MDS Attacks + CROSSTalk

@elliptic_shiho

概要

- Meltdown [3]は大まかには既知とする
  - わからないところがあれば都度聞いてください
  - 参考: [10], [11]
- MDS Attacks, CROSSTalk共にMeltdownの応用手法
  - MDS Attacksはある性質を持ったMeltdownの応用手法の総称
  - CROSSTalkはMDS Attacksの手法を応用したもの
- 対象は主にSkyLake

概要

- MDS Attacks [16]
  - MDS ⇔ Microarchitectural Data Sampling
  - CPU内の様々なバッファをMeltdownの応用で取得する手法の総称
    - I/Oバッファ, L1キャッシュとL2以下/メモリとの中継ぎをするバッファ, ロード命令のバッファ, ストア命令のバッファ, …
  - このうち今回はRIDL[5]を解説
  - 並行して報告されたZombieLoad [9]を含め, Meltdownから派生した脆弱性のーサブカテゴリを成している

概要

- CROSSTalk [1]
  - 2020/06/09に公開されたRIDLの応用手法
  - これまでのMeltdown系手法では異なる物理コアで実行されているプロセスの実行内容は不可能であったが, 一部の命令についてはこれが可能であることを示した
  - ここではrdrand命令, rdseed命令を対象とし, 実際に出力を(他の物理コアから)推測することに成功した

準備

- ページング
- Intel CPUのキャッシュ機構
- Transient Execution
- Intel TSX

ページング

- ユーザプロセスが物理メモリアドレスを直接扱うのは都合が悪い
  - 他プロセスへの干渉が簡単なのはセキュリティ的にどうか
- それぞれのプロセスに独立な論理メモリ空間(仮想メモリ)を割り当てることでプロセスごとにメモリ空間を分離
  - ページングは仮想メモリアドレスと物理メモリアドレスの対応付けをする手法の一つ

ページング

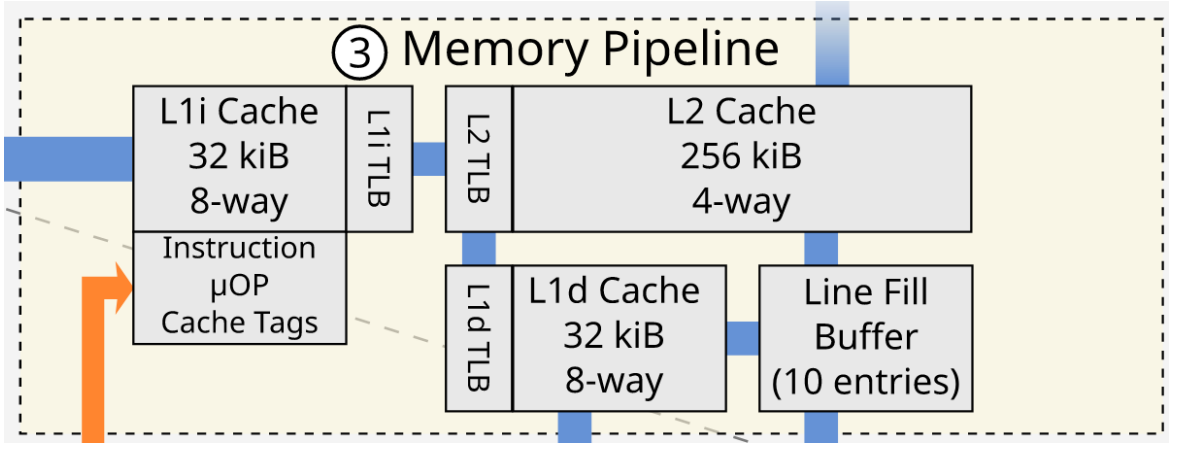
- ページング方式では, 仮想メモリ空間をページと呼ばれる小さな単位(e.g. 4KiB)に分割して管理
  - ページ単位で仮想・物理メモリアドレスの対応表(ページテーブル)を作る
  - アクセス権等もページ単位で管理
- ページング方式では, ページへの初回アクセス時にPage faultというfaultを発生させ, 物理メモリへの紐付けを動的に行う
  - 紐付けはカーネルの作業
  - 物理アドレスの紐付けが可能な仮想アドレスをmapped, そうでない場合をunmappedと呼ぶ

ページング

- ページテーブルは複雑
  - 例: プロセスごとに独立なのであるプロセスから見た0x40000と別のプロセスから見た0x40000は全く別の物理メモリにマップされる
  - これをメモリ上にあるテーブルを(4KiB分割の場合は最高4回)引いて解決するが, 多段間接参照は一般に極めて重い処理
- このためTranslation Lookaside Buffer(TLB)と呼ばれるバッファで仮想 – 物理の対応付けをキャッシュしておく
  - タスクスイッチ毎に消去される (最近はそうでもないらしいが未確認)

Intel CPUのキャッシュ機構

- メモリからデータを読むときのフローを概説
  - L1d, L2が主な登場人物

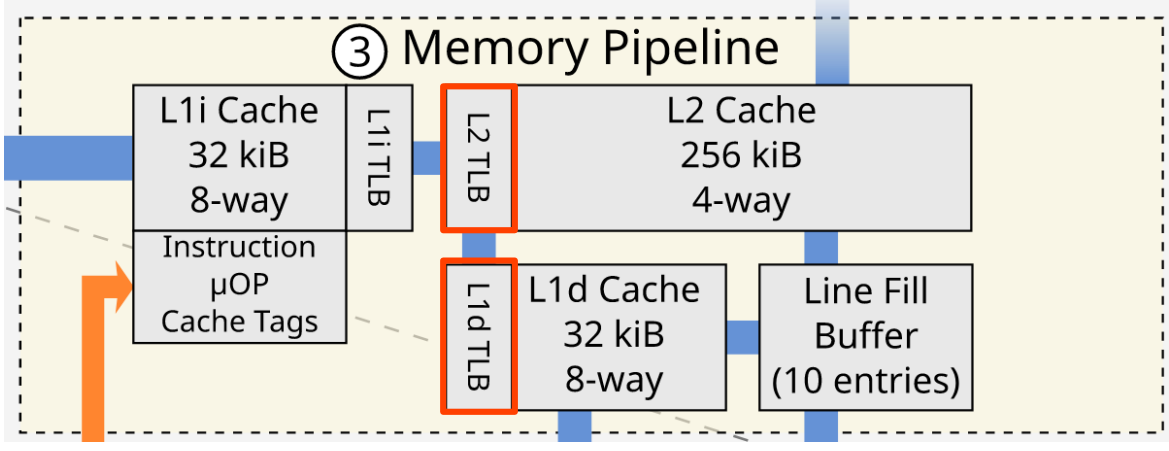


画像: [5] Fig.9より引用

Intel CPUのキャッシュ機構

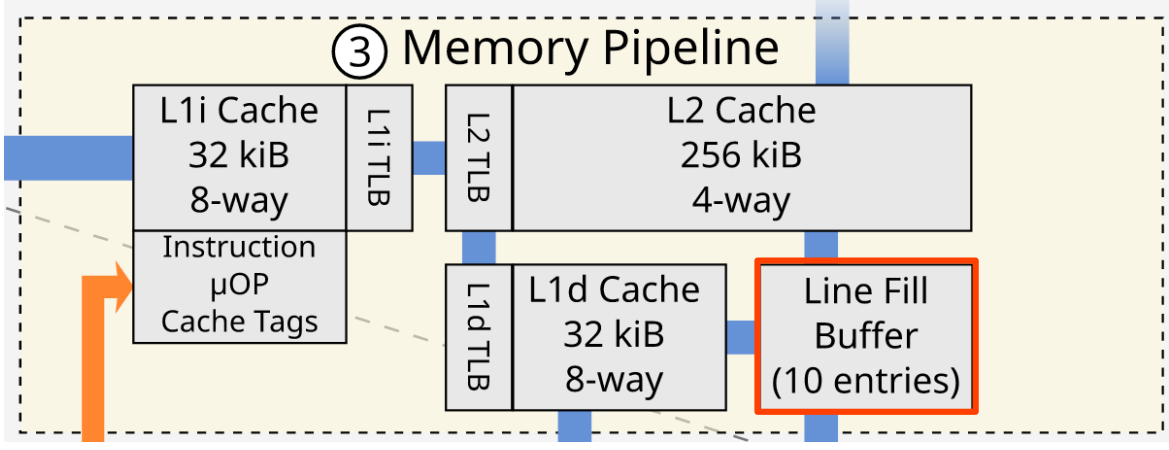
- あるアドレスAをロードしたい
  - まずTLBをあたって物理アドレスへの変換を試みる

- AがTLBに存在すればそのまま, なければページテーブルを引いて解決
- このアドレス変換中は命令はブロックされる



Intel CPUのキャッシュ機構

- 得た物理アドレスで, Line Fill Buffer(LFB)を検索
  - LFBとは?



Intel CPUのキャッシュ機構

- LFBは主にCPUの外とキャッシュの中継ぎをするバッファ
  - ロード･ストア命令のスループット向上が主な目的
- ロード時には対象のアドレスをLFBに登録する
  - LFBにアドレスが登録されたらL2, L3, メモリの順に探す処理が非同期に走る
  - オペレーション終了時にデータをL1dキャッシュに書き込んでLFB側は解放
  - ロード命令の実装を単純化する意図もある? (発表者予想)

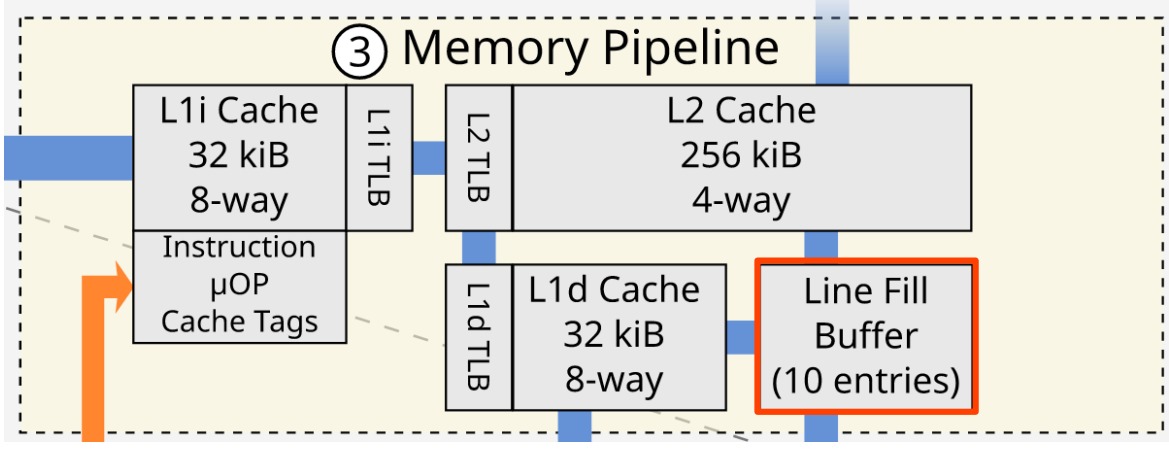
Intel CPUのキャッシュ機構

- ストア時もエントリ確保の上アドレス･データを書き込む
  - 特に, L1dキャッシュから追い出す操作がLFBを確保してデータを書き込むだけになる
    - ノンブロッキングな追い出し
    - ストアは後ろで非同期に行われるので, L1dキャッシュへの書き込みがよりスムーズになる
    - メモリ書き込み前にロードが走った時はLFBからキャッシュに差し戻せばよい
- 実際にはLoad buffer / Store bufferも絡むが, 簡単のため省略している

Intel CPUのキャッシュ機構

- 以上を踏まえて解説に戻る

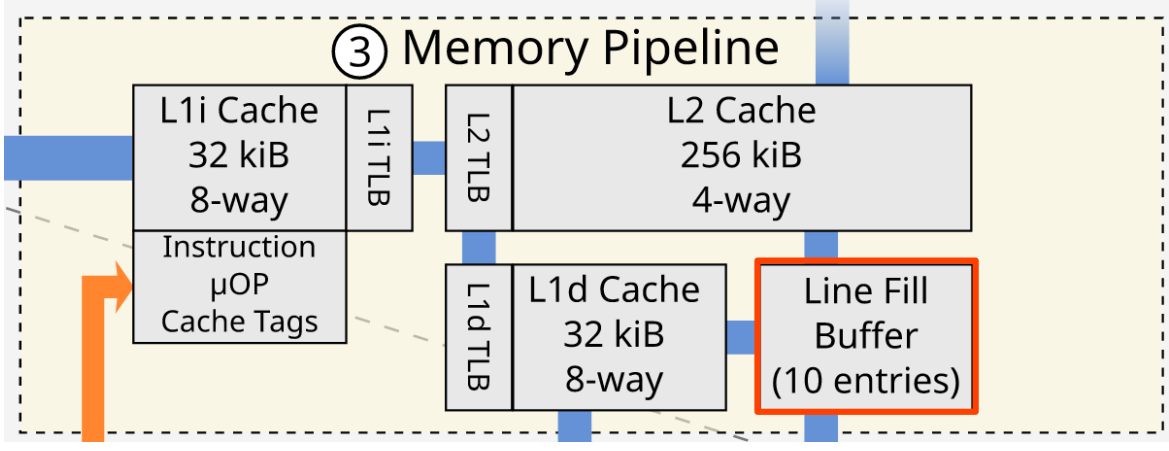
- LFBの内容はL1dと同じかより新しいはず
- 当該アドレスに対しLFBエントリ(ロード)が存在するならL1dに反映されるのを待てばよい



