Нарушение защиты памяти между задачами Meltdown.

Полянин М.А.
20 июля 2019.

> "
> *аппаратурой авторы оптимизации выполнения нити x86 также не занимаются, но вот опасность генерации непроверенных запросов к памяти в многозадачной системе должна была их насторожить и заставить отвлечься от программной модели процессора*
>
> *процессор никогда не должен выполнять сбойные в терминах своего контекста обращения, авторы x86 не заметили проблему таких запросов, поскольку вероятно решили так:*
>
> > *- код выполняется в будущем, если в будущем контекст будет правильным, то этот код можно выполнить прямо сейчас;*
> >
> > *- если же в будущем контекст будет неправильным, то результат кода выполненного в прошлом можно будет в будущем выбросить, словно код не выполняется сейчас;*
>
> *Кто на ком стоял? Даже написать эти условные выражения, оперируя одновременно тремя временами, было не просто. И заваленные проблемами авторы x86 перепутали гипотетическое с реальным.*
>
> *Это примерно также как десятки тысяч фермеров в теории рыночной экономики, которая тоже не отличает реальную экономику от вымышленной, в теории рыночной экономики озабоченно бродят десятки тысяч фермеров, которые все время производят какие то излишки, как чудо-горшочки которые варят кашу, и чтобы полученные излишки не выбрасывать, фермеры упорно их обменивают на рынке с целью продать подешевле. А потом правила такой рыночной теории применяют к реальному миру и тысячелетия бедствий человечества вполне закономерный итог такого применения.*
>
> *вернувшись к процессору, оперируя формальными правилами и возможностью отменить непосредственный результат выполнения, авторы x86 не сообразили что отменить само сбойное обращение к памяти, после того как оно случилось, будет уже нельзя. Оперируя одновременно тремя временами они два раза выполнили операцию "не". Запутаться легко, если не питать дополнительной неприязни к нарушениям механизмов защиты предлагаемых*

процессором.
"

1.
Какие проблемы внес производитель x86, что привело к возникновению Meltdown?

Первая претензия сразу к производителю x86 на то, что обращения к памяти с нарушениями прав доступа должны отклоняться.

Для этого все обращения к памяти x86 должны были бы проверяться по базе, пределу и правам доступа, эти данные управления доступом находятся в дескрипторном кэше сегментного регистра и в TLB кэше для страницы, это все расположено внутри процессора, в таких корневых модулях как ALU, и скорость доступа к таким данным такая же высокая как к обычному регистру типа EAX.

Поэтому проверка таких прав намного быстрее чем само обращение к памяти и казалось бы такая проверка даже выгоднее, ведь системная память это узкое место и запросы туда должны быть максимально достоверными.

a)
Однако можно взглянуть на такой код

```
if(a<b)asm mov es, 0x01
if(c<d)asm mov ebx, 0x02
if(e<f)asm mov eax, es: [ebx]
```

из него видно что
- или надо выполнить сериализацию в точке "mov eax, es: [ebx]", что в совокупности с расходами на перегрузку дескриптора вызовет гнев программистов;
- или права доступа во время упреждающего "mov eax, es: [ebx]" можно казалось бы и не проверять, непонятно какой сегмент и какая страница используются при реальном выполнении;

выполнить упреждающий контроль записи в регистры ES и EBX и сразу по всем условным веткам не такая простая задача

более того, из за "эффектов косвенного изменения данных" вероятно не всегда возможно гарантированно предсказать модификацию адресов ES и EBX, и придется или перестраховаться и выполнять бесполезную сериализацию когда эти адреса не изменяются, или необходимая сериализация будет пропущена

в случае с Meltdown необходимая сериализация была пропущена

б)
Также у обращений к памяти с нарушениями прав доступа есть и иная проблема, на связанная с нарушениями программной защиты задач Meltdown.

