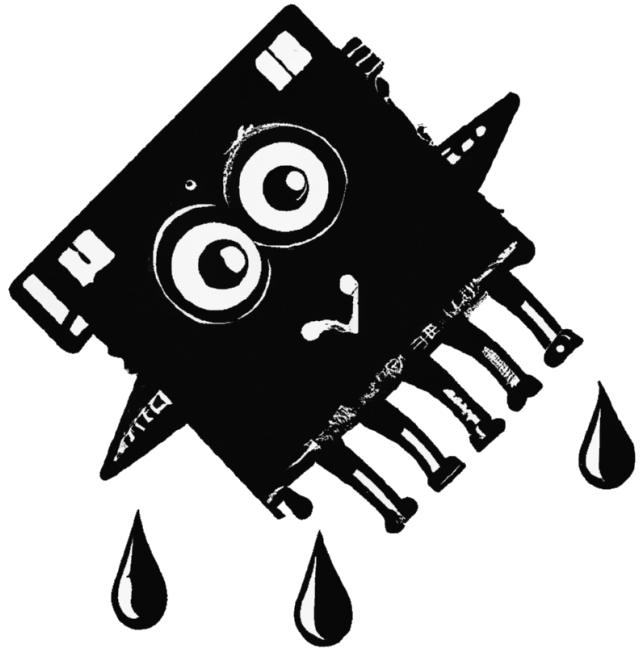
SGX脆弱性調査 - Downfall

Ao Sakurai

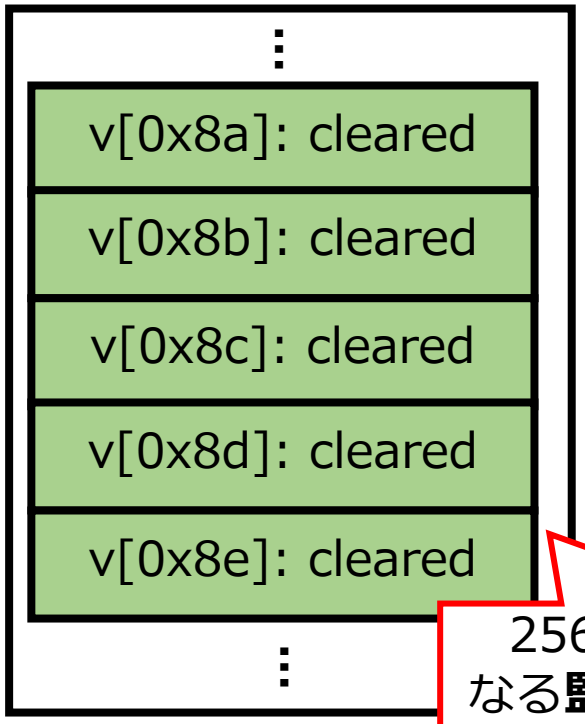
本資料の目標

- ベクトルレジスタからの過渡的漏洩を悪用するMeltdown型攻撃である、Gather Data Sampling（GDS）等を可能にするDownfall脆弱性について説明を行う。



FLUSH+RELOAD攻撃

- **FLUSH+RELOAD**：キャッシュラインを**フラッシュ**（クリア）し、攻撃対象に何らかの活動させた後、**再度アクセス**する攻撃[14]
  - **再アクセス時**に**アクセス時間が短ければ**、**そのキャッシュ値**を**攻撃対象が使用**したという事がわかる（データ自体の推測）



キャッシュ

256要素（=1バイト）からなる**監視用配列**の各要素に対応する**キャッシュをフラッシュ**

秘密情報0x8cを**インデックス**として**監視用配列にアクセス**

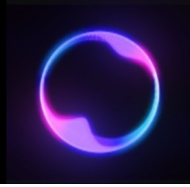
```
ret = func();
```

攻撃対象コード

監視用配列インデックス	アクセス時間[ナノ秒]
0x8a	1.52
0x8b	1.76
0x8c	**0.80**
0x8d	1.47
0x8e	1.63

監視用配列の全スロットに**アクセス**し、**アクセスが速かったもの**はキャッシュにあり**funcに使用された**（=**秘密バイトが0x8c**）と分かる

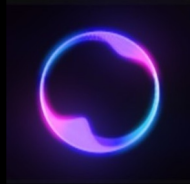
コンテキストスイッチ



- **コンテキストスイッチ**：CPUが**処理する対象**を**変更**する動作
  - プロセス間の切り替え、SGXのEnclave内外での切り替え、ユーザモードからOSのカーネルモードへの切り替え

- コンテキストスイッチが発生した場合、**仮想アドレス空間**と**CPUレジスタの状態**（コンテキスト）の**切り替え**が発生する
  - よって、例えば切替後のプロセスが切替前のプロセスのメモリやレジスタにアクセスする事はできない

SIMDとベクトルレジスタ（1/3）

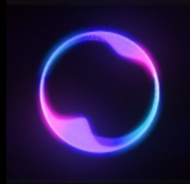


- **SIMD**：Single Instruction Multiple Dataの略。**同じ操作**を**異なるデータ**で**並列に実行**するような処理
  - 例：8個のデータを、単一のSIMD命令で並列で一気にビット反転する

- SIMDには、現在主流であるアーキテクチャのビット数である64bitよりも大きい、専用に用意されている**ベクトルレジスタ**（**ワイドレジスタ**）を用いる

- Intel SSE対応のCPUであれば128bit、AVXやAVX2対応であれば**256bit**、AVX512対応であれば**512bit**のベクトルレジスタが用意されている

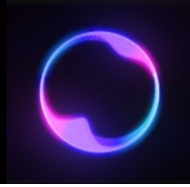
SIMDとベクトルレジスタ（2/3）



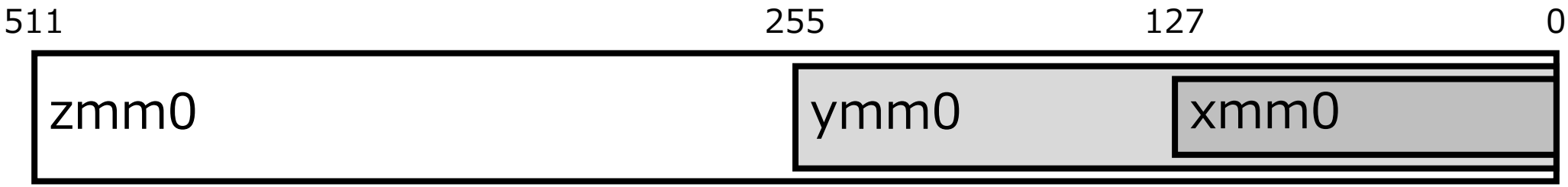
- 64ビットレジスタにおけるeaxのように、ベクトルレジスタにもレジスタ名が付与されている

- 128bitレジスタはxmm$n$、256bitレジスタはymm$n$、512bitレジスタはzmm$n$という命名規則になっている
  - 各末尾のnはレジスタのインデックス番号（例：xmm0）

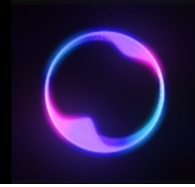
SIMDとベクトルレジスタ（3/3）



- 同じインデックス番号の異なるサイズのベクトルレジスタがある場合、より**大きいレジスタ**はより**小さいレジスタ**を**内包する**構成となっている
  - 例：zmm0はymm0とxmm0を含み、ymm0はxmm0を含む
  - 通常のレジスタにおけるraxとeaxのような関係と同様



Gather命令（1/4）

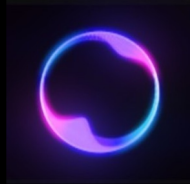


- 主に**メモリの各所に分散して存在**している非連続なデータを**効率的に収集しロード**する命令
  - **%rsi**：ベースアドレス
  - **%xmm2**：収集してロードするデータの位置を指定する、ベースアドレスに対するインデックスを格納するインデックス配列
  - **%xmm1**：マスクレジスタ。ここで0であるようなビット位置のデータは、インデックス配列に格納されていても収集を行わず無視する

    （例：マスクレジスタの2bit目が0なら、インデックス配列の2要素目に対応するデータの収集は行わない）
  - **%xmm3**：収集結果を格納するベクトルレジスタ

```
vpgatherdd = %xmm1, 0(%rsi, %xmm2, 2), %xmm3
```

Gather命令（2/4）

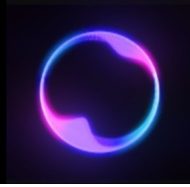


- 前ページの例では、32bitの値（**dword**）4つを収集して**128bitの**<u>**ベクトルレジスタ**</u>である**xmm3レジスタに格納**する

- 収集するデータの位置は、**(%rsi + %xmm2[i] * 2)**という形で決定される
  - 収集対象が4個であるため、$0 \leq i < 4$である

- また、インデックス配列の内特定要素に対応する場所のデータは収集不要である等の場合は、対応する**マスク配列**の**要素**を**0**にする事で**収集対象から除外**する事ができる

