

11. SGX攻撃編④

Ao Sakurai

2025年度セキュリティキャンプ全国大会
L3 - TEEビルド&スクラップゼミ

本セッションの目標



- SGXに対する過渡的実行攻撃の内、「ZombieLoad」「Downfall」を紹介する
- これらの攻撃は実践するには極めて高度であるため、あくまでも本ゼミでは解説を行うに留める

ZombieLoad

マイクロコードアシスト



- フォールトの処理やページテーブルの内容の変更のような**複雑な操作**は、通常事前に定義された**マイクロコードルーチン**に対して**マイクロシーケンサ**を向けさせる
 - マイクロシーケンサ：命令処理に使用するマイクロ命令の組み合わせを決定する機構
- その後、実行ユニットは**イベントコード**（マイクロコードイベント時の例外処理コード）を例外の起きたマイクロ命令に関連付け、その**イベントコードに対応するマイクロ命令**を読み出す
- 上記の一連の処理により、**例外や複雑な操作の処理**を行うマイクロコード上の機能を**マイクロコードアシスト**という

Intel TSX (1/2)



- **Intel TSX** : Intel Haswell CPUで導入された、トランザクション処理のためのx86拡張命令セット
 - Transactional Synchronization Extensionsの略
- 特定のコード領域を**トランザクション的（排他处理的）**に実行し、その**コード領域全体が正常に完了**した時点で、そのメモリ操作を**アトミックなコミット**として他の論理CPUに反映させる

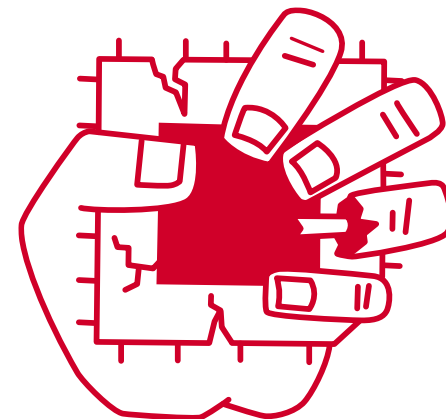


- トランザクション処理中に何らかの問題が発生した場合、アトミック性を保つために**TSXアボート**を発動させる
- TSXアボートが発動すると、実行自体を**トランザクション前**のアーキテクチャ状態に**ロールバック**し、トランザクション領域で実行された**全ての操作を破棄**する
- TSXアボートは、トランザクション内で変更されたアドレスから別の論理CPUから読み書きする事による**競合的なメモリ操作**等、様々な問題によって誘発される

ZombieLoad (1/2)



- ロード操作時に**フォールト**や**マイクロコードアシスト**が発生した際に、**過渡的実行ウィンドウ**においてLFBに存在する古い値が漏洩する事が観測された
 - LFBは**全ての論理CPUによって使用**され、プロセスや権限の**区別を行わない**点にも注意
- このようなLFBからの漏洩を発生させるロード (**ゾンビロード**) を悪用し、過去に使用された**秘密情報を漏洩**させる**過渡的実行攻撃**が**ZombieLoad**である





- **L1DとL2以上のメモリ階層**とのインタフェースとして機能する**LFBからの漏洩**であるため、攻撃の直前では原則として**キャッシュのクリア**を行い、**LFB経由でメインメモリからフェッチ**させる
 - これにより、フェッチに伴い**LFBにもそのデータが残留**する
- Meltdownのように**アドレスで漏洩対象を選択する事は出来ない**。
良くも悪くも、**直前にロードやストアされた任意のデータの漏洩**であるため、この特性を上手く使ってやる必要がある

ZombieLoadの原因



- ZombieLoadの漏洩の原因だが、MeltdownやForeshadow、他のMDS攻撃であるRIDLやFalloutと異なり、実は**根本原因が明らかになっていない**
- 論文では**Stale-Entry**仮説というものを立てているが、その実証については**見送っている**

Stale-Entry仮説 (1/2)



- マイクロコードアシストが発動されると、**命令パイプラインをフラッシュするマシНКリア**が発生し、また**既に実行中の命令**についても**強制終了**させる
- 言うまでもなく**かなりの遅延が発生するため**、その**遅延を最小限に抑えるべく**、**物理アドレスの一部が一致する限り**、**フィルバッファエントリ**（LFBのエントリ）が過渡的領域で**楽観的にマッチング**されると予想される

Stale-Entry仮説 (2/2)



- この楽観的なマッチングにより、あるロードで例外が発生すると、**直前のロードやストアで有効だったような誤ったLFBエントリで過渡的に処理が継続され、その値の漏洩が発生する**
 - 本来もう使われてはならない古い値を使用する事になるため、これは **Use-After-Free脆弱性** である
- LFBは、前述の通り **論理CPU（ハイパースレッド）** 間で競合的に**使用される**事がIntelにより文書化されている
- よって、古いLFBエントリは**並行するハイパースレッド**からも**漏洩を行う**可能性がある
 - ちなみに、並行するハイパースレッドの事を**シブリングスレッド**や**シブリング論理コア/CPU**と呼ぶ
 - シブリングスレッドから漏洩を行う攻撃の方が寧ろ多い

ZombieLoadにおける漏洩元 (1/5)



- ZombieLoadにより漏洩する値がどこから来ているのかについても論文中で具体的に検証されている
- まず、あるページを**キャッシュ不可** (Uncacheable) とし、既存の**キャッシュをフラッシュ**する事で、そのページからのメモリロードは全て**キャッシュ階層を迂回**するように仕向ける
 - こうする事で、毎回メインメモリから**直接LFBに向かう**事になり、**キャッシュには値が残らないがLFBには残る**状態になる

ZombieLoadにおける漏洩元 (2/5)



- この状態でZombieLoadを実行すると、i7-8650Uで1秒あたり平均5.91B/sのレートで**漏洩が発生**したため、**この漏洩はLFBに由来する**ものであると帰着できる
- この時、MEM_LOAD_RETIRED.FB_HITというLFBヒットを示すパフォーマンスカウンタが数千回を示していたため、**間接的にLFB由来**である事を**裏付けている**

ZombieLoadにおける漏洩元 (3/5)



- しかし、全ての漏洩がLFB由来であるかという**実はそうではなさそう**であると論文中で仮説を立てている
 - 同じMDS攻撃の一員である**RIDL**の論文における結論や、Intelによる攻撃分析レポートでは、LFBからのみ漏洩するとしていた
- **LFBへのリクエストやアクセスが発生しない**状況を作るために、**TSXのトランザクション処理**を使用する

ZombieLoadにおける漏洩元 (4/5)



- トランザクション内部で書き込みを行う場合、**使用するデータが書き込みセット内に存在**している必要がある
 - 書き込みセットは必ず**L1キャッシュ内に存在**していなければならない
- 処理中に書き込みセットからデータを退避すると**TSXアボートが発動**されるため、**トランザクションにおけるデータは確実にL1キャッシュから提供**される
- **L1キャッシュはLFBよりもCPUの中心に近い位置**に存在するため、**L1から提供**される限り**LFBへのアクセスが発生する事は無い**

ZombieLoadにおける漏洩元 (5/5)



- このように、**LFBへのアクセスを封じた状況**でも、ZombieLoadによる**データの漏洩が確認**された
 - 数kB/sというかなりの漏洩レートが観測されている
 - LFBからの漏洩を間接的に示すMEM_LOAD_RETIRED.FB_HITやMEM_LOAD_RETIRED.L1_MISSパフォーマンスカウンタは増えていない
 - 一方、MEM_LOAD_RETIRED.L1_HITは数千回を記録している
- よって、少なくとも**漏洩はLFBからだけではない**事は分かるが、具体的にどこから漏洩しているかは不明
 - 紛らわしいが、上記の漏洩が**L1から来ているとは限らない**
- **LB等の他のμ-Archバッファ**が関与している可能性があるが、詳細は不明

データサンプリング (1/5)



- ZombieLoad (やRIDL) は、前述の通り**アドレスで漏洩対象を指定する事はできない**
 - かつ、ZombieLoadで漏洩する値は、**例外の発生したアドレス上の値に由来するものではない**。あくまでも**直前にロードやストアされた値**が漏洩する
- しかし、ZombieLoadを発動させるロードに渡す仮想アドレスの**下位6bit** (= **64通り**表現可能) により、LFB (サイズは**64B**) 内のどのエントリを使用するかを**バイト単位で一意に指定する事が出来る**
- よって、ZombieLoadでは**直前のロードやストアの値**を、この**下位6bitによる指定**と組み合わせながら**LFBから漏洩させる事が肝**となる

データサンプリング (2/5)



- 様々なMeltdown型攻撃において、漏洩対象のどこまでを攻撃者が明示的に指定できるかを表した図 ([1]より引用)

	Page Number			Page Offset	
Meltdown	51	Physical	12	11	0
	47	Virtual	12		
Foreshadow	51	Physical	12	11	0
	47	Virtual	12		
Fallout	51	Physical	12	11	0
	47	Virtual	12		
ZombieLoad/ RIDL	51	Physical	12	11	6
	47	Virtual	12		