Intel CPUのキャッシュ機構

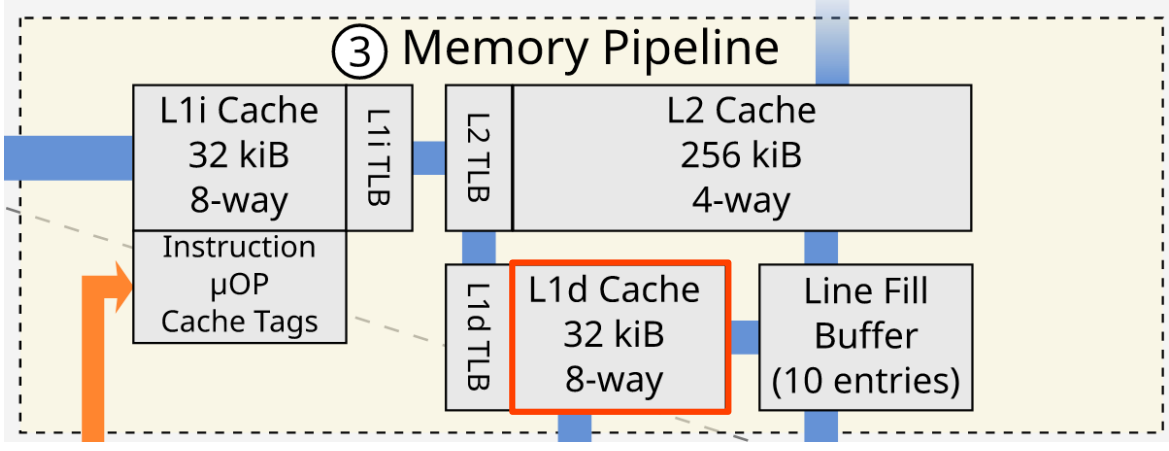
- 以上を踏まえて解説に戻る

- LFBエントリ(ストア)の場合はストアする内容をそのままL1dにコピー



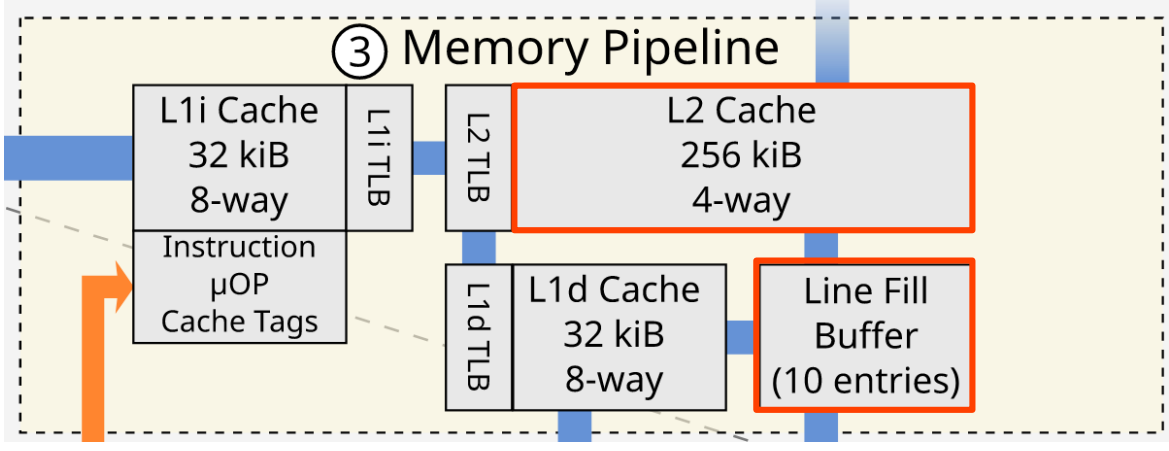
Intel CPUのキャッシュ機構

- LFBの検索の後か同時平行でL1dキャッシュも見る
  - 当然ヒットすればそこからデータを取得するが, LFBエントリが存在する場合はその限りではない



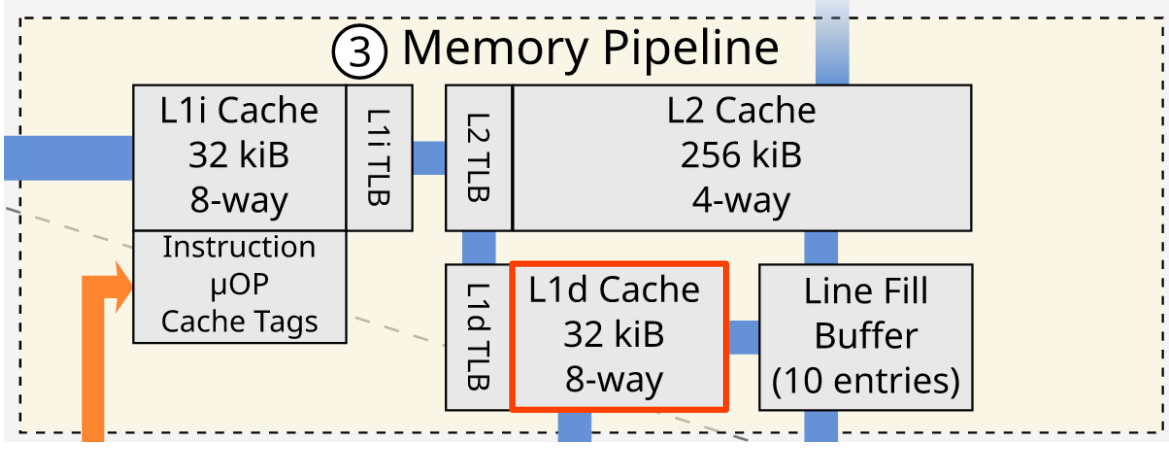
Intel CPUのキャッシュ機構

- これでもデータが見つからない場合はL2以降を見るが，LFBに書き込んでおけば適当に処理してくれる
- 先程のLFBのロード操作そのまま



Intel CPUのキャッシュ機構

- LFBによりL1dにアドレスAのデータがロードされたので, これを読めばよい



Transient Execution

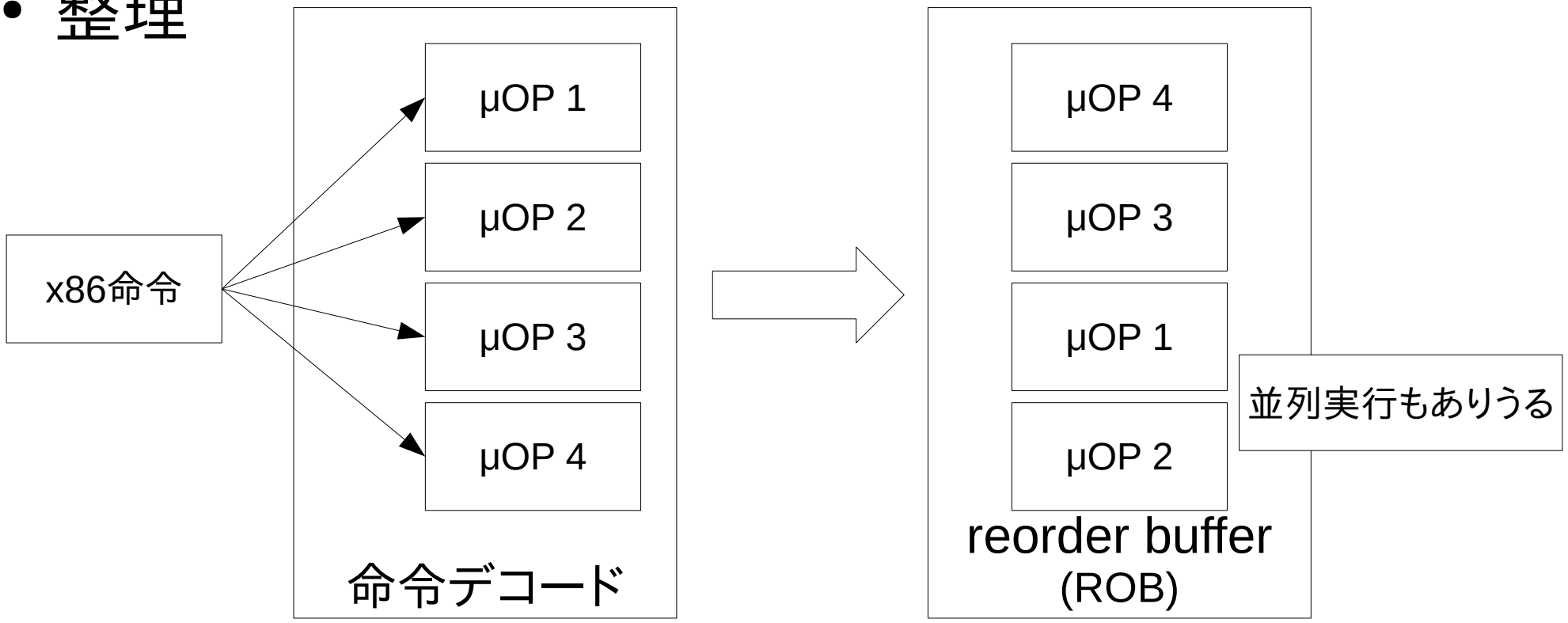
- Intel CPUは命令を細かく分解して実行する
  - 以降分解前をx86命令, 分解後をμOPと区別
  - 基本4つまでに分解するが, それ以上の場合もある
- μOPは適宜依存関係を壊さない範囲で高速実行できるように並び替えられる(OoO: Out-of-Order execution)
  - 先々を読んで効率的になるなら投機的に実行してみる
  - 並び替え･実行の状況はReOrder Buffer(ROB)で管理

Transient Execution

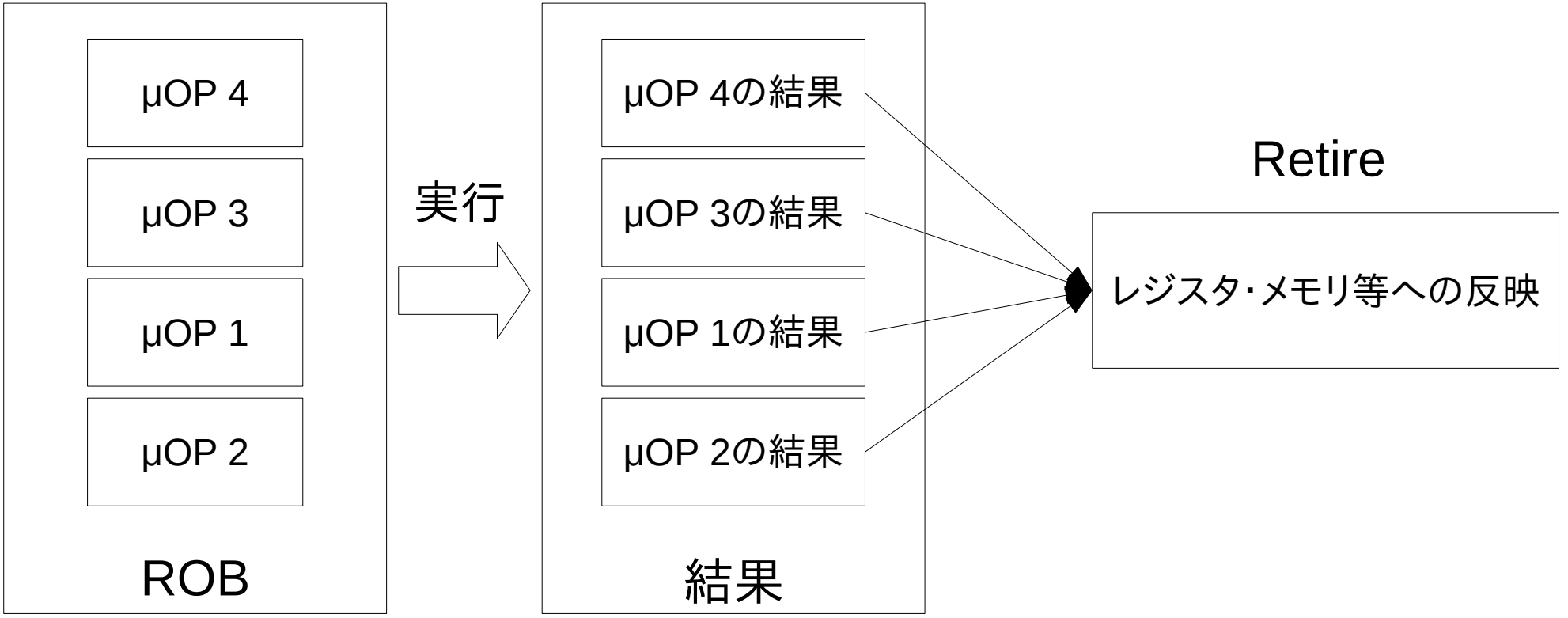
- x86命令1個分のμOPsが実行終了するごとにレジスタ･メモリ等への変更を実際に適用する
  - この適用処理のことをretireという

Transient Execution

- 整理



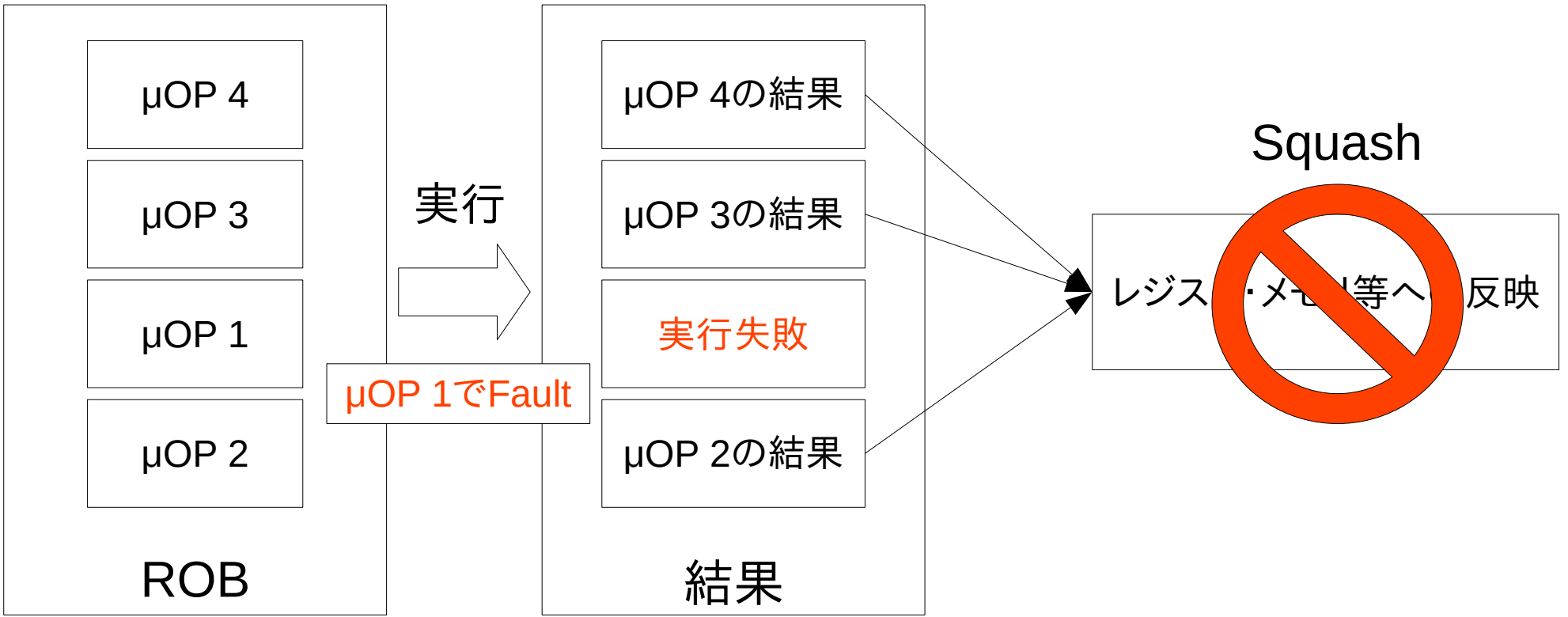
Transient Execution



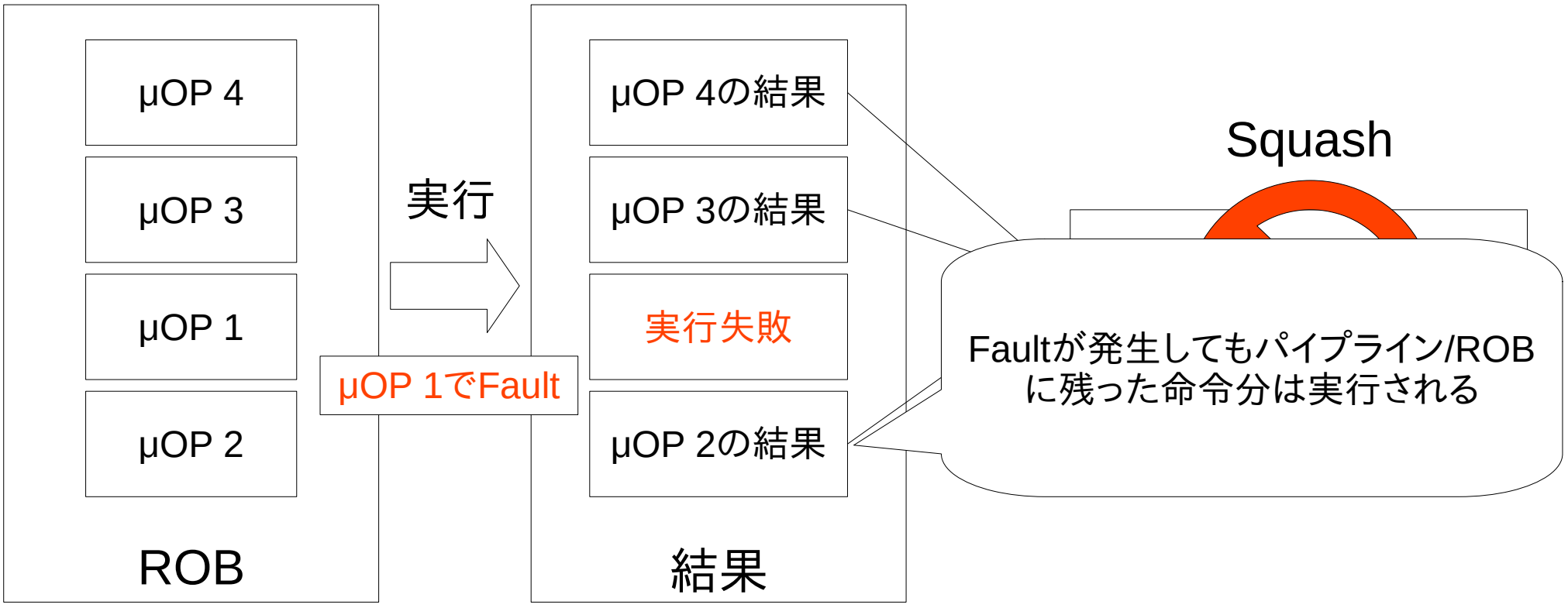
Transient Execution

- μOPsが途中でfaultにより失敗した場合を考える
  - x86命令としては実行失敗の状態
- 失敗したμOPを含め, μOPsが与えるはずだった変更は全てロールバックされる
  - このロールバックをsquashと呼ぶ
- Squash時は変更をロールバックするが, キャッシュの値やTLBの値はロールバックしない(できない)
  - これが副作用となり, Meltdownが起きる

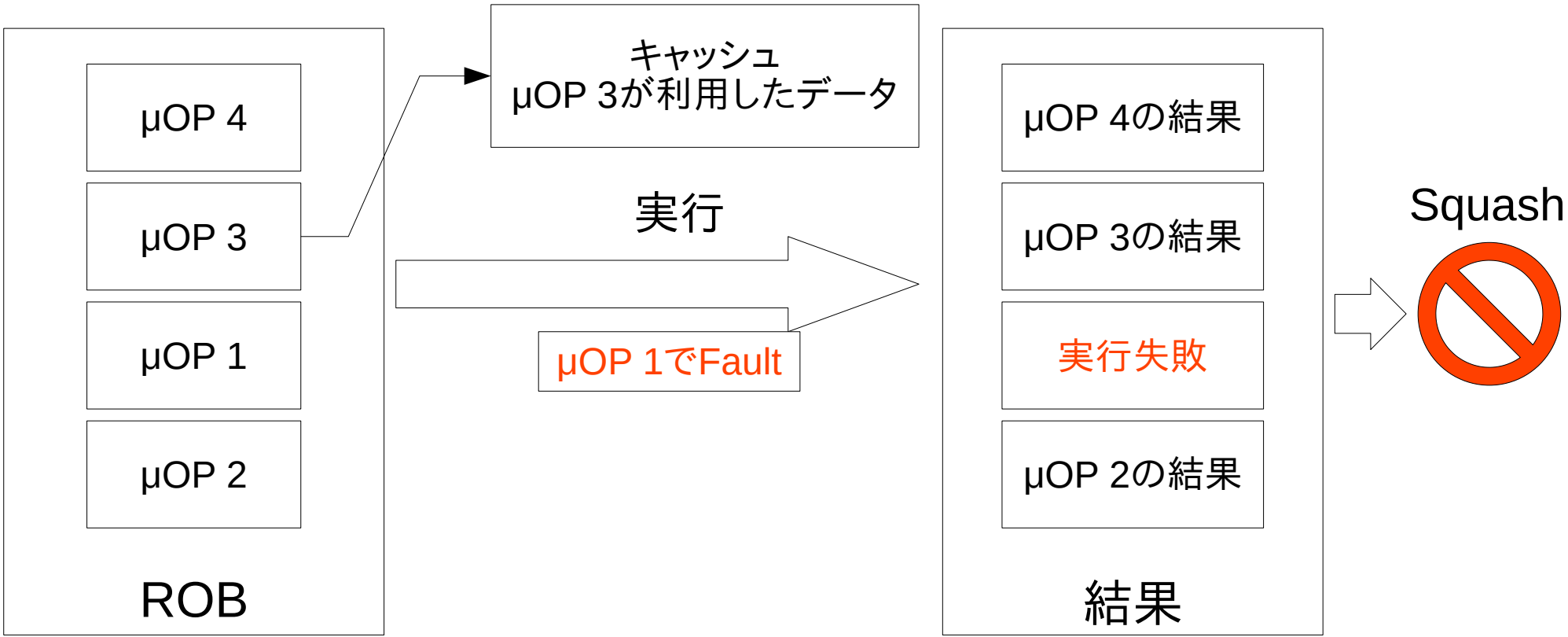
Transient Execution



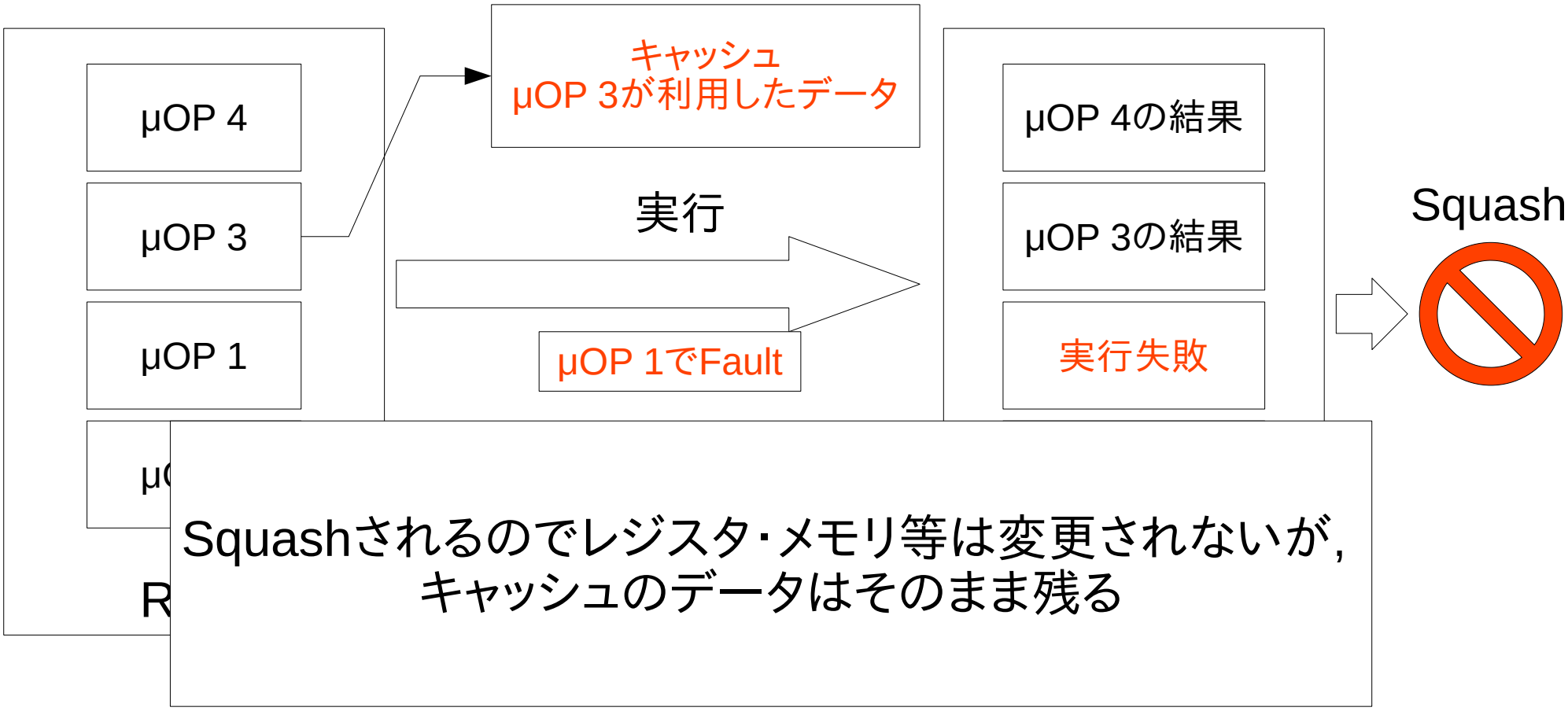
Transient Execution



Transient Execution



Transient Execution



Transient Execution

- x86命令としては失敗しても, 途中のμOPsはfaultしたもの以外は正しく動いている
  - これら(squashされる正常動作したμOP)をTransient instruction(s)という
- Transient instructionを利用する攻撃手法のことをTransient execution attackといい, そのような実行そのものをここではTransient executionという
  - 詳しくは[14]を参照

Meltdown

- Meltdownの大まかな流れ
  - array: 大きい配列
  - CACHE_SIZE: キャッシュラインのサイズ
  - target_addr: mappedかつ読めないアドレス

Meltdown

- x = array[*target_addr * CACHE_SIZE]のようなメモリ読み出しをするとき, target_addrのアクセス権チェックとロードが同時に走る(OoOによる)
- target_addrにはアクセス不可能なのでfaultが発生, ロード命令としてはsquashする
  - Faultは適当に無視する
- しかし, arrayの*target_addr * CACHE_SIZE番目の値は副作用としてキャッシュに乗る

Meltdown

- キャッシュに乗っているかどうかはアクセス速度で判定可能
  - 事前に配列arrayの全要素をキャッシュから追い出しておく
  - *target_addr * CACHE_SIZEが添字なので, 配列arrayに順にアクセスして高速にアクセスできた添字がキャッシュに乗っている
- Flush&Reloadと呼ばれる手法

Meltdown

- Linuxは適当なプロセスからカーネル空間のメモリが(アクセス権限は存在しないが)見えるようになっていた
  - システムコールの高速化のため
  - しかし, ページングのアクセス権限チェックがMeltdownで役立たなくなってしまったことから, カーネル空間の任意のメモリが任意のプロセスから読めることに!
- ユーザ空間のプロセスの仮想メモリにカーネル空間のメモリをマップしないことで対処(KPTI: Kernel Page Table Isolation)

Intel TSX

- TSX ⇔ Transactional Synchronization eXtensions
- もともとはトランザクショナルメモリを実現するための拡張だが, ここではエラー発生時に巻き戻される性質のみを利用する
  - 使うのはxbegin, xend命令2つだけ

Intel TSX

- xbegin命令からxend命令までの間に挟まれた処理を1つのトランザクションとみなし, メモリ操作に対しatomicにこれを実行する
  - トランザクション中にエラーが起きた場合は全メモリ操作をロールバック
  - トランザクションの外にfaultが伝播することがない
  - ある種の投機的実行

Intel TSX

- Meltdownではエラーハンドリングが面倒なのがネック
  - 通常longjmpとsignalを使う
  - TSXを利用することでとても単純なコードに

Intel TSX

- TSXを利用したMeltdown PoC

```
flush(array);

if (xbegin() == SUCCESS) {
  unused = array[*target_addr * CACHE_SIZE];
  xend();
}

datum = reload(array);
```

MDS Attacks

- Microarchitectural Data Sampling Attacks
  - 名前の通り, メモリやディスク, ポートへのアクセスではなくCPU内部の実行ユニットからデータを取得(sampling)する手法群
- 今回はRIDL Attackを取り上げる
  - CROSSTalk AttackはRIDLの応用
  - IntelはRIDLのことをMFBDS(Microarchitectural Fill Buffer Data Sampling)と名付けている [15]

RIDL

- RIDL ⇔ Rogue In-flight Data Load
  - LFBを利用してIn-flightな(今まさに動いているデータを)取得する攻撃
- Meltdownとは異なり, 特定のアドレスを読むことはできない
  - バッファ領域のデータを読むことになるため, 同じ物理コア上で実行されている任意のプロセスの入出力データが読める
  - Meltdownでは仮想メモリから読めるならなんでも読めるが, KPTIのように見えなくされると読めない