Эта иная проблема в том, что процессор x86 при упреждающем выполнении может хаотически генерировать произвольные запросы чтения к памяти. Мы увидели эту вторую проблему когда произошла проблема Meltdown.

Обычно на ошибочные запросы к памяти любого происхождения и внимания не обращаешь, потому что у процессора всегда включена защита доступа к памяти для последовательного выполнения, а при спекулятивном выполнении она оказывается отключается и эти запросы вылетают на шину (чем бы она ни была).

Некоторые аппаратные устройства отображаются на память процессора и чтение по случайным адресам теоретически может вызвать аппаратную ошибку на устройствах не готовых к такому развитию событий и не имеющих помехозащищенного протокола для обмена с процессором (особенно для простых устройств, которые реагируют на чтение как на запись).

Проблема эта чисто гипотетическая, потому что из-за firmware кроме самого автора процессора никто для него устройства делать не станет.

Вообще интересно что процессоры с таким спекулятивным выполнением являются на своей шине аппаратными источниками хаотических сбойных запросов чтения памяти:
- запросов к несуществующей памяти (частично блокируется управляющими регистрами контроллера);
- запросов к запрещенной межпроцессорной памяти (частично блокируется управляющими регистрами контроллера);
- и т.п.;
это готовый эмулятор стресса системы, встроенный в процессор.

в случае с Meltdown потеря программной защиты памяти задач была вызвана как раз аппаратными особенностями работы на шине процессора такого устройства как кэш, из за этих сбойных запросов чтения памяти исходящих от процессора

2.
Как исправить обнаруженные проблемы?

Решений обнаруженной проблемы видится как бы два
- всегда выполнять бесполезную сериализацию и утратить производительность
- исправить проблемы связанные с генерацией процессором сбойных запросов чтения памяти

Надо выбрать вариант "исправить проблемы связанные с генерацией процессором сбойных запросов чтения памяти, производящихся в интересах выполнения нити".

потому что получаются не просто сбойные запросы, а запросы для обслуживания выполнения программы и логически нет никаких препятствий чтобы выполнять какие-либо запросы к памяти если это нужно для работы программы, в таком контексте запросы вполне осмысленные, не более бесполезные чем кэш-промахи

исправить же проблемы сбойных запросов можно следующим путем
- необходимо выполнять проверку доступа каждого обращения к памяти по текущим кэшам сегментного регистра и страницы

    это всегда гарантирует, что если у вас предыдущее состояние процессора и доступа к памяти корректное, то процессор не сможет выполнить сбойное в терминах этого контекста обращение

    это позволит задавать запрещенные области памяти в дескрипторе сегмента и в таблице страниц, что защитит аппаратуру от помех исходящих от процессора

    такое решение защитило бы привилегии задач и в случае с Meltdown

- необходимо иметь возможность привилегированными командами программно включать/отключать спекулятивное выполнение

    это нужно чтобы процессор мог работать пока первичное корректное состояние таблиц и дескрипторов не установлено или когда генерация таких запросов недопустима

3.
Виноват ли производитель x86 в возникновении Meltdown и если да то в чем именно?

Задача тех, кто делал "вычисление вперед" для x86 была такой:
- распараллелить последовательный поток;
- вычислять вперед (спекулятивно);
- сохранять временные состояния процессора;
- и т.п.

И в итоге сделать все так, чтобы именно для этого выполняющегося потока не возникло никаких побочных эффектов, чтобы для этого

выполняющегося потока все выглядело так, словно команды выполнялись друг за другом, чтобы
- исключения не возникали в точке позади текущей позиции выполнения;
- позже выполненные операции не влияли на предыдущие выполненные операции;
- и т.п.

Такая гарантия отсутствия побочных эффектов для выполняющегося потока это задача весьма непростая, отказы этой системы повышения производительности сулят серьезные случайные сбои при выполнении, но явных проблем в этой системе пока не обнаружили.