Gather命令（3/4）



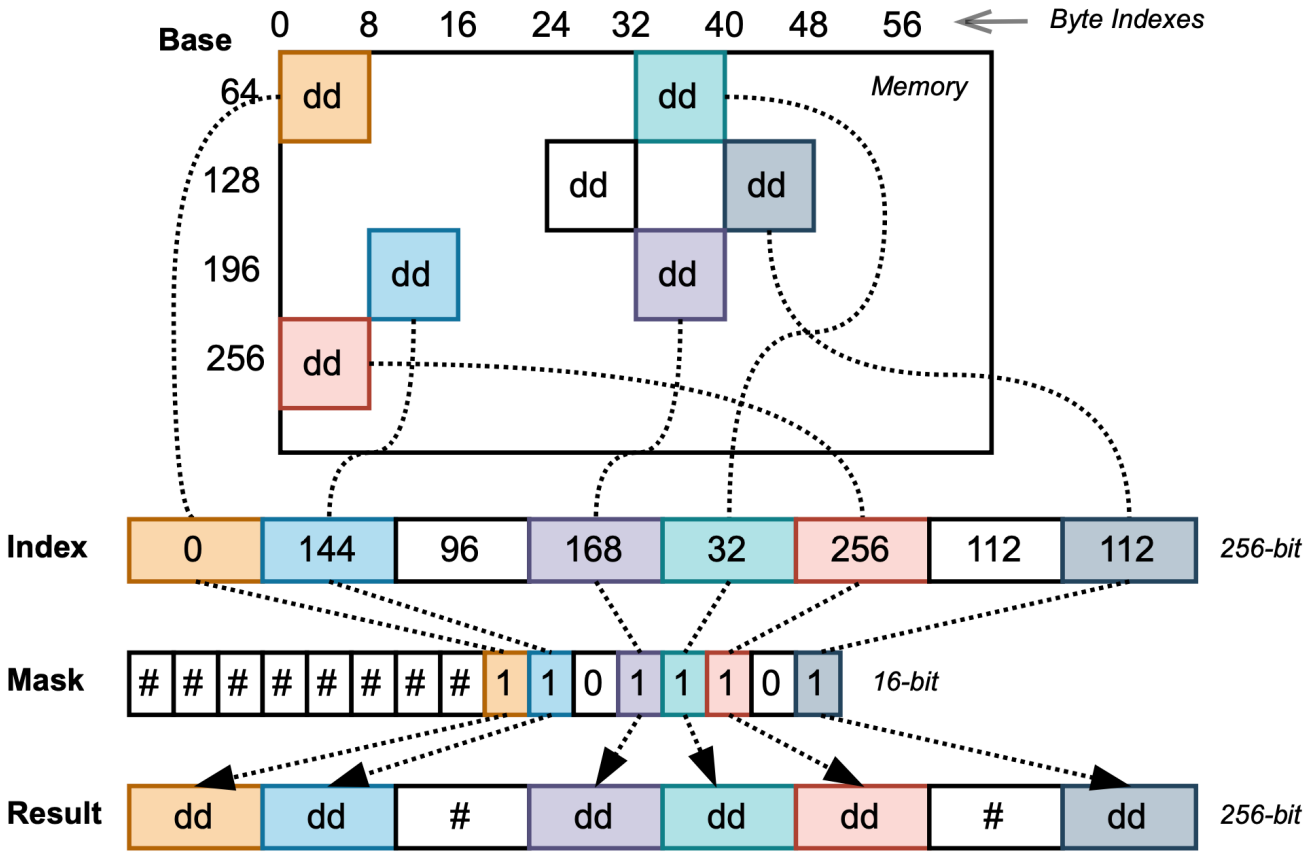
- このGather命令でメモリの各所に散らばったデータを効率的に**収集**して**ベクトルレジスタにロード**した上で、**他のSIMD命令**を実行すると**効率的にSIMD処理を行う事ができる**

- AVX-512の場合、マスク配列専用の**マスクレジスタ**knが用意されており、以下のようにGather命令を記述する事ができる
  - rsiがベースアドレス、zmm2がインデックス配列、k1がマスクレジスタ、zmm3が格納先ワイドレジスタ

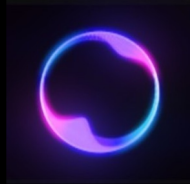
```
vpgatherdd 0(%rsi, %zmm2, 1), %zmm3{%k1} // AVX-512
```

Gather命令（4/4）

- Gather命令による動作の様子を図示すると以下のようになる
  （図は[1]より引用）



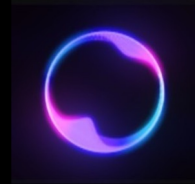
マイクロアーキテクチャ（μ-Arch）



- **μ-Arch：命令セットアーキテクチャ**よりも**ローレベル**な、**CPUの内部構造やデータフロー**を**定義**する**設計レベル**の事
  - 有名所としては**キャッシュメモリ**もμ-Archに含まれる
  - その他、**直近の分岐履歴**等を記録する**LBR**（Last Branch Record）や、アウトオブオーダー実行等で未処理のストア命令を記録しておく**ストアバッファ**等が存在する

- μ-Archに対する攻撃は、**過渡的実行攻撃**（Transient Execution Attacks）の**アウトブレイクが発生**した**2018年**以降**急激に増えている**
  - この分類では、**過渡的実行に依存しない類の攻撃の例**のみを挙げ、**過渡的実行攻撃**に関しては**別分類**としている

Gatherのμ-Archによる最適化



- CPUは、以下のような**マイクロアーキテクチャによる最適化**により Gather命令の実行を**高速化**している：

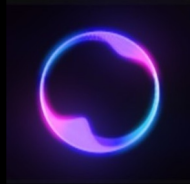
➢**0であるマスクビット**に対応するメモリ位置からは、そもそもロード処理自体行わずに**処理から排除**する。

➢**同一キャッシュライン**から複数の値を収集する場合、**そのキャッシュラインを保持**しておく。

➢複数の読み出しを**並列かつ投機的に実行**し、少なくとも1つの読み出しに失敗したら**結果を破棄**する。

➢Gather処理中に割り込みが入った場合に途中から再開できるよう、既に実行された**部分的な読み取り結果を保持**しておく。

Gather命令の最適化から見えるデジャヴ



- 複数回読み出し時の**キャッシュライン保持**や、割り込み発生時の再開のための**部分結果保持**には、**CPUパッケージ内の何らかのバッファを使用**しているのでは？
  - Foreshadow（L1D）、ZombieLoad（LFB）、Fallout（SB）、LVI（L1D、LFB、SB、LP、FPU）のような漏洩が発生するのでは？

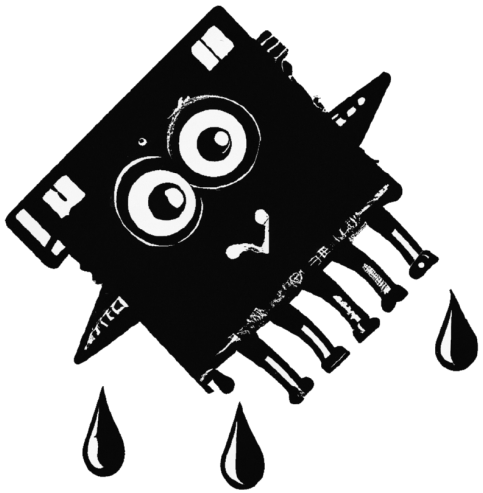
- **投機的に実行**して駄目ならアーキテクチャ状態（CPUやメモリの実際の状態）への反映時（**命令リタイア時**）に**破棄**、という挙動は、**過渡的領域**において**何かしらの脆弱性**を抱えているのでは？
  - 過渡的実行中にキャッシュ等に秘密情報に依存する値の痕跡を残す攻撃はもはや恒例である

Downfall（1/2）

- お察しの通り、**Gather命令**に伴い**ベクトルレジスタ内の古い値**が**過渡的に漏洩**する「**Gather Data Sampling（GDS）**」が発見された

- さらに、ForeshadowやMDSからのLVIへの接続の類推から、**GDSによる漏洩値**を**後続の過渡的命令**への**注入に転用**する「**Gather Value Injection（GVI）**」の実現にも成功している

- これらのGDSやGVIを悪用した攻撃を**Downfall**と呼んでいる[1]



Downfall（2/2）

- Gatherに伴い漏洩するデータの漏洩元は、論文では「一時バッファ」「内部バッファ」「SIMDレジスタバッファ」と表現しており、いまいち**実体が釈然としない**

- Intel公式による解説によると、前述の通り**ベクトルレジスタ**が**漏洩元**であると言及されている

- **論文執筆中には実体が知れなかった**が、エンバーゴ（情報開示禁止期間）中に**Intelが突き止めて判明した**可能性などが憶測できる
  - ちなみに、Downfallのメインページではベクトルレジスタであると明示的に言及している

