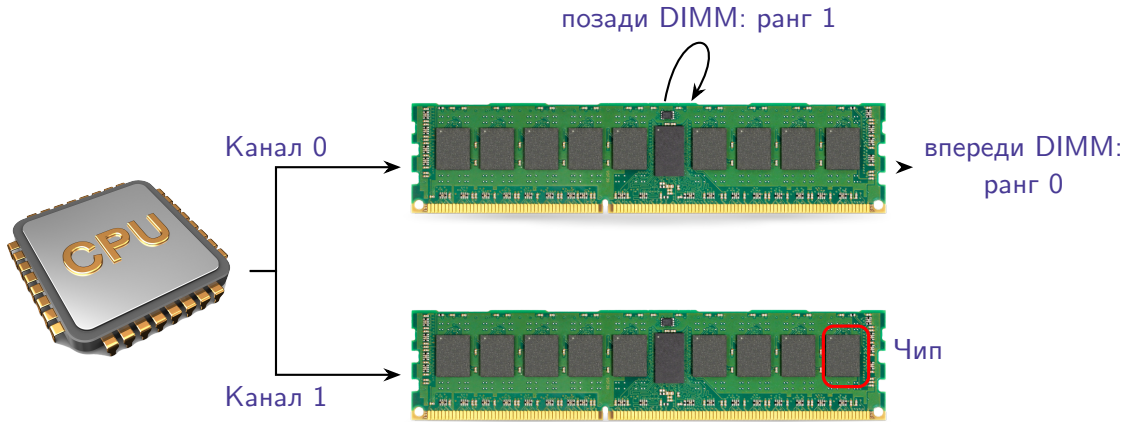
Физическое строение



Архитектура DRAM

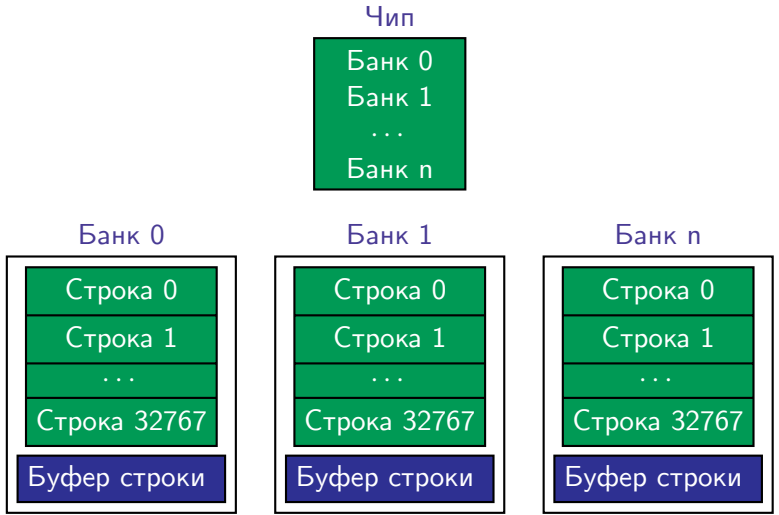
Современные модели DRAM имеют обычно от **1 до 4 рангов**, количество которых увеличивается при перемножении с количеством банков. Такого рода параллелизм позволяет **уменьшить промах строки**.

Физическое строение



Архитектура DRAM

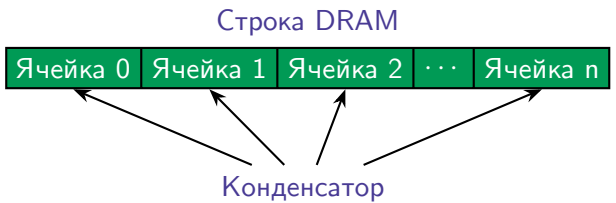
Физическое строение



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



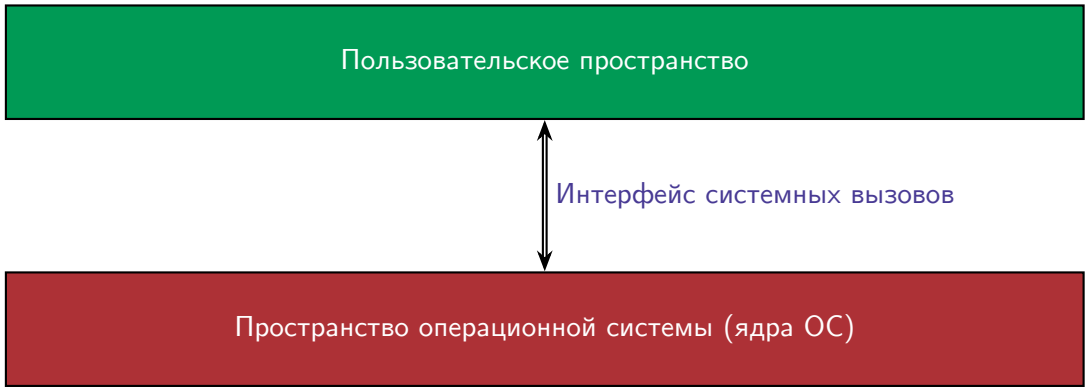
Архитектура DRAM

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

Изолирование памяти

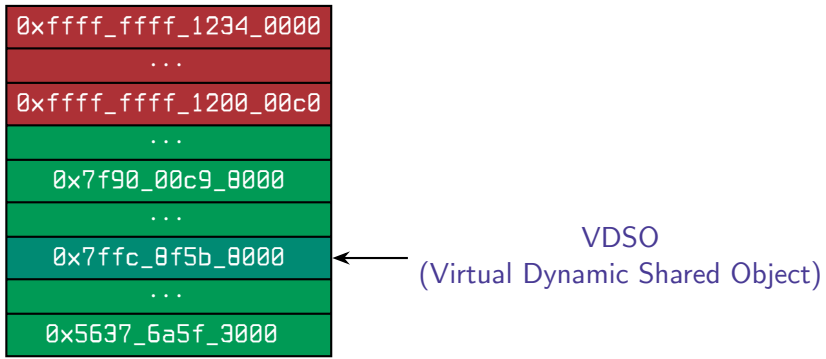


Система изолирования памяти

Так как мультипроцессорная обработка данных становится всё популярнее, то и **задача изолирования памяти различных процессов** также становится актуальнее.
Существует программная изоляция в виде разделения на **пользовательское пространство** и **пространство** операционной системы (**ядра ОС**).

Изолирование памяти

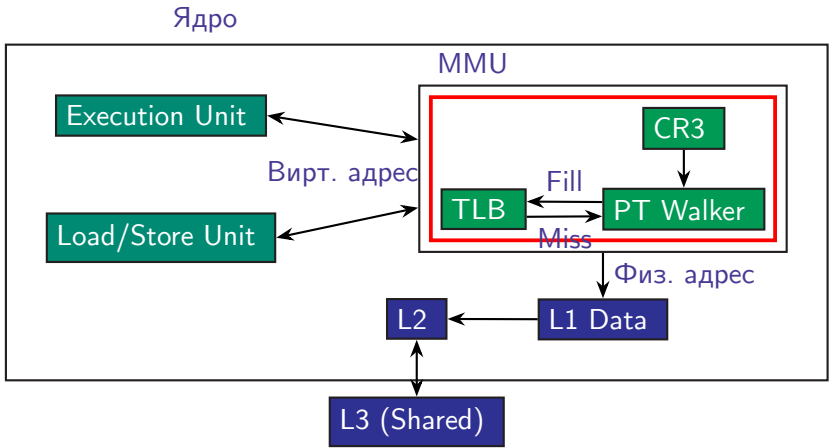
```
$ cat /proc/self/maps
```



Пример карты памяти для процесса `cat`

Также на первых порах была введена такая технология, как **виртуальная память**, которая представляла из себя **изолированные сегменты, ссылающиеся на физическую память**. На уровне виртуальной памяти операционная система могла иметь определённые права доступа к конкретному сегменту по конкретному смещению. Соответственно, **различные процессы** использовали **различные сегменты виртуальной памяти**.

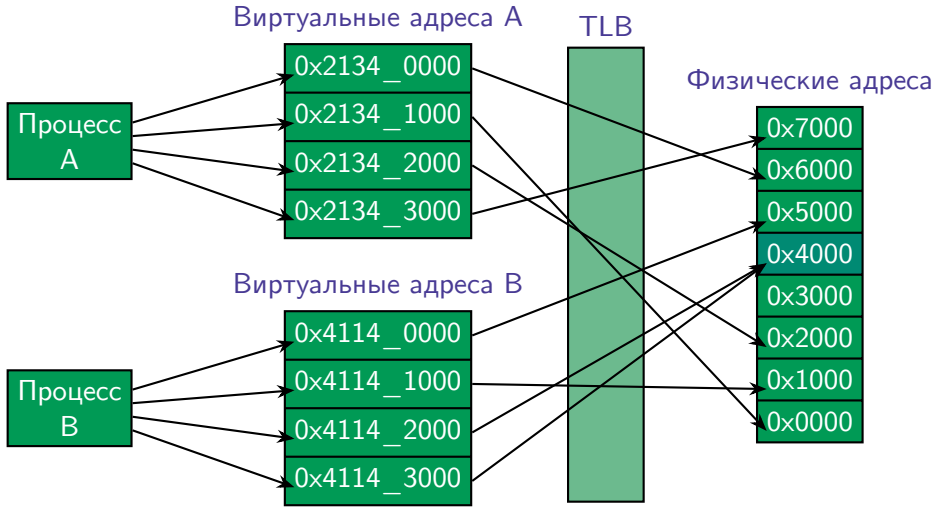
Виртуальная память



Абстрактная архитектура элементов ядра, работающих с данными

Ниже **будет рассказано** о работе этих элементов.

Трансляция адресов

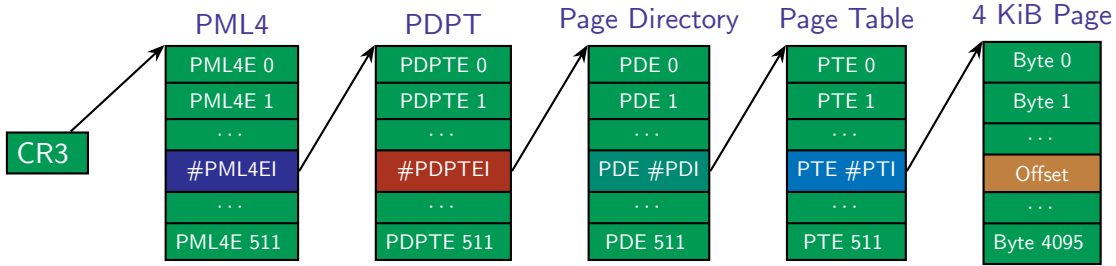


Пример трансляции адресов (0x4000 — разделяемая страница памяти)

В итоге существует **два понятия**: **виртуальный адрес** памяти, который доступен процессу и **физический адрес** памяти, по которому процесс напрямую обращаться не может.

Вместо сегментов памяти, современные процессоры используют так называемые **страницы памяти**, которые представляют из себя **участки памяти фиксированного размера**. И виртуальная, и физическая память разбивается на подобные страницы, при этом страницы виртуальной памяти ссылаются на страницы физической с помощью указания номеров требуемых страниц. Размеры страниц памяти в современных процессорах варьируются, но самыми маленькими обычно являются страницы **размером 4KB или 1KB**. Страницы физической и виртуальной памяти выровнены **в соответствии с их размером**, т. е. страница физической памяти размером в 4KB выровнена со страницей виртуальной памяти такого же размера.

Трансляция адресов. x86-64



Трансляция адресного пространства для страниц в 4KB на x86-64 процессорах

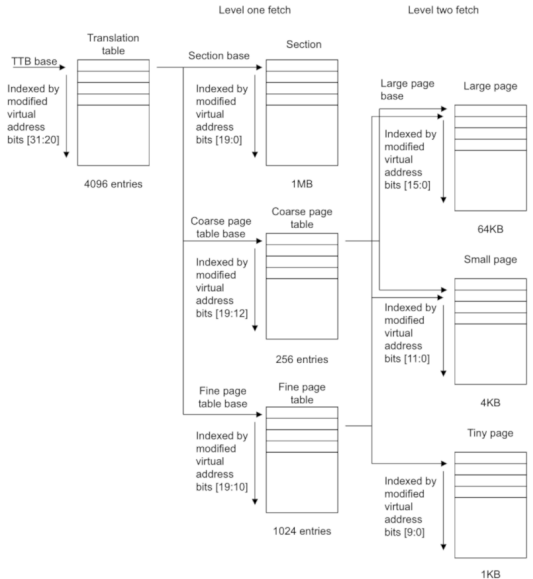
Массив, который отображает 48-битное пространство — занимает 512GB, что **не рационально**. Используется **многоуровневая трансляция**.

CR3 регистр изменяется в соответствии с **контекстом** выполнения, что обеспечивает **изоляцию**.

Современные процессоры Intel имеют 4 уровня трансляции адресов (см. рисунок 22]).

1. PML4 (page map level) — 48-битное виртуальное адресное пространство на 512 регионов по 512GB.
2. Таблица указателей директории страниц PDPT (page directory pointer tables) — на 512 записей по 1GB виртуальной памяти. Эта 1GB виртуальная страница может напрямую ссылаться на 1GB страницу или страницу директорий PD (page directory).
3. PD — 512 ячеек по 2MB.
4. PT (page table) — 4KB.

Трансляция адресов. ARM



Что касается ARM архитектуры, то там складывается похожая картина. Есть несколько отличий: используются **отображения других размеров** (1MB — секции, 64KB — большие страницы, 4KB — маленькие страницы и 1KB — крошечные страницы, на поздних версиях большего размера); используется регистр c2 (часть `CP15` регистров) (**Translation Table Base Register TTBR**) для получения указателя на таблицу в физическом адресном пространстве, содержащую описания секции или страницы (или и того и другого); в отличии от Intel используется **двухуровневая трансляция** (см. рисунок 23) и другие. На современных ARM процессорах (Cortex-A) используется два регистра для хранения физического адреса таблицы трансляции адресов (`TTBR0` и `TTBR1`). Обычно один из них используется для **пользовательского пространства**, а другой для **пространства адресов ядра**. Это может служит причиной, почему некоторые атаки работают на x86-64 процессорах, но **не работают на ARM**. При попытке доступа из пользовательского контекста выполнения к регистру, который отвечает за адресацию в ядерном пространстве, возникает исключение.

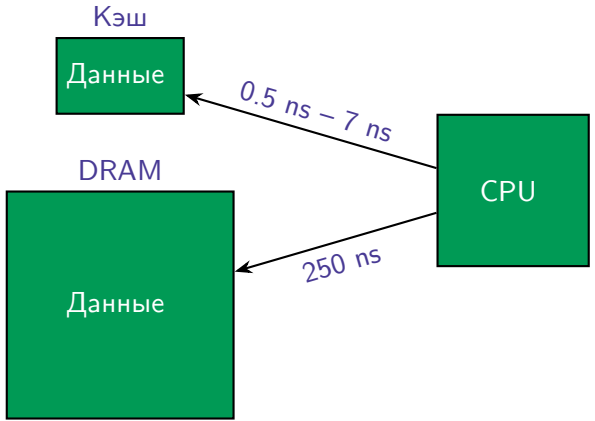
План

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

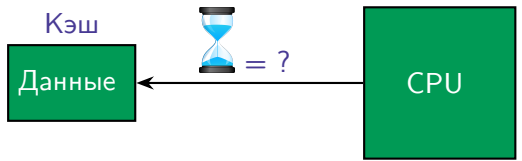
Атаки на кэш



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**.

Атаки на кэш



Для атаки на кэш необходимо знать точное время цикла обращения к ячейке памяти

Для проведения успешной кеш-атаки необходимо **знать точное время цикла обращения к ячейке памяти**. Ранние кеш–атаки использовали для этих целей **системные счётчики производительности**, но этот способ **неэффективен**, поскольку эти счётчики на ARM-процессорах доступны только в привилегированном режиме.

Однако в 2016 году были предложены **три альтернативных источника** синхронизации, доступные в том числе и в непривилегированном режиме. Один из них — запуск **параллельного синхронизирующего потока**, который непрерывно инкрементирует глобальную переменную. Читая значение этой переменной, злоумышленник может измерять время цикла обращения к ячейке памяти. Об остальных методах **будет рассказано позже** в ходе повествования.

Кроме того, в **ARM–процессорах** действует так называемая политика псевдослучайного замещения, в результате действия которой, **вытеснение из кеша происходит менее предсказуемо**, чем в процессорах Intel и AMD. Тем не менее в 2016 году была **продемонстрирована эффективная кеш-атака** даже в таких зашумленных условиях.

Evict + Time

1. Измерить время выполнения программы–жертвы
2. Вытеснить определённый набор кэша
3. Снова измерить время выполнения программы–жертвы и сравнить

Атакующий запускает несколько вычислений и затем измеряет время, потребовавшееся на завершение данных вычислений. Для определения влияния на конкретный набор кэша, атакующий предварительно вытесняет этот набор перед вторым запуском вычислений. Для второй половины запусков вычислений атакующий производит сравнение времени, и в случае, если время вычислений изменилось, значит именно этот набор кэша использовался для вычислений.

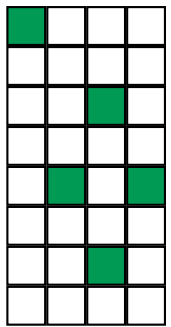
Evict + Time

Измерить время выполнения программы–жертвы

0.30 ms

```
long long *rsa_encrypt(...) {
  ...
  for (i = 0; i < size; i++){
    encrypted[i] = rsa_modExp(
      message[i],
      pub->exponent,
      pub->modules);
  }
  ...
}
```

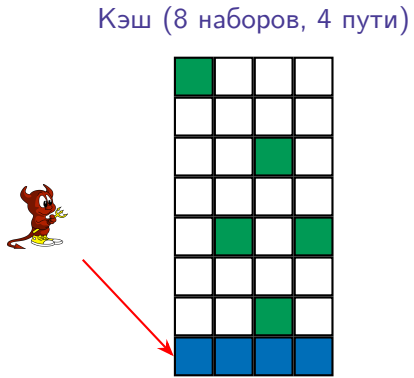
Кэш (8 наборов, 4 пути)



Кэш должен быть прогретым.