а)
И вопрос простой, при таких поставленных задачах, смогли бы вы сами или мог бы кто нибудь еще реально предугадать такой иной проблемный эффект Meltdown, как:
- доступ к защищенным данным;
- прямое использование значений этих защищенных данных;
на этапе спекулятивного выполнения для выбора способа измеримого изменения состояния аппаратных устройств компьютера?

Такое просто невозможно было предусмотреть тем, кто делает выполнение нити, поскольку на доступных программисту ресурсах программной модели процессора никаких эффектов от спекулятивного выполнения нет, а других вещей они не понимают и понимать не могут, они еще много какими профессиями не владеют.

б)
Можно обругать производителя за превращение процессора в источник хаотических сбойных запросов чтения памяти, однако мы выяснили что хотя производитель этого добился для себя случайно, но такие запросы в интересах выполнения нити логически сбойными не являются, они не более сбойные чем кэш-промах

аппаратурой авторы оптимизации выполнения нити также не занимаются, но вот опасность генерации непроверенных запросов к памяти в многозадачной системе должна была их насторожить и заставить отвлечься от программной модели процессора и выяснить у соседних инженеров вопрос "не наносится ли урон всей системе от обращений процессора к запрещенным регионам памяти, в добавок от обращений скрытых от программиста", это их упущение не доказать что сбойного обращения к памяти и печальных от этого последствий никогда не будет (независимо произойдет после этого Meltdown или нет).

процессор никогда не должен выполнять сбойные в терминах своего контекста обращения, авторы x86 не заметили проблему таких запросов, поскольку вероятно решили так:
- код выполняется в будущем, если в будущем контекст будет правильным, то этот код можно выполнить прямо сейчас;
- если же в будущем контекст будет неправильным, то результат кода выполненного в прошлом можно будет в будущем выбросить, словно

код не выполняется сейчас;

Кто на ком стоял? Даже написать эти условные выражения, оперируя одновременно тремя временами, было не просто. И заваленные проблемами авторы x86 перепутали гипотетическое с реальным.

Это примерно также как десятки тысяч фермеров в теории рыночной экономики, которая тоже не отличает реальную экономику от вымышленной, в теории рыночной экономики озабоченно бродят десятки тысяч фермеров, которые все время производят какие то излишки, как чудо-горшочки которые варят кашу, и чтобы полученные излишки не выбрасывать, фермеры упорно их обменивают на рынке с целью продать подешевле. А потом правила такой рыночной теории применяют к реальному миру и тысячелетия бедствий человечества вполне закономерный итог такого применения.

Вернувшись к процессору, оперируя формальными правилами и возможностью отменить непосредственный результат выполнения, авторы x86 не сообразили что отменить само сбойное обращение к памяти, после того как оно случилось, будет уже нельзя. Оперируя одновременно тремя временами они два раза выполнили операцию "не". Запутаться легко, если не питать дополнительной неприязни к нарушениям механизмов защиты предлагаемых процессором.

4.
Любая катастрофа это чаще всего сочетание нескольких неблагоприятных факторов, а не единственный промах.

Появление на x86 сбойного обращения к памяти это серьезное упущение, но непосредственно эта проблема процессора x86 не вдет к эффекту Meltdown.

Но вы же сами видите произошедшую проблему, почему все же это случилось, в чем заключается та систематическая ошибка для процессора x86, которая привела к такой неприятности как нарушение защиты памяти между задачами при оптимизации выполнения?

Мы даже перефразируем этот вопрос чтобы подвигнуться к пониманию: "если взять и устранить конкретную ошибку процессора x86, гарантирует ли это что подобные ошибки в программах для x86 впредь не появятся?".

Нет, не гарантирует. Но почему это так?

Вот простое объяснение, представьте что вы возьмете адреса данных со стека своей функции и начнете эти адреса записывать в статической памяти. Если вы будете так делать, то спустя какое то время ваши программы с таким кодом дадут сбой. Неизбежно.