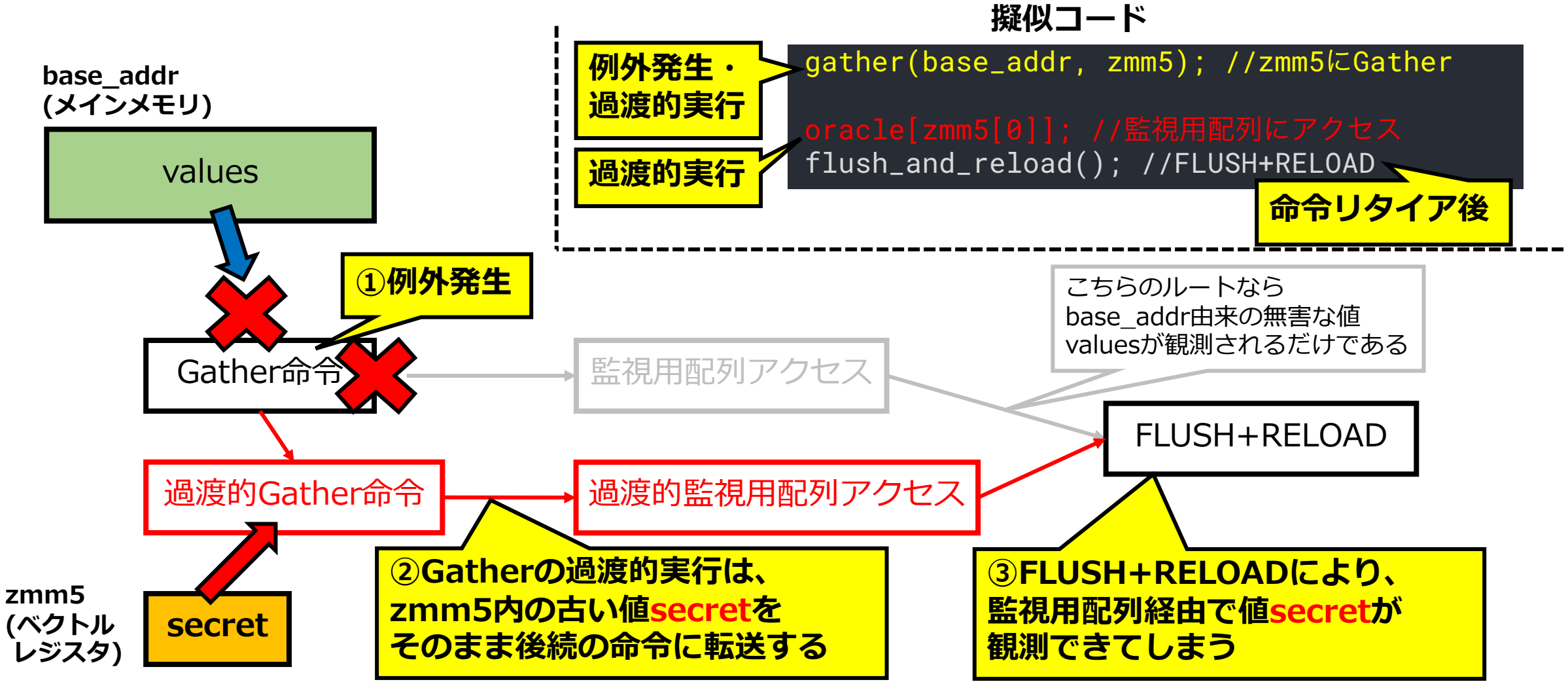
GDSのPoC実装（1/7）

- 以下のコードは、GDSを実行するためのPoCコードである

```
// Step (i): Increase the transient window
lea addresses_normal, %rdi
clflush (%rdi)
mov (%rdi), %rax

// Step (ii): Gather uncacheable memory
lea addresses_uncacheable, %rsi
mov $0b1, %rdi
kmovq %rdi, %k1
vpxord %zmm1, %zmm1, %zmm1
vpgatherdd 0(%rsi, %zmm1, 1), %zmm5{%k1}

// Step (iii): Encode (transient) data to cache
movq %xmm5, %rax
encode_eax

// Step (iv): Scan the cache
scan_flush_reload
```

GDSのPoC実装（2/7）

- GDSの概要を示した図は以下の通り：



GDSのPoC実装（3/7）

■ステップ(i)

キャッシュクリアを行う事で**キャッシュミスを誘発**する。
キャッシュミスはCPU的には遅延以外の何物でもないため、
投機的実行（過渡的実行）の**ウィンドウ（実行時間）**を**増幅**させ、
過渡的に転送された値をキャッシュに残す**時間的猶予が増える。**

■ステップ(ii)

まず、このステップの最初の行で対象メモリアドレスを**キャッシュ不可**（Uncacheable）としている。かつ、ステップ(i)で**キャッシュをクリア**しているため、**Gather命令**において**キャッシュミス**が発生する。
これにより、動作は中断されないが、裏で**過渡的実行が発動**する
（キャッシュミスは遅いため、CPU的には投機的に解決したい）

GDSのPoC実装（4/7）

■ステップ(iii)

Gatherの過渡的実行により、（恐らくzmm5）**ベクトルレジスタ**内に**残留**している**古いdword**（4バイト値）が**後続の命令に過渡的に転送**される（ここでは単一バイトのみ漏洩するものとする）。

過渡的に漏洩したdwordの**各バイトをインデックス**として、4×256の**監視用配列**に**過渡的にアクセス**し痕跡を残す。

例：dword値が0x8c34c592である場合、監視用配列をA[4][256]とすると

A[3][0x8c], A[2][0x34], A[1][0xc5], A[0][0x92]

のように過渡的にアクセスする。

GDSのPoC実装（5/7）

■ステップ(iv)

過渡的実行の終了、つまり命令リタイア後、**FLUSH+RELOAD攻撃**により過渡的に漏洩した値を監視用配列経由で観測する。

前述の例の場合、ステップ(i)で**キャッシュクリア済み**（**FLUSH**）であり、かつA[3][0x8c], A[2][0x34], A[1][0xc5], A[0][0x92]にのみ**過渡的にアクセス**し**キャッシュが残っている**ため、これらの要素への**アクセス時間**だけ**他に比べて高速**である。

よって、アクセスが高速であったような要素の**インデックス経由**で、**漏洩したdword値を復元**する事ができる（**RELOAD**）。

GDSのPoC実装（6/7）

- もしベクトルレジスタ内に残留していた**古いdword**が、本来**攻撃者のアクセスできない秘密情報**であった場合、この時点で**秘密情報の漏洩が発生した**事になる

- ForeshadowやZombieLoad同様、キャッシュラインプリフェッチャの誤動作により監視用配列のキャッシュが汚染される事を防ぐため、監視用配列の各バイト監視用の要素（スロット）は最小で128B、最大で4096B（1物理ページ分）間隔を空ける必要がある

GDSのPoC実装（7/7）

- このPoC実装のGDSをTiger Lake CPU上で実行した所、**並行するハイパースレッド**（**シブリングスレッド**）から**1秒間**に**809個のdword値を漏洩**させる事ができた

- また、前述の**ステップ(i)～(iii)を繰り返す**事で、dwordが完全な形でキャッシュに残る**確度を高める**事ができる

- 実際に(i)～(iii)を32回実行してから(iv)を実行した所、**1秒間に903個のdwordを漏洩**させられた

GDSのトリガー方法（1/4）

- GDSは、前述の通り**キャッシュ不可メモリ**へのアクセスや、あるいは**Write Combiningメモリ**へのアクセスでも発生する
  - **Write Combining**：書き込みを後でまとめて行うようなメモリモード。このメモリモードである場合もキャッシュが迂回される

- また、Gatherに伴う**あらゆるフォールト**によってもGDSが**誘発される**事が確認できている
  - カーネルやメモリ保護キーへの無効なアクセスに伴う**パーミッションフォールト**
  - マッピングされていないメモリアクセスによる**ページフォールト**
  - 非正規アドレスへのアクセスに伴う**アドレス生成フォールト**

GDSのトリガー方法（2/4）

- また、PTEの**Accessedビットが0**であるページに**アクセス**した際にも、GDSによってかなり**転送レートの低めな漏洩が確認**された
  - ZombieLoadからの類推からすると、このようなアクセスに伴う**マイクロコードアシスト**が原因そうだが、トリガーとして確定はできていない

- transient.failにて整理されている**過渡的実行攻撃の分類**で言うと、Meltdown-US、Meltdown-MPK、Meltdown-NC、Meltdown-P、Meltdown-UC、Meltdown-AをGDSは悪用している
  - 順にMeltdown本家（カーネルアクセス違反）、メモリ保護キー違反、非正規アドレス違反、ページフォールト、キャッシュ不可、アラインメントされていないメモリオペランドを悪用するものである

- Downfall発見時点でIntelによりTSXが無効化されているため、Meltdown-TAAは当てはまらない

GDSのトリガー方法（3/4）

- 以下のコード例のように、**フォールトやアシストを誘発**するような**異常な（Exotic）アドレス**に一切アクセスせずに、**過渡的実行を誘発**して**GDSを発動**させる方法も存在する

```
// Step 1: Increase the transient window a lot
lea addresses_normal_helper, %rdi
.set i,0
.rept 8
clflush 64*i(%rdi)
mov 64*i(%rdi), %rax
.set i,i+1
.endr
xchg %rax, 0(%rdi)

// Step 2: Gather cachable memory (no fault)
lea addresses_normal, %rsi
...
```