- ところで、従来のメモリベースのサイドチャネル攻撃は、**メモリアクセス位置を特定し、実行時間等からその時点での処理で使用されている秘密情報を漏洩させる**
 - 早い話、「この実行時間でこのアクセスパターンならこの値がこの処理で使われているに違いない」といった推測が可能
 - **メモリアクセス位置と実行中の処理（命令ポインタにより管理される）の実行時間を照らし合わせて秘密情報を推測するため、メモリアクセス位置と命令ポインタを関連付けるような攻撃である**
- 一方、**Meltdown**は本来アクセスしてはならない**アドレス**に過渡的にアクセスする事で、**その値をサイドチャネル的に観測する**
 - キャッシュに痕跡を残す必要はあるとはいえ、こちらは**対象アドレスからデータを比較的直接的に引きずり出している**

データサンプリング (4/5)

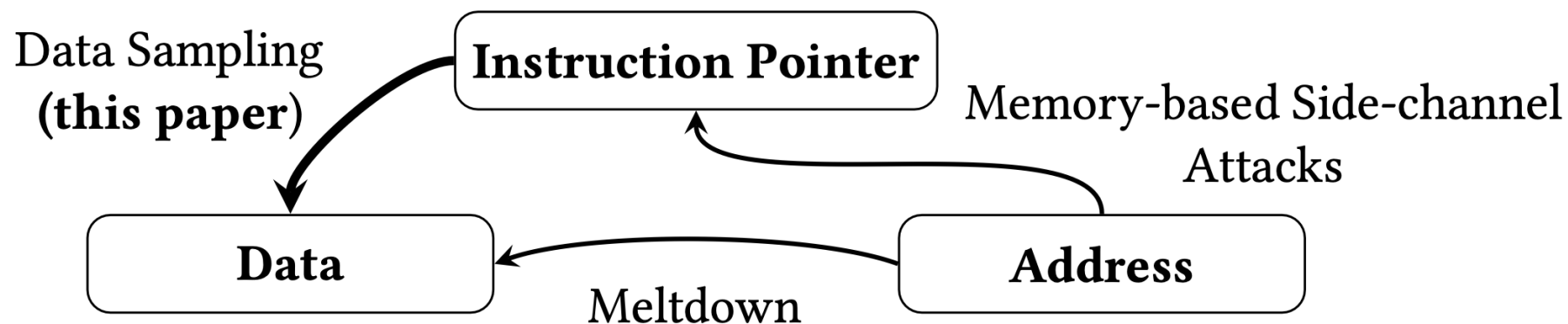


- 一方、ZombieLoadは**直前の処理**（**命令ポインタ**が現在指しているよりも前の命令）で使用された**データ**を、**LFB内**からバイト単位で**任意に指定して漏洩**させる事ができる
- よって、言わばZombieLoadは**命令ポインタ**（直前の処理）と**データ**を**関連付ける**ような攻撃と言える
- そして、このような攻撃の事を**データサンプリング攻撃**と呼ぶ事に行っている。ZombieLoad、RIDL、Falloutは全て**μ-Arch**上でこれを行うため、**MDS攻撃**（**Microarchitectural Data Sampling**）と命名されている

データサンプリング (5/5)



- 命令ポインタ、データ、アドレスを、それぞれの攻撃（従来型、Meltdown、データサンプリング攻撃）がどのように関連付けているかを表した図（[1]より引用）



RIDLとの比較



- 比較的ZombieLoadと類似しているMDS攻撃であるRIDLとの比較表は以下の通り（バリエーションについては後述）：

	RIDL	ZombieLoad
漏洩元	LFB、LP	（少なくとも）LFB
漏洩の発生するロード	キャッシュされないロード 操作のみ（LFB）	全てのロード（LFB）
漏洩の発生するストア	全てのストア（LFB）	全てのストア（LFB）
既知のバリエーション数	1か2	5
悪用されているフォールト	ページフォールト	マイクロコードアシスト、 ページフォールト
軽減策で修正済みか	はい	いいえ
MDS耐性のあるCPUで 動作するか	いいえ	はい（バリエーション2）



ZombieLoadはSGXだけを侵害するための攻撃ではないため、様々なシナリオにおいてデータの漏洩を行う事ができる。

■ユーザ空間からの漏洩

非特権攻撃者が、同時実行中の**他のユーザ空間アプリケーション**によって**ロードやストアされた値**を漏洩させる**プロセス間攻撃**に悪用可能。攻撃者は攻撃対象の**シブリングスレッド**で動作する

■カーネル空間からの漏洩

非特権攻撃者は、**カーネル空間**で実行された**ロードやストアで使われた値**も漏洩可能。攻撃者はシブリングスレッドだけでなく、カーネルからユーザへの**コンテキストスイッチ**時に、攻撃対象と**同一論理コア**上で攻撃を行う事もできる。



■ Intel SGXからの漏洩

SGXのEnclave内部で実行されたロードとストアから漏洩させる事も可能。勿論、**Enclave内データを対象としていても同様**。
攻撃者は攻撃対象 (Enclave) の**シブリングスレッド**で動作する。
SGXの脅威モデルの性質上、**特権攻撃者**を前提とする。

■ 仮想マシンからの漏洩

VMの境界を超えて、**他のVM**のロードやストアから漏洩させる事もできる。攻撃者VMは攻撃対象VMの**シブリングスレッド**で動作する。
攻撃者は**信頼不可能な仮想マシンの内部で実行**しているため、**特権攻撃** (ゲストページのPTE変更等) を行う事が可能。

※PTE : ページテーブルエントリ (Page Table Entry)



■ハイパーバイザからの漏洩

VM内の攻撃者が、**ハイパーバイザ**がロードやストアする値を漏洩させる事もできる。

こちらにも**信頼不可能なVM内で実行**しているため、**特権コードの実行**を制限されない。



- ZombieLoadには、具体的な攻撃アプローチとして**5つのバリエーション**（変種）が存在し、それぞれ**ゾンビロードを誘発するための方法が異なっている**
 - 論文中の実践的な攻撃実験でよく使われているのはバリエーション1～3



- あるユーザ空間の**仮想メモリページ v** がアーキテクチャ的に正しく**物理アドレス p** を指している状態であるとする
- この時、**カーネルページ**または**アクセス不能な仮想メモリページ k** を用意し、 **k も物理ページ p を指す**ように仕向ける



- この状態で v からキャッシュをクリアしながら、ユーザ空間から k にアクセスするとマイクロコードアシストが発生し、過渡的にLFBから直近のロードやストアで使用された値が漏洩する**ゾンビロード**が発生する
 - LFBからの漏洩を確実にするため、**キャッシュクリアとアクセス（ロード）は同時に行う必要がある** [3][4]
 - ただし、このバリエーション1は**Meltdown耐性を有するマシンでは無力**である
- キャッシュのインデックス付けは（大抵は）**物理アドレスを基準に行われる** [2]ため、 v のキャッシュフラッシュは k のキャッシュフラッシュと**同義**である

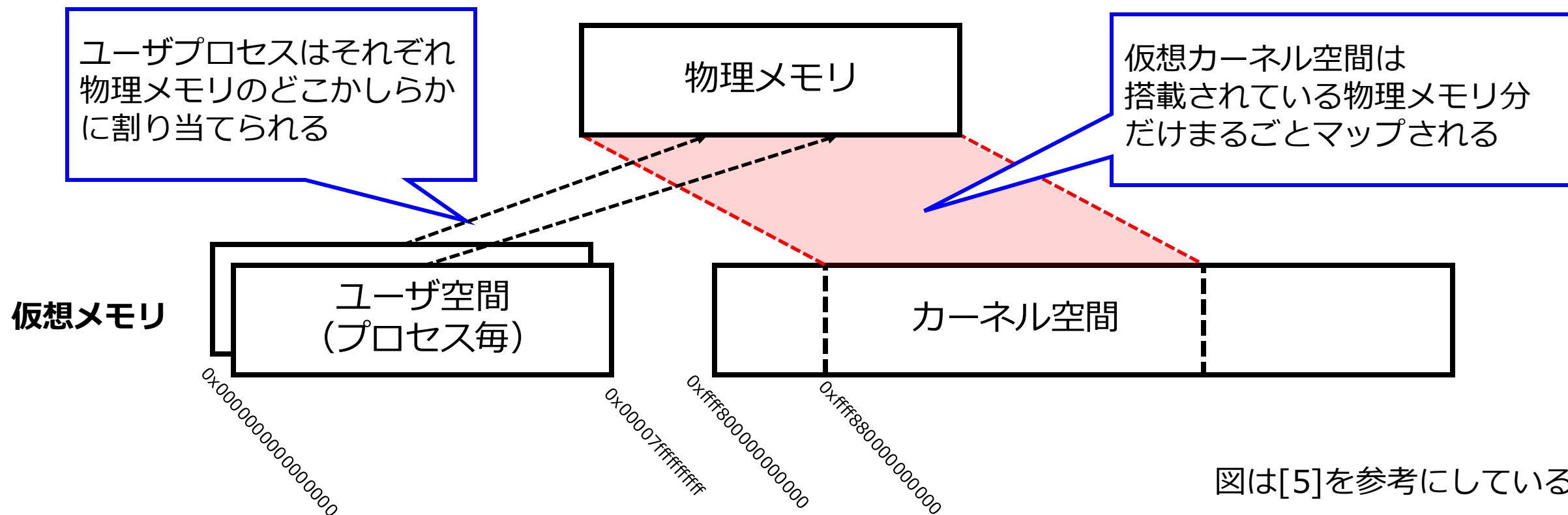


- カーネルは、**カーネルの仮想メモリが特定の仮想アドレス（ベースアドレス）から始まるようにし、かつ搭載メモリ分だけ物理メモリに直接マップする**（Direct Physical Map; ダイレクト物理マップ）
- このカーネルの**ベースアドレスを基準**とし、物理アドレス p を**オフセット的に使用する事で、仮想カーネルアドレスである k を攻撃者が取得**する事ができる
- 物理アドレス p は、**`/proc/pageinfo`**から取得する（要特権）、**PTEditor**を用いる（非特権で可能[4]）、別の**サイドチャネル攻撃で奪取**する等により入手する事ができる