Есть еще очень много всяких подобных вещей, которые программист никогда не делает и не делает именно по той причине что такие

действия прямо ведут к проблемам и ошибкам выполнения.

И аналогично для x86, нарушение защиты памяти между задачами Meltdown вызваны вовсе не блоком оптимизации выполнения для x86, а именно такими действиями программистов пишущих ОС, действиями которые прямо ведут к проблемам и ошибкам выполнения.

5.
И давайте эти действия немного перечислим.

а) вопрос, что в адресном пространстве задачи в проблемных ОС делают привилегированные данные?

Посмотрите на архитектуру x86, это вовсе не большой процессор Z80 с 64 битной памятью. Наверное в детстве мы все о таком мечтали, нам казалось что это будет хорошо и не хватает именно такого.

Но доступ к сервису ОС в x86 должен идти только через шлюз, если вы хотите применить хороший механизм защиты, необходимо активировать систему защиты.

Оказалось что на больших процессорах взаимодействие программ между собой приобретает новое по сравнению с малыми компьютерами значение, определяет характер происходящих проблем.

В ОС на базе x86 в адресном пространстве все должно быть наоборот, попав в код ОС вы имеете доступ к адресному пространству приложения, а из приложения никакие защищенные участки памяти ОС недоступны, их не адресуют сегменты, на них не отображены страницы.

У приложения есть только точки входа в ОС, вызов которых приводит не к упреждающему выполнению кода пользователя, а к смене контекста кодом встроенным в процессор.

И сделано такое разделение именно для этого, чтобы исключить такие казусы как Meltdown. Чтобы исключить больше проблем связанных с непредвиденными случаями сбоя в аппаратуре или в программах.

Нарушение этой схемы защиты x86 это умышленная ошибка программиста ОС, который пренебрегает схемой безопасности встроенной в процессор, и как ни странно, пренебрегает только из-за того, что есть другие процессоры в которой такой схемы шлюзов нет.

Процессор x86 может работать как другие, в чем то более простые процессоры, именно этот режим x86 и используется в проблемных ОС. Это их попытка написать код ОС для абстрактного "обычного процессора", у которого "механизмы защиты обязательно есть, но какие

непонятно".

б) вопрос, что в адресном пространстве задачи в проблемных ОС делают данные соседних процессов?

Аппаратура защиты x86 не такая изощренная как ей надо было бы быть, авторы x86:
- допускали что их центральный процессор может применяться даже как контроллер;
- также они поддержали работу кода ОС написанного для "обычного процессора";
что оставило свой след на архитектуре.

Но вот в защищенной от сбоев ОС у вас практически нет никаких вариаций, кроме того как следовать схеме ролей для компонентов ОС жестко встроенной в x86.

Не буду приводить пример кода, но попытки нецелевого использования компонентов защиты x86 сразу приводят к тому, что это порождает код входа в сервисы ОС состоящий из многих сотен команд и многих десятков обращений к памяти, и этот workaround код применяется только для того чтобы справиться с самой аппаратурой защиты x86.

И такой workaround вход в сервис ОС намного хуже чем команда INT_XX с автоматическими действиями x86 по смене уровней привилегий или нитей.

При взаимодействии задач в системе x86 вам надо:
- с одной стороны скрыть адресные пространства ОС от приложений
- с другой стороны предоставлять адресное пространство любого приложения для сервиса ОС
- с третей стороны скрыть адресные пространства нитей друг от друга

Аппаратура защиты x86 прямо не поддерживает у задачи модули, поэтому внутри нити все монолитное как на i8086, приватные сегменты модулей:
- доступны друг другу (стек у модулей тоже общий);
- активируются через явную загрузку сегментного регистра (сегмент и так чаще всего только один);
так что принципиального урона нет, но защита приватных сегментов модулей утрачивается.