※論文より引用している。恐らくこの後ろにGather命令本体が来るはずである

GDSのトリガー方法（4/4）

- 前ページの例では、キャッシュクリアにより**キャッシュミス**を誘発させた上で、**アトミックなLMS**（Load-Modify-Store）**命令**である**xchg命令**を実行している

- アトミックなLMS命令では、対象を**排他的にロック**した上でロード・変更・ストアの一連の処理を行うものであるため、CPUからすると**非常に時間的コストの高価な処理**である

- その上キャッシュミスによりさらなる遅延を仕組まれているため、これらの**遅延を軽減**しようとCPUが画策して**過渡的実行が発動**し、結果として**GDSが発生**してしまう

マスクビットが0である場合の挙動

- GDSを発生させるGatherにおいて、ある**インデックス**に対応する**マスクビットが0**である場合、そのインデックスに対応する**ベクトルレジスタ**からは、**GDSにより漏洩**させる事は**できなかった**

- **フォールト**によりGDSが誘発される場合でも、そのような**異常**（Exotic）**アドレス**に**アクセスしない場合**でも**同様**

- これは、**前述のマイクロアーキテクチャ最適化**により、マスクビットが0であるようなデータは**そもそも読み出さない**ようにされるからであると考えられる
  - **GDSを引き起こすGather側のマスクビット**の話である点に注意。**攻撃対象**とするベクトル命令におけるマスクビットについては**また別**である

GDSの影響を受ける命令（1/5）

- **ベクトルレジスタ**に**読み出し**たり、あるいは**一時的なバッファ**として**ベクトルレジスタを使用**する**命令**が、原理的に**GDSによって漏洩**させられてしまう事になる

- 手法の詳細は省略するが、自動的あるいは手動でテストを行う事により、実際に**GDS**により**使用した値が漏洩**してしまうような命令を洗い出して**一覧化**している
  - 攻撃者の実行するGatherによって使用した値が漏洩するような、**攻撃対象となる命令の一覧**である
  - 別の攻撃対象命令から**漏洩させるために使用**できる**命令の一覧でない**点に注意

GDSの影響を受ける命令（2/5）

- GDSによる影響を受ける命令は以下の通り：

Instruction buckets:	(v)(vp)(p)blend*{19}	(v)(vp)(p)cmp*{217}
(v)(vu)(u)comi*{8}	(v)insert*{12}	(v)(vp)(p)align*{4}
(v)(vp)maskmov*{4}	(v)(vp)(p)mov*{47}	(v)perm*{22}
(v)(vp)compress*{4}	(v)(vp)gather*{8}	(v)(vp)max*{12}
(v)scale*{4}	(v)(vp)(p)shuf*{17	(v)rsqrt*{7}
(v)sqrt*{6}	(v)fixup*{4}	(v)fpclass*{10}
(v)getmant*{4}	(v)(vp)xor*{5}	(v)(vp)or*{5}
(vp)rol*{4}	(v)pack*{4}	(vp)(p)srl*{10}
(v)(vp)andn*{5}	(v)(vp)and*{5}	(v)getexp*{4}
(vp)lzcnt*{2}	(v)lddqu{1}	(vp)dpwssd*{2}
(v)dbpsadbw{1}	(vp)sadbw{1}	(v)rndscale*{4}
sha*{6}	(vp)madd*{4}	(vp)ror*{4}
(v)cvt*{74}	(v)dpp*{4}	(v)gf2p8*{6}
(v)(vp)(p)hadd*{10}	(vp)(p)abs*{7}	(vp)(p)clmul*{7}
(v)phmin*{2}	(v)(vp)min*{12}	(v)popcnt*{4}
(v)div*{4}	(v)(vp)broadcast*{17}	(v)fm*{36}
(v)(vp)(p)test*{12}	(vp)multishift{1}	(v)(vp)(p)mul*{13}
(v)rcp*{7}	(v)round*{8}	(v)reduce*{4}
(v)range*{4}	(v)(vp)expand*{6}	(vp)ternlog*{2}
(v)addsub*{2}	(v)(vp)add*{12}	(v)(vp)sub*{12}
(vp)conflict*{2}	(vp)(p)sll*{9}	(vp)(p)sra*{8}
(vp)dpbus*{2}	rep(ne) mov*{8}	xsave/xrstor*{2}
fxsave/fxrstor*{3}	(v)(vp)(p)hsub*{10}	(vp)sign*{3}
(v)(vp)unpck*{12}	(v)fnm*{24}	(vp)(p)ins*{6}
(vp)shl*{6}	(vp)2intersect*{2}	(v)mpsad*{2}
(vp)shr*{6}	(vp)avg*{2}	(v)aes*{12}

※{n}は影響を受けるそのカテゴリの命令の合計数、(v)(vp)(p)はベクトル命令の接頭辞、
*部分は取り扱うデータの型によるバリエーション（接尾辞）

GDSの影響を受ける命令（3/5）

■SIMD読み出し

メモリから**ワイドデータ**（128/256/512bitのデータ）を**読み出す**
全てのSIMD演算がGDSの影響を受ける。

例：読み出しのみを行うvmov*命令、読み出しとXOR演算を実行するvpxor*命令

■SIMD書き込み

compress命令のみが影響を受ける。

■暗号学的拡張命令

AES-NIや**SHA-NI**のような**暗号学的拡張命令**が、**値のロード等**で
内部的にベクトルレジスタを使用するため、これらの拡張機能を
用いたAESやHMAC-SHAから**平文データや秘密鍵**が漏洩する。

GDSの影響を受ける命令（4/5）

■高速メモリコピー

memcpyやmemmoveにおける高速なメモリコピーのために用いられている、rep命令とmovs*命令の組み合わせが、内部でベクトルレジスタを用いているために影響を受ける。
（rep命令はmovs系命令をループさせる命令）

■レジスタコンテキストのリストア

コンテキストスイッチに伴う**レジスタコンテキスト**の**ストアやリストア**時にもベクトルレジスタが使われるため、**GDSの影響を受ける**。

具体的には、**xsave**や**xrstor**で扱われる標準のレジスタと、**fxsave**や**fxrstor**で扱われるワイドレジスタ双方が対象となり、**後者はSGXのAEXやERESUMEで使用**されている。

GDSの影響を受ける命令（5/5）

■ダイレクトストア

ある**64バイト**の値を、コピー元アドレスからコピー先アドレスに**直接コピー**する**ダイレクトストア操作**も**内部でベクトルレジスタを使用**しており、GDSの影響を受ける。

ちなみに、ダイレクトストアはキャッシュを迂回して行われる[4]。

■誤検出のケース

movのような標準的なメモリ読み出しからの漏洩も確認されたが、これは裏で**定期的にOSのタイマ割り込み**等で**実行**される**xrstor命令によるもの**であると判明した。

これは、ForeshadowやZombieLoadのように**ゼロステップ処理**でSGXから**xrstorを狙って漏洩させられる**というヒントとなっている。

エントリサイズからの考察

- 前述の通り、論文執筆時点ではSIMDレジスタバッファが何物なのか判然としていなかった可能性が推測される

- 論文中では、**AVX-512対応**であれば**512ビット**（64バイト）の**zmmレジスタ**へのロードデータの**任意の部分を漏洩**でき、**非対応**であれば**最大32バイト**しか**漏洩できない**事を確認している

- この事からも、**ベクトルレジスタが漏洩元である**事を当時であっても**ある程度推測**できる
  - ただし、仮に**別の不明な内部バッファが存在**し、AVX-512への**対応時のみ大きくなっている**可能性も、この推測だけでは**拭いきれない**