ZombieLoad v1 – カーネルマッピング (4/5)



- ユーザ空間メモリは**プロセスごと**に**物理メモリにマップ**される一方で、カーネルメモリは**物理メモリ全体に丸ごとマップ**されるため、ユーザページとカーネルページは**重複**し得る
 - PTEのU/Sビット等により、アクセス違反が発生しないよう管理している

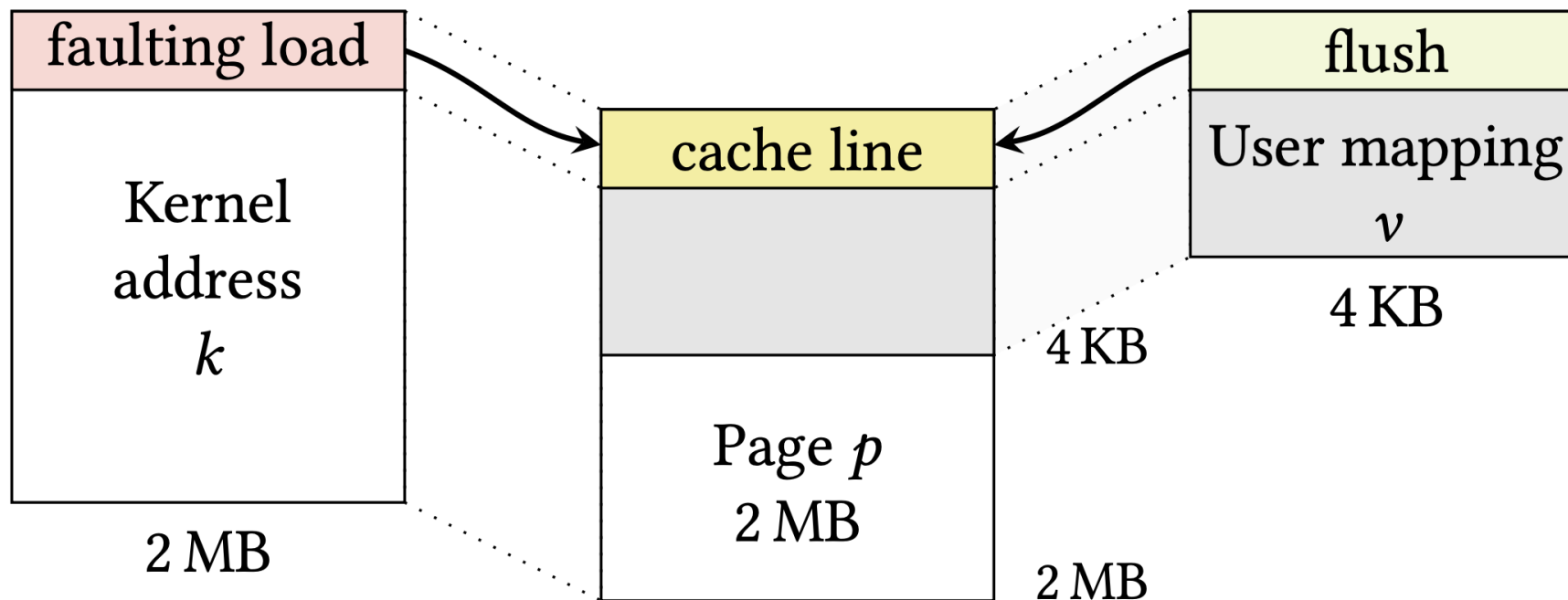


図は[5]を参考になっている

ZombieLoad v1 – カーネルマッピング (5/5)



- カーネルは、通常2MBの巨大なページでマッピングされるため、 v に対応する物理ページを、 k に対応する巨大な物理ページが覆い被さるようなイメージとなる（図は[1]より引用）





- ある**仮想アドレス v** を通してユーザがアクセスできる**物理アドレス p** が存在するとする
 - ユーザ空間に割り当てられた任意のページがこの要件を満たすため、前提とするまでもない**普遍的な状態**である
- その状態で、シブリング論理コア等からTSXトランザクション内の読み取りセットに**競合を発生**させ、その上でトランザクション内で**正当なロードを実行**させ、**TSXアボート**を誘発する
 - 競合の誘発は、前述の通りトランザクションに使用するデータの変更等によって行う事ができる



- TSXアボートにより、TSXは**フォールト**し結果がコミットされなくなるが、このフォールトはアーキテクチャ的なフォールトではなく、**ゾンビロード**につながる**過渡的なフォールト**である
- このバリエーション2は、**Meltdown耐性**のあるマシンや、論文執筆当時**MDS耐性**があるとされていたマシンでも動作する点が明確な強みである
 - ただし、言うまでもなくマシンが**TSXをサポート**している必要がある

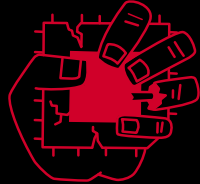


- ある1つの**物理ページ p** を同時に指す、**2つの別々の仮想アドレス v, v_2** が存在するとする
 - 共有メモリやメモリマップトファイルのように、2つの別々のプロセスから同じ位置のデータ（物理ページ）にアクセスする状況を考えると分かりやすい
- v からは p にアーキテクチャ的に**正当にアクセスできる**とする
- 一方、 v_2 についてはPTEの**Accessedビット**（Aビット）を**0**にする
 - **Aビット**：そのページが使用された場合に1になるPTE上のビット
 - **Linux**ではAビットのクリアには**特権**が必要だが、**Windows**では**定期的にこれをクリアする**事が確認されている



- 対応するPTEの**Aビットが0**の状態で**ページにアクセス**すると、Aビットを立てるために**マイクロコードアシスト**が発行される
- 前述の通り、LFBからの漏洩を確実にするには**キャッシュがフラッシュされた状態を維持**しなければならないため、 v で**キャッシュフラッシュ**するのと**同時**に v_2 へ**アクセス**する
 - 前述の通り、キャッシュは**物理アドレスを基準にインデックス付け**されるという点に注意。 v のフラッシュは v_2 のフラッシュと同義
- この状態で v_2 にアクセスを行う事で、**マイクロコードアシストを誘発**し、かつキャッシュが確実にクリアされているのでLFBからの**ゾンビロード**を誘発する事ができる

ZombieLoad v1～3を実行できる環境



- バリエーション1～3がそれぞれどのような環境と権限で実行可能であるかをまとめた表を以下に示す（[1]より引用）：
 - 黒丸**：実行可能、**白黒半分ずつの丸**：特定のHW構成で可能、**白丸**：実行不可

Scenario \ Variant	1		2		3	
						
Unprivileged Attacker	○	◐	◐	◐	●	○
Privileged Attacker (root)	●	●	◐	◐	●	●



- **Enclave内**のメモリに対して**外からアクセス**すると、**読み出し値は0xff**となり、**書き込みは無効化**される
 - この挙動を**アボートページセマンティクス (APS)** という
- 具体的には、非EnclaveモードでEnclaveメモリにアクセスすると、アドレス変換結果が上記のような無効化挙動を取るアボートページに置換されるが、この置換処理は**マイクロコードアシスト**が行う
- ここで、**Enclaveの物理アドレス p** にマッピングされた**仮想アドレス v** が存在するとする



- Enclave外から v にアクセスすると**APSが発動**するが、APSの適用前に、直前に（シブリングスレッドが）ロードまたはストアした値である**LFBエントリ**を**過渡的に漏洩**させる**ゾンビロード**が発生する
- 論文における攻撃の動作確認実験では、**TSXトランザクション内で**前述のような v にアクセスして**ゾンビロードを発動**させている
 - Foreshadowでも使われている手法だが、TSXトランザクション内で実行すると、OSのフォールト処理に伴う、キャッシュやその中間経路であるLFBの**汚染を抑制**する事ができる



- このバリエーション4では、 p のキャッシュを**フラッシュ**する代わりに、TSXトランザクションを行う前に**単にアクセスするだけ**で良い事も確認された
 - APSの処理が**キャッシュ階層内で完結**しない、つまりどのような場合でも**LFBを通る**仕様であると予想されるため、キャッシュクリアしなくてもLFBから漏洩するのであると考えられる
- **マイクロコードアシストの誘発にSGXのAPSを用いているだけ**であるため、攻撃対象のLFBエントリは**Enclave由来のものでなくとも構わない**という点に注意