Evict + Time

Вытеснить определённый набор кэша



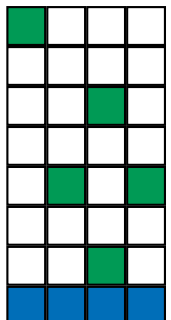
Evict + Time

Снова измерить время выполнения программы–жертвы и сравнить

0.28 ms

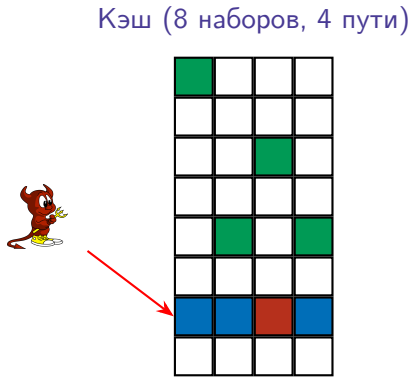
```
long long *rsa_encrypt(...) {
  ...
  for (i = 0; i < size; i++){
    encrypted[i] = rsa_modExp(
      message[i],
      pub->exponent,
      pub->modules);
  }
  ...
}
```

Кэш (8 наборов, 4 пути)



Evict + Time

Вытеснить определённый набор кэша



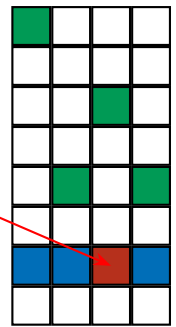
Evict + Time

Снова измерить время выполнения программы–жертвы и сравнить

0.56 ms

```
long long *rsa_encrypt(...) {
  ...
  for (i = 0; i < size; i++){
    encrypted[i] = rsa_modExp(
      message[i],
      pub->exponent,
      pub->modules);
  }
  ...
}
```

Кэш (8 наборов, 4 пути)



`Evict + Time` метод даёт детальную информацию об используемом наборе кэша, но **«шум»**, получаемый от других источников, мешает корректно вычислить время исполнения задачи. По этой причине может потребоваться **несколько тысяч повторений запусков** для получения, например, целого AES ключа. Данный метод требует от атакующего возможности **точного измерения времени начала и окончания вычислений**. Это может оказаться **неосуществимым** в случае выполнения атаки **асинхронно**, так как атакующий не имеет возможности запускать вычисления по своему усмотрению. Преимущество `Evict + Time` метода в том, что ему **не требуется работать с разделяемой памятью**.

Усложнённые адресация данных и правила вымещения из кэша на современных процессорах делает этап очищения данных из набора кэша сложнее и, соответственно, саму атаку с помощью метода `Evict + Time`.

Prime + Probe

1. Заполнить определённые наборы кэша
2. Передать управление программе–жертве
3. Определить какие наборы кэша всё ещё заполнены нашими данными

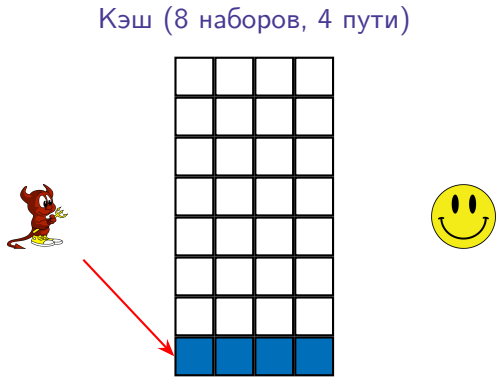
Вторая техника атаки представленная Osvik'ом оказалось более мощной. Атакующий в течении времени заполняет (primes) набор кэша, а затем измеряет как долго происходит повторный доступ к этим данным.

Первое применение атаки было **нацелено на L1 кэш**. Однако, **обратная разработка** работы **кэшей последнего уровня** открыла возможность проведения атаки и на данный вид кэшей (2009). Существует множество примеров успешной атаки на криптографические алгоритмы из состава всё того же OpenSSL.

2011 — был представлен пример атаки на **соседние облака и прослушивание соседних виртуальных машин**. Однако, эти атаки были совершены на микроархитектуры **с простой организацией**, где отсутствовали срезы кэшей и сложные функции по вычислению адресов. Позже — и на современные процессоры, в том числе атака на AES BouncyCastle (**ARM**), атака на **TrustZone**, **Intel SGX анклав**.

Prime + Probe

Заполнить определённые наборы кэша

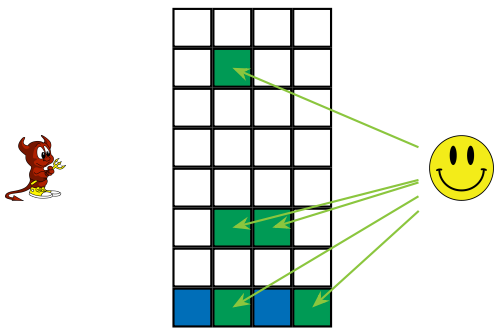


Атака проиллюстрирована в трёх шагах. Атакующий непрерывно заполняет набор кэша, используя доступ к своей памяти и позднее измеряет время доступа (шаг 1 и 3).

Prime + Probe

Передать управление программе–жертве

Кэш (8 наборов, 4 пути)

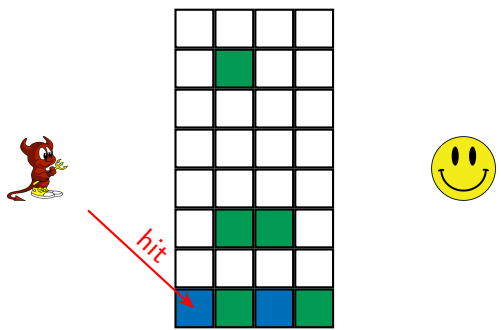


На шаге 2 жертва, возможно, обращается к участку памяти (не общей), которая расположена в том же наборе кэша.

Prime + Probe

Определить какие наборы кэша всё ещё заполнены нашими данными

Кэш (8 наборов, 4 пути)



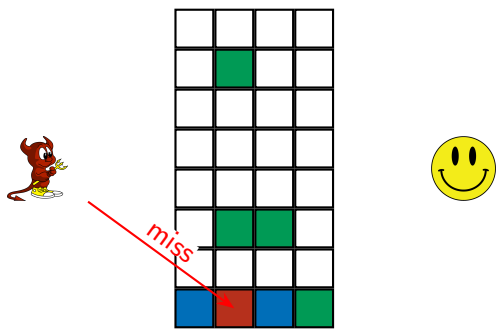
Если жертва обращалась в тот же набор кэша, то время доступа к данным на шаге 3 увеличится, так как один из путей кэша будет заменён, в противном случае время доступа будет меньше.

Время, затрачиваемое на повторный доступ к набору кэша **пропорционально количеству путей**, которые были заменены другими процессами. **Большое время** при измерении означает, что **по крайней мере один путь кэша был заменён**, и наоборот — меньшее время указывает на то, что ни один из путей кэша не был заменён.

Prime + Probe

Определить какие наборы кэша всё ещё заполнены нашими данными

Кэш (8 наборов, 4 пути)



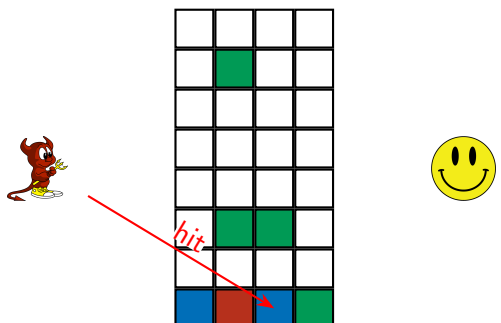
Если жертва обращалась в тот же набор кэша, то время доступа к данным на шаге 3 увеличится, так как один из путей кэша будет заменён, в противном случае время доступа будет меньше.

Время, затрачиваемое на повторный доступ к набору кэша **пропорционально количеству путей**, которые были заменены другими процессами. **Большое время** при измерении означает, что **по крайней мере один путь кэша был заменён**, и наоборот — меньшее время указывает на то, что ни один из путей кэша не был заменён.

Prime + Probe

Определить какие наборы кэша всё ещё заполнены нашими данными

Кэш (8 наборов, 4 пути)



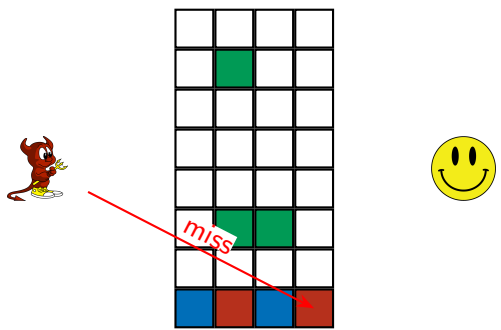
Если жертва обращалась в тот же набор кэша, то время доступа к данным на шаге 3 увеличится, так как один из путей кэша будет заменён, в противном случае время доступа будет меньше.

Время, затрачиваемое на повторный доступ к набору кэша **пропорционально количеству путей**, которые были заменены другими процессами. **Большое время** при измерении означает, что **по крайней мере один путь кэша был заменён**, и наоборот — меньшее время указывает на то, что ни один из путей кэша не был заменён.

Prime + Probe

Определить какие наборы кэша всё ещё заполнены нашими данными

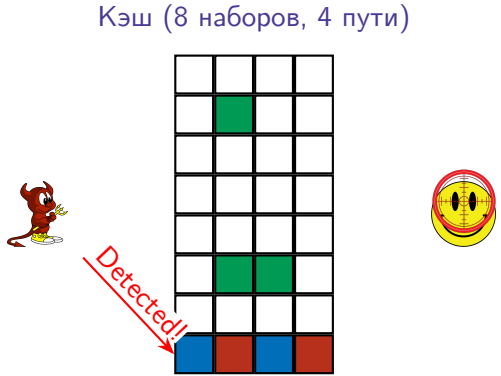
Кэш (8 наборов, 4 пути)



Если жертва обращалась в тот же набор кэша, то время доступа к данным на шаге 3 увеличится, так как один из путей кэша будет заменён, в противном случае время доступа будет меньше.

Время, затрачиваемое на повторный доступ к набору кэша **пропорционально количеству путей**, которые были заменены другими процессами. **Большое время** при измерении означает, что **по крайней мере один путь кэша был заменён**, и наоборот — меньшее время указывает на то, что ни один из путей кэша не был заменён.

Prime + Probe



Атака имеет такую же **детализацию** направленности атаки, как и Evict + Time атака, т. е. **набор кэша**. **Точность атаки больше**, чем в случае с Evict + Time, так как измеряется время **прямого доступа к данным**, а Evict + Time измеряет время через запуски вычислений, а не напрямую. Но также, как и с Evict + Time атакой, **усложнение функций адресации и правил вымещения** из кэша делает проведение атаки затруднительным.

Flush + Reload

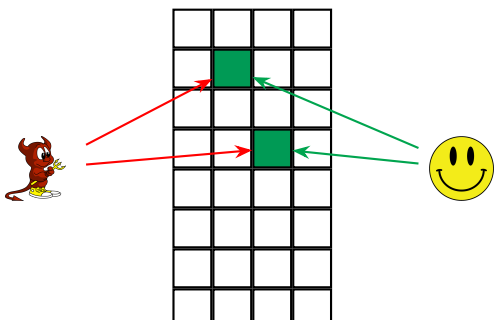
Данная атака считается **наиболее эффективной** атакой на кэш. Целью данной атаки является не просто набор кэша, а **отдельная линия кэша**, более того, у атакующего существует возможность проверить закэширована ли та или иная область памяти. Атака `Flush + Reload` выполняется в три фазы.

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

Flush + Reload

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

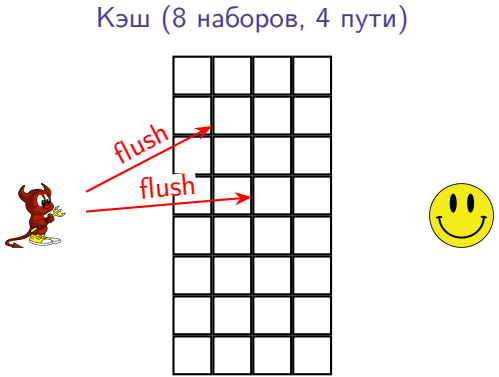
Кэш (8 наборов, 4 пути)



`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)

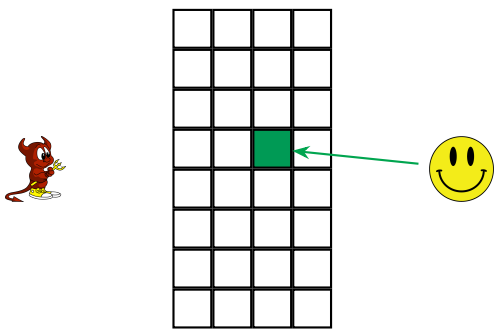


Во время первой фазы контролируемый участок памяти удаляется из структуры кэша (удаляется линия кэша с помощью инструкции **clflush** в случае с Intel).

Flush + Reload

Передать управление программе–жертве

Кэш (8 наборов, 4 пути)

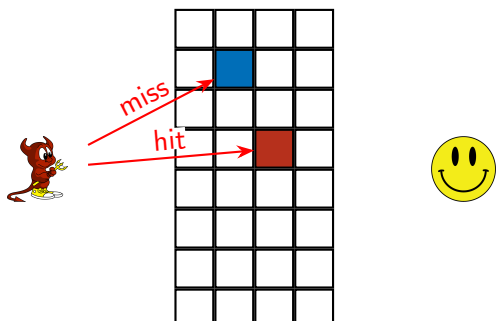


Во второй фазе программа–шпион находится в режиме ожидания, **давая жертве время воспользоваться** участком памяти.

Flush + Reload

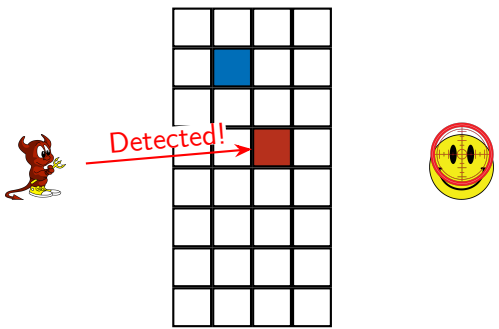
Определить какие линии кэша были загружены программой–жертвой снова

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**.

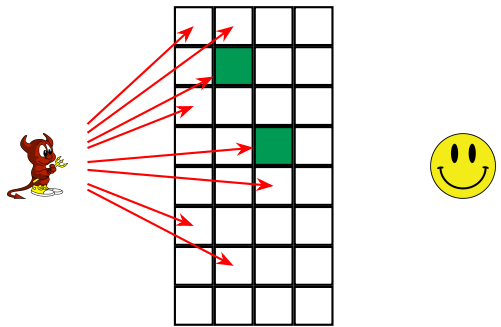
Flush + Reload



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.

Flush + Flush

Кэш (8 наборов, 4 пути)

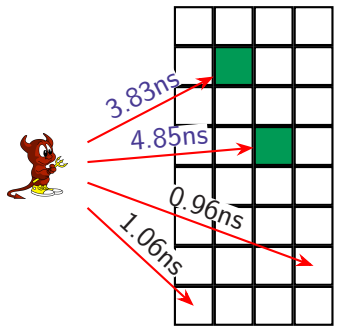


Количество и продолжительность обращений к памяти может быть измерено, а атаки на кэш — обнаружены

Атаки типа `Flush + Reload` и `Prime + Probe` производят **большое количество обращений** к памяти, продолжительность которых **можно измерить** (при помощи системных счётчиков производительности), по этой причине они **могут быть опознаны** процессором.
Атака **Flush + Flush** представляет из себя **стелс–версию** атаки `Flush + Reload`.

Flush + Flush

Кэш (8 наборов, 4 пути)



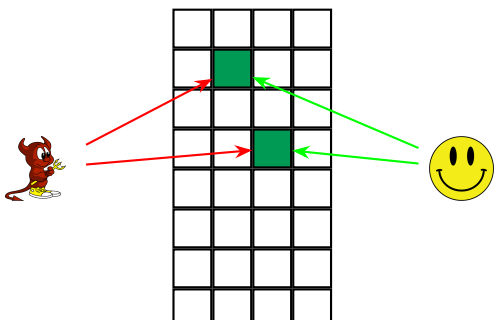
Инструкция для сброса кэша срабатывает за различное время в зависимости от того, находятся ли сейчас какие-либо данные в кэше или нет

Было выяснено, что **инструкция для сброса кэша срабатывает за различное время** в зависимости от того, находятся ли сейчас какие-либо данные в кэше или нет. В случае, если данные, которые подвергаются вытеснению **присутствуют**, то вытеснение происходит **медленнее**. Таким образом вместо повторной загрузки данных в кэш, атакующий снова вытесняет кэш и также измеряет время.

Evict + Reload

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

Кэш (8 наборов, 4 пути)

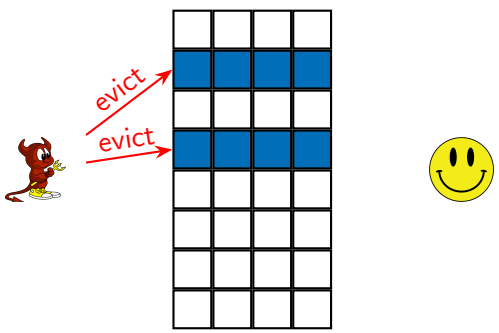


Атака представляет из себя **модифицированную версию Flush + Reload** и имеет смысл для архитектур, где **инструкция вытеснения** из кэша доступна **только в привилегированном режиме**, например, для ARM.

Evict + Reload

Вытеснить содержимое кэш–линии (код или данные)

Кэш (8 наборов, 4 пути)

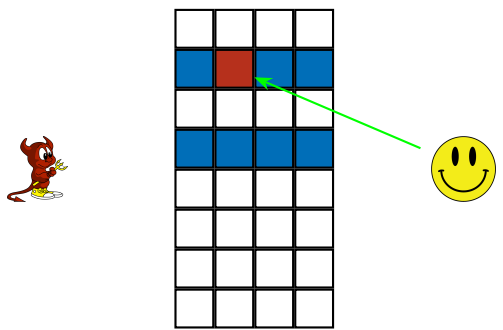


Суть атаки заключается в том, что **для вытеснения** нужной линии кэша происходит **заполнение кэш–памяти большим количеством взаимосвязанных адресов**, в результате чего механизм, отвечающий за вытеснение, сам начнёт вытеснять нужную нам линию кэша.