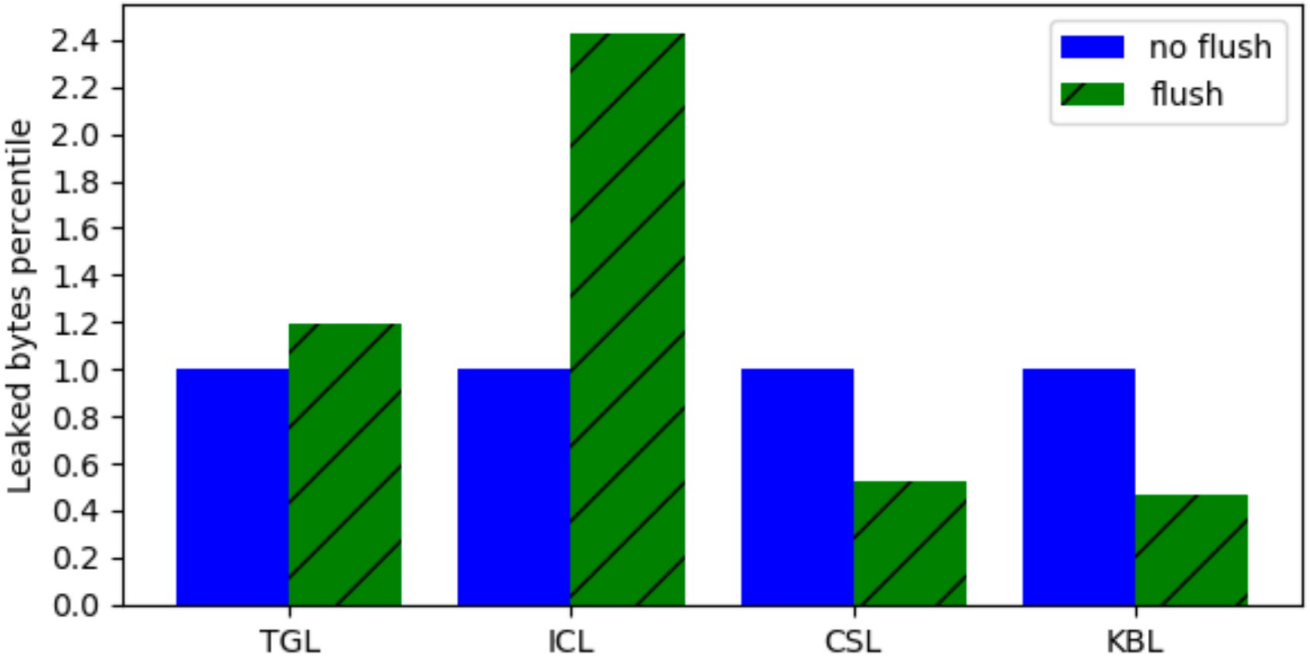
漏洩元の推測（1/4）

- ここで、**Foreshadow**や**MDS**で**漏洩元**となっていた**L1Dキャッシュ**や**マイクロアーキテクチャバッファ**が**GDSの漏洩元では**ない事を確定させておく必要がある

- そこで、**VERW命令**により**マイクロアーキテクチャバッファ**を**フラッシュ**し、**MSR**（モデル固有レジスタ）経由で**L1Dキャッシュ**を**フラッシュ**してみる
  - VERW命令は本来全く関係ないあまり使われない命令であったが、MDSの発見以降μ-Archバッファをフラッシュする副次的機能を付与されている

漏洩元の推測（2/4）

- 結果、フラッシュの有無に関わらず漏洩したため、GDSは既存の攻撃におけるμ-Archバッファとは無関係であると結論付ける事ができる
  - フラッシュ後の方が漏洩レートが高くなっているものについては、副次的な要因で過渡的実行ウィンドウが拡大されたためであると推測される



図は[1]より引用

漏洩元の推測（3/4）

- 既存のMDSやMMIO Stale Data脆弱性を抱えていないTiger Lake CPUでVERW命令を行うと、μ-Archバッファをフラッシュする必要がないため、以下のようにVERWのサイクル数が小さくなる

CPU Generation	GDS		MDS				VERW
	SMT	Switch	SMT	Switch	>TAA	>MMIO	Cycles
Tiger Lake	ϑ	ϑ	θ	θ	θ	θ	80
Ice Lake	ϑ	ϑ	θ	θ	θ	Δ	592
Cascade Lake	ϑ	ϑ	θ	θ	ϰ	Δ	324
Kaby Lake	ϑ	ϑ	ϑ	Δ	ϰ	Δ	696

ϑ Vulnerable　θ Not affected　ϰ TSX disabled　Δ Buffer flush

図は[1]より引用

漏洩元の推測（4/4）

- その状態でも、**SIMDメモリアクセス**のみがGDSの**影響を受けている**事から、**既存の攻撃で悪用**されたμ-Archバッファとは**別**の、**SIMD演算に関連するバッファ**から**漏洩している**と推測される
  - 論文中では「**SIMDレジスタバッファ**」という（不明な）バッファであると言及している

- 実際、前述の通りエンバーゴ期間を経た後に、Intel及びDownfallのトップページ[2]でそれが**ベクトルレジスタであった**事が開示されている
  - この推測の仕方からしても、**論文執筆時点ではその実体が不明瞭であった**事が窺える

Downfall攻撃の実践例

Downfall攻撃の実践例

- ここまでで説明したGDSを応用した、以下の4つの攻撃について順に説明を進める
  - プロセス間秘密チャネル
  - 任意のデータの盗聴
  - Gather Value Injection
  - SGXへの攻撃

プロセス間秘密チャネル

プロセス間秘密チャネル（1/9）

- **秘密チャネル**：本来データ転送のために用意されているものではない要素を用いて構築された、秘密裏にデータ転送を行う通信路
  - 英名：Covert Channel
  - 過渡的実行攻撃でのキャッシュサイドチャネル攻撃におけるキャッシュも、まさに**過渡的領域**から**命令リタイア後**への**秘密チャネル**である

- 攻撃対象プロセスで使用された値を攻撃側プロセスからGDSで盗聴する、プロセス間秘密チャネル攻撃について考える

プロセス間秘密チャネル（2/9）

- 過渡的実行攻撃においては、**秘密チャネル**として主に前述の通り**256スロットの監視用配列**の**キャッシュ**を利用する事が多い
  - あるバイトについて、それをインデックスとして監視用配列に過渡的にアクセスし、FLUSH+RELOADで後から検知する

- **Meltdown**や**Foreshadow**では256スロットの監視用配列を**1つ**用意して**1バイト**ずつ、**ZombieLoad**では256スロット監視用配列を**3つ**用意して**3バイト**ずつ**漏洩値の観測**を行っている

- 一方、GDSでは**32個の256スロット監視用配列**を用意し、64バイトまたは32バイトのベクトルレジスタから**最大32バイトを同時に漏洩**させる**マルチワードデータサンプリング**を考える

プロセス間秘密チャネル（3/9）

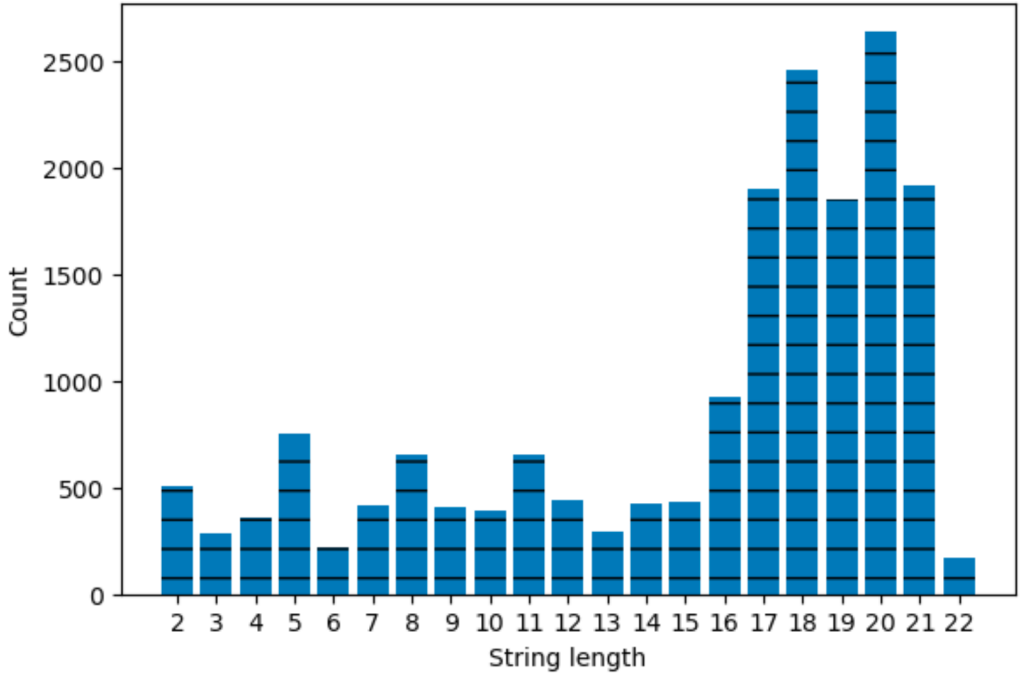
- 理論的には最大32バイト同時に漏洩させる事ができそうだが、実際にはどの程度の同時漏洩が可能であるのかを実験的に確認する

- 攻撃対象の論理スレッド上にて、**64バイトの連続データ**をvmov命令でSIMD読み出しし、それを並行するシブリングスレッドからGDSで**可能な限り同時に複数バイトを漏洩**させる
  - 具体的には、連続データはA..Za..z0..9#!の64バイトの連続データである。ただし、ピリオド2つは表記上の省略を表している

- ベクトルレジスタ内の特定の要素をスカラー値として抽出するvextract*命令やpextr*命令、そしてベクトル内の要素の並び替えや特定要素の取得を行うvperm*命令（Permute命令）を駆使してマルチワード漏洩を試みる