- バリエーション4ではSGXのEnclaveメモリを用いていたが、代わりに**キャッシュ不可メモリ**[6]にアクセスする事でも同様にして**ゾンビロード**を誘発可能
- キャッシュ不可であるようなページにアクセスすると、**ページミスハンドラ**が**マイクロコードアシスト**を**発動させる**挙動を利用している
 - **ページミスハンドラ**：TLB（PTEのキャッシュのようなもの）に対象の仮想アドレスが存在しない場合に、ページテーブルを参照して解決を図るページウォークを行う機構[7]

パフォーマンス評価 (1/2)



- 様々なCPUのマシンにおける、バリエーション1～3の動作状況。
“✓”は動作する事、“✗”は動作しない事、“-”はTSXがそのCPUでは無効化されているかサポートされていない事を示す。
(図は[1]より引用)

Setup	CPU (Stepping)	μ -arch.	Variant		
			1	2	3
Lab	Core i7-3630QM (E1)	Ivy Bridge	✓	-	✓
Lab	Core i7-6700K (R0)	Skylake-S	✓	✓	✓
Lab	Core i5-7300U (H0)	Kaby Lake	✓	✓	✓
Lab	Core i7-7700 (B0)	Kaby Lake	✓	✓	✓
Lab	Core i7-8650U (Y0)	Kaby Lake-R	✓	✓	✓
Lab	Core i7-8565U (W0)	Whiskey Lake	✗	-	✗
Lab	Core i7-8700K (U0)	Coffee Lake-S	✓	✓	✓
Lab	Core i9-9900K (P0)	Coffee Lake-R	✗	✓	✗
Lab	Xeon E5-1630 v4 (R0)	Broadwell-EP	✓	✓	✓
Cloud	Xeon E5-2670 (C2)	Sandy Bridge-EP	✓	-	✓
Cloud	Xeon Gold 5120 (M0)	Skylake-SP	✓	✓	✓
Cloud	Xeon Platinum 8175M (H0)	Skylake-SP	✓	-	✓
Cloud	Xeon Gold 5218 (B1)	Cascade Lake-SP	✗	✓	✗

パフォーマンス評価 (2/2)



- バリエーション1～3について、**Core i7-8650U**のマシンにおいて漏洩（伝送）速度を計測する実験を行っている
- **v1**の場合、v4のようにTSXで例外を抑制しながら実施した所、平均伝送速度は**5.30kB/s**、真陽性率**85.74%**であった
- **v2**の場合、平均伝送速度は**39.66kB/s**、真陽性率は**99.99%**だった
- **v3**は、**TSX無し**では平均伝送速度**0.08kB/s**で**52.7%**の真陽性率、**TSX有り**の場合平均伝送速度**7.73kB/s**で真陽性率**76.28%**であった
 - 論文中では、TSX無しの方は「シグナルハンドラと併用」と書いてあるが、これはTSX未使用なのでOSの例外ハンドラが発動する事を指していると考えられる

SGXシーリング鍵の抽出 (1/10)



- **特権攻撃者**が、Enclave内で実行される**sgx_get_key**関数から、この関数で生成された**鍵**を**ZombieLoad**で**漏洩**させるシナリオについて考える
 - sgx_get_keyは、主にEnclave内で使用される各種の重要な鍵を導出する**EGETKEY**というCPU命令のラッパー関数である
- sgx_get_keyからの漏洩レートを測るベンチマーク用Enclaveからの漏洩の他、**自己署名QE**からの**PSKの漏洩**についても行っている
 - PSKについては[こちら](#)を参照
- 特権攻撃が可能であるため、ForeshadowやLVIよろしく**SGX-Step**を利用しながら攻撃を進める
 - 1命令ずつ厳密に割り込みながらの攻撃が可能となる

SGXシーリング鍵の抽出 (2/10)



- **機密情報が格納されているような状態でCPU状態を固定し、それに対して攻撃を繰り返して逐次秘密情報を抽出**したい場合がある
 - ZombieLoadやForeshadowのように、1バイトずつしか漏洩させられない攻撃でCPUレジスタ状態から16バイトの鍵を抽出する、といった場合に必要
- 攻撃を実施するコード部分に対応する**ページの実行権限を剥奪**すると、同じコード（命令）部分で**AEXとERESUME**が繰り返される**ゼロステップ処理**が発生する
- ゼロステップ処理中は**Enclave実行が一切先に進まない**ため、延々と**同一のCPUレジスタ状態**がSSAにストア/SSAからロードされるのが繰り返され、結果**CPU状態が固定**される
 - AEX、ERESUME、SSAの説明は[こちら](#)を参照

SGXシーリング鍵の抽出 (3/10)

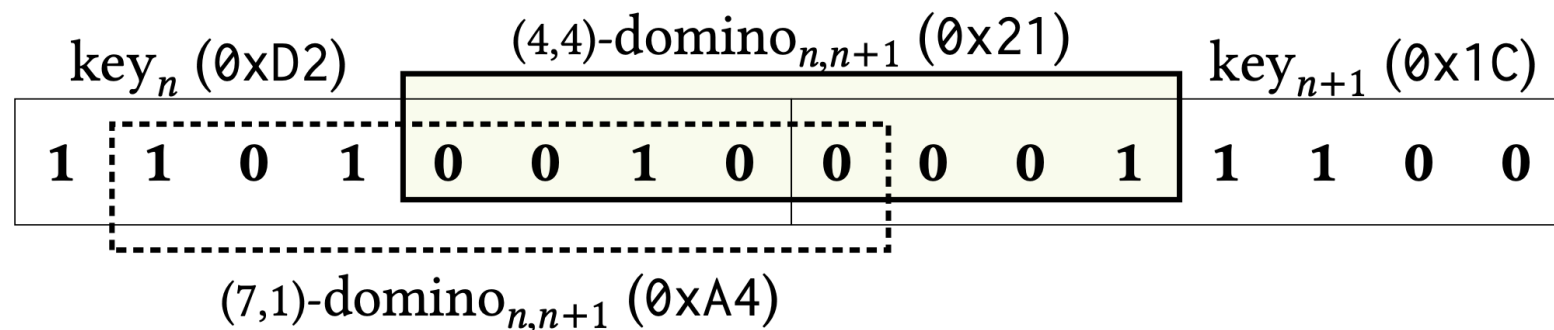


- 攻撃対象で使用する**キャッシュへのヒット**を**観測**した瞬間にZombieLoadを開始する等により、**興味のない処理**から漏洩する**ノイズ**を**大幅に制限**できる
- しかし、それでも攻撃の裏でOSやプロセス等が行う**無関係なロード**からの値が漏洩してくる事もあり、これ自体を**抑制**しきる事は**不可能**に近い
- そこで、漏洩させたい鍵バイトだけを観測するだけでなく、**隣り合う鍵バイトのそれぞれ一部分を内包**するような**ドミノバイト**を観測する手法を実施する

SGXシーリング鍵の抽出 (4/10)



- 以下の図のように、隣接する鍵それぞれ（白実線矩形部）を漏洩させると共に、それらのそれぞれ一部を含むドミノバイト（薄緑矩形部）の漏洩と抽出も行う（図は[1]より引用）



- 各鍵バイト位置で攻撃を繰り返し、最も漏洩の多いバイトが正しいと仮定するだけでなく、その候補にドミノバイトの内容が一致して初めて正解であると判断するようにする
 - 隣接する2つの鍵バイトと、それにまたがるバイトであるドミノバイトとの整合性が取れていれば、そこに誤りは無いだろうというロジック

SGXシーリング鍵の抽出 (5/10)



- ZombieLoadでは**LFBエントリ内の位置をバイトレベルで指定して漏洩させる事しか出来ないため、単なる鍵バイトの観測だけでは**
ドミノバイトを漏洩させる事は出来ない
 - 例えば、**連続するLFBエントリのそれぞれ後ろ4ビットと前4ビットずつの漏洩、というビットレベルの指定は出来ない**
- そのため、過渡的実行により**AES鍵全体の内どの部分の漏洩も可能**であり、かつそれを過渡的に任意の処理に利用できるという前提が必要
 - この条件を満たせば、ZombieLoad発動後の**過渡的ドメインのガジェット**において、AES鍵全体から**ドミノバイトを構築**する事ができる

SGXシーリング鍵の抽出 (6/10)



- n バイト目の鍵バイトと $n + 1$ バイト目の隣接する鍵バイトについて、例えばそれぞれの4バイトずつからなるドミノバイトがある時、これを(4, 4)-ドミノバイトと書く事にする
- `sgx_get_key`に対する攻撃では、(7, 1)から(1, 7)までの全てのパターンのドミノバイトを漏洩させる事で、さらにノイズが混じらないように確度を高めている
 - 言うまでもなく、(8, 0)や(0, 8)は単一鍵バイト内で完結しているのでドミノバイトではない
- 上記のような全パターンのドミノバイト一式をスライディングウィンドウと呼んでいる

SGXシーリング鍵の抽出 (7/10)



- `sgx_get_key`は、内部で独自の`intel_fast_memcpy`という関数を呼び出し、**ベクトルレジスタ**である`xmm0`を経由させる形で、**128bit単位のmove命令でコピー**している事が判明した
 - **ベクトルレジスタ**：SIMD命令等でよく使われる、そのアーキテクチャのビット数よりも大きいようなレジスタ。`xmm`レジスタは128bit長である
 - 余談だが、このベクトルレジスタからの過渡的漏洩が**Downfall**として数年後の2023年に報告されている
- よって、この独自の`memcpy`を構成する機械語レベルでの**最後の命令でゼロステップ処理**を行い、`xmm0`にSGX鍵を内包しているようなCPU状態を繰り返しSSAからロードする状況を作る