Evict + Reload

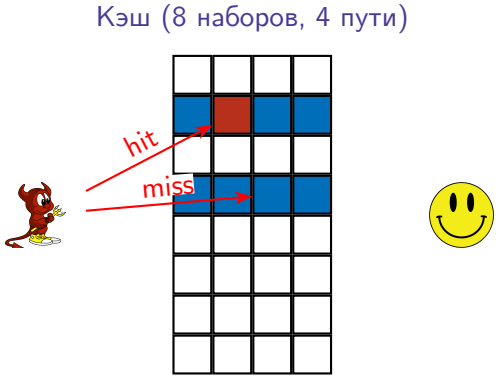
Передать управление программе–жертве

Кэш (8 наборов, 4 пути)



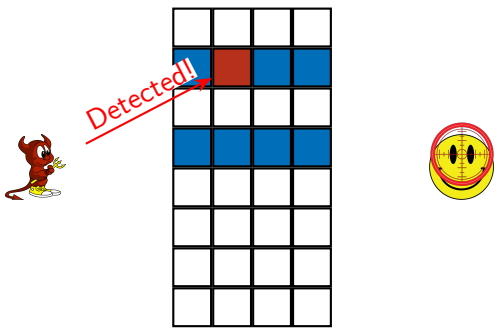
Evict + Reload

Определить какие линии кэша были загружены программой–жертвой снова



Evict + Reload

Кэш (8 наборов, 4 пути)



Новый (2017 год) тип атаки на кэш от Systems and Network Security Group at VU Amsterdam.

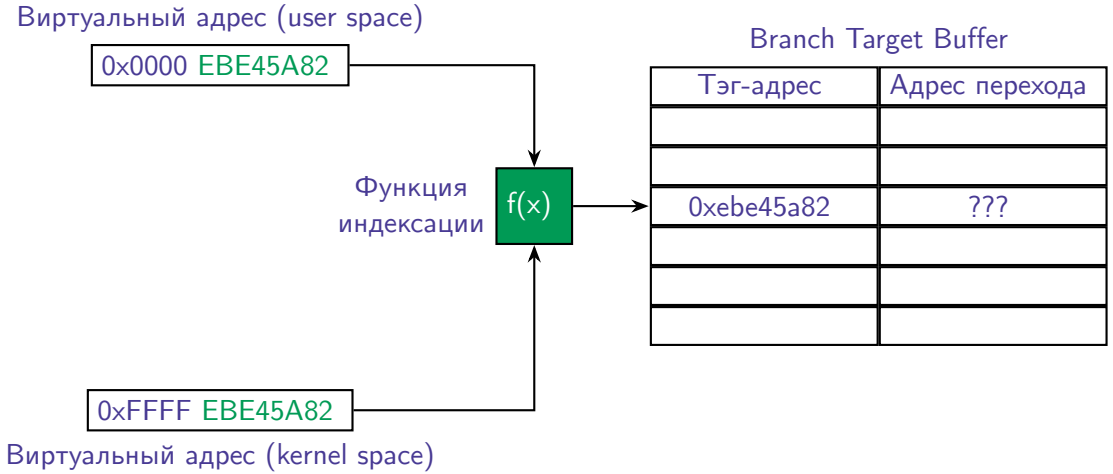
Атака представляет из себя **модифицированную версию Evict + Time**. Были использованы **новые методы для расчёт времени** (Time to Tick — ввиду ограничений на стандартный счётчик, который теперь отсчитывает такты, не измеряя точное время доступа к памяти, отсчитываются такты после обращения к памяти; Shared Memory Counter — параллельный счётчик на отдельном ядре; пример на рисунке **??**). Также **использовалась атака Prime + Probe** на MMU (Memory Management Unit), для возможности проведения атаки на кэш последнего уровня.

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 смешение, полученное в результате KASLR.
Возникает два типа коллизий:

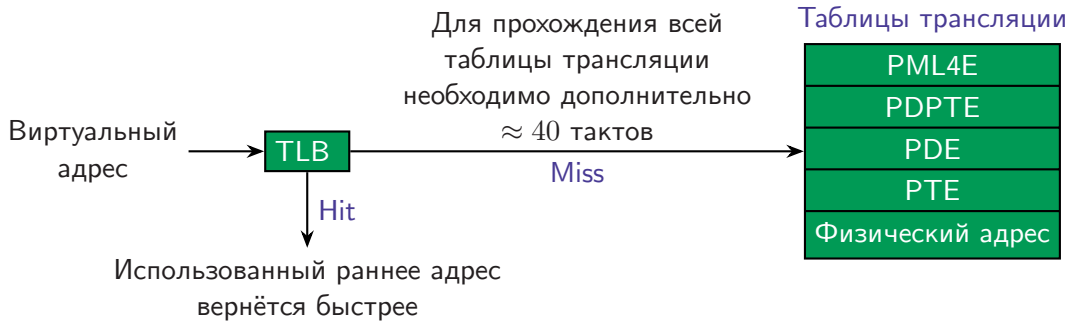
1. cross domain collisions — user и kernel space;
2. same domain collisions — разные user-space процессы, позволяет

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**).

План

Типы атак

- Атаки, основанные на срабатывании исключительных ситуаций
  - Атаки на систему дедупликации памяти

Атаки, основанные на срабатывании исключительных ситуаций

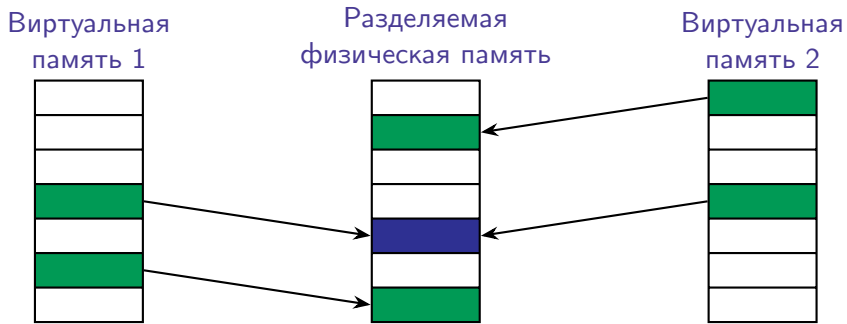
- прерывание планировщика
- инструкции прерывания
- ошибка отсутствия страницы в памяти
- поведенческие изменения (например, возврат кода ошибки)

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**.

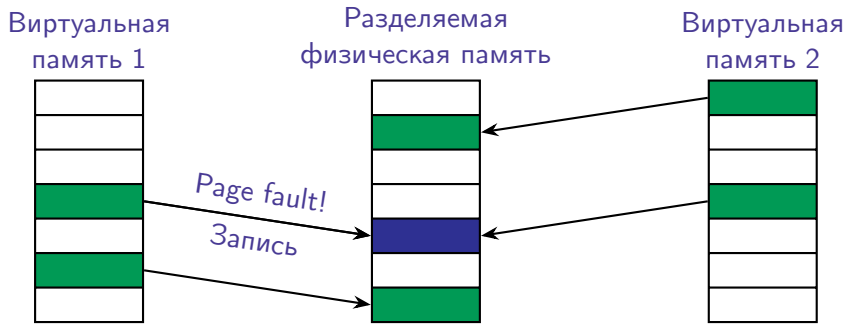
Атаки на систему дедупликации памяти



Дедупликация памяти

Подсистема дедупликации контент–ориентированных страниц **сканирует всю системную память на идентичные физические** страницы памяти и **склеивает их в одну** физическую страницу.

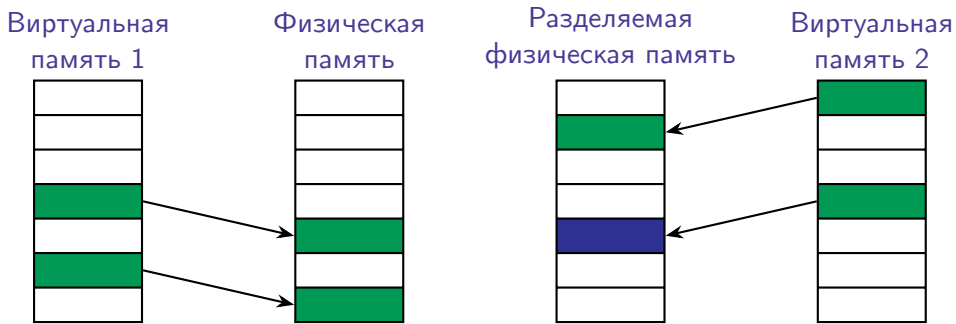
Атаки на систему дедупликации памяти



Процесс 1 пытается записать данные в разделяемую память

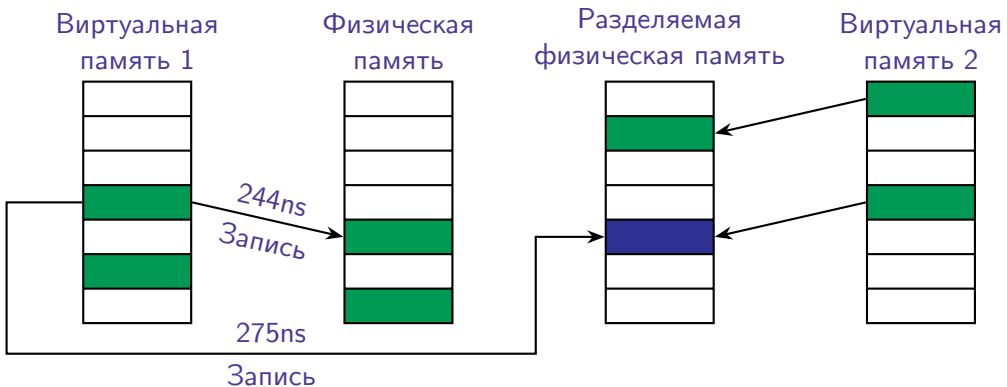
Подсистема дедупликации проецирует несколько **идентичных копий физических страниц** памяти **на одну** разделяемую копию, с доступом в режиме **«копирование при записи»**. В результате при запросах на чтение каждый процесс получает данные из одной и той же страницы. Если же процесс хочет записать данные, то перед тем, как он сможет это сделать, для него создаётся отдельная копия страницы. В результате **запись в разделённую страницу вызывает «страничный отказ»**.

Атаки на систему дедупликации памяти



Запись в дедуплицированную память происходит в режиме Copy-on-Write

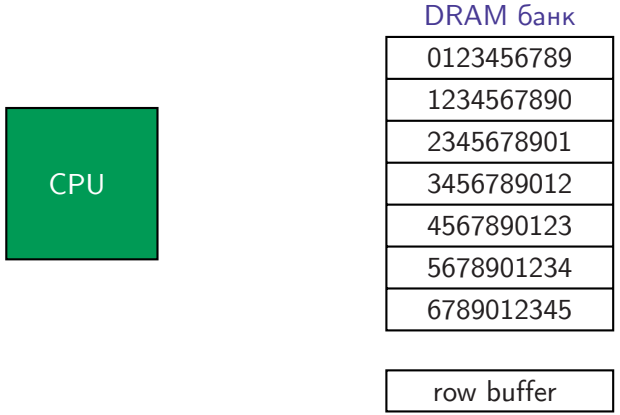
Атаки на систему дедупликации памяти



Следовательно, **запись в разделяемую страницу происходит значительно медленнее**, чем запись в обычную страницу. Злоумышленник, **способный создавать страницы в целевой системе**, может использовать эту разницу во времени, чтобы обнаружить факт **существования интересующих его страниц**.

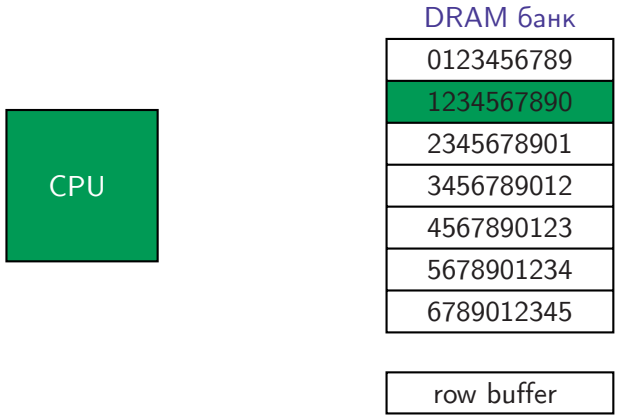
План

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



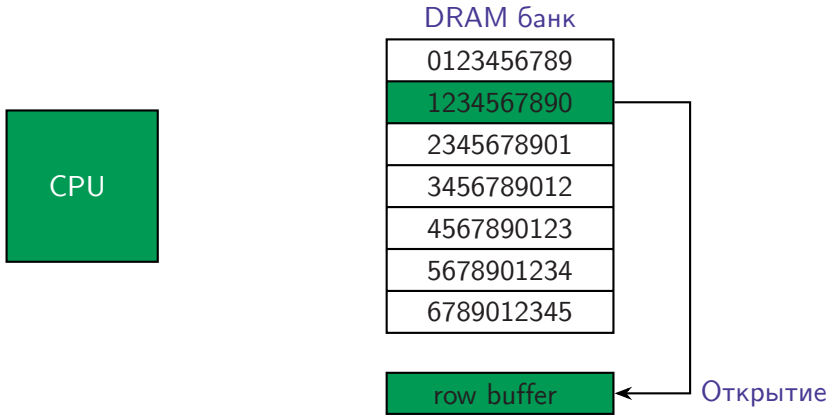
Работа DRAM (ещё раз)

Атаки на DRAM



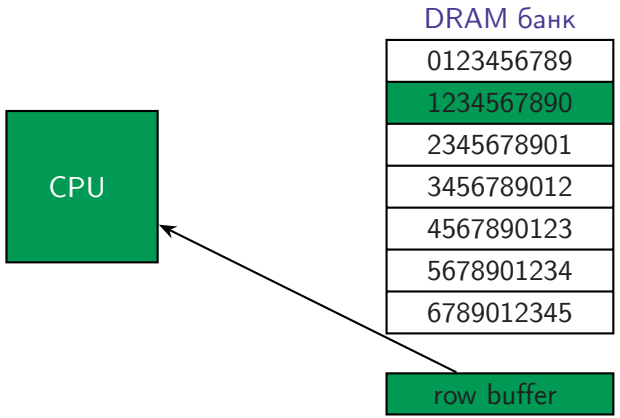
CPU запрашивает на чтение строку №1

Атаки на DRAM



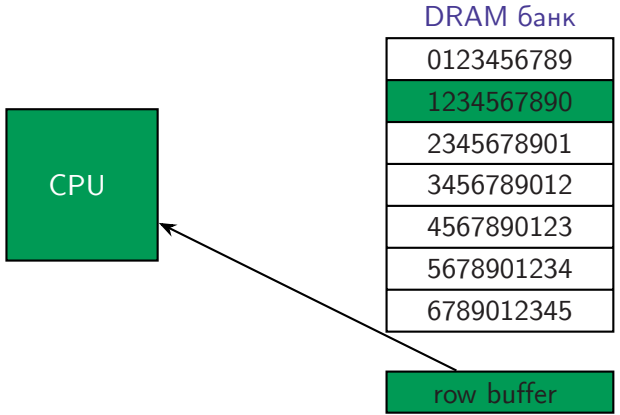
DRAM открывает строку №1

Атаки на DRAM



CPU читает строку №1 из буфера строки

Атаки на DRAM

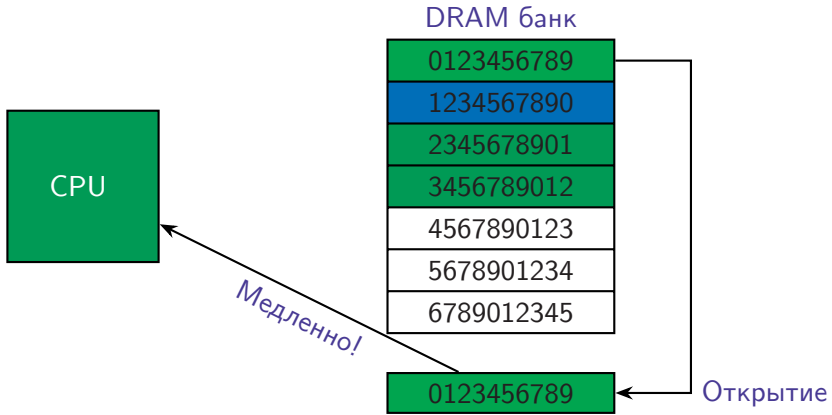


CPU снова запрашивает на чтение строку №1, которая уже есть в буфере строки, чтение происходит быстрее

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

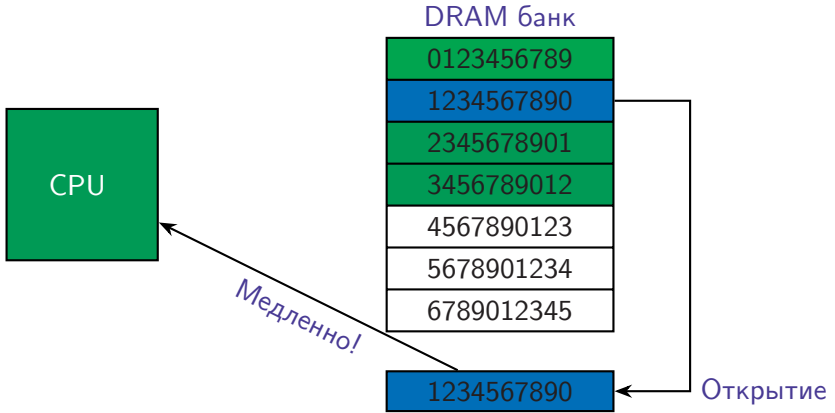
Row hit атака (Flush + Reload)



Атакующий запрашивает строку №0, содержимое которой принадлежит атакующему

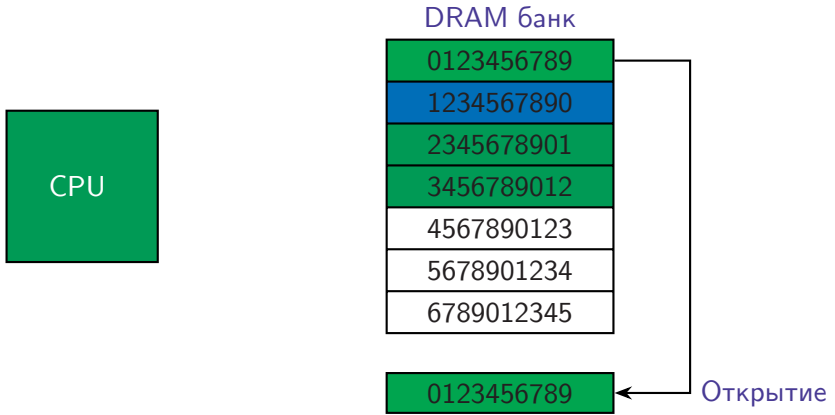
В случае атаки при **попадании строки**, атакующий и жертва используют **одну и ту же строку** DRAM.