プロセス間秘密チャネル（4/9）

- Tiger Lake CPU上で前ページで述べたマルチワード漏洩手法を試した所、以下の図に示す通り、**最大同時漏洩数**は**22バイト**であり、殆どの場合**16～21バイトの同時漏洩数**であった
  - 32バイトの同時漏洩ができない原因としては、過渡的実行ウィンドウの限界やノイズの混入などが考えられるが、論文中では言及されていない



図は[1]より引用

プロセス間秘密チャネル（5/9）

- 同時漏洩数の次は、漏洩するデータの**データパターン**について実験的に確認する

- **アルファベット列**（A～T）と**Xの羅列**、そして**Yの羅列**の3つの羅列を用意し、**同一または他の羅列と同時**にvmov命令で**ロード**し、そこから**GDSを用いて抽出する**事を試みる
  - dwordを収集するGatherとqwordを収集するGatherの両方について実験を行う

プロセス間秘密チャネル（6/9）

- 実験の結果、以下のようなデータ漏洩パターンが観測された
  - 非常に分かりにくいが、3行で1つの塊として見た際に、一番上の塊がアルファベット列、2番目がXの羅列、3番目がYの羅列のロードについての結果を示している
  - その上で、それぞれ1行目はアルファベット列との同時ロード、2行目はXの羅列との同時ロード、3行目はYの羅列との同時ロードを示している

図は[1]より引用

プロセス間秘密チャネル（7/9）

- このように、**同時に発生するベクトル命令**によって**読み取り結果**に**混合が発生**するため、攻撃の裏で行われている処理による**意図せぬノイズが入る**事もままある
  - よって、単一のdwordあるいはqwordよりも大きい連続した正しいデータを漏洩させられる保証はない

- 論文では、ベクトルレジスタ内の各要素同士の並べかえを行う**Permute命令**を用いる事で、目当てのdwordやqwordのみを抽出できるとしている
  - が、前ページで示したデータ漏洩パターンが決定的なものであるのか書かれていないため、どこまでコントロールできるのかは不明

プロセス間秘密チャネル（8/9）

- 前ページまででその性質を実験的に確かめた、GDSによるマルチワードデータサンプリングを用いて、実際に**プロセス間で秘密チャネルを構築**し漏洩速度の**ベンチマークを取る**

- **vmov**命令、**rep mov**（高速メモリコピー）、**fxrstor**命令、**aes**命令の3つからそれぞれ使用したデータをGDSを用いて漏洩させる

- さらに、過渡的実行を誘発させる方法として、**フォールト使用**、**キャッシュ不可能メモリ使用**、**いずれも不使用**（前述のアトミックなLMSを用いた方法など）の3パターンについて測定する

プロセス間秘密チャネル（9/9）

- ベンチマークの結果は以下の図の通り（図は[1]より引用）：

CPU Generation	vmov			rep mov			fxrstor			aes		
	●	∪	×	●	∪	×	●	∪	×	●	∪	×
Tiger Lake	4128.78	5584.57	5870.3	3318.15	1438.53	1414.55	92.35	1465.13	178.68	688.27	1763.57	1101.7
Ice Lake	0.73	2.48	6.25	11.67	58.13	30.97	0.0	0.57	3.05	0.1	6.68	7.42
Cascade Lake	133.27	72.47	2424.83	19.23	14.23	2569.78	76.2	3.98	1209.13	8.0	75.77	1395.7
Kaby Lake	0.03	26.45	11.12	0.2	3.87	70.2	0.03	0.1	0.07	0.0	0.13	2.03

● Cacheable no fault　∪ uncacheable　× Page fault

- **Tiger Lake CPU**において**ページフォールト**を使用したGDSにより**vmov**命令から漏洩させるシナリオが**5870.3byte/s**と最高効率であった事が分かる

任意のデータの盗聴

任意のデータの盗聴（1/2）

- 次に、**任意の静止データ**（Data-at-Rest）をGDSによって盗聴する攻撃を考える
  - **Data-at-Rest**：ここでは、**攻撃対象アドレス**に**マッピング**されているが**使用されていない**データを指す。本来は「保存データ」と言い、ある処理において使用されていない補助記憶装置上のデータを指す

- この攻撃では、CPUが**静止データ**をベクトルレジスタに**プリフェッチ**する事により、ソフトウェアが**読み込んでいない**にも関わらずGDSが漏洩させられる状況が**2通り**ある事を悪用する
  - **境界外（OOB; Out-Of-Bounds）プリフェッチ**
  - **NOPプリフェッチ**

任意のデータの盗聴（2/2）

- **境界外プリフェッチ**：ソフトウェアは**本来nバイト**のみ**読み取り**を行うはずなのにも関わらず、CPUが**最大x個のキャッシュライン**（**$64\times x$バイト**）をプリフェッチし**漏洩**させてしまう挙動

- **NOPプリフェッチ**：ソフトウェアは**本来0バイト**を読み込む（つまり「何もしない命令」である**nop命令**を実行する）にも関わらず、CPUが**最大x個のキャッシュライン**をプリフェッチし**漏洩**させてしまう挙動

- これらは、**マスク付きmove命令**（maskmov）や**繰り返しmove命令**（rep mov）からのGDSによるデータの漏洩に悪用する事ができる

マスク付きmove命令への攻撃

- **マスク付きmove命令**：対応する**マスクビット**が**1**であるような要素のみmove（コピー）を行うようなSIMD命令
  - **Gather命令**における**マスクレジスタ**のそれと全く同様のイメージ
  - マスクビットが**全て0**である場合は、本来は**NOP命令**となるはず
- この時、**単一のdword**を読み取ったり、そもそもマスクビットが**全て0**である場合でも、攻撃対象アドレスから**64バイト**を**CPUがプリフェッチ**してしまう
  - GDSの**攻撃側のGather命令のマスクレジスタ**とは**全く別の議論である点に注意**。前述の通り、Gather側はマスクビットが0だと収集が行われない
- 当然、本来アーキテクチャ的にアクセスされるはずがない要素（**静止データ**）も**プリフェッチ**されてしまうため、このような静止データが**GDSにより漏洩**させられてしまう

繰り返しmove命令への攻撃（1/2）

- コピーする連続データのバイト数を***t***とした時、繰り返しmove命令である**rep mov{t}**命令を実行する場合について考える
  - 前述の通り、**memcpy命令**や**memmove命令**で内部的に使用される

- この命令により、本来は%rcx * sizeof(t)がアドレス%rsiからアドレス%rdiにコピーされるはずである
  - 早い話が**memcpy(%rdi, %rcx * sizeof(t), %rsi)**のようなイメージ

- しかし、Tiger Lake CPUで試した所、データ粒度や%rcxの値に関わらず、**キャッシュライン2つ分**（**128バイト**）までrep movから**GDSによりデータを漏洩**させられる事が分かった
  - コピーサイズが128バイトよりも小さい場合でもこの挙動が発生する

繰り返しmove命令への攻撃（2/2）

- rep mov命令は**広範な場面で使用**されているため、以下の理由によりセキュリティ上のリスクが大きいものとなっている：
  - rep movは**大きな連続秘密データの転送**に用いられる事が多いmemcpyで使われるため、そこかしこで**秘密が漏洩するリスク**となり得る
  - 最大で**128バイトの境界外データ**を**漏洩**させてしまうため、ある種の**過渡的バッファオーバーフロー**ともみなせる挙動が行われてしまう
  - rep movに対するGDSにより、攻撃者は本来アクセスできない領域のデータを取得できてしまうため、**混乱した代理ガジェット**として機能してしまう可能性がある
- rep mov命令によるこの過剰なプリフェッチは、rep mov命令の**投機的な動作に起因**する可能性があると参考文献[5]が示す研究において確認されている

データ漏洩ガジェット（1/5）

- ここでは、特に**繰り返しmove命令**からGDSにより**任意の静止データを漏洩**させる攻撃について詳細に見ていく

- 伝統的なバッファオーバーフロー（BoF）攻撃やSpectre攻撃には脆弱ではないが、興味対象のデータをベクトルレジスタに取り込み、結果としてGDSによりそれを漏洩できてしまう**3通りのコード列（ガジェット）**を紹介する

データ漏洩ガジェット（2/5）

- 3通りのガジェットのコード列は以下の通り：

```
// Gadget 1: Safe check
if(copySize < sizeof(local) &&
  copySize+index < sizeof(source)){
    memcpy(local, source+index, copySize);
}

// Gadget 2: Safe no-op check
if(copySize >= sizeof(local) ||
  copySize+index >= sizeof(source)){
    copySize = 0;
}
memcpy(local, source+index, copySize);

// Gadget 3: Buggy unexploitable
if(copySize < sizeof(local))
    memcpy(local, source+index, copySize);
```

データ漏洩ガジェット（3/5）

■ガジェット1：安全なチェック

- **境界外**の**読み取り**や**書き込み**の双方を回避するための**正しい入力サニティチェック**を行っている例
  - **サニティチェック**：境界外参照が起きないかのチェックの事