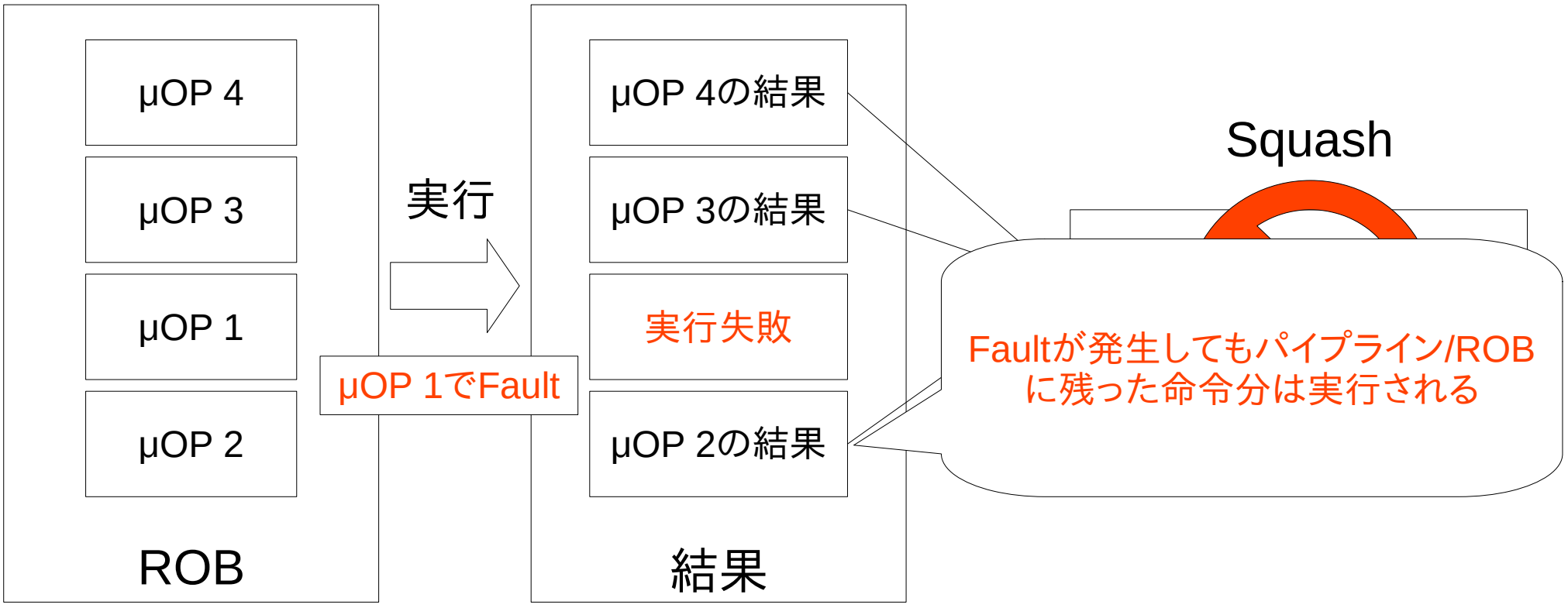
RIDL

- 注: 以降の記述はあんまり裏を取っていません
  - 大元の論文[5]にも載っておらず, 直接Intelの出したpatentを読むしかない箇所が多い
  - 今回は[6]で有志が読み解いたものといくつか別途発見できたpatent, Intel SDMを参考に記述している

RIDL

- μOPが途中でfaultしてもパイプライン/ROBに残っているならば実行されるという話をした

RIDL


μOP 4
μOP 3
μOP 1
μOP 2
ROB
実行
μOP 1でFault
μOP 4の結果
μOP 3の結果
実行失敗
μOP 2の結果
結果
Squash
Faultが発生してもパイプライン/ROB
に残った命令分は実行される

RIDL

- μOPが途中でfaultしてもパイプライン/ROBに残っているならば実行されるという話をした
  - いわば「このx86命令はfaultします」というフラグを立てただけにすぎない
    - 以降fault flagと呼ぶ
  - 本当にそれだけしか影響がないのか

RIDL

- 実は, fault flagが立っているとき, μOPのアドレス比較・サイズ比較のロジックが全て停止する
  - [6]では消費電力削減を疑っている (真相は不明)
- しかも, fault flagが立っているからといって(faultした命令の)実行自体が必ずしも停止するわけではない
  - 具体的に説明する

RIDL

- 0x0(null pointer)をロードすることを考える
  - 通常であればunmapped page
  - 当たり前ながらTLBには存在しない
- MMU(Memory Management Unit)はページテーブルを読むためにロード処理を中断する
  - しかしページテーブルにも存在しないのでエラー
  - MMUはROBに対しfaultを宣言する

RIDL

- ROBはfault宣言を受け取ってfault flagを立てる
- 一方, MMUによりブロックされていたロード命令はここでブロック解除されて動きだす
  - 物理アドレスとして何かが与えられるはずだが, 物理アドレスの取得に失敗しているので適当な信号が与えられるのみ
- ここで通常通りLFBを見に行く

RIDL

- fault flagが宣言されているので, アドレス比較ロジックが全て成功する
  - よってLFBに存在する全ての確保済エントリのアドレスと一致する判定になる
  - 最初にヒットしたものを(0x0に対応する)LFBエントリと誤認し, これをL1dキャッシュにコピーしてロード処理終了
- μOP / x86命令としてはfaultするので,「キャッシュに何かたまたまヒットしたLFBエントリの値がコピーされた状態」で停止する
- 後はMeltdownと同様にキャッシュを読めばよい

RIDL

- 適当なLFBのデータをキャッシュに持ってくることができた
  - 0x0である必要性はなく, ページテーブルに存在しないならば0xdeadbeefでも0xc0ffeeでもなんでもいい
- LFBエントリは10個しか存在しないため, 何度か試せば望みのエントリを読むことも可能と考えられる
  - しかしロード･ストアが頻繁に起きる場合には厳しい

RIDL

- これを踏まえたRIDLの疑似コードは以下の通り

```
flush(array);

if (xbegin() == SUCCESS) {
  unused = array[*((unsigned char*) NULL) * CACHE_SIZE];
  xend();
}

datum = reload(array);
```

- ほぼMeltdownに同じ

RIDL

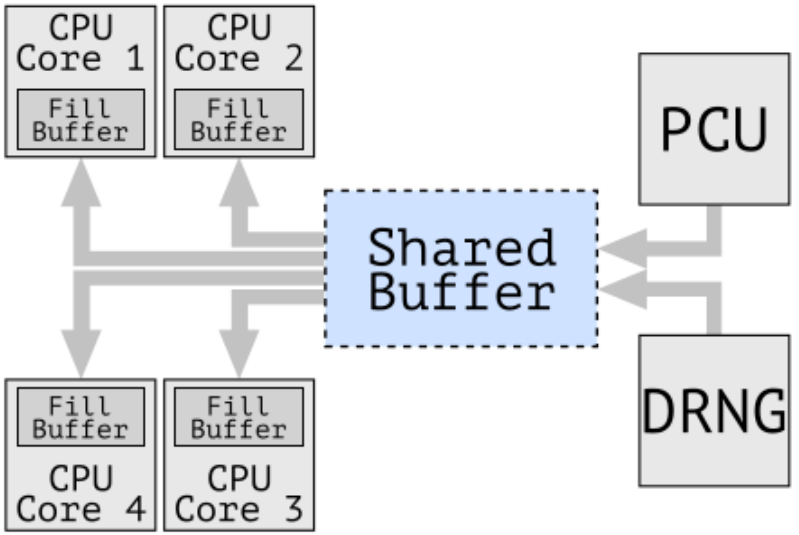
- RIDLのメリット
  - メモリアドレスに左右されず, 全てのロード・ストアデータを読める
    - KPTIを回避できる
  - 同じ物理コア上にあれば, VM guestから他のVM guestのデータを読むことすら可能

RIDL

- RIDLのデメリット
  - 同じ物理コア上という制約
    - 実質Hyper Threadingが前提
  - 狙ったLFBに毎回当たるとは限らない
    - そもそも上書きされる可能性のほうが高い
  - メモリアドレス指定ができないので, 相手方の通信に極めて強く依存する

CROSSTalk

- CROSSTalkは特定の命令に対しcross-coreにその出力を観測可能な手法
  - RIDLが理解できていれば, 以下の図で十分かも?



画像: [1] Fig. 2

CROSSTalk

- 最近のIntel CPU(Ivy bridge以降)は乱数生成命令を備えている
  - rdseed命令: 疑似乱数生成器(PRNG)の出力を得る
  - rdrand命令: 暗号論的安全なPRNGの出力を得る
    - 他のPRNGのシードに使うことが想定されるのがrdseed [17]
- これらはDRNG(Digital Random Number Generator)と呼ばれるCPUコアとは独立した機能ブロックからデータを取得する
  - 実は全コアで共通のバッファ(staging buffer)が存在し, 一旦そこを経由してLFBへとコピーされる

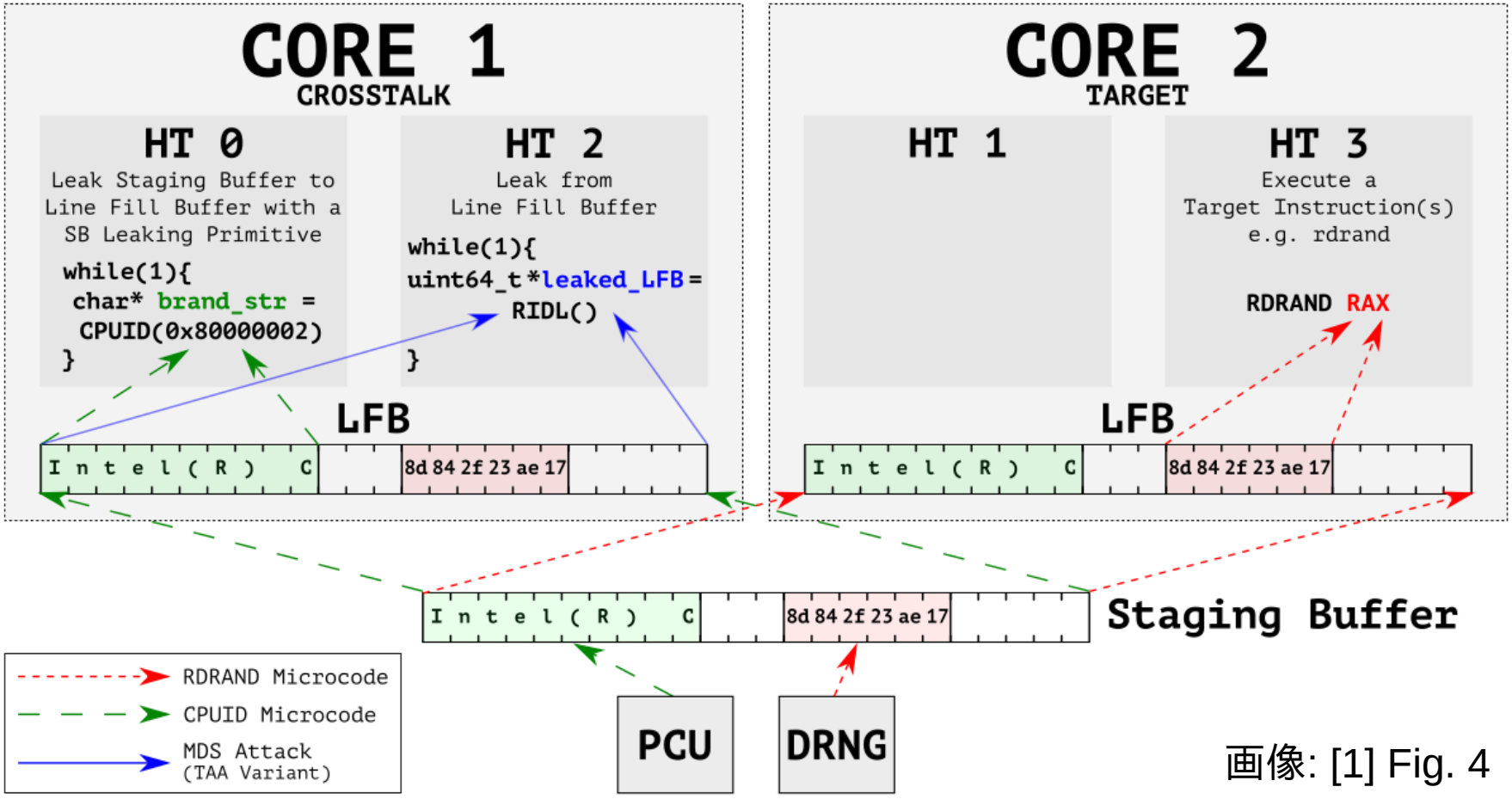
CROSSTalk

- 一方, CPU関連の情報取得や電源管理に用いられるcpuid命令, rdmsr命令もコアの外にあるPCU(Power Controller Unit)という機能ブロックと通信する
  - こちらもstaging bufferを経由してLFBにコピーされる
- Staging bufferの存在自体はエスパーだが, どうやら特許にmailbox interfaceなる手法が含まれており, そこから検証した結果発見したようである[1, VIII Discussion]

CROSSTalk

- Staging bufferは64byteしかないが, 利用法としてはmailboxである
  - Staging bufferのうちcpuidはここからここまで, rdrandはここからここまで, ...のようにある程度使う場所が固定になっている
  - cpuidのbrand string関連が64byte全てを使うので, これの後に各命令を実行して差分を見れば調べられる [1, Fig. 5]
- Staging bufferの中身は毎回LFBにコピーされるので...?

CROSSTalk



画像: [1] Fig. 4

CROSSTalk

- これによりrdrand, rdseed命令の出力を読み取ることができた
  - Intel側の調査[2]では, 更にIntel SGX enclave内のegetkey命令も読めることが判明
- Hyper threading OFFの環境でも動作
- 修正後はrdrand/rdseed等の命令を実行した後staging bufferの当該領域をクリアするようになった
  - μOP数にして450倍以上の性能劣化 (rdrand: 16 → 7565, rdseed: 16 → 7564)

References

- 注: 引用番号は順不同かつスライド中触れていないものも存在する
  - 必要に応じて読んでほしい
- Intelのpatent関連は対応付けが膨大すぎて覚えきれず && 書ききれず, SDM/Optimization Manualは言わずもがなということで省略

References

[1] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ***CROSSTALK: Speculative Data Leaks Across Cores Are Real***

https://download.vusec.net/papers/crosstalk_sp21.pdf

[2] ***Deep Dive: Special Register Buffer Data Sampling***

https://software.intel.com/security-software-guidance/insights/deep-dive-special-register-buffer-data-sampling

[3] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. ***Meltdown: Reading Kernel Memory from User Space***

https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf

References

[4] ***MDS: Microarchitectural Data Sampling***

https://mdsattacks.com/

[5] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze,Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. ***RIDL: Rogue In-flight Data Load***

https://mdsattacks.com/files/ridl.pdf

[6] ***About the RIDL vulnerabilities and the "replaying" of loads***

https://stackoverflow.com/questions/56187269/about-the-ridl-vulnerabilities-and-the-replaying-of-loads

References

[7] ***Spectre/Meltdownとその派生***

https://www.slideshare.net/herumi/spectremeltdown

[8] ***The Microarchitecture Behind Meltdown***

http://blog.stuffedcow.net/2018/05/meltdown-microarchitecture/

[9] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ***ZombieLoad: Cross-Privilege-Boundary Data Sampling***

https://zombieloadattack.com/zombieload.pdf

References

[10] ***本当にわかるSpectreとMeltdown***

https://www.slideshare.net/hktechno/spectre-meltdown-110262623

[11] ***図解でわかるSpectreとMeltdown***

https://speakerdeck.com/sat/tu-jie-dewakaruspectretomeltdown

[12] Victor Costan and Srinivas Devadas. 2016. ***Intel SGX Explained***

https://eprint.iacr.org/2016/086.pdf

References

[13] Stephan van Schaik and Alyssa Milburn. 2019. ***RIDLed with CPU bugs***

https://hardwear.io/netherlands-2019/presentation/RIDLed-with-CPU-bugs-hardwear-io-nl-2019.pdf

[14] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin and Daniel Gruss. 2019. ***A Systematic Evaluation of Transient Execution Attacks and Defenses***

https://arxiv.org/abs/1811.05441

[15] ***Deep Dive: Intel Analysis of Microarchitectural Data Sampling***

https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling

References

[16] ***MDS Attacks: Microarchitectural Data Sampling***

https://mdsattacks.com/

[17] ***The Difference Between RDRAND and RDSEED***

https://software.intel.com/content/www/us/en/develop/blogs/the-difference-between-rdrand-and-rdseed.html