Row hit атака (Flush + Reload)



Атакующий запрашивает строку №1, содержимое которой частично принадлежит и атакующему, и жертве

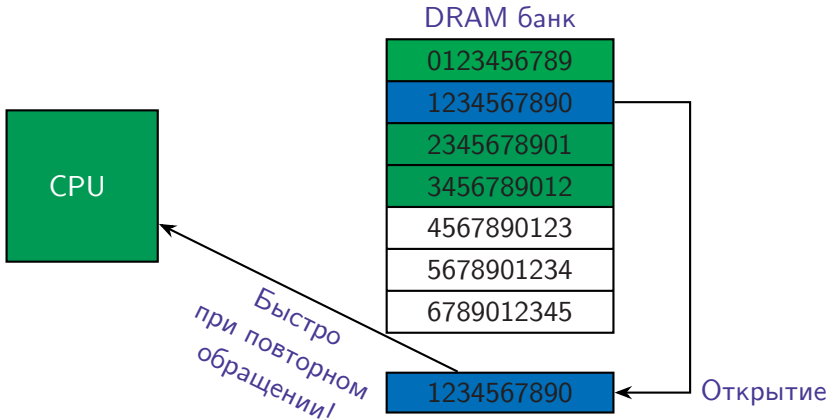
Row hit атака (Flush + Reload)



Сбросим (вытесним) буфер и передадим управление программе–жертве

Атакующий загружает другую строку в буфер строки (тем самым закрывая открытую строку), что похоже на начало атаки **Flush + Reload**.

Row hit атака (Flush + Reload)

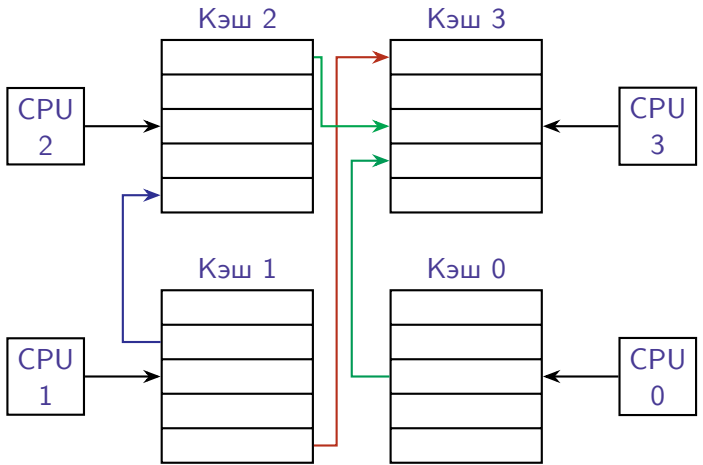


В случае, если жертва обращалась к данному адресу, то это можно вычислить по времени повторного обращения

Если жертва **обращалась** к общей уже закрытой строке, то данная строка снова будет загружена в буфер строки, что выяснится позже при повторном обращении атакующего к этой строке (**время обращения будет меньше**).

План

Скрытые каналы

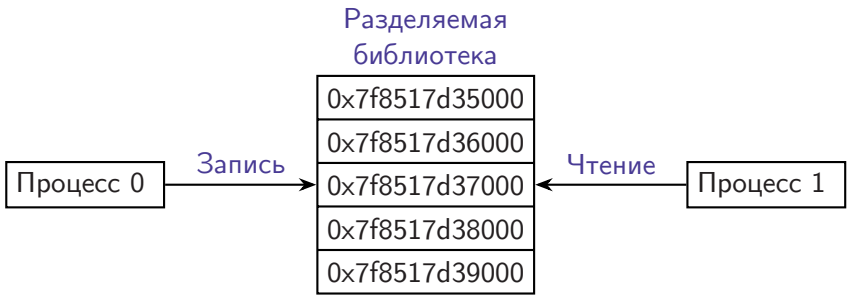


Скрытые каналы между процессорами

Современные облачные системы часто имеют **несколько процессоров**, установленных в материнскую плату с **мультисокетом**. Кэшы 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**.

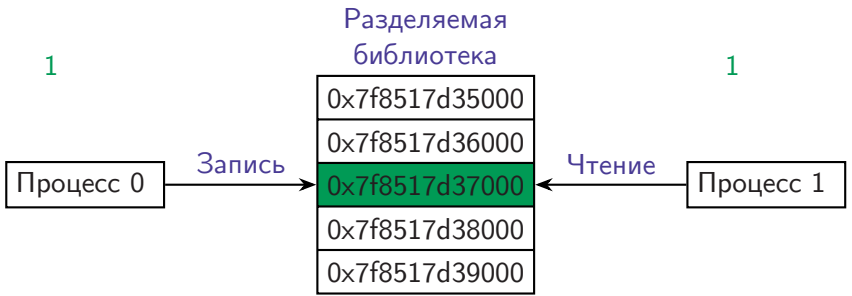
Пример работы



Скрытые каналы между приложениями на базе кэшей

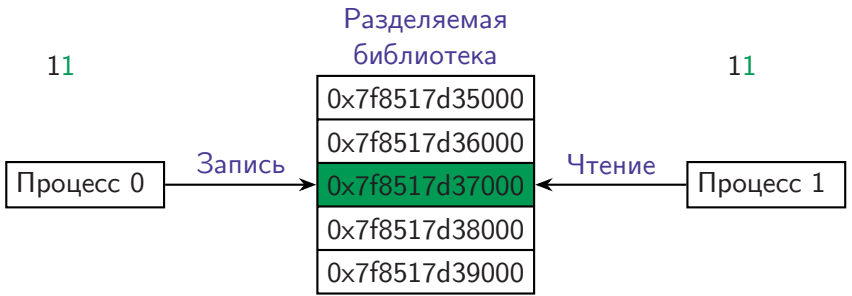
Основная идея скрытого канала заключается в том, что **отправитель и получатель согласовывают набор адресов памяти** какой-нибудь **разделяемой библиотеки**. Они используют для передачи информации загрузку **ячейки в кеш или ее выгрузку оттуда**. Например, если такая-то ячейка находится в кеше, то это **единичка**, а если нет, то **нолик**. В нескольких работах представлена пакетная реализация передачи данных с возможностью повторного запроса недоставленных пакетов; здесь используется «бит отправки» и «бит подтверждения», которые реализованы по такому же принципу, а также контрольная сумма введена (а-ля **TCP по скрытому каналу**).

Пример работы



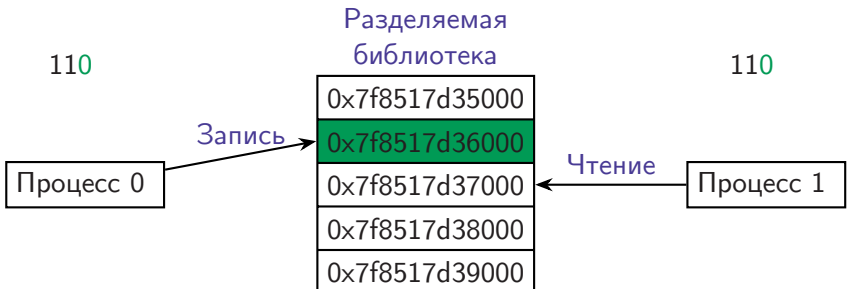
Скрытые каналы между приложениями на базе кэшей

Пример работы



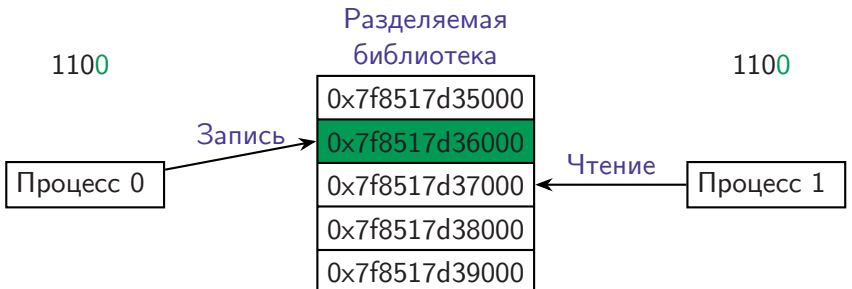
Скрытые каналы между приложениями на базе кэшей

Пример работы



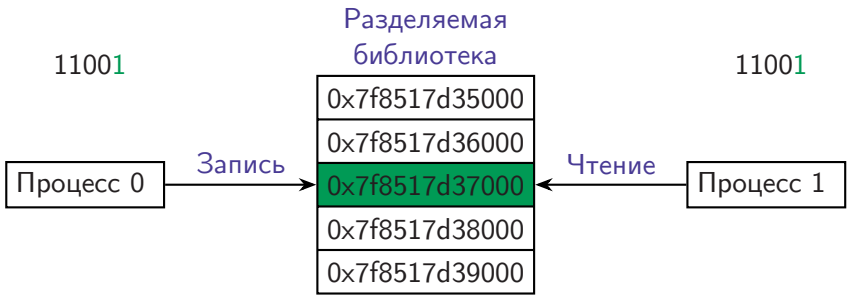
Скрытые каналы между приложениями на базе кэшей

Пример работы



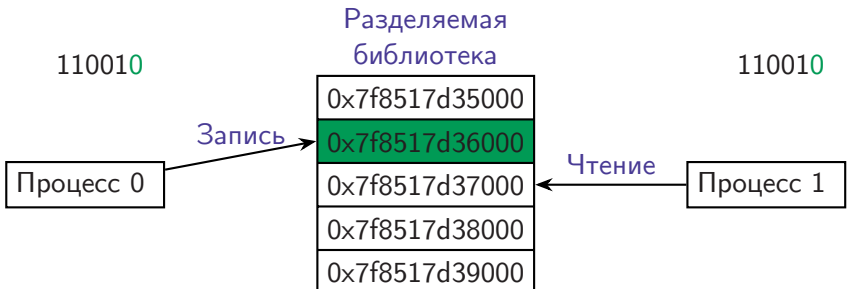
Скрытые каналы между приложениями на базе кэшей

Пример работы



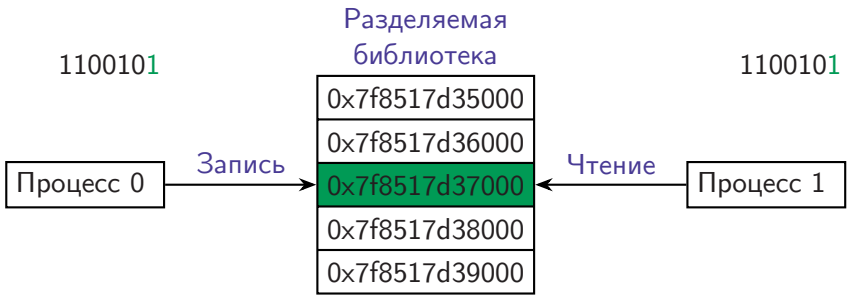
Скрытые каналы между приложениями на базе кэшей

Пример работы



Скрытые каналы между приложениями на базе кэшей

Пример работы

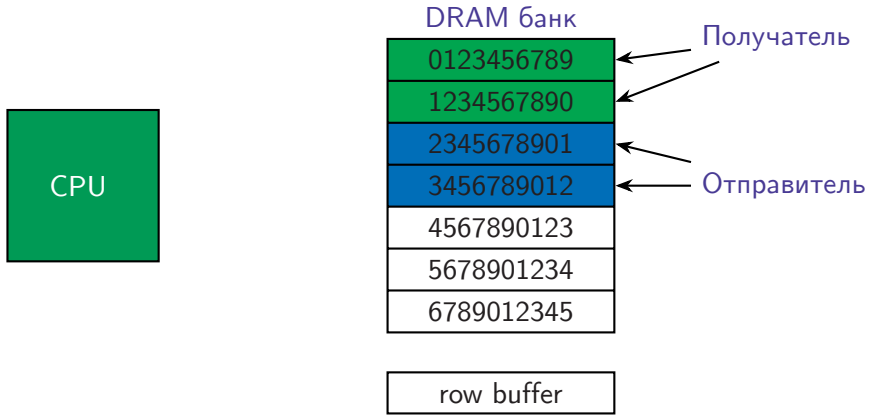


Скрытые каналы между приложениями на базе кэшей

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.

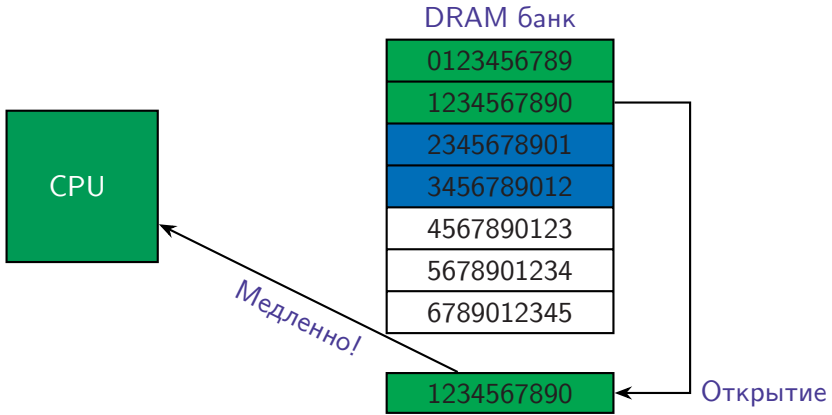
DRAM (row miss атака)



Отправитель и получатель используют один и тот же банк памяти

В случае атаки при **промахе строки**, получатель и отправитель пользуются **разделяемым банком**, но не строкой.

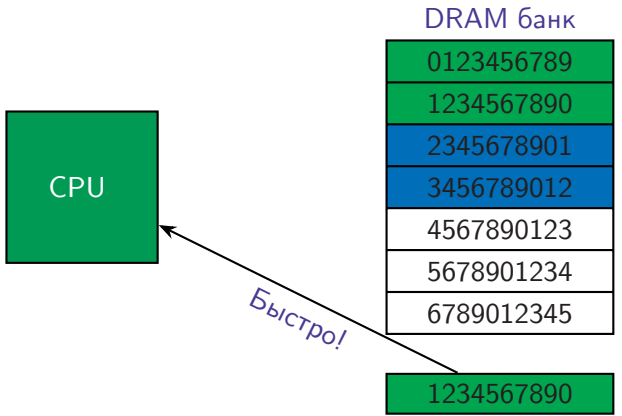
DRAM (row miss атака)



Получатель проверяет время доступа к своей памяти

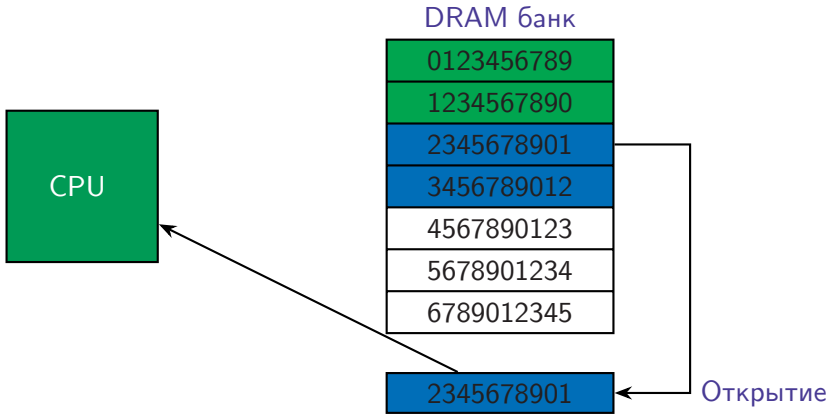
Получатель постоянно открывает одну и ту же строку в банке. Как только отправитель открывает какую-либо другую строку, то у получателя возрастает задержка во время открытия своей строки.

DRAM (row miss атака)



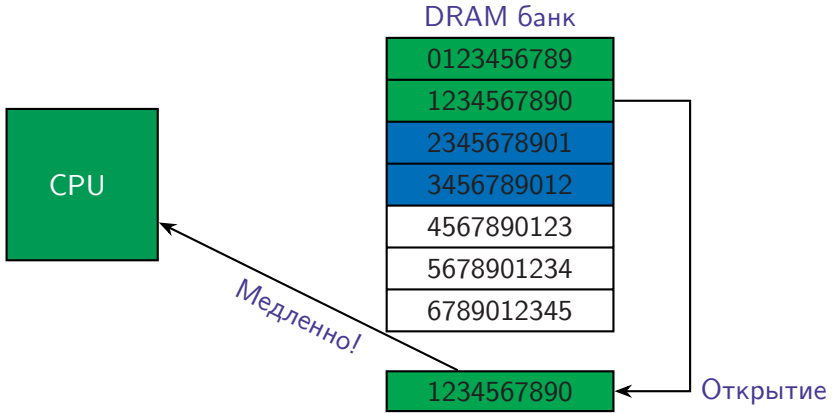
Повторное чтение своей памяти будет происходить быстрее

DRAM (row miss атака)



Отправитель получает доступ к своей памяти

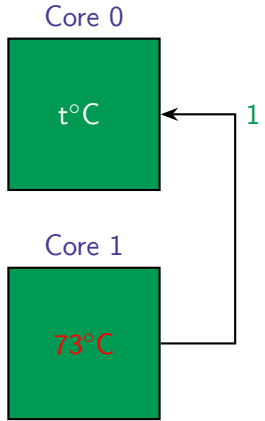
DRAM (row miss атака)



Получатель при своей следующей попытке чтения памяти получит промах строки, соответственно большее время ожидание

Таким образом налаживается связь: если в определённый промежуток времени не было промаха строки, то это 0, если промах случился, то 1.

Тепловой канал

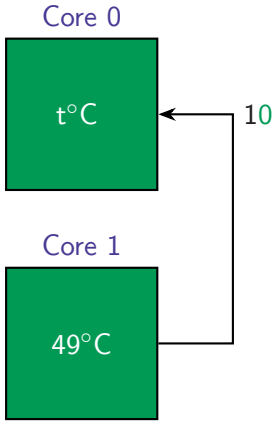


Скрытый канал на основе теплового следа

В 2015 году — эффективно работает даже при **наличии жёстких контрмер**: **псевдоизоляции адресного пространства** критических процессов, либо посредством **выделения памяти случайным образом** и т. п. Рассматриваемая методика — **анализ тепловой активности микропроцессорной системы**.

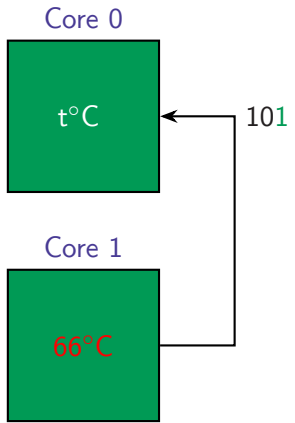
Здесь используются следующие два эффекта. **Во-первых, остаточные тепловые следы в ядре** сохраняются, даже когда процесс прекратил своё исполнение, и этот **след частично передаётся следующему процессу по расписанию**. Во-вторых, тепловой след **влияет и на другие ядра** (если эти несколько ядер на одном чипе расположены).

Тепловой канал



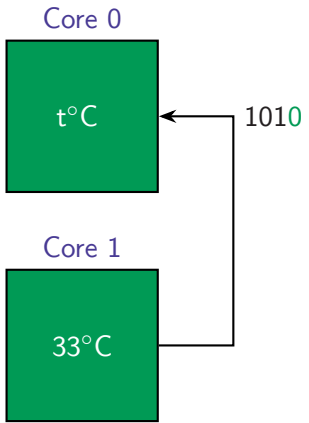
Скрытый канал на основе теплового следа

Тепловой канал



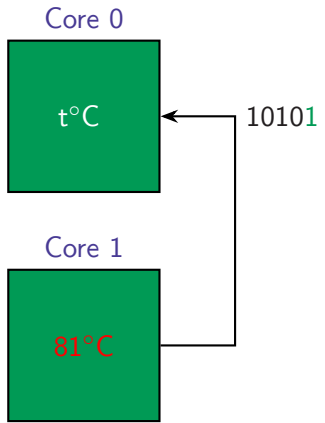
Скрытый канал на основе теплового следа

Тепловой канал



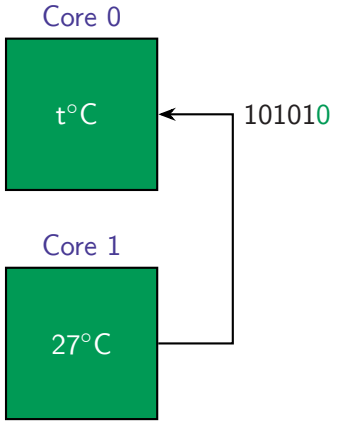
Скрытый канал на основе теплового следа

Тепловой канал



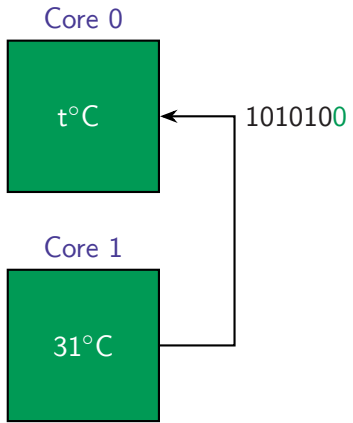
Скрытый канал на основе теплового следа

Тепловой канал



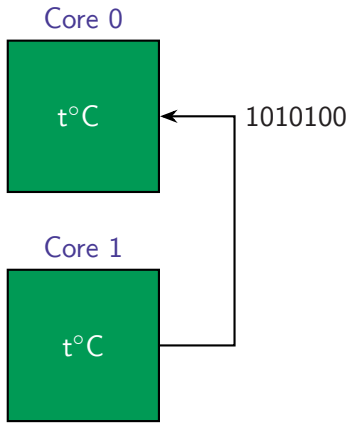
Скрытый канал на основе теплового следа

Тепловой канал



Скрытый канал на основе теплового следа

Тепловой канал



Скрытый канал на основе теплового следа

Так, пропускная способность «теплового канала» на сервере Intel Xeon (с двумя процессорами по восемь ядер) составляет 12,5 бит/с. Номер кредитной карты по такому каналу передаётся за промежуток от пяти секунд до четырёх минут.

Также в 2015 году в рамках демонстрации возможностей скрытого теплового канала было показано, что эта методика позволяет **идентифицировать приложения на основе их тепловых следов**. При этом с течением времени эффективность и **производительность скрытых тепловых каналов будет только расти**, потому что пользователю предоставляется все более подробная информация о температуре системы — чтобы предпринимать **эффективные меры для охлаждения**.

План

Атаки, основанные на аппаратных дефектах

Аппаратные дефекты можно эксплуатировать с помощью исполнения кода

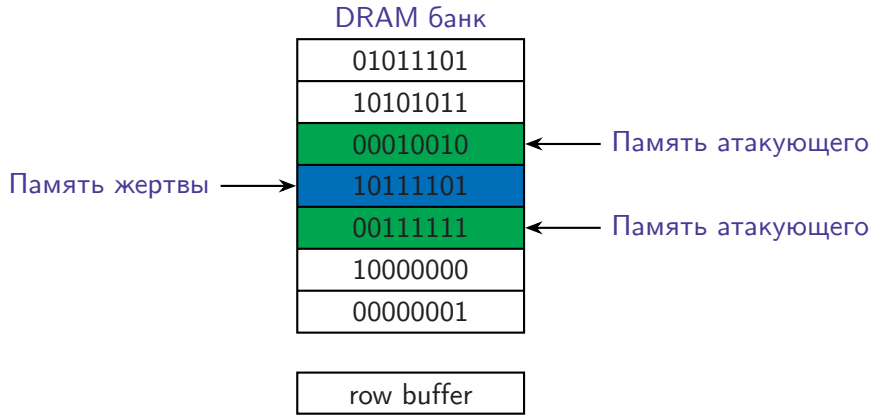
Обычно предполагается, что безопасность системы и программная безопасность опирается на безопасность аппаратную и на то, что в аппаратном средстве нет ошибок. Однако, это не так, и **аппаратные средства не идеальны**, особенно часто дефекты встречаются в случаях, когда работа производится **за границами 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.

План

Атаки, основанные на аппаратных дефектах

- Rowhammer

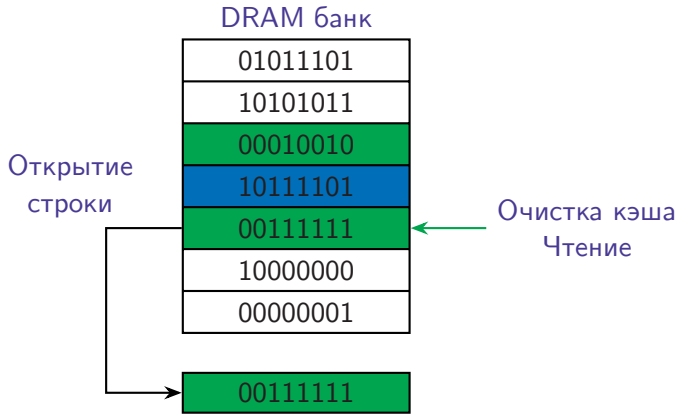
Rowhammer



Для эксплуатации атака должна быть направлена на память, расположенную в одном и том же банке, но в разных строках

Первая атака, основанная на дефекте микроархитектуры, названном впоследствии **Rowhammer**, была найдена Kim в 2014 году. Она запускалась из программного обеспечения и могла наносить вред безопасности системы.

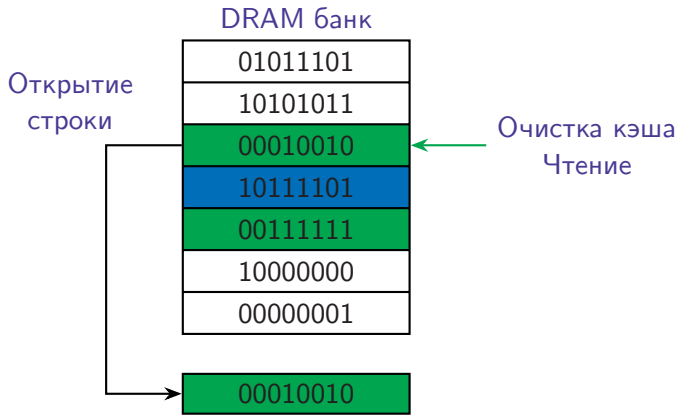
Rowhammer



«Долбим» память вокруг памяти жертвы

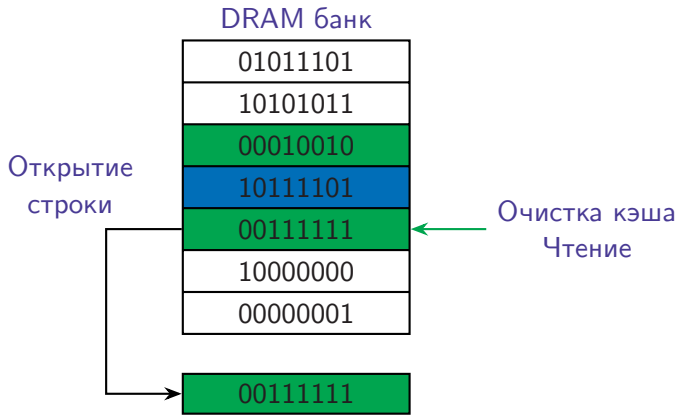
Атакующий постоянно **получает доступ к DRAM памяти**, при этом **сбрасывает кэш** для того, чтобы **часто открывать-закрывать строки** DRAM. Если строки DRAM находятся в физической близости друг с другом, то на одной из строк может **самопроизвольно переключиться бит**. Память, в которой самопроизвольно переключается бит, могла быть недоступна атакующему и даже принадлежать другому домену безопасности.

Rowhammer



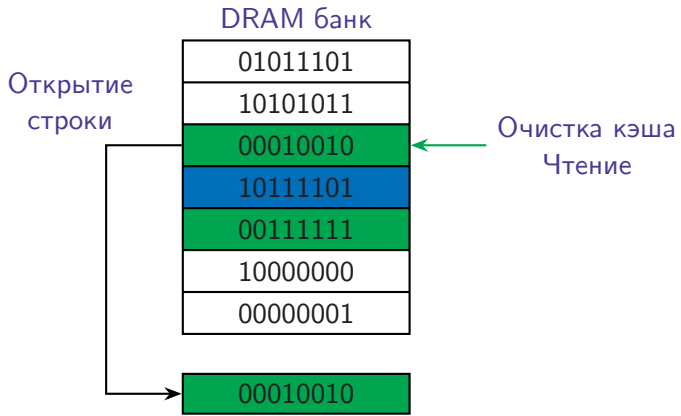
«Долбим» память вокруг памяти жертвы

Rowhammer



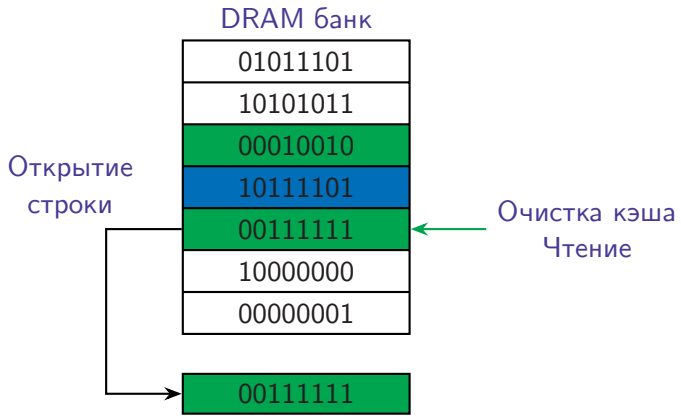
«Долбим» память вокруг памяти жертвы

Rowhammer



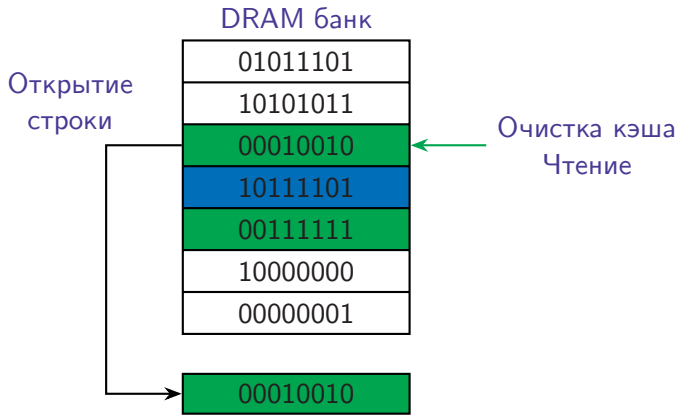
«Долбим» память вокруг памяти жертвы

Rowhammer



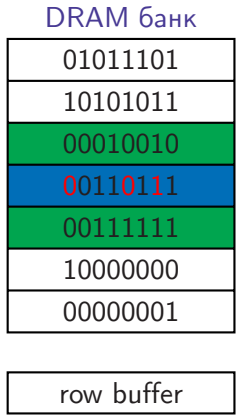
«Долбим» память вокруг памяти жертвы

Rowhammer



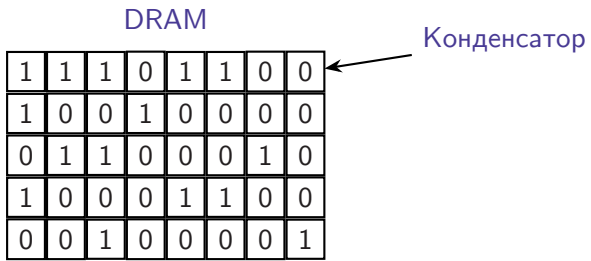
«Долбим» память вокруг памяти жертвы

Rowhammer



В результате — самопроизвольное переключение битов памяти

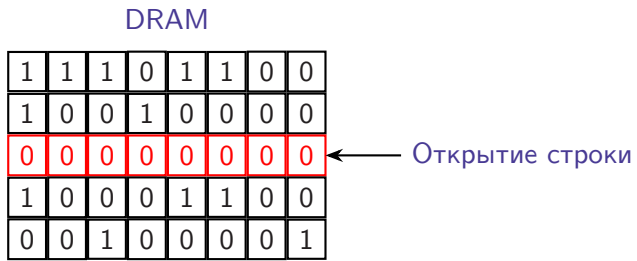
Rowhammer



Что происходит при Rowhammer?

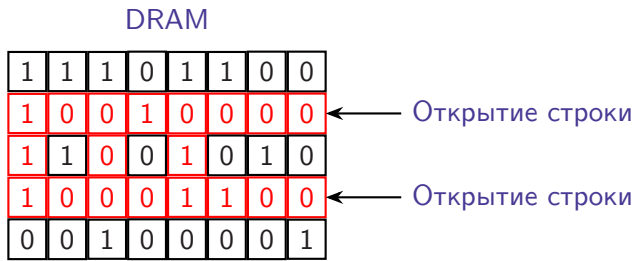
Почему же так происходит? Каждая ячейка DRAM — это конденсатор; 0 и 1 — это **заряженное или разряженное** состояние конденсатора. Каждая ячейка в сетке **связана с соседней ячейкой проводом**.

Если какая-либо ячейка активируется, то напряжение подаётся как на её конденсатор, так и на **все остальные конденсаторы той же строки**.



На всю строку подаётся питание

Rowhammer



Активация строк в современных DRAM

Поскольку ячейки памяти по мере технологического прогресса становятся всё меньше и меньше и **всё ближе** друг к другу, помехи, вызванные активацией строки памяти, очень часто **влияют на заряды конденсаторов соседних** строк.

Первые реализации атак с использованием эффекта Rowhammer полагались либо на вероятностные методы (из-за чего в процессе атаки могло произойти незапланированное обрушение системы), либо на специализированные функции управления памятью: дедупликацию памяти, паравиртуализацию MMU, интерфейс pagemap. Однако подобные функции на современных устройствах либо недоступны вообще, либо отключены по соображениям безопасности.

План

Необходимые примитивы

- 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.

План

Атаки, основанные на аппаратных дефектах

- Разновидности Rowhammer

Разновидности Rowhammer

- Flip Feng Shui — целенаправленный Rowhammer

+ методика «массажирования памяти» для атаки на конкретный адрес
− при условии наличия систем разделения памяти (дедупликация, виртуальные машины и т. п.)

В 2016 году была представлена методика **«массажа памяти» Flip Feng Shui (FFS)** — новый вектор эксплуатации эффекта Rowhammer, который позволяет злоумышленнику **возбуждать предсказуемые битовые перескоки** в произвольном месте физической памяти и иметь полный контроль над этим процессом даже при полном отсутствии уязвимостей в атакуемом программном обеспечении. В рамках демонстрации методики FFS **скомпрометирован механизм обновления**, используемый операционными системами **Ubuntu/Debian**. Компрометация удаётся в случае, **если используется дедупликация памяти, виртуальные машины работают на одной системе или используется разделяемая память**.

Разновидности Rowhammer

- Flip Feng Shui — целенаправленный Rowhammer
- Throwhammer — удалённая атака

\+ удалённо

\− remote direct memory access (RDMA)

В 2018 была описана новая атака — **Throwhammer**: Rowhammer Attacks over the Network and Defenses. Она позволяла проводить атаки типа Rowhammer **удалённо**, достаточно было послать необходимые сетевые пакеты. Единственное условие — машины должны быть подключены **через удалённый прямой доступ к памяти (remote direct memory access, RDMA)**, что характерно только для высокопроизводительных кластеров и облачных технологий.

Разновидности Rowhammer

- Flip Feng Shui — целенаправленный Rowhammer
- Throwhammer — удалённая атака
- Nethammer — улучшенная удалённая атака

+ удалённо
– Intel CAT
– драйверы сетевых устройств используют инструкции очистки кэша
– используется некэшируемая память

В 2018 году была описана ещё одна удалённая атака использующая Rowhammer — **Nethammer**: Inducing Rowhammer Faults through Network Requests. Данная атака позволяла изменять биты памяти с помощью **специально сконфигурированного сетевого пакета**, который вызывал чтение по установленным участкам памяти.
Для Rowhammer атаки требуется также очистка кэша. В данной работе этот вопрос «решён» тремя способами:

1. на жертве используется **Intel CAT технология**, которая размещает данные в определённом порядке, что позволяет атакующему вытеснять из кэша необходимые ему данные;
2. драйверы сетевых устройств используют **инструкции очистки кэша** (`clflush`, например);
3. на жертве **используется некэшируемая память**, поэтому очищать кэш не требуется вовсе.