Если короче и без подробностей, то в системе x86 у вас есть только нити связанные с TSS.

Аппаратных дескрипторных таблиц на x86 две и автоматически перегружается только регистр LDTR, но все системные данные включая дескрипторы LDT и TSS хранятся только в GDT, отчего ваша любимая команда будет LGDT.

Но в итоге этими таблицами достигается то, что приватные данные каждой нити хранятся в отдельных таблицах, ситуация защиты между нитями точно такая же, как в случае защиты между приложением и ОС.

в) вопрос, что в проблемных ОС делают страницы?

Страничный механизм x86 предусмотрен вовсе не для защиты памяти, защитой памяти в процессоре x86 занимается сегмент.

Если вы посмотрите на схему программной модели процессора x86, то увидите что в его сегментном регистре, который всегда находится в самом быстром кэше процессора, много скрытых полей, часть которых берется из дескриптора сегмента и содержит такие данные как "база" и "предел сегмента".

Каждое обращение к памяти процессора x86 всегда проходит через вполне определенный сегмент и параметры "база" и "предел сегмента" для этого сегмента всегда доступны процессору в кэше сегментного регистра и должны проверяться процессором при любом обращении к памяти.

В проблеме meltdown нет нарушения предела сегмента, читаются существующие данные, которые защищены только таблицей страниц.

страницы процессору x86 нужны вовсе не для защиты памяти, а для
- отображения большой виртуальной памяти задачи частями на малую физическую память

- виртуализации на 32 битных процессорах x86 линейных адресов для задач с большими сегментами
    на 32 битных процессорах x86 максимальный размер сегмента равен размеру физической памяти

    а еще и модель памяти у процессора x86 особенная (модель памяти x86 см. ниже), которая часто заставляет задействовать полный 32 битный диапазон адресов сегмента, при том что в таком сегменте в середине будет много неиспользуемой памяти

    на 32 битных процессорах x86 режим PAE не встроен в дескриптор сегмента, не включая страницы несколько больших сегментов нельзя расположить в физической памяти рядом друг с другом

    и поэтому если на 32 битных процессорах не включить страницы, то уже две задачи с большим максимальным размером сегмента (статическим или с динамически изменяемым во время выполнения), не смогут работать одновременно из-за пересечения их линейных адресов

> это не так на 16 битных x86, где предельный размер сегмента 64К, что намного меньше размера физической памяти, а 64К это заодно и максимальный размер страницы, так что на 16 битных x86 страницы можно не включать

страницы процессору x86 нужны для работы с памятью одной нити, а не для взаимодействия между нитями

и это не первая "ошибка защиты страниц процессора x86", из за некорректного использования построителями ОС страничного механизма процессора x86, страничного механизма который для такого совершенно не предназначен по своей природе

на сколько мне известно, все "ошибки защиты страниц процессора x86" никогда не были связаны с нарушением защиты сегмента

использовать страничный механизм процессора x86 заместо сегмента можно, но сила защиты страниц меньше и в результате будет больше проблем с нарушением защиты

6.
процессор x86 обладает множеством иных особенностей:

а) в процессоре x86 нет регистров общего назначения, что пришло к нему от i8080

все регистры x86 это операнды команд и результаты их выполнения, процессор x86 ориентирован строго на операции типа a=b+c с данными a,b,c размещенными в памяти

но адреса для a,b,c было бы невозможно прямо выразить в опкоде без применения регистров, это будет очень громоздкий опкод, адресов в общем три штуки

поэтому вы выполняете операции типа a=b+c за несколько опкодов x86, в общем
- загоняя данные из памяти в регистры
- выполняя операцию над регистрами
- и сохраняя результаты из регистров в память

конечно нет нужды буквально загонять каждый операнд в регистр x86, есть возможность комбинировать операции и доступ к памяти, если адресов памяти одновременно используется мало, но смысл всего этого сохраняется именно такой