- **ソフトウェアレベル**では**安全**だが、GDSによりソースバッファの**境界を超えた漏洩が可能**であり、また攻撃者がインデックスを入力できるため、攻撃の幅も広い

- 例えば、sizeof(source) = 64、index = 63、copySize = 1である場合、サニティチェックには合格するが、rep movに伴う**境界外プリフェッチ**により**後続の128バイト分が漏洩**してしまう
  - いわばインデックス64～191に相当する部分

データ漏洩ガジェット（4/5）

■ガジェット2：安全なNOPチェック

- コピーサイズとインデックスをチェックし、それが境界外アクセスに繋がっていると判断した場合、単純に**copySizeを0にする**実装

- このようなゼロサイズmemcpyはC言語やrep movにて行われるが、最適化により**NOP命令に置換**される
  - よって、アーキテクチャ的にはコピー処理自体が試行されない

- しかし、この場合もCPUによる**NOPプリフェッチ**によって値が取得され、**GDS**によってそれを**漏洩**させる事ができてしまう

データ漏洩ガジェット（5/5）

■**ガジェット3：バグはあるが従来型攻撃への悪用はできない例**

- ユーザにはアクセスできない**非公開なlocalバッファ**に対して
  コピーを行う例

- localの**メモリ破壊は発生しない**一方、**インデックスのチェックを行っていない**ため、 sourceバッファの**境界外読み取り**は発生する

- ただし、**localバッファ**が**非公開**なため、本来はlocal経由で**sourceの境界外**を**読み取る事はできない**

- しかし、この場合もsource+indexの位置を起点としたCPUによる**境界外プリフェッチ**が発生し、**GDSによる漏洩**ができてしまう

ユーザ空間からのカーネルメモリの読み取り（1/4）

- ここまで説明したガジェット1～3を実際に実装し、**ユーザ権限の攻撃者**が**GDS**を用いて**カーネルデータを漏洩**させる実験を行う

- 従来型のソフトウェア攻撃でカーネルを侵害できないよう、indexやcopySizeといったユーザ入力は**ioctl**を通じて受け付ける、**ローダブルカーネルモジュール**（LKM）を作成する
  - **LKM**：後付で追加できる、OSカーネルを拡張するオブジェクトファイル。/dev/moduleのようにマウントして使用する。勿論削除（アンマウント）も可能。
  - **ioctl**：ドライバとやり取りをするためのシステムコール

- コピー先の**localバッファ**についても、全ガジェットにおいて**非公開バッファ**であるとする

ユーザ空間からのカーネルメモリの読み取り（2/4）

- あるプロセスを用いてユーザ空間から**ioctl経由**で**indexと copySizeを提供**し、**並行するシブリングスレッド**上のプロセスで**GDSを実行**する

- 攻撃者は、**indexを操作**する事で**目当てのデータ**を**プリフェッチ**させGDSにより漏洩させる事ができる

- まずdwordを1つ漏洩させ、次に2バイトだけ先に進める。
  前の反復の後半2バイトと現在の反復の前半2バイトが一致したら、
  エラーが発生していないとして受理する、という操作を繰り返す
  - これにより観測結果を確実性の高いものにする事ができる

ユーザ空間からのカーネルメモリの読み取り（3/4）

- まず、**ガジェット1**を悪用し、Tiger Lake CPU上でこの攻撃を実行する

- sourceバッファは**キャッシュラインサイズ**（64バイト）に**アライン**されているものとする
  - sourceバッファがキャッシュラインサイズぴったりだと、プリフェッチャがsourceの次のキャッシュライン2つ分を読むように判断し取得するため、境界外の128バイトをベクトルレジスタに持って来られる

- それぞれ10回攻撃を実行した所、128バイトの境界外のカーネルデータを**平均1.04秒**で**漏洩**させる事に成功した

ユーザ空間からのカーネルメモリの読み取り（4/4）

- また、**238バイトのLinuxバナー文字列**を、ガジェット2では**1.37秒**、ガジェット3では**1.58秒**で漏洩させられた
  - **Linuxバナー文字列**：ビルドバージョンやタイムスタンプが記載されている、カーネルメモリ上の文字列[6][7]。OS起動時によく目にする。

- Linuxカーネルのバイナリでは**2728個のrep mov系命令**が存在し、ソースコードでは合わせて**25992個のmemcpy**及び**memmove**（前述の通り、内部でrep mov系命令を用いている）が見つかった

- このように、**原因となる命令**が**広範に存在する上**、本来**バグですら無いコード**（ガジェット1・2）でも**GDSによって漏洩**が発生するため、**対策に難儀するであろう**事が想像できる

Gather Value Injection

Gather Value Injection（1/6）

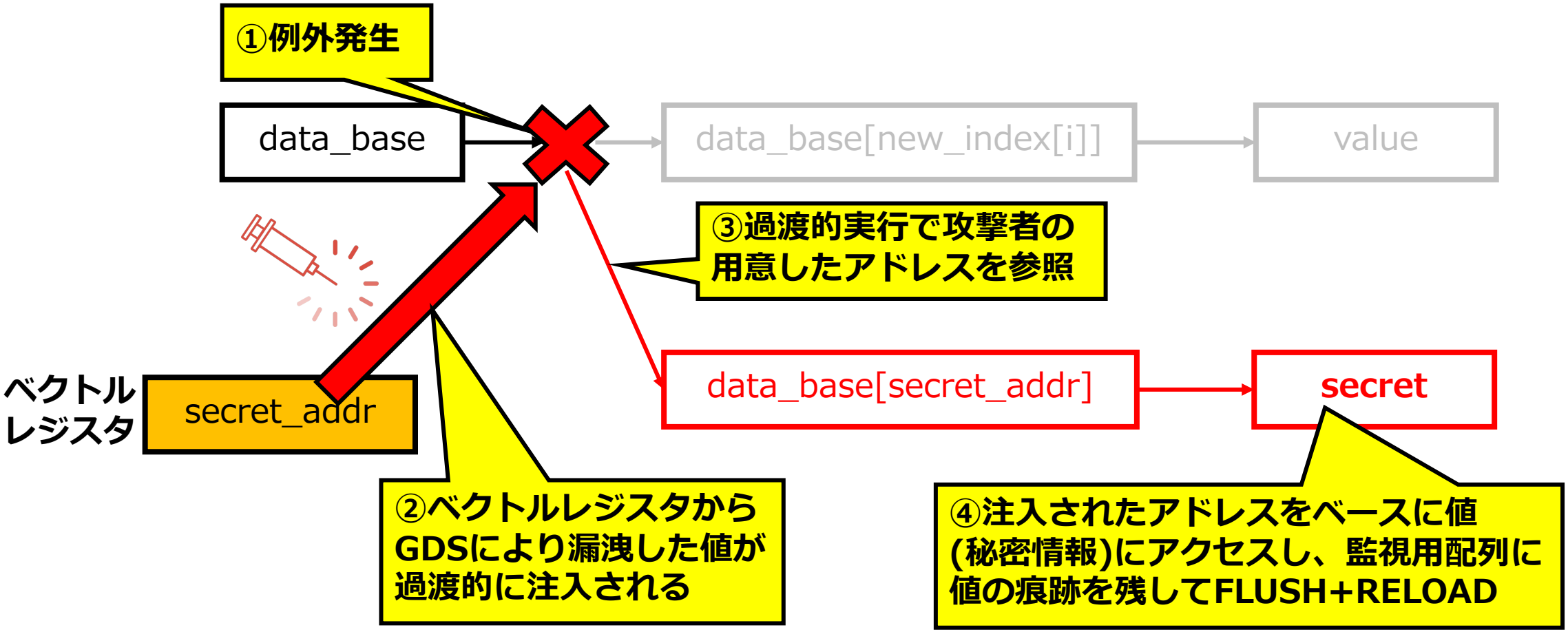
- GDSは**Meltdown型**の過渡的実行攻撃であるため、LVI同様ベクトルレジスタから漏洩した値を後続の命令への**注入に転用**できる事が推測できる

- 実際に、Downfallでは**Gather Value Injection（GVI）**としてGDSを**LVI的な攻撃に転用**する事に成功している
  - LVIについての詳細はSGX攻撃編③のスライドを参照

- LVIとは異なり、ほぼSGX専用の攻撃というわけではない

Gather Value Injection（2/6）

- GVIの概要図は以下の通り
  - 比較的LVI-SBに類似している



Gather Value Injection（3/6）

- Downfallの論文では、2通りのGVIガジェットのコード例を示している：
  - [i]がついている部分はSIMD的に処理される部分だと考えられる

```
// Gadget A: Gather followed by a load
new_index[i] = gather(index_base, index[i]);
value = data_base[new_index[0]];
leak_to_side_channel(value);

// Gadget B: Double gather
new_index[i] = gather(index_base, index[i]);
values[i] = gather(data_base, new_index[i]);
leak_to_side_channel(values[i]);
```