Разновидности Rowhammer

- Flip Feng Shui — целенаправленный Rowhammer
- Throwhammer — удалённая атака
- Nethammer — улучшенная удалённая атака
- Drammer — атака на ARM

– ARMv7 — непривилегированный сброс кэша невозможен
– ARMv8 — инструкция сброса кэша отключена на уровне ядра
– системный вызов `cacheflush()` — сброс кэша только до второго уровня
– вытеснение из кэша с помощью вычислений — медленно
+ Android ION allocator — некэшируемая DMA память

Drammer — новый вид атаки, способный проводит атаку Rowhammer на мобильных устройствах. До этой атаки утверждалось, что ARM устройства якобы неуязвимы по причине **медленной скорости чтения памяти**.
Как уже известно, для атаки подобной Rowhammer **требуется проводить сброс кэша**. На устройствах на **ARMv7** чипе произвести сброс кэша **невозможно**. Однако, в ядре Android существует системный вызов **cacheflush()**, который можно вызвать из пользовательского контекста выполнения, очищает только до второго уровня — **недостаточно** для атаки на DRAM. Что касается **ARMv8**, то там присутствуют инструкция, позволяющая сбрасывать кэш, но она **отключена на уровне ядра**.
Также существует метод вытеснения из кэша с помощью различных **вычислений**, но к сожалению данная атака не подходит по причине того, что на практике (на архитектурах ARMv7 и ARMv8) вычисления производятся **с недостаточной скоростью**.
В итоге был использован **Android ION аллокатор памяти**, который позволяет работать с **некэшируемой DMA памятью**.

Разновидности Rowhammer

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

✓ механизм вычисления времени доступа к памяти и другим ресурсам — WebGL

✓ общие ресурсы — кэш GPU

✓ знание физического расположения данных в памяти GPU — обратная разработка с помощью атак по сторонним каналам (примитивы представлены выше)

✓ быстрый доступ к памяти — WebGL + GPU

Новый вид атаки типа Rowhammer, направлен на эксплуатацию **GPU на мобильных устройствах**. Для атаки **используется WebGL механизм**, встроенный в браузер (Firefox).

- **механизм вычисления времени доступа к памяти** и другим ресурсам (используется механизм **WebGL для создания высокоточных таймеров**)
- **общие ресурсы** (**кэш GPU**, работа которого не документирована, но в результате обратной разработки было выяснено, что для оптимизации кэша используются весьма простые алгоритмы)
- знание **физического расположения данных** (было получено в результате проведения атак по сторонним каналам)
- **быстрый доступ к памяти** (получается за счёт использования GPU + WebGL)

В итоге был разработан эксплоит на JS, позволяющий обойти ASLR механизм, скомпрометировать данные.

План

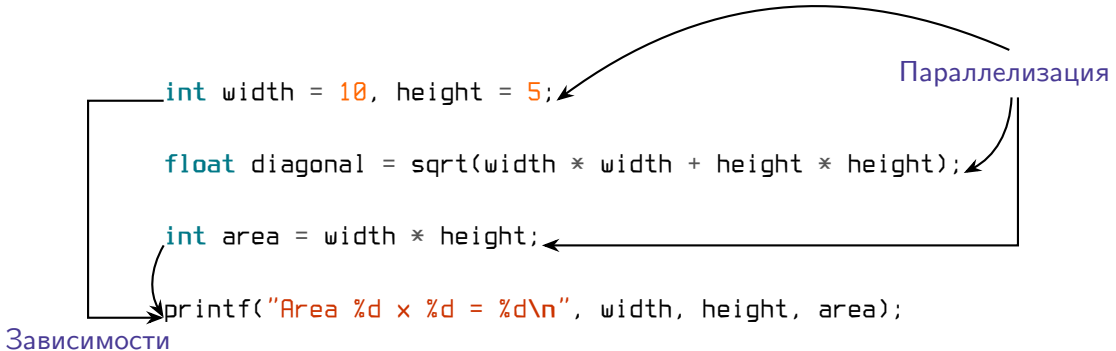
План

Variant 3

CVE-2017-5754: спекулятивное чтение недоступных данных

Meltdown

Выполнение не по порядку



В разделе теории было уже рассказано, что инструкции могут выполняться не по порядку, а некоторые даже спекулятивно, если у них нет зависимостей.

Чтение недоступной памяти

- Если программа читает память,то
  1. проверяются права
  2. память считывается
- Если программа пытается читать недоступную память, то
  1. происходит ошибка
  2. выполнение останавливается

Но что будет, если проверка прав и чтение будут выполняться не по порядку?

Чтение недоступной памяти

```
*(volatile char*) 0; // ошибка чтения, Выполнение прерывается
temp = array[84 * 4096]; // Выполнение Вне очереди?
```

1. Попытка чтения недоступной памяти
2. Проверка прав на чтение
3. Параллельно: спекулятивное выполнение последующей операции
4. Параллельно: запись данных в кэш

Чтение недоступной памяти

```
*(volatile char*) 0; // ошибка чтения, Выполнение прерывается
temp = array[84 * 4096]; // Выполнение Вне очереди?
```

С помощью атаки на кэш `Flush + Reload` выясняется, что обращение к 84 странице памяти состоялось!

Эксплуатация

```
unsigned char value = *(unsigned char *)ptr;
unsigned long index = (((value >> bit) & 1) * 0x100) + 0x200;
maccess(&data[index]);
```

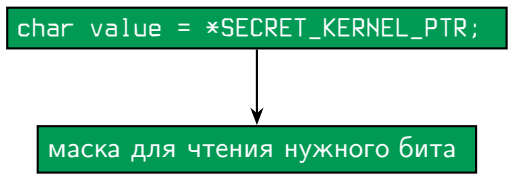
ptr — указатель на интересующую нас память.
data — контролируемый атакующим массив.

Эксплуатация

```
char value = *SECRET_KERNEL_PTR;
```

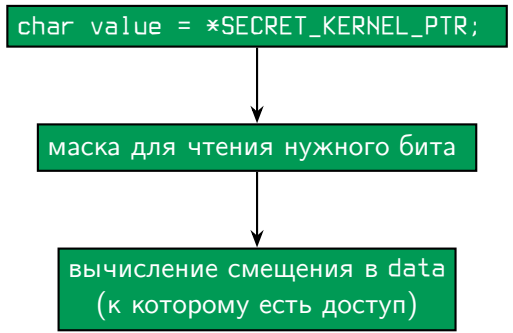
```
unsigned char value = *(unsigned char *)ptr;
```

Эксплуатация



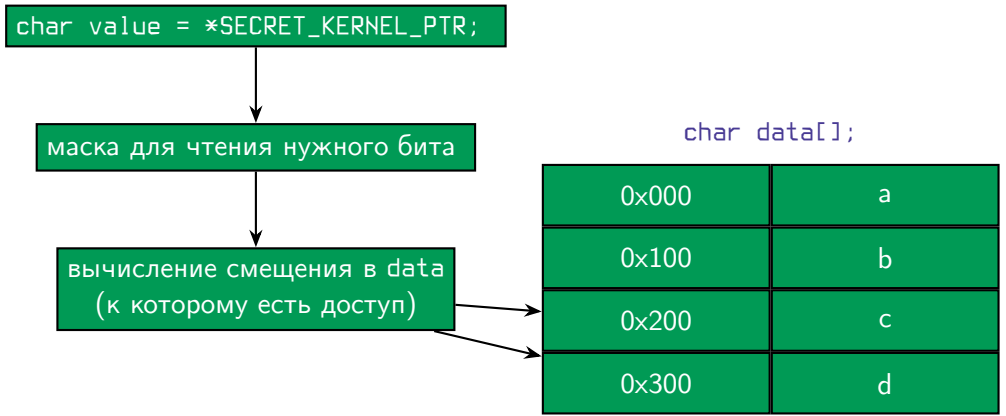

```
unsigned long index = (((value >> bit) & 1) * 0x100) + 0x200;
```

Эксплуатация

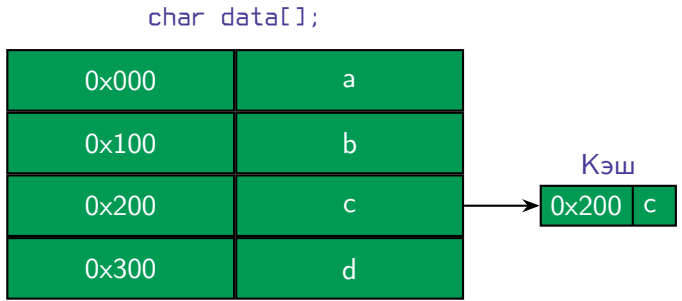



```
unsigned long index = (((value >> bit) & 1)* 0x100) + 0x200;
```

Эксплуатация

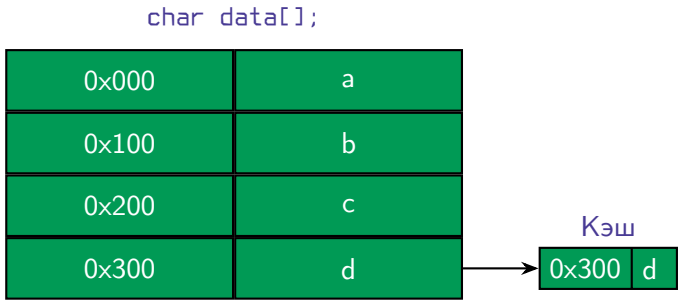


Эксплуатация



При обращении к массиву по определённому индексу данные попадают в кэш

Эксплуатация



При обращении к массиву по определённому индексу данные попадают в кэш

ASM

```
LDR X1, [X2]        ; X2 - указатель на данные, которых нет В кэше,
                    ; также В TLB не должно быть данного адреса
CBZ X1, over        ; переход, который В итоге будет соВершён,
                    ; но инструкции ниже Всё равно исполнятся
LDR X3, [X4]        ; X4 - указатель на данные В пространстВе памяти ядра
LSL X3, X3, #imm    ; получение нужного бита данных
AND X3, X3, #0xFC0 ; ВыравниВание с размером страницы памяти
LDR X5, [X6,X3]     ; X6 - адрес массиВа атакующего
over
```

Читаем недоступные пользователю данные из памяти, лучше, если адрес не будет храниться в TLB, также данных не будет в кэше.

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

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

План

Variant 3a

CVE-2018-3060: спекулятивное чтение недоступных данных

Meltdown

По сути всё тот же meltdown.

ASM

```
LDR X1, [X2]        ; X2 - указатель на данные, которых нет В кэше,
                    ; также В TLB не должно быть данного адреса
CBZ X1, over        ; переход, который В итоге будет соВершён,
                    ; но инструкции ниже Всё раВно исполнятся
MRS X3, TTBR0_EL1   ; TTBR0_EL1 - системный регистр,
                    ; недоступный для чтения пользоВателем
LSL X3, X3, #imm    ; получение нужного бита данных
AND X3, X3, #0xFC0 ; ВыраВниВание с размером страницы памяти
LDR X5, [X6,X3]     ; X6 - адрес массиВа атакующего
over
```

Всё также, как и в случае с чтением недоступной пользователю памяти, только здесь производится **спекулятивное чтение системного регистра**.

К **ARM процессорам не применимо**, так как даже в ходе спекулятивного выполнения данные из системных регистров не читаются.

+ предотвращение Variant 3.

В зависимости от уровня привилегий заменять значения системных регистров фиктивными

План

Variant 1

CVE-2017-5753: обход проверки границ

Spectre

Спекулятивное выполнение

- CPU пытается предугадать будущие переходы
  - ... учась на произошедших
- происходит спекулятивное выполнение инструкций выбранного перехода
- если переход угадан верно
  - ... быстрое выполнение
- если переход угадан неверно
  - ... отброс результата спекулятивного выполнения

Вскользь уже была упомянута тема с предугадыванием переходов. Какие же **побочные эффекты** могут нас ожидать при такой архитектуре?

Побочные эффекты

Рассмотрим, в каких случаях происходит спекулятивное выполнение.

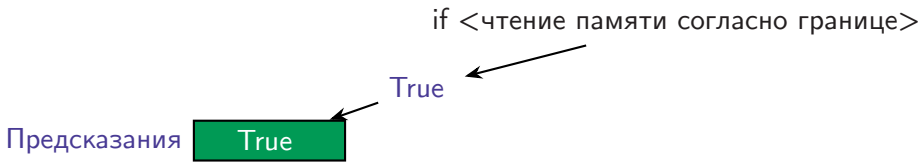
if <чтение памяти согласно границе>

Побочные эффекты

if <чтение памяти согласно границе>

Если чтение участка памяти происходит согласно заявленным границам.

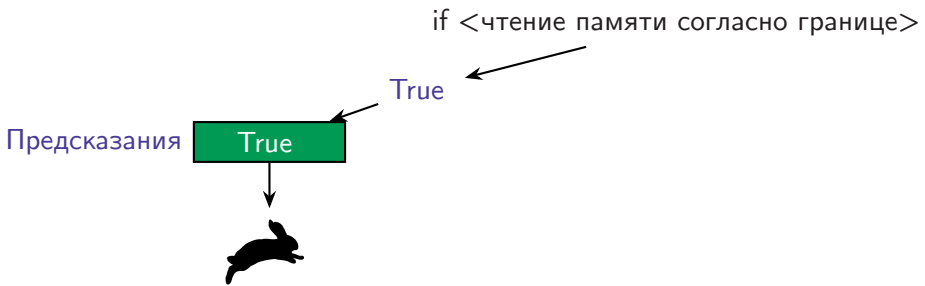
Побочные эффекты



Чтение согласно границам, предсказатель также думал, что чтение будет происходить согласно границам:

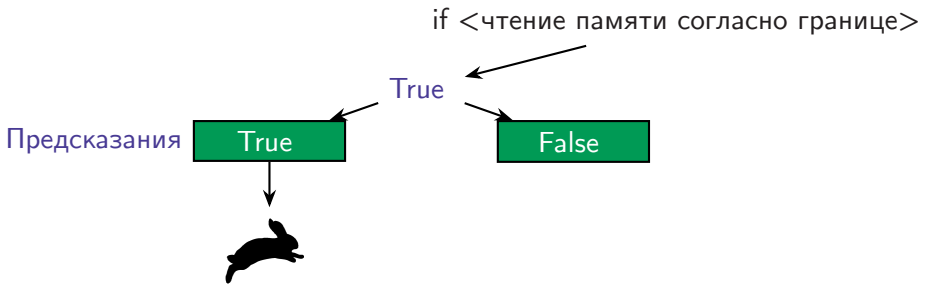
- спекулятивно выполнится код чтения в соответствии с границами памяти

Побочные эффекты



В итоге выполнение кода произойдёт быстро, так как уже заранее был выполнен, результаты получены.

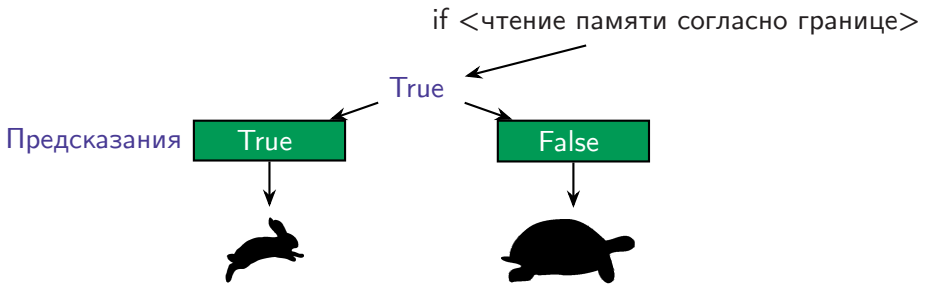
Побочные эффекты



Чтение согласно границам, предсказатель же думал, что чтение памяти выйдет за границы.

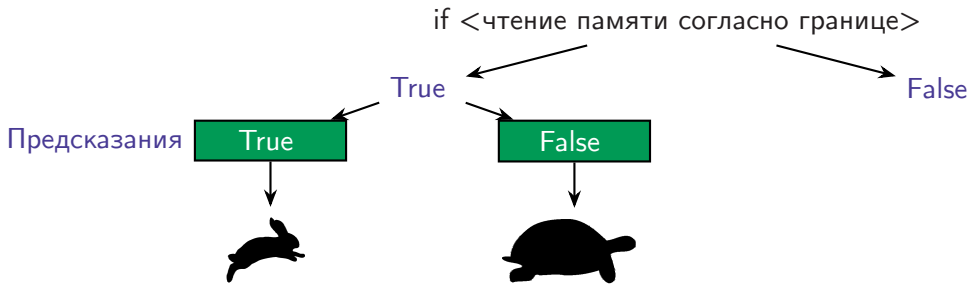
- спекулятивно будет выполняться последующий за чтением памяти код
- спекулятивное выполнение отбросится, выполнение кода будет происходить заново

Побочные эффекты



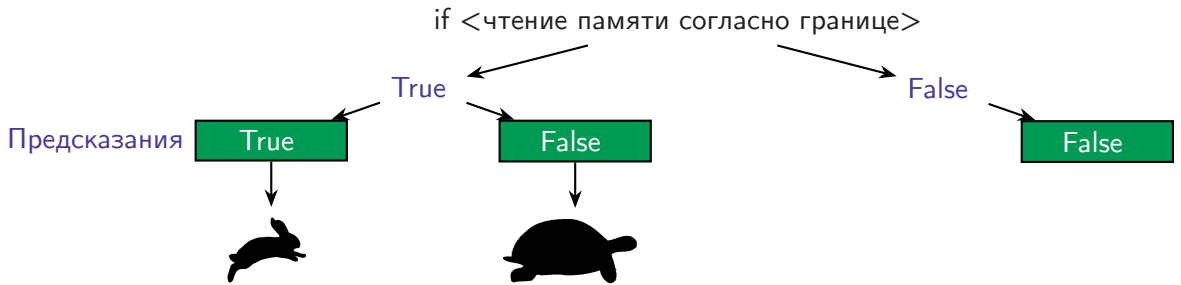
В итоге из-за расхождения реальности с предсказанием, скорость выполнения кода будет ниже.

Побочные эффекты



Если чтение участка памяти происходит за заявленными границами памяти.