SGXシーリング鍵の抽出 (8/10)



- この状態でZombieLoadにより鍵を漏洩させる攻撃を、まずは前述の**ベンチマーク用Enclave**に対しCore i7-7700のマシン上で実行した
 - レポートキーを漏洩させ正しいかをチェックする事で成功確率を検証している
- 結果、100回中30回、つまり**30%の確率**で鍵候補の中に**全鍵バイトが内包**されている状態が確認された
 - 実際に**完全な鍵を回復**できたのはその内**3%**の試行だけであり、割合としてはかなり低いと言わざるを得ない
- 残り70%では全鍵バイトは観測できず、内31%では**平均して10バイト**含まれており、残り39%では**一切内包されていなかった**

SGXシーリング鍵の抽出 (9/10)



- 次に、**実験用QE**から**PSKを漏洩**させる事を試みている
 - QEは本来ビルド・署名済みのイメージが直接インストールされてそれを用いるが、実験ではLinux-SGXのSGXPSWで公開されているQEのコードからビルドしデバッグ用の鍵で署名した、**実験用のQEに対して攻撃**している
- 結果として、実際にPSKを漏洩させ、実験用にPSKでシーリングしストアしたAttestationキーをアンシーリングし取得する事に成功した
 - 恐らくこのAttestationキーは正規のプロビジョニング手続きを経てデプロイされたものではなく、実験用にダミー的に用意した値

SGXシーリング鍵の抽出 (10/10)



- **PSKはMRSIGNERポリシ、つまりEnclave署名鍵に依存する**
シーリング鍵であるため、この**実験用QE**から漏洩したPSKでは、
Intel公式の**PvE**（QEと署名鍵が同じ）で**正規のPSK**により
シーリングされた**Attestation**キーは**アンシーリング出来ない**
- 論文中ではIntelが署名した**公式QE**からの漏洩は行っていないが、
もしできてしまうと漏洩したPSKで**正規のAttestation**キーが**暴かれ**、
RAのセキュリティ保証が崩壊してしまう

VM間秘密チャネル (1/7)



- **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VMの秘密情報**を漏洩させて**攻撃者のVMに転送する秘密チャネル**を、**ZombieLoad**によって**実現し攻撃を実行**する
 - プロセス間、カーネル、SGX、ハイパーバイザといった他のシナリオでも可能ではあるが、ここではVM間攻撃を考える
- この二者のVMはシブリングスレッド上にそれぞれ存在するとする

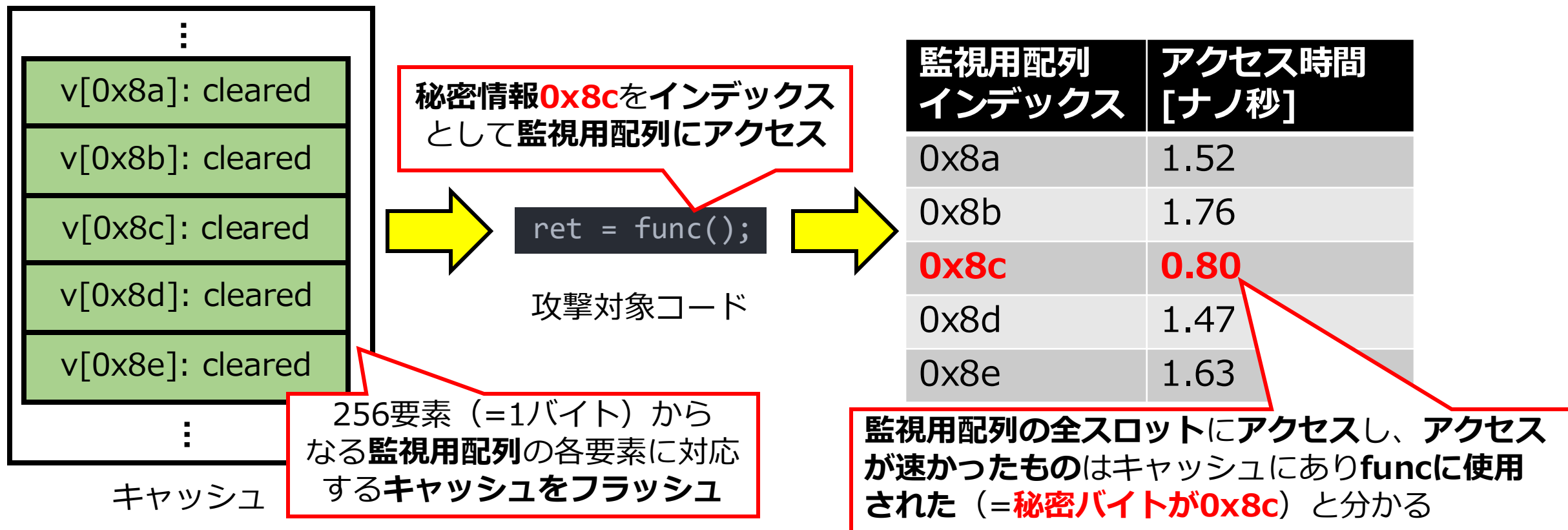


- 送信側 (**攻撃対象VM**) では、漏洩させたい値を複数のメモリアドレスに展開し、それぞれから繰り返しロードする
 - これにより、複数のLFBエントリ (CPUモデルによってエントリ数は**10か12個**) に漏洩させたい値が持ち込まれる確率が上がり、**ZombieLoad**により漏洩する確率も上がる
- 受信側 (**攻撃側VM**) では**ZombieLoadを実行**し、送信側がロードした値を漏洩させる。漏洩させた値は監視用配列の**キャッシュに痕跡を残し、FLUSH+RELOAD**で過渡的実行終了後に**観測**する
 - Foreshadow等と同様、1バイトずつの漏洩攻撃になる
 - これもForeshadow同様、キャッシュラインプリフェッチャの誤爆の影響を阻止するため、**監視用配列の各スロットはページサイズ間隔**で配置する
 - スロット数は後述の理由から $256^2 \times 4096$

(復習) FLUSH+RELOAD攻撃



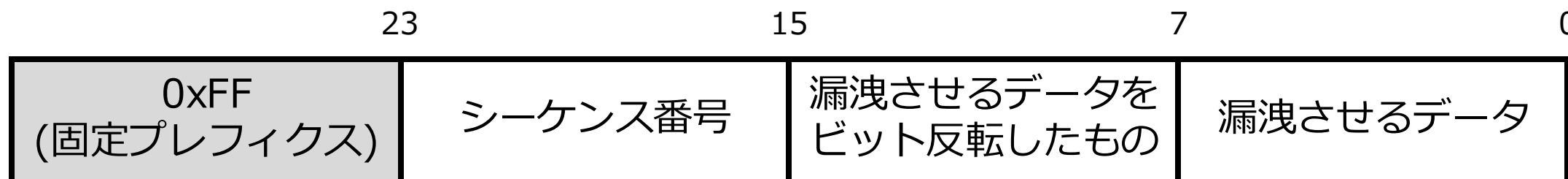
- **FLUSH+RELOAD** : キャッシュラインをフラッシュ（クリア）し、攻撃対象に何らかの活動させた後、**再度アクセス**する攻撃
 - 再アクセス時にアクセス時間が短ければ、そのキャッシュ**値**を**攻撃対象が使用**したという事がわかる（データ自体の推測）



VM間秘密チャネル (3/7)



- 例に漏れずZombieLoadは**無関係な処理**に由来する**ノイズ**も**漏洩させる**ため、以下の図に示すような**32bitのパケット**を漏洩・転送させる事でそのような**ノイズの排除**を試みる



- 監視用配列は**1バイトずつの漏洩**のみを行えるため、**パケット全体**を命令リタイア後に観測する事は**出来ない**が、**過渡的領域でエラー検出**を行い、**有効なバイトデータ** (0x00yy) と**シーケンス番号** (0xFFzz) のみを**キャッシュに残す**事が出来る
 - その後、FLUSH+RELOADで観測する

VM間秘密チャネル (4/7)



- 送信側は、前図の右から**1バイト目**には**漏洩させるデータ**を、**2バイト目**には**漏洩させるデータをビット反転させたもの**を格納する
- 受信側はこのパケットを**ZombieLoad**で**漏洩**させ、**2バイト目**に対し**1バイト目**を**XNOR演算**してそのまま**2バイト目を上書きする**
 - 元データとビット反転したデータのXNORであるため、誤りがなければ2バイト目は必ず**0b00000000**となる

VM間秘密チャネル (5/7)



- この2バイト目と1バイト目からなる**16bitの値**を**インデックス**として以下のように**監視用配列にアクセス**する

```
uint8_t value = oracle[leaked_data * 0x1000];
```

- もし誤りが無い場合、前述の通り16bitの内**上位8bitがオールゼロ**になるため、上記の過渡的アクセスは必ず**監視用配列の256スロットのいずれかにアクセス**する
- 一方、誤りがあると**上位8bit**のいくつかのビットが**1になる**ため、`leaked_data * 0x1000`は言わば257スロット目以降の**境界外へのアクセス**を行う
 - 境界外は後続のFLUSH+RELOADでの探知を行わないため、結果的に**誤りが発生した場合は完全に無視される**形になる

VM間秘密チャネル (6/7)



- 後は、受信側で**FLUSH+RELOAD**により監視用配列のキャッシュから目当ての**秘密バイトを観測**するだけである
- パケットの前図右から3バイト目にはバイトの**シーケンス番号**を格納しているため、これを参照する事で**漏洩させたデータを順番に並べ替える事**も出来る
 - この順番通りの並べかえは、AES鍵のように**順序が重要なデータ**を漏洩させる場合に**有用**である
- シーケンス番号はプレフィックスの0xFFと併せて0x**FF**yy、秘密バイトは誤りが無ければ0x**00**zzの形で監視用配列のインデックスとなるため、**観測時相互に識別可能**である
 - 誤りがある場合は0x00nnに対するFLUSH+RELOADでアクセス時間が**全要素均一**となる事で検知でき、その場合は**シーケンス番号自体不要**である



- 論文では、Core i7-8650Uを搭載するローカルマシン上のQEMU KVMで動作する2つのVM間での転送と、あるパブリッククラウドの同一ベアメタル上のVM間での漏洩と転送を行う実験を実施している
 - どのパブリッククラウドであるかは当該クラウド事業者により口封じされているらしい
- 前者のシナリオでは、TSXによる例外抑制を用いながらのZombieLoad v1を用いて最大26.8kbpsの伝送レートを記録している
- クラウドのシナリオでは、そのクラウドでTSXを使用できなかったため、TSX無しのZombieLoad v1で最大1.99kbpsの伝送レートを達成している

Downfall



- **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)

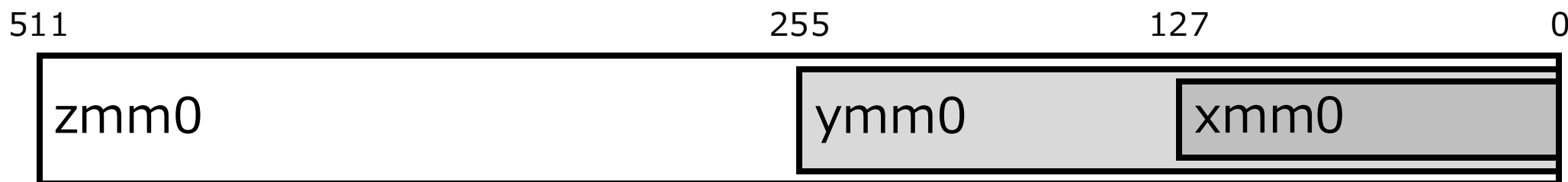


- 64ビットレジスタにおけるraxのように、ベクトルレジスタにもレジスタ名が付与されている
- 128bitレジスタは xmm_n 、256bitレジスタは ymm_n 、512bitレジスタは zmm_n という命名規則になっている
 - 各末尾の n はレジスタのインデックス番号（例： xmm_0 ）

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





- 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のベクトルレジスタ**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)



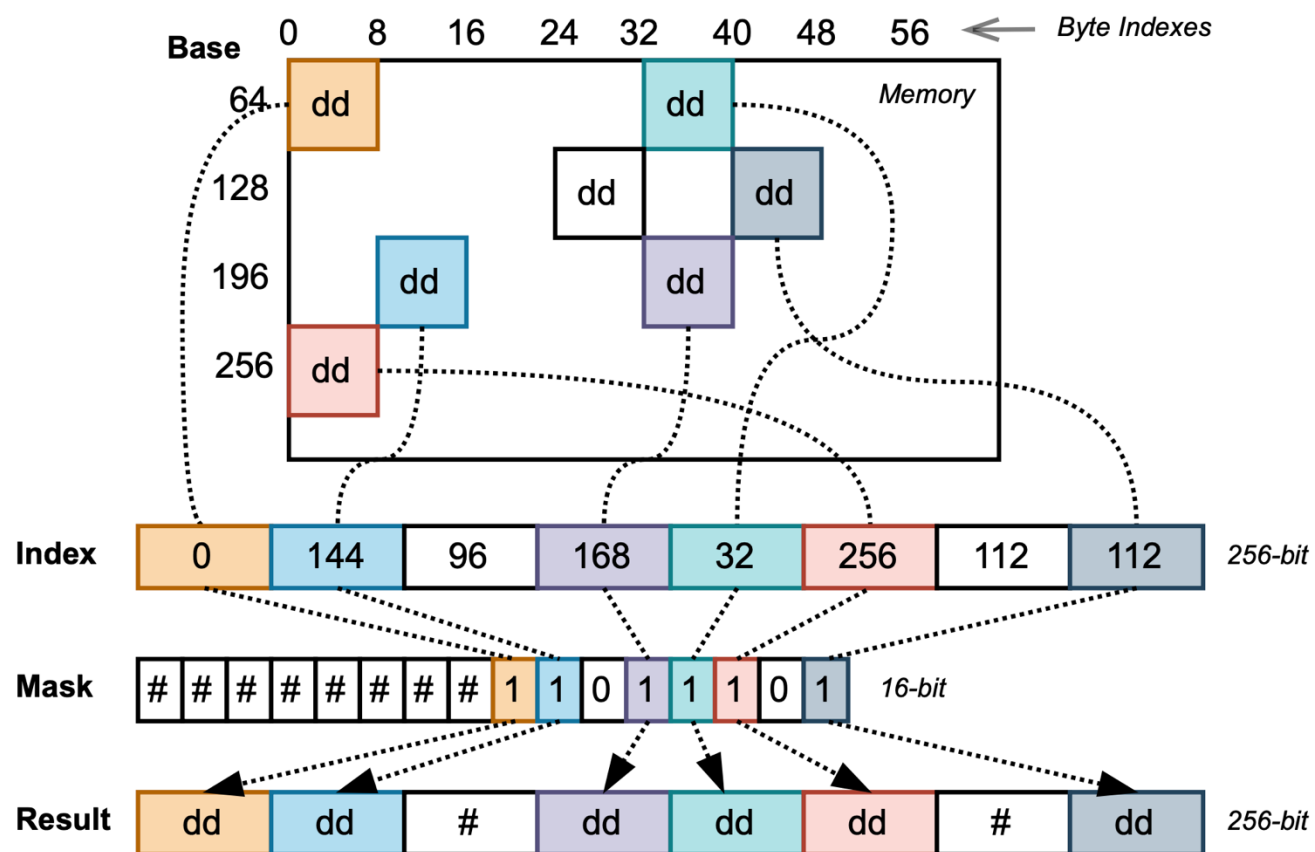
- このGather命令でメモリの各所に散らばったデータを効率的に**収集**して**ベクトルレジスタにロード**した上で、**他のSIMD命令**を実行すると**効率的にSIMD処理を行う事ができる**
- AVX-512の場合、マスク配列専用の**マスクレジスタ** k_n が用意されており、以下のようにGather命令を記述する事ができる
 - rsiがベースアドレス、zmm2がインデックス配列、k1がマスクレジスタ、zmm3が格納先ワイドレジスタ

```
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なになる
(図は[8]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

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

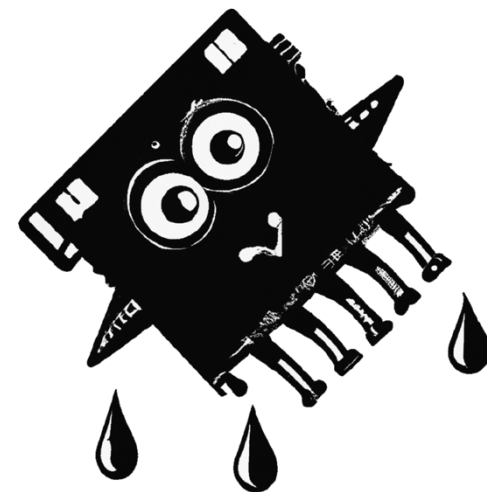


- 複数回読み出し時の**キャッシュライン保持**や、割り込み発生時の再開のための**部分結果保持**には、**CPUパッケージ内の何らかのバッファを使用**しているのでは？
 - Foreshadow (L1D) 、ZombieLoad (LFB) 、Fallout (SB) 、LVI (L1D、LFB、SB、LP、FPU) のような漏洩が発生するのでは？
- **投機的に実行**して駄目ならアーキテクチャ状態 (CPUやメモリの実際の状態) への反映時 (**命令リタイア時**) に**破棄**、という挙動は、**過渡的領域**において**何かしらの脆弱性**を抱えているのでは？
 - 過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[8]





- 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



- 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
```

-
- The diagram illustrates the execution flow of a Gather command and the resulting security issue due to a transitionary execution state. It is divided into two main sections: **擬似コード** (Pseudo-code) and **実行フロー** (Execution flow).
- 擬似コード (Pseudo-code):**
- ```
gather(base_addr, zmm5); //zmm5にGather
oracle[zmm5[0]]; //監視用配列にアクセス
flush_and_reload(); //FLUSH+RELOAD
```
- 実行フロー (Execution flow):**
- base\_addr (メインメモリ)** (base\_addr (Main memory)) points to **values**.
  - zmm5 (ベクトルレジスタ)** (Vector register) contains **secret**.
  - The **Gather命令** (Gather instruction) is executed. A red 'X' indicates an **例外発生・過渡的执行** (Exception occurrence / Transitory execution) state.
  - This leads to **過渡的Gather命令** (Transitory Gather instruction), which then leads to **過渡的監視用配列アクセス** (Transitory monitoring array access).
  - The **監視用配列アクセス** (Monitoring array access) leads to **FLUSH+RELOAD**.
  - The **FLUSH+RELOAD** state leads to **命令リタイア後** (After instruction retirement).
- 説明 (Explanation):**
- ①例外/遅延発生** (Exception/delay occurrence): The Gather command is interrupted, leading to a transitory state.
  - ②Gatherの過渡的执行は、zmm5内の古い値secretをそのまま後続の命令に転送する** (Transitory execution of Gather transfers the old value secret from zmm5 to the subsequent command as is): The secret value is transferred to the monitoring array access.
  - ③FLUSH+RELOADにより、監視用配列経由で値secretが観測できてしまう** (Due to FLUSH+RELOAD, the value secret can be observed via the monitoring array): The secret value is observed through the monitoring array.
- 補足 (Additional information):**
- これらのルートなら base\_addr由来の無害な値 valuesが観測されるだけである (In these routes, only harmless values values derived from base\_addr are observed).



## ■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）



## ■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dword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0x**8c34c592**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。



## ■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**することができる (**RELOAD**)。



- 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



- 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**



- 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



- 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



- 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



- 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で使用される値が、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：

|                             |                       |                     |
|-----------------------------|-----------------------|---------------------|
| <b>Instruction buckets:</b> | (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、\*部分は扱うデータの型によるバリエーション（接尾辞）



## ■ 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。



## ■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。



## ■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[11]。

## ■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。



- 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**

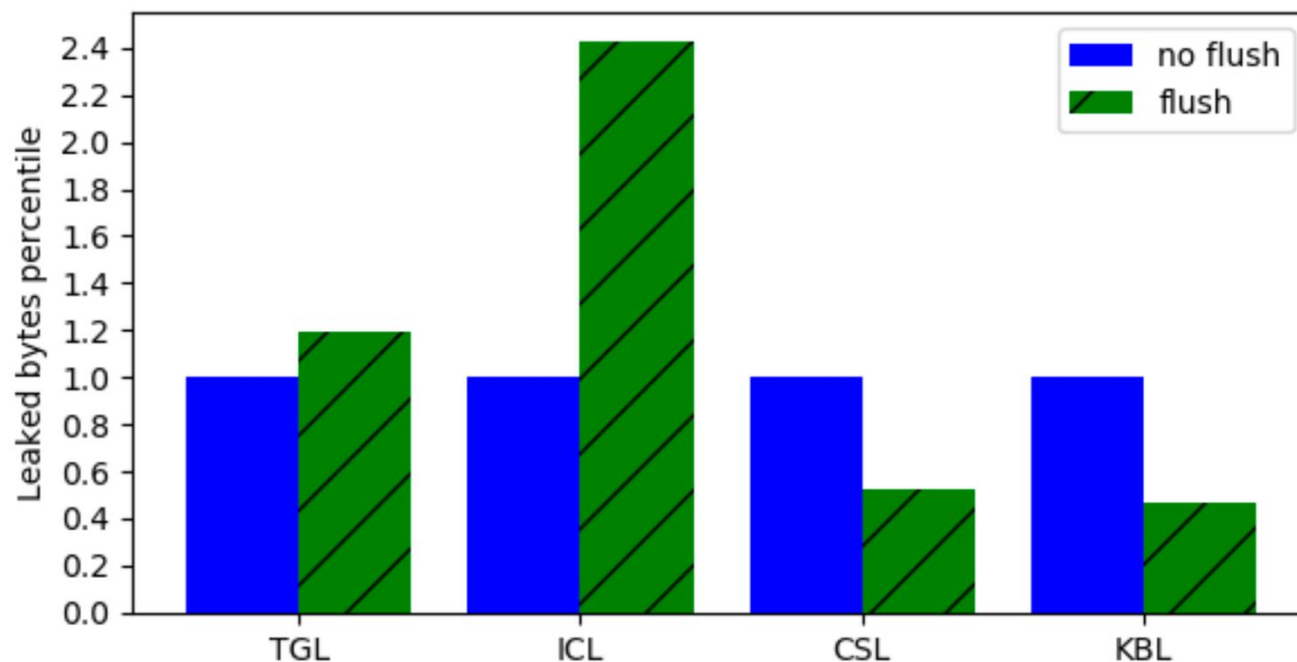


- ここで、**Foreshadow**や**MDS**で漏洩元となっていた**L1Dキャッシュ**や**マイクロアーキテクチャバッファ**が**GDSの漏洩元ではない**事を確定させておく必要がある
- そこで、**VERW命令**により**マイクロアーキテクチャバッファ**を**フラッシュ**し、**MSR**（モデル固有レジスタ）経由で**L1Dキャッシュ**を**フラッシュ**してみる
  - VERW命令は本来全く関係ないあまり使われない命令であったが、MDSの発見以降μ-Archバッファをフラッシュする副次的機能を付与されている

# 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 $\mu$ -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[8]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、 $\mu$ -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 Cycles |
|----------------|-----|--------|-----|--------|------|-------|-------------|
|                | SMT | Switch | SMT | Switch | >TAA | >MMIO |             |
| Tiger Lake     | ⌘   | ⌘      | ⊖   | ⊖      | ⊖    | ⊖     | 80          |
| Ice Lake       | ⌘   | ⌘      | ⊖   | ⊖      | ⊖    | △     | 592         |
| Cascade Lake   | ⌘   | ⌘      | ⊖   | ⊖      | ⌘    | △     | 324         |
| Kaby Lake      | ⌘   | ⌘      | ⌘   | △      | ⌘    | △     | 696         |

⌘ Vulnerable    ⊖ Not affected    ⌘ TSX disabled    △ Buffer flush

図は[8]より引用



- 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[9]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



- 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



- **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



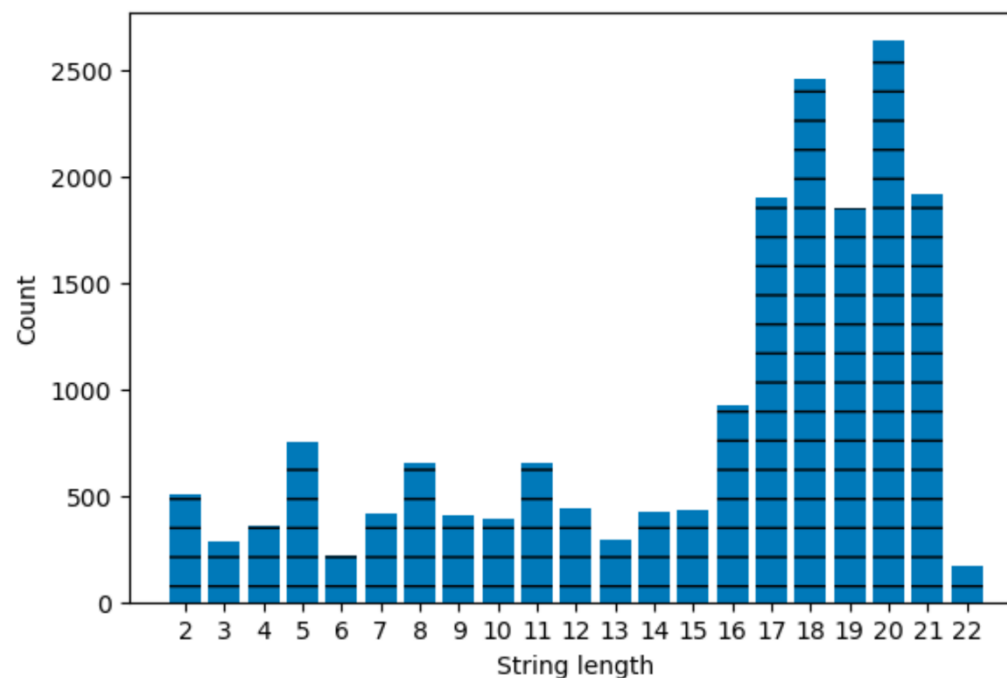
- 過渡的実行攻撃においては、**秘密チャネル**として主に前述の通り**256スロットの監視用配列のキャッシュ**を利用する事が多い
  - あるバイトについて、それをインデックスとして監視用配列に過渡的にアクセスし、FLUSH+RELOADで後から検知する
- **Meltdown**や**Foreshadow**では256スロットの監視用配列を**1つ**用意して**1バイト**ずつ、**ZombieLoad**では256スロット監視用配列を**3つ**用意して**3バイト**ずつ**漏洩値の観測**を行っている
- 一方、GDSでは**32個の256スロット監視用配列**を用意し、64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