именно поэтому регистров x86 так мало (контекст процессора x86 маленький) и регистры x86 такие асимметричные, странные для тех кто знаком с иными процессорами

попытки использовать эти регистры как общие данные между несколькими идущими подряд разными операциями типа a=b+c это собственное развлечение программистов

б) модель памяти у процессора x86 особенная

процессор x86 снабжен системой сегментов, но жестко завязан на "плоскую модель памяти С образца 1970-х годов", это выглядит как единственный сегмент данных, представленный сегментными регистрами DS и SS, при этом
- у DS и SS общая база;
- DS растет вниз и его предел самый первый байт 0, через DS
    - защищается адрес 0 как NULL
    - доступен весь физический сегмент данных, кроме адреса 0
    - доступны данные DATA,BSS, они начинаются снизу от адреса 0 + выравнивание 2 или 4 и растут вверх
    - доступны данные STACK, они начинаются сверху от адреса 0 - выравнивание 2 или 4 и растут вниз
- SS растет вниз и его предел это адрес посредине сегмента задаваемый с помощью sbrk(), через SS
    - защищается адрес 0 как NULL
    - защищаются данные DATA,BSS от автоматического декремента (e)SP при создании фрейма стека
    - в операциях процессора x86 данные DATA,BSS также недоступны
    - доступны только данные STACK

через DS весь сегмент данных оптимально доступен для всех обычных опкодов x86 для доступа ко всем данным
через SS только данные STACK оптимально доступны для всех опкодов x86 ориентированных на стек

еще есть регистр CS, который позволяет организовать еще один отдельный от DS/SS сегмент, это сегмент кода который содержит функции, а эти функции общие на все копии одной задачи (сегмент данных напротив, отдельный и свой для каждой копии одной задачи), при этом
- CS растет вверх и его предел равен размеру кода задачи, через CS
    - не защищается адрес 0 как NULL
    - отметим что "указатель на данные" не совместим с "указателем на код" (адресует разные сегменты при равном значении)
    - защищается код CODE от записи и даже от чтения
    - доступен весь сегмент кода CODE

через CS весь сегмент кода оптимально доступен для всех опкодов x86 ориентированных на выполнение команд

такая схема сегментов CS, DS/SS позволяет

иметь "указатель на данные", как требует "плоская модель памяти"
эффективно использовать опкоды x86
эффективно использовать память разделяя один код на несколько копий задачи

такая схема раздельных сегментов используется на 16 битных процессорах x86

на 32 битных процессорах x86 в такой схеме предельный размер сегмента DS/SS уменьшается на размер CS, при этом
- начало DS (предел DS) поднимается вверх на размер CS, уменьшая максимальный размер сегмента DS/SS;
- SS не изменяется;

такая схема раздельных сегментов используется на 32 битных процессорах x86 при включении страниц

это потому, что силами аппаратуры x86 невозможно одновременно разместить в памяти сегменты одной нити, суммарный размер которых больше 4 Гбайт, их линейные адреса начинают перекрываться

на 32 битных процессорах x86 PAE это "серверная" возможность, которая позволяет запускать много маленьких (по 4 Гбайта) 32 битных задач на большой памяти, заместо "клиентской" возможности запуска одной большой 32 битной задачи с общей адресуемой памятью более 4 Гбайт и причина тому 32 битный линейный адрес, сегменты тут не виноваты

есть вариация когда на 32 битных процессорах x86 предельный размер сегмента приложения задан в ОС намного меньше 4 Гбайт, при этом
- начало DS (предел DS) поднимается вверх до нужного максимального размера сегмента;
- SS не изменяется;

такая схема раздельных сегментов используется на 16 битных процессорах x86
такая схема раздельных сегментов используется на 32 битных процессорах x86 при выключении страниц

есть вариация когда сегмент кода размещается впереди данных, создавая только один CS/DS/SS сегмент, при этом
- у CS, DS и SS общая база (это логически один сегмент);
- есть вариации начала DS (вариации предела DS с доступом к CS через DS или без доступа);
- SS не изменяется;

такая схема сегментов используется на 16 битных процессорах x86
такая схема сегментов используется на 32 битных процессорах x86 при включении страниц

вот и все, при всей потенциальной гибкости сегментной архитектуры x86 попытки реально как то иначе использовать процессор x86 это прямо и в открытую враждовать с архитектурой этого процессора и процессор отплатит вам тем же самым, враждой, злостной перегрузкой регистра ES, отсутствием alloca и т.п.

> как правило размер сегмента кода статический, а используемый размер сегмента данных динамически зависит от тех данных которые поступают приложению для обработки при каждом конкретном запуске и размер варьируется в очень широких пределах, поэтому расположение сегментов CS/DS/SS именно такое, позволяющее распределять память данных между стеком и кучей по потребности
>
> процессор x86 позволяет использовать межсегментные адреса, но это относится к поддержке IPC силами процессора, а в обычном режиме выполнения сегментный контекст процессора не изменяется и используются только смещения

попытка эмулировать на процессоре x86 какой то иной процессор ничем хорошим не закончится

в) еще процессор x86 не поддерживает нити так как вы бы их ожидали увидеть

на нити в x86 нет сегментов и системы команд, вернее с точки зрения программиста, из за слияния DS/SS процессор x86 поддерживает нити, но не поддерживает процессы (не поддерживает статические данные общие для всех нитей)

статические данные общие для всех нитей для x86 доступны через IPC (неэффективно, через вспомогательные сегментные регистры и если для 32 битных режимов x86 для статических данных процесса есть регистры FS и GS, то для 16 битных режимов это только перегрузка сегментов из памяти)

на 32 битных процессорах x86 доступная память нити 4 Гбайт делится на
- данные процесса (хороший доступ через FS)
- данные нити (самый эффективный доступ через DS/SS)
- кода процесса (самый эффективный доступ через CS)

Варианты нитей конечно бывают разные, например с помощью страниц для x86 можно создавать нити-функции как ветвления стека, т.е. для каждой нити создается своя таблица страниц в которой с момента вызова функции создается свой стек отображением тех же адресов на отдельную физическую память, а вся остальная память DS/SS остается для всез нитей общей (память процесса)

Но с помощью сегментов x86 такого сделать нельзя, для сегментов x86 каждая нить это отдельный SS, который наследует копию данных предыдущего SS или не наследует и новая нить получает пустой стек.

г) процессор x86 имеет только встроенные в опкоды префиксы сегментов

и раз уж разговор зашел о Meltdown и ошибке разработчиков x86, то поддержка процессором x86 только статически компилируемой адресации сегментов в виде префиксов в опкоде это еще одна серьезная ошибка разработчиков x86

которые начиная с i286 не сообразили что для модульной архитектуры надо уметь индексировать сегментные регистры, а не только статически задавать их префиксы в опкоде

причина та же самая что и с Meltdown, они делают процессор а не пишут программы и сами по себе они не очень понимают почему если через опкод можно адресовать регистр AX, то через такого же типа опкод нельзя адресовать регистр сегмента, ведь и тот и другой регистры

решение разработчиков x86 сделало сегменты процессора x86 враждующими с программистом, сегменты не любят все программисты x86, которые в добавок не любят маленький размер 16 битных сегментов

на самом деле сегменты это очень здорово, это аппаратная поддержка модулей, но сегменты нуждаются в поддержке опкодов как минимум в виде индексирования, без этой поддержки сегменты практически бесполезны, невозможно написать код, который бы использовал сегменты как параметр

Здесь проблема только в том, что программисты производителя процессора x86 не ставили такую задачу, а те кто делают железо x86, те сами программы не пишут и за все 40 лет они вряд ли осознали в чем тут проблема.

===
Конец текста