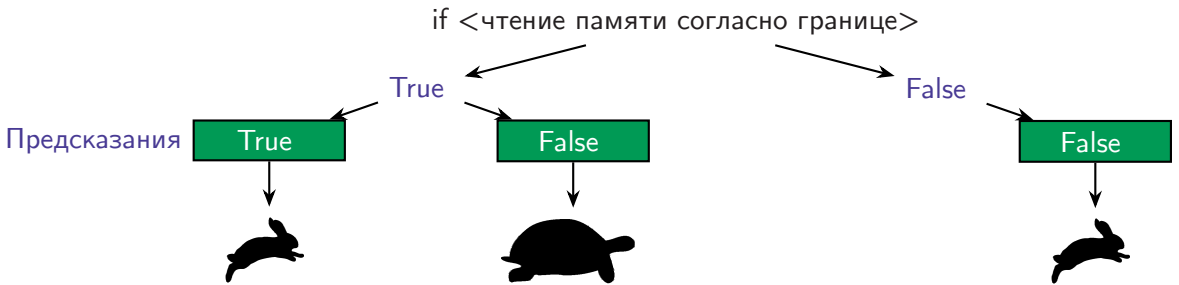
Побочные эффекты



Чтение участка памяти за границами, предсказатель также думал, что чтение будет происходить за границами.

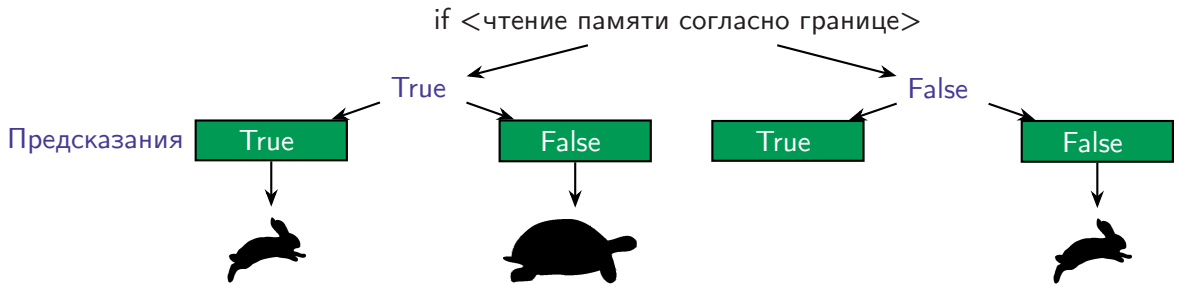
- спекулятивно будет выполняться последующий за чтением памяти код

Побочные эффекты



Код исполнится быстро, так как предсказатель угадал и спекулятивно выполнял последующий код.

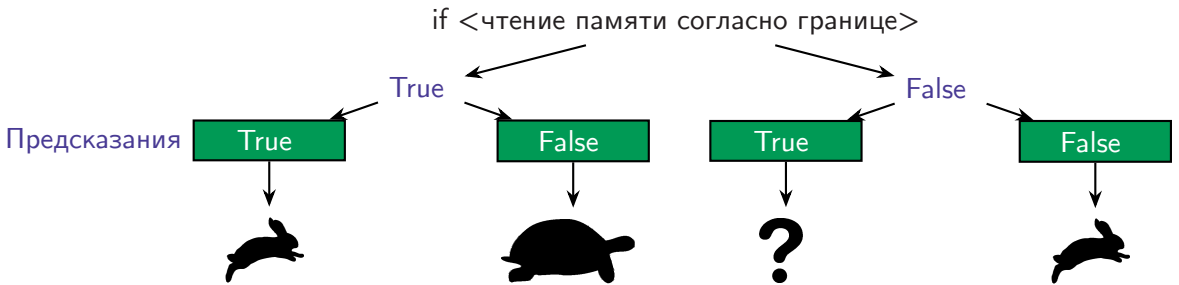
Побочные эффекты



В случае же, если предсказатель ошибся и решил, что вычисления будут происходить в границах памяти:

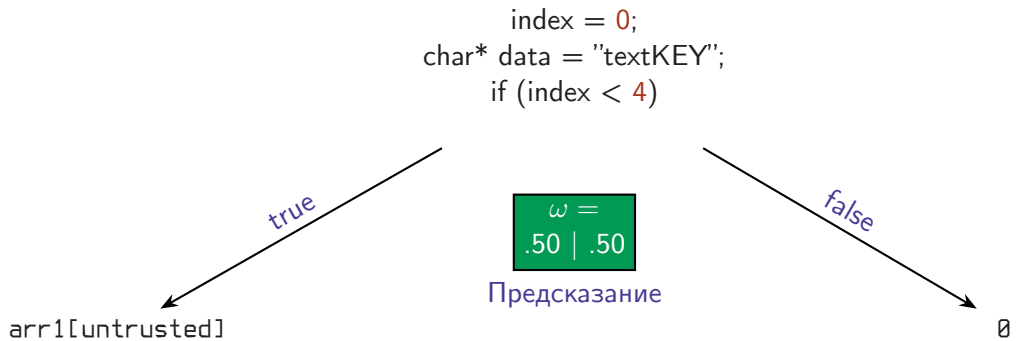
- спекулятивно выполнится код, который на практике выходит за границы памяти

Побочные эффекты

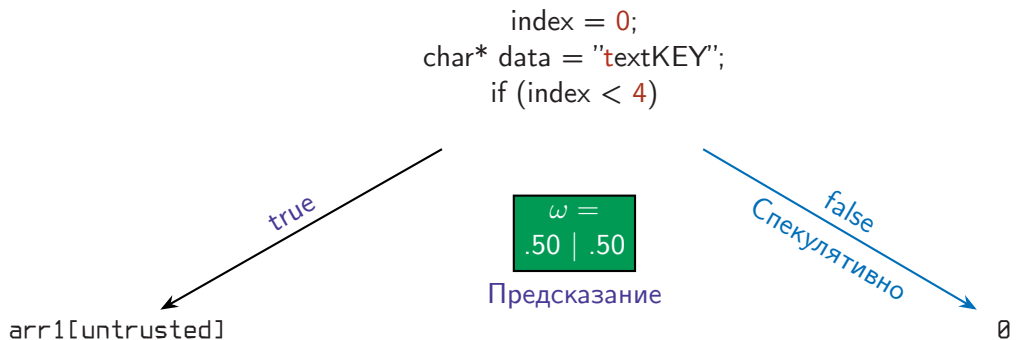


Спекулятивно выполнится код, который на практике **выходит за границы памяти**.

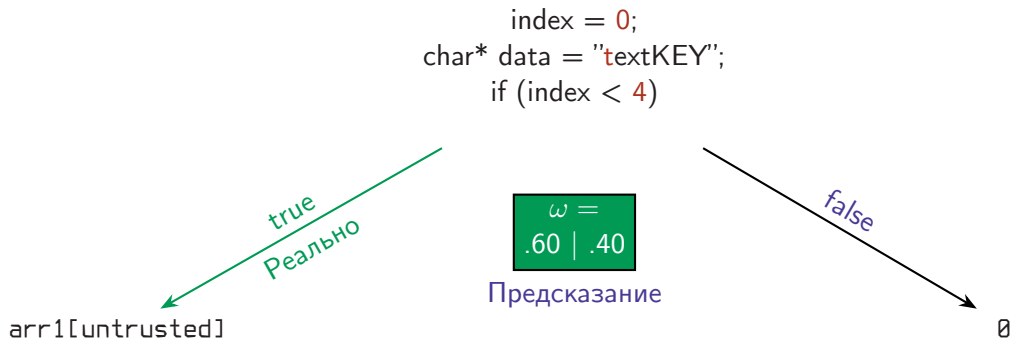
Тренировка предсказателя переходов



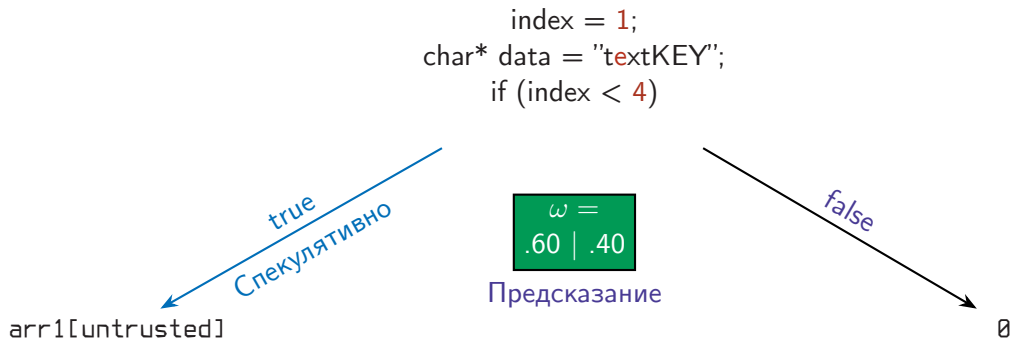
Тренировка предсказателя переходов



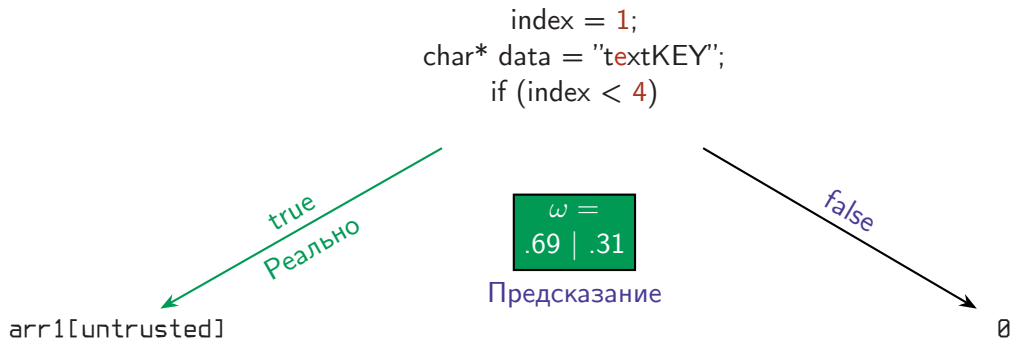
Тренировка предсказателя переходов



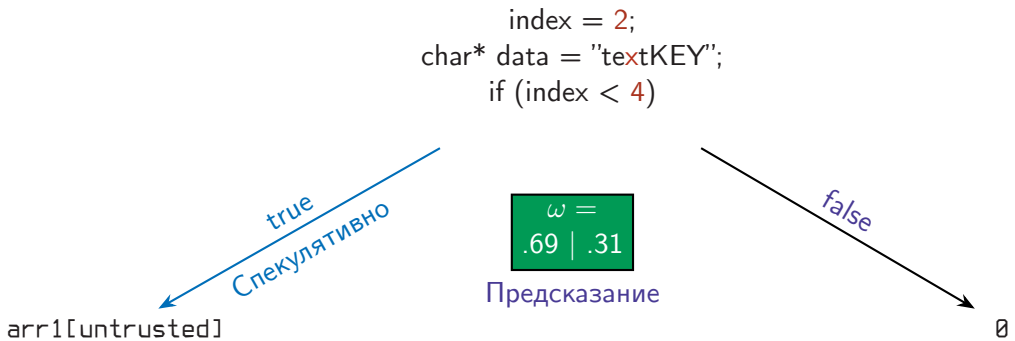
Тренировка предсказателя переходов



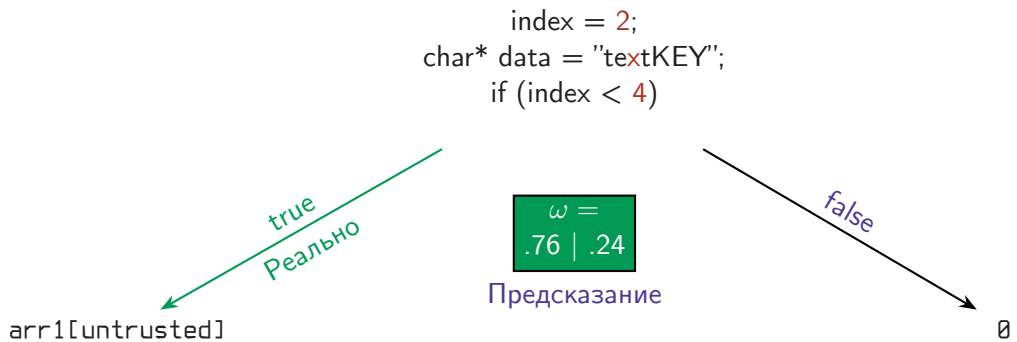
Тренировка предсказателя переходов



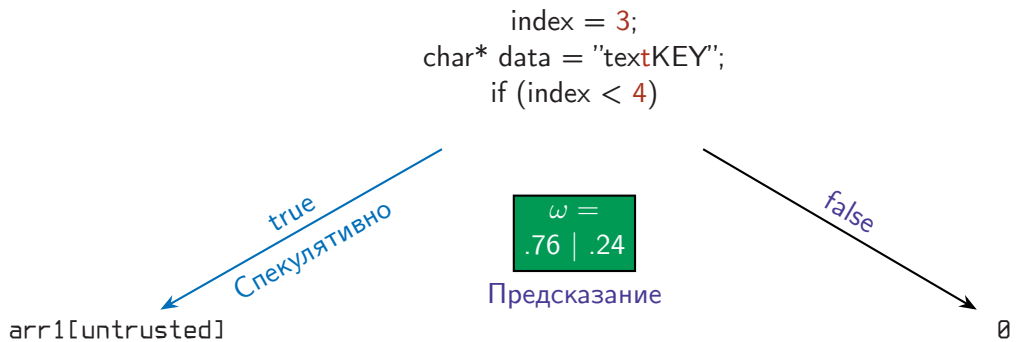
Тренировка предсказателя переходов



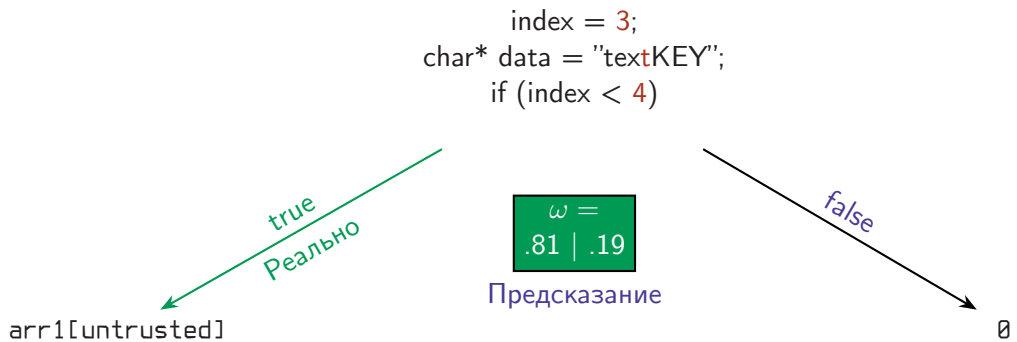
Тренировка предсказателя переходов



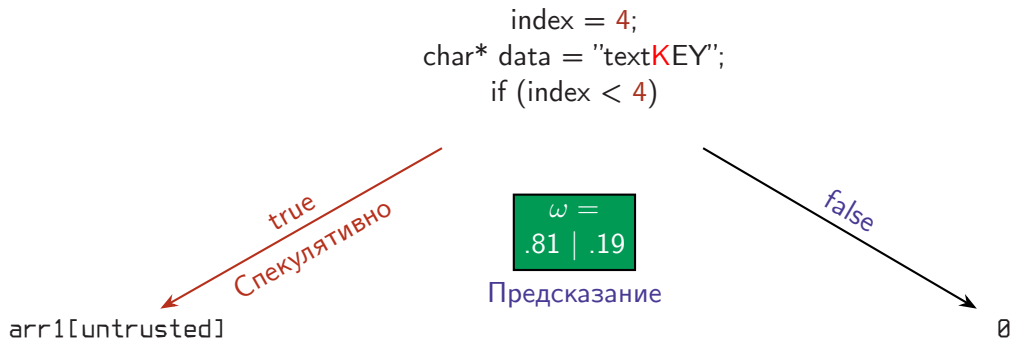
Тренировка предсказателя переходов



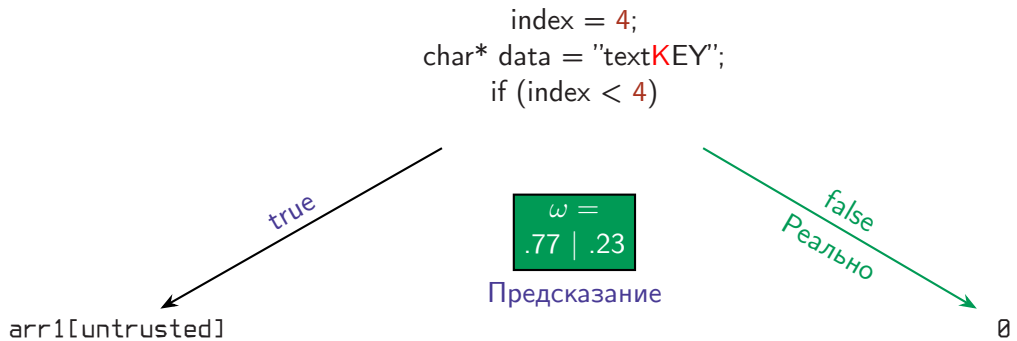
Тренировка предсказателя переходов



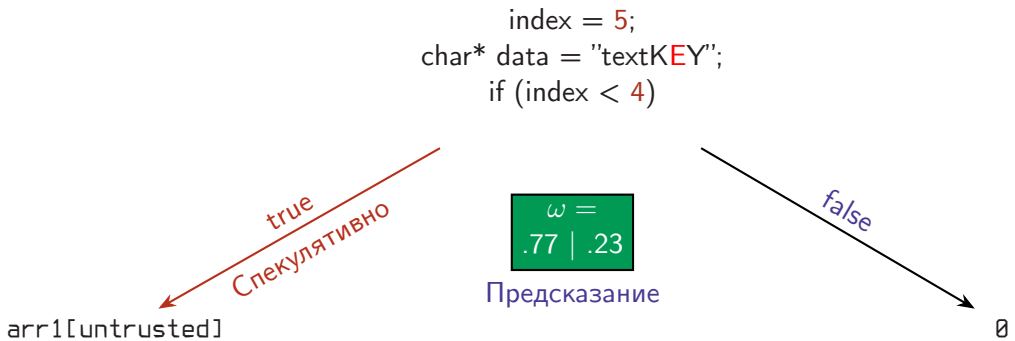
Тренировка предсказателя переходов



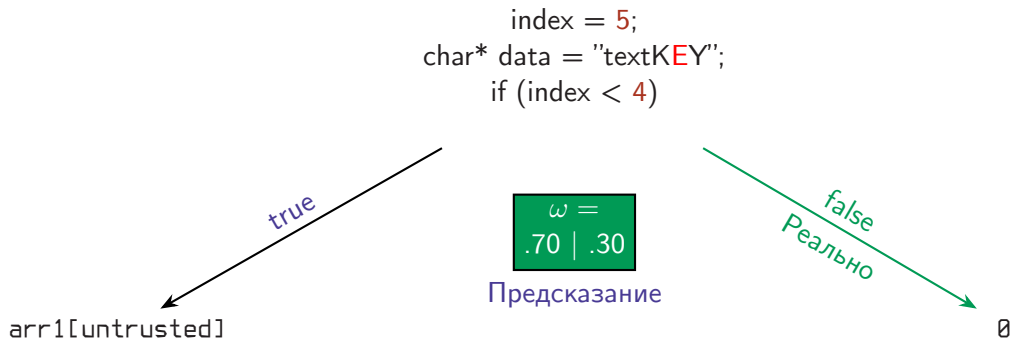
Тренировка предсказателя переходов



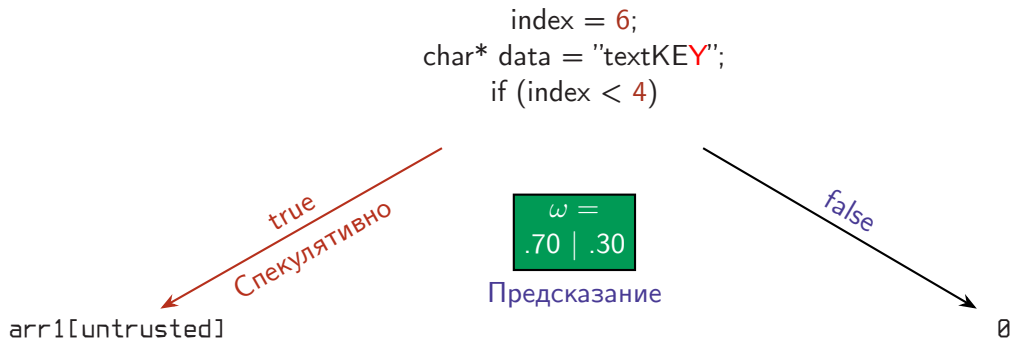
Тренировка предсказателя переходов



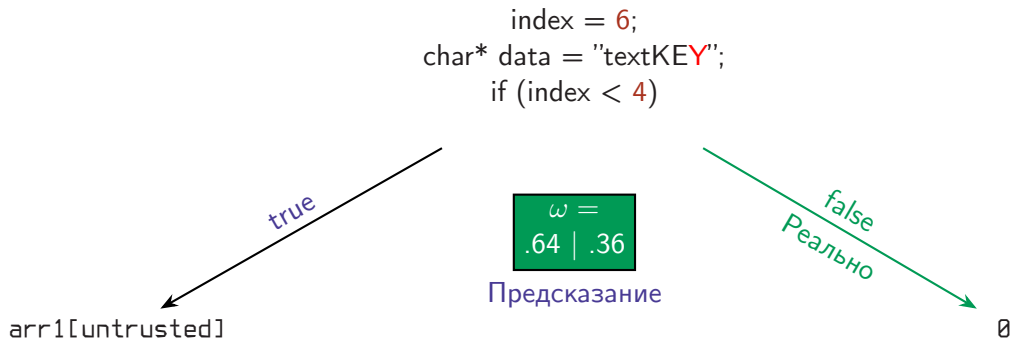
Тренировка предсказателя переходов



Тренировка предсказателя переходов



Тренировка предсказателя переходов



Обход проверки границ

```
struct array {
  unsigned long length;
  unsigned char data[];
};
struct array *arr1 = ...; /* небольшой массив */
struct array *arr2 = ...; /* массив размером 0x400 */
unsigned long untrusted_offset_from_user = ...;
if (untrusted_offset_from_user < arr1->length) {
  unsigned char value = arr1->data[untrusted_offset_from_user];
  unsigned long index2 = ((value & 1) * 0x100) + 0x200;
  if (index2 < arr2->length) {
    unsigned char value2 = arr2->data[index2];
  }
}
```

Существует множество вариантов эксплуатации данной уязвимости, рассмотрим одну из них.
Разберём всё по порядку.

Обход проверки границ

```
struct array {
  unsigned long length;
  unsigned char data[];
};
struct array *arr1 = ...; /* небольшой массив */
struct array *arr2 = ...; /* массив размером 0x400 */
```

Объявляется два массива.
Первый — целевой массив, в котором будет происходить обход границ.
Второй — массив для применения атаки на кэш.

Обход проверки границ

```
/* переменная, управляемая атакующим */
unsigned long untrusted_offset_from_user = ...;

/* проверка границ */
if (untrusted_offset_from_user < arr1->length) {

  /* спекулятивное выполнение, получение значения недоступной памяти */
  unsigned char value = arr1->data[untrusted_offset_from_user];

  /* получение значения бита интересующей области памяти */
  unsigned long index2 = ((value & 1) * 0x100) + 0x200;

  /* атака на кэш */
  unsigned char value2 = arr2->data[index2];
}
```

Проверка границ происходит как в примере представленном ранее.
Атака на кэш происходит как в примере, рассказанном ранее.
Код следующий за проверкой границ называется «гаджетом», как и в случае с ROP.
Многое зависит от устройства кэшей различных уровней, TLB, BTB (branch target buffers).

ASM

```
LDR X1, [X2]      ; X2 - указатель на arr1->length
CMP X0, X1        ; X0 содержит untrusted_offset_from_user
BGE out_of_range
LDRB W4, [X5,X0]  ; X5 содержит arr1->data
AND X4, X4, #1
LSL X4, X4, #8
ADD X4, X4, #0x200
LDRB X7, [X8, X4] ; X8 содержит arr2->data
out_of_range
```

Упрощённый asm код
Спекулятивно можно выполнить так же **ROP гаджеты**, заставить выполнять операции, тем самым считывать данные из **закрытых ресурсов**, например, крипто-чип.

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

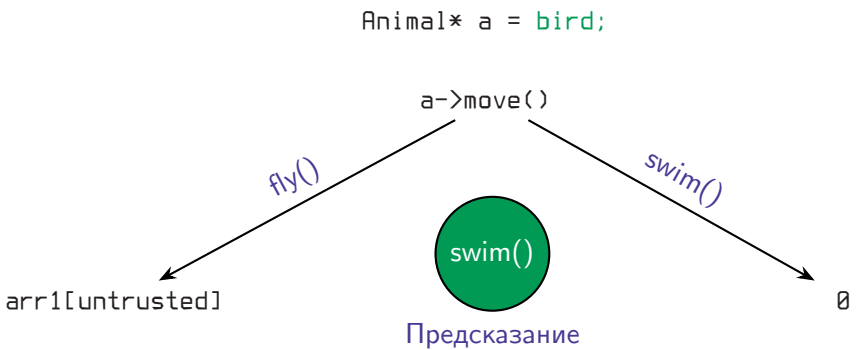
План

Variant 2

CVE-2017-5715: тренировка предсказателя переходов

Spectre

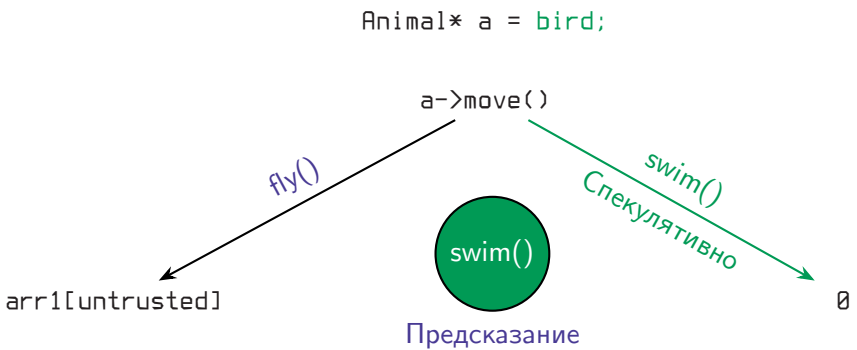
Предсказатель переходов и его тренировка



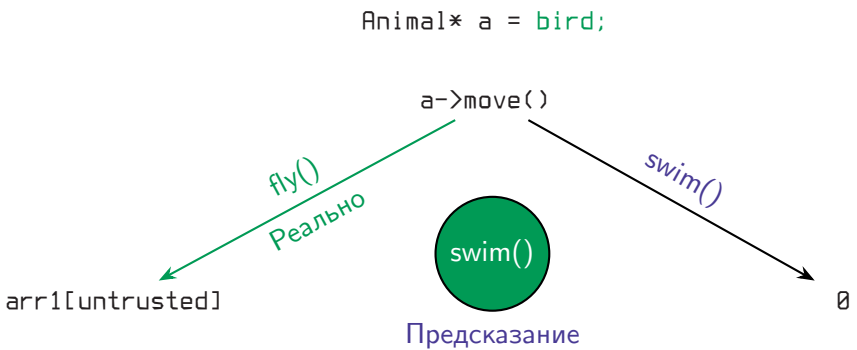
Ранее уже рассказывалось про предсказатель переходов, а также про буфер.

Для атаки требуется досконально знать, **как работает предсказатель переходов**.

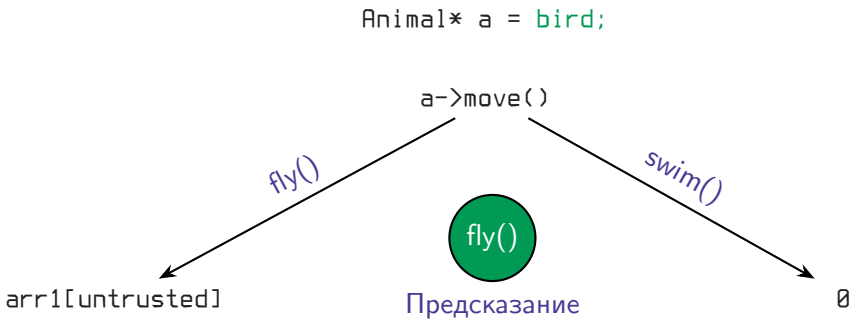
Предсказатель переходов и его тренировка



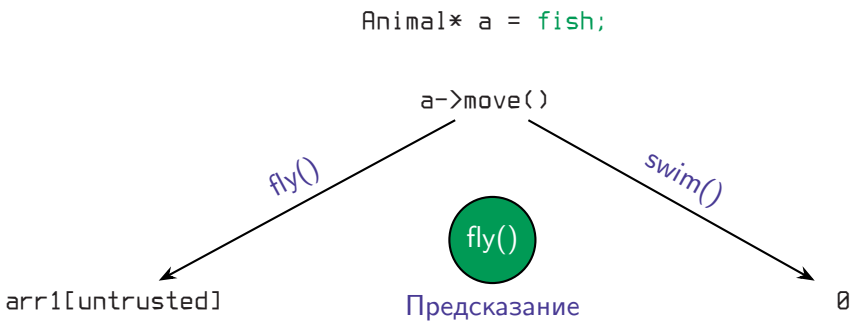
Предсказатель переходов и его тренировка



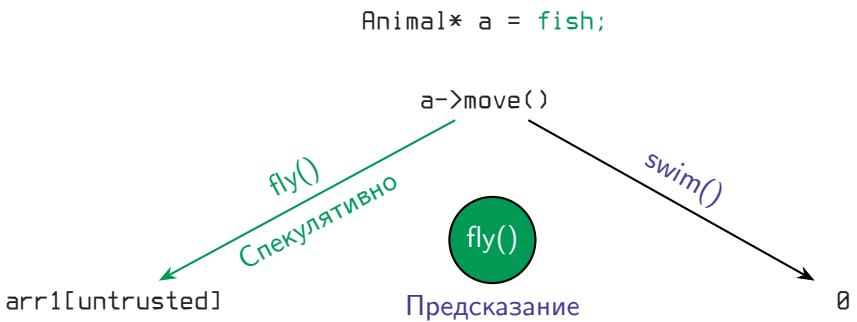
Предсказатель переходов и его тренировка



Предсказатель переходов и его тренировка

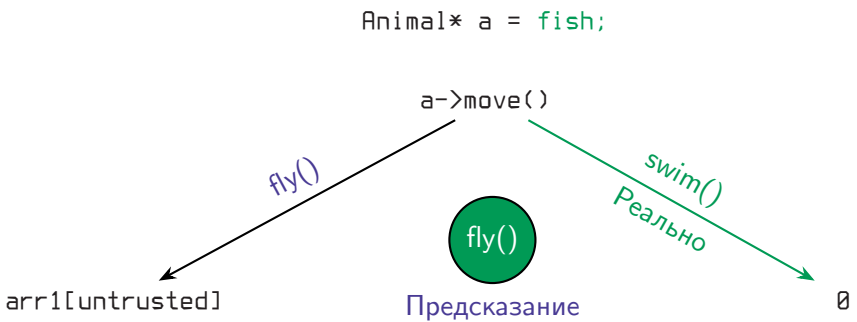


Предсказатель переходов и его тренировка

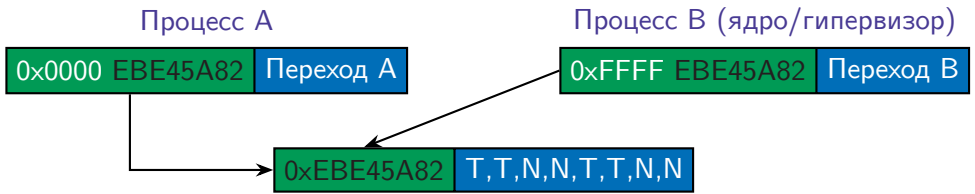


Внимание, спекулятивно выполняется натренированная нами ветка!
В итоге будет исполняться код, который совершит атаку на кэш.

Предсказатель переходов и его тренировка



Предсказатель переходов и его тренировка



В BTB используются виртуальные адреса, а также возникают коллизии

Из-за того, что возникают коллизии в таблице BTB, мы можем **натренировать** его таким образом, чтобы спекулятивно исполнялся нужный нам переход.

Существует **не один способ** тренировки предсказателя переходов.

Также существует возможность создания ROP цепочки из **гаджетов программы-жертвы** и натренировать на него, но для этого требуется знать адрес. Чтобы **узнать адрес перехода** можно применить **атаку по сторонним каналам** на предсказатель переходов.

И снова спекулятивное выполнение

```
if (untrusted_offset_from_user < array1_size)
  y = array2[((array1[untrusted_offset_from_user] & 1) * 0x100) + 0x200];
```

Ничего нового, используется всё тот же код, чаще всего ROP цепочка, которая приводит к атакам на кэш.

Предотвращение

Частично такое же, как и в случае с Variant 1

- отключение предсказателя переходов (Indirect Branch Restrict Speculation)
- очистка буфера предсказателя переходов при переключении контекста (Indirect Branch Predictor Barrier)
- выключение/включение MMU
- retpoline — «оборачивание» косвенных переходов

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

План

Variant 4

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

Spectre

Сначала чтение, потом запись

```
STR X1, [X2]   ; X2 - адрес памяти, который ещё не известен
...
LDR X3, [X4]   ; X4 содержит тот же адрес, что и X2
<произвольная обработка X3>
LDR X5, [X6, X3]  ; спекулятивное выполнение со старым адресом
```

Во многих современных процессорах применяются интересные техники оптимизации, а именно: **загрузка данных** из памяти по определённому адресу производится **раньше, чем запись** в тот же участок памяти в случае, **если адрес** на этапе записи данных **ещё не известен**.
В итоге у нас **спекулятивно выполняются** все операции **со старым адресом** в регистре (в том числе запись в кэш).

Читаем данные EL1

```
STR X1, [X2]
...
ERET            ; Возврат на более нижний уровень исключений
...
LDR X3, [X4]    ; X4 содержит такой же физический адрес, как и X2,
                ; но Виртуальный адрес отличается
<произвольная обработка X3>
LDR X5, [X6, X3]
```

Если **адрес один и тот же** (виртуальный и физический), то данная уязвимость эксплуатируема только **на одном уровне исключений** (exception level for ARM).
В случае, если есть возможность **чтения на одном уровне исключений**, а **загрузки на другом**, и при этом используется **один и тот же физический адрес**, то есть возможность эксплуатации уязвимости **на разных уровнях**.

Спекулятивное чтение одного и того же регистра

```
STR X1, [SP]
...
LDR X3, [SP]
<произвольная обработка X3>
LDR X5, [X6, X3]
<произвольная обработка X5>
LDR X7, [X8, X5]
```

Удивительно, но факт — мы можем читать данные из памяти, обращаясь по одному и тому же регистру.

Такое возможно благодаря **современным оптимизациям на некоторых архитектурах**. В случае, если у нас содержимое SP, например, **отсутствовало в кэше**, то **при записи** мы получим **cache miss** и задержку, позволяющую нам **спекулятивно прочитать всё те же данные**. Такое возможно по причине того, что процессор в RoB отслеживает, когда данные из регистра попали в кэш и соответственно ускоряет исполнение инструкций.

PoC нет!

Спекулятивный запуск непривилегированного кода

```
STR X1, [SP]
...
LDR X3, [SP]
...
BLR X3
```

Так же, как и в обычном Spectre, мы можем составить цепочку ROP гаджетов и спекулятивно их запустить, чтобы **записать нужные нам данные в кэш**.

Сначала запись, потом чтение

```
...
LDR X3, [X4]
<произвольная обработка X3>
LDR X5, [X6, X3]
....
STR X1, [X2] ; X2 содержит тот же адрес, что и X4
```

Утверждается, что существует возможность прочитать данные спекулятивно, записанные также спекулятивно, но позже. Это возможно в случае, если регистр для чтения будет высчитываться гораздо дольше регистра для записи.

PoC нет!

Предотвращение

- вставка «барьеров»
- отключение реорганизации операций чтения и записи
- SafeSpec

План

Meltdown & Spectre

Производные и не только

Производные и не только

Spectre-NG

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

TotalMeltdown?

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
- ML атака на TLB, с помощью Hyper-Threading технологии (Intel,

План

Заключение

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**
- 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**
- 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**
- 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**.