- 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



- 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[8]より引用

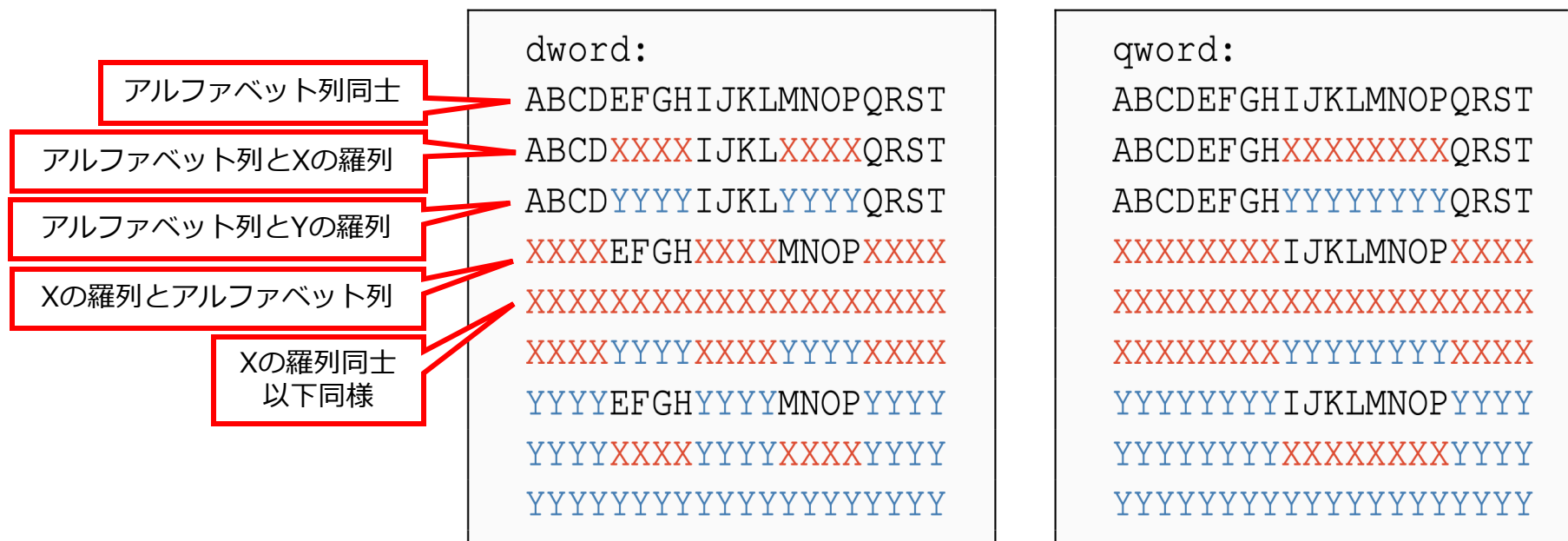


- 同時漏洩数の次は、漏洩するデータの**データパターン**について実験的に確認する
- **アルファベット列** (A~T) と**Xの羅列**、そして**Yの羅列**の3つの羅列を用意し、**同一または他の羅列と同時に**vmov命令で**ロード**し、そこから**GDSを用いて抽出する**事を試みる
  - dwordを収集するGatherとqwordを収集するGatherの両方について実験を行う

# 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ですが、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[8]より引用



- 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



- 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4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



- ベンチマークの結果は以下の図の通り（図は[8]より引用）：

| CPU Generation | vmov    |         |         | rep mov |         |         | fxrstor |         |         | aes    |         |        |
|----------------|---------|---------|---------|---------|---------|---------|---------|---------|---------|--------|---------|--------|
|                | ●       | U       | ×       | ●       | U       | ×       | ●       | U       | ×       | ●      | U       | ×      |
| 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    U 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**



- **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 $\%rcx * \text{sizeof}(t)$ 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( $\%rdi, \%rsi, \%rcx * \text{sizeof}(t)$ )**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[12]が示す研究において確認されている



- 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);
```



## ■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



## ■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



## ■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いoctlを通じて受け付ける、**ロード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



- 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



- 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[13][14]。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

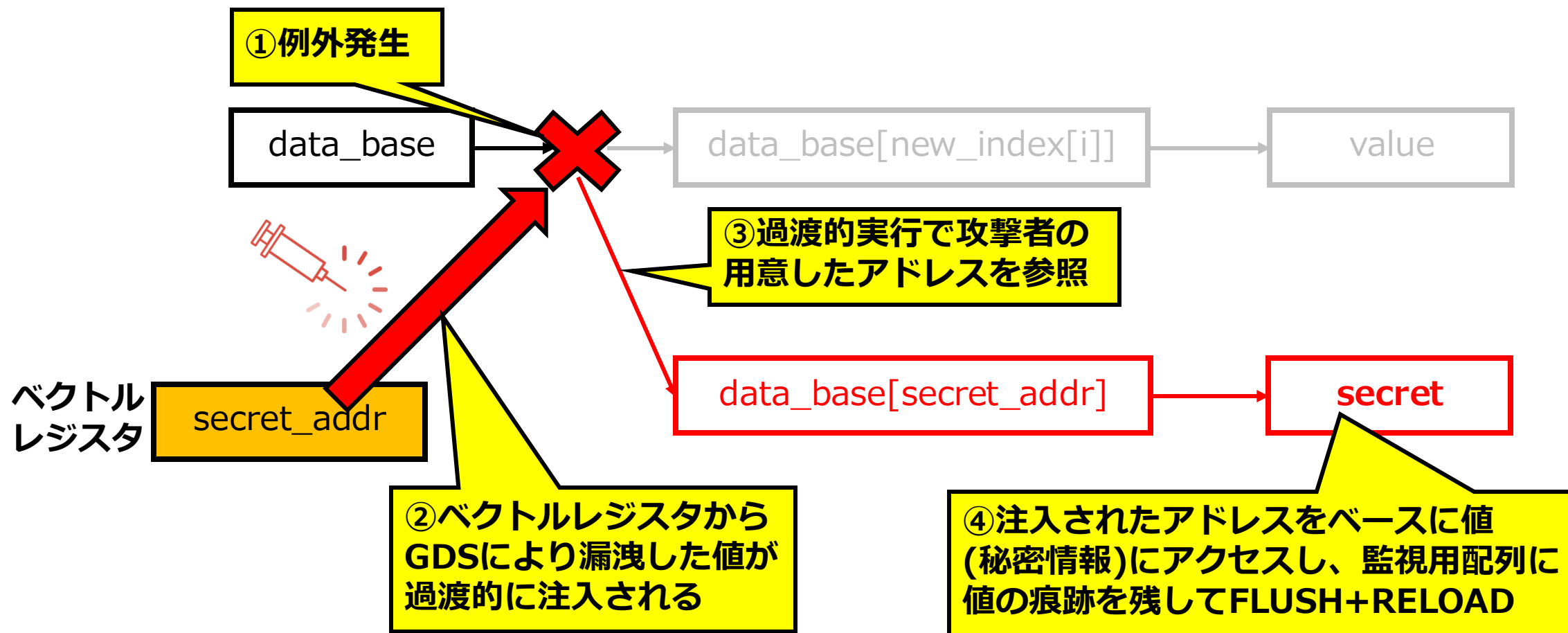


- GDSは**Meltdown型**の過渡的実行攻撃であるため、LVI同様ベクトルレジスタから漏洩した値を後続の命令への**注入に転用**できる事が推測できる
- 実際に、Downfallでは**Gather Value Injection (GVI)** としてGDSを**LVI的な攻撃に転用**する事に成功している
  - LVIについての詳細はSGX攻撃編③のスライドを参照
- LVIとは異なり、ほぼSGX専用の攻撃というわけではない

# 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]);
```



- 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



- 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



- 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[15]**の**実装**が挙げられる

# SGXへのGDS攻撃



- 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



- 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



- 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



- 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\_data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

輕減策



- **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



- LVI等と同様、Gather命令の後ろに**Ifence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Ifence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



- 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**[17]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**



- 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
- このことから、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

# 参考文献 (1/3)



- [1] "ZombieLoad: Cross-Privilege-Boundary Data Sampling", Michael Schwarz et al., <https://zombieloadattack.com/zombieload.pdf>
- [2] "キャッシュの書き込みポリシーと仮想記憶", 天野英晴 (慶應義塾大学), <https://www.am.ics.keio.ac.jp/parthenon/cache2.pdf>
- [3] "Rapid Prototyping for Microarchitectural Attacks", Catherine Easdon et al., <https://www.usenix.org/system/files/sec22-easdon.pdf>
- [4] "ZombieLoad Attack", Daniel Gruss et al., [https://gruss.cc/files/zombieload\\_36c3.pdf](https://gruss.cc/files/zombieload_36c3.pdf)
- [5] "64bitでのアドレス空間", 2023/9/27閲覧, <https://wiki.bit-hive.com/linuxkernelmemo/pg/64bit%E3%81%A7%E3%81%AE%E3%82%A2%E3%83%89%E3%83%AC%E3%82%B9%E7%A9%BA%E9%96%93>
- [6] "ページング入門", 2023/09/28閲覧, <https://os.phil-opp.com/ja/paging-introduction/#peziteburunoxing-shi>
- [7] "Intel SGX Explained", Victor Costan & Srinivas Devadas, <https://eprint.iacr.org/2016/086.pdf>

## 参考文献 (2/3)



[8]“Downfall: Exploiting Speculative Data Gathering”, Daniel Moghimi,  
<https://downfall.page/media/downfall.pdf>

[9]“Downfall Attacks”, Daniel Moghimi, <https://downfall.page/>

[10]“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>)

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

[12]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 (3/3)



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

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

[15]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>

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

[17] Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis, Daniel Moghimi et al., <https://www.usenix.org/system/files/sec20-moghimi-medusa.pdf>