Gather Value Injection（4/6）

- ガジェットAでは、GDSによってnew_indexに**ベクトルレジスタ由来の値**を**過渡的に代入**し、**後続の過渡的命令**における配列参照のインデックスとして**注入**している
  - GDSによりnew_indexに格納された値を境界外を指すような**不正な値**にする事で、本来アクセスしてはいけない**秘密情報を参照**できてしまう

- ガジェットBも基本的にAと同様だが、valuesに対する代入（過渡的注入）を**SIMD的**に行う事で、**より効率的**に攻撃による秘密の盗聴が可能となる

- ベクトルレジスタを不正なインデックスで埋め尽くすため、攻撃者は並行するシブリングスレッド上でそのような値のvmovを繰り返す

Gather Value Injection（5/6）

- 実際にGVIによって秘密情報を漏洩させる実験も論文では行っている
  - いずれのガジェットにおいても、キャッシュに痕跡を残しサイドチャネル的に観測するのはGDSやLVIと同様

- GDS同様、Gather対象をキャッシュ不可能として過渡的実行を誘発する。また、Gatherの前にキャッシュミスを誘発させる事で過渡的実行ウィンドウを拡大させている

- Tiger Lake CPUで10秒間**GVI**を実行するのを100回繰り返した所、1秒間に平均**8734.3バイト**の境界外データを漏洩させる事に成功
  - ガジェットA・Bのどちらが使用されたのかは論文中に明記されていない

Gather Value Injection（6/6）

- これもGDS同様に、GVIも必ずしも**過渡的実行の誘発**に**異常（Exotic）アドレス**を**用いる必要はない**

- 先程のガジェットのような、Gather命令を用いた二重インデックスを用いている例として、耐量子公開鍵暗号の1つである**CRYSTALS-KYBER[8]**の**実装**が挙げられる

SGXへのGDS攻撃

SGXへのGDS攻撃（1/4）

- 攻撃の実例の最後として、**おまけ程度**に**SGXのEnclave**から**シーリング鍵を抽出**する攻撃を考案し実現に成功している

- Enclaveのコードページを実行不能とし**ゼロステップ処理**を発動させる事で、AEXの度に**同一ハイパースレッド上**で**GDSを実行**する
  - ゼロステップ処理に関してはForeshadowの解説（SGX攻撃編③）を参照

- これにより、AEXやERESUMEの内部実装である[9]**fxsave**や**fxrstor**から、予めベクトルレジスタに格納しておいた**既知の古い値**が**漏洩する事が判明**した
  - Kaby Lake向けの当時最新のマイクロコードアップデートを適用した状態でも漏洩可能であった
  - さらに、**ハイパースレッド不使用**でも**攻撃に成功**した

SGXへのGDS攻撃（2/4）

- シーリング鍵の中でも、**EPID-RAにおける信頼性の根拠そのもの**である**Attestationキー**をシーリング/アンシーリングするために使用される、**PSK**（Provisioning Seal Key）を**抽出**する
  - PSKはAttestationキーを取り扱うPvEやQEにより使用される

- 当然、PSKが漏洩すれば**Attestationキー**も簡単に**PSKで復号し抽出**できてしまう

- 他の攻撃の場合と同様、**Attestationキーが漏洩**する事で**QUOTE構造体の偽造**が可能となり、Intelにより失効してもらわない限り**RAの信頼性が破綻**してしまう

SGXへのGDS攻撃（3/4）

- シーリングのためにSGXSDKで用意されている**sgx_seal_data関数**は、内部で**EGETKEY命令**のラッパーである**sgx_get_key関数**を呼び出している

- sgx_get_key関数は、まず**初め**に**AES鍵の鍵伸長**のために**l9_aes128_KeyExpansion_NI関数**を呼び出しているが、これが**AES-128のマスター鍵**を**ベクトルレジスタxmm0にロード**している
  - AESの鍵伸長についてはLVIの解説（SGX攻撃編③）を参照

- よって、**sgx_seal_dataの最初**で**一時停止**し、SGX-Step等を使用しつつ**ゼロステップ処理**を行えば、xmm0のコンテキストを内包する**SSA**から**AES鍵**（＝**PSK**）を**抽出**できる

SGXへのGDS攻撃（4/4）

- 攻撃対象のl9_aes128_KeyExpansion_NI関数は以下のようなコードである：

```
<l9_aes128_KeyExpansion_NI>:
endbr64
vmovdqu (%rsi),%xmm0
vpslldq $0x4,%xmm0,%xmm2 // <-- Zero Stepping
```

- 実際にvpgatherddとvpermdd（Permute命令）を使用したGDSを10秒間実行してAEから**4つの異なるdwordを抽出**し、最も高い頻度で出現したものを組み合わせた所、**PSKを構築する事に成功**した
  - PvEかQEのどちらを攻撃したのかは明記されていないが、sgx_seal_keyを攻撃するという記述から、**PvEを狙っていると推察**できる

軽減策

軽減策（1/4）

- **ハイパースレッディングの無効化**は部分的に有効だが、動作性能に影響がある上、SGXへの攻撃のような**コンテキストスイッチに伴うGDS**による漏洩は**対策できない**
  - そもそもハイパースレッドを攻撃に用いていないため

- 影響を受ける**SIMD命令の禁止**や**Gatherの無効化**は、**動作性能の著しい低下**や**ソフトウェア互換性の喪失**を招く可能性があり、様々なシナリオにおいて**致命的**となり得る

軽減策（2/4）

- LVI等と同様、Gather命令の後ろに**lfence命令**を**挿入**する事で、後続の命令への過渡的転送を阻止しGVIを阻止する事が可能

- **コンパイラが信頼可能**かつ実行バイナリ中の命令を攻撃者が選択できない状況であれば、**コンパイラ**が**Gather命令の内部**に**lfenceを挿入**する事でGDSも対策できる
  - SGXであれば上記の対策を盛り込んだEnclaveイメージを作成した上で、**RAでMRENCLAVEをチェック**する事により信頼し軽減を実現できる

- 実際に、IntelはGDSとGVIを軽減するための、Gatherからの過渡的転送を防ぐマイクロコードアップデートをリリース予定である
  - 恐らく既にリリースされている

軽減策（3/4）

- MeltdownタイプやMDSタイプの攻撃を発見するファジングベースのテストツールとして、 **Transynther**[10]というものがある
  - **ファジング**：異常値を入力する事でシステムの欠陥を検出するテスト手法

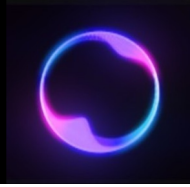
- 従来のTransyntherでは**Gather命令についてのテスト**を**生成していなかった**ために、今までは**GDSを発見できていなかった**

軽減策（4/4）

- そこで**Gatherのテストのみを行う**ようTransyntherを改造して実行した所、様々な場合における**GDSの自動的な発見**に成功した
  - 並行するシブリングスレッドを使う場合と使わない場合の双方にて、様々な誘発条件（Meltdown-MPK/UC/US）に基づくGDSを発見

- この事から、他のMeltdownタイプの過渡的実行攻撃と同様、**Gatherのような命令**でもその**Meltdown型脆弱性**の**自動的な発見**に**有効**であると考えられる

本セクションのまとめ



- Downfallは、GDSやGVIという攻撃・脆弱性の総称である
- GDSは、ベクトルレジスタに残留する古い値を、過渡的なGather命令が後続の命令に転送し、キャッシュサイドチャネル的に不正に観測できてしまう危険性のある攻撃である
- GVIでは、GDSにより漏洩した値を、LVIと同じく後続の命令で使用する値として注入し、データ漏洩や制御フローのハイジャックのような不正な動作を実現する攻撃である
- SGXだけでなく、プロセス間、カーネル、VM等の様々な境界をまたいで漏洩を行う事ができる

参考文献（1/2）

[1]”Downfall: Exploiting Speculative Data Gathering”, Daniel Moghimi, https://downfall.page/media/downfall.pdf

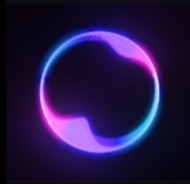
[2]”Downfall Attacks”, Daniel Moghimi, https://downfall.page/

[3]”Gather Data Sampling”, Intel, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/gather-data-sampling.html
（魚拓：https://archive.is/NCTdf）

[4]MOVDIR64B — Move 64 Bytes as Direct Store, 2023/10/15閲覧, https://www.felixcloutier.com/x86/movdir64b

[5]Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing, Oleksii Oleksenko et al., https://arxiv.org/pdf/2301.07642.pdf

参考文献（2/2）



[6]コメントから読む Linux カーネル, Sano Taketoshi,
http://archive.linux.or.jp/JF/JFdocs/readkernel.html

[7]linux/init/version-timestamp.c, GitHub,
https://github.com/torvalds/linux/blob/b85ea95d086471afb4ad062012a4d73cd328fa86/init/version-timestamp.c#L28

[8]CRYSTALS–Kyber: a CCA-secure module-lattice-based KEM, Joppe Bos et al.,
https://eprint.iacr.org/2017/634.pdf
実装：https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/Kyber-Round3.zip

[9]Intel® Software Guard Extensions Programing Reference, Intel,
https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf