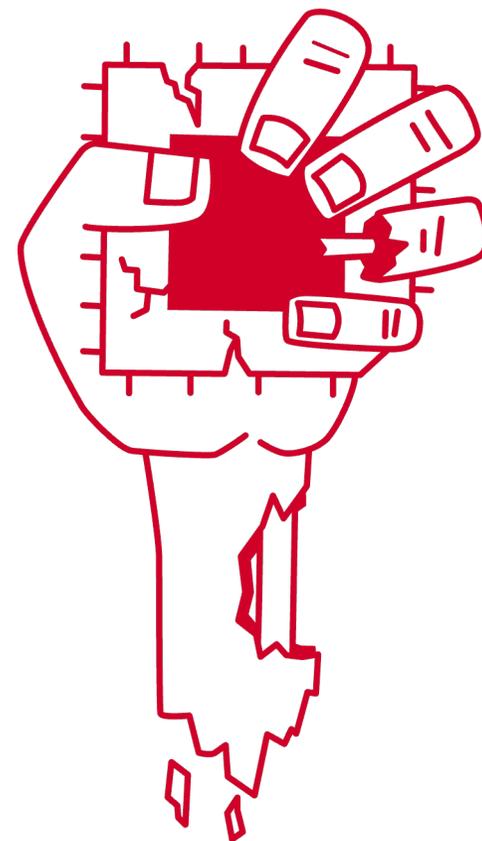


# SGX脆弱性解説 - ZombieLoad

Ao Sakurai



- Meltdown型の過渡的実行攻撃であり、Microarchitectural Data Sampling (MDS) 攻撃の一種でもあるZombieLoad攻撃について理解する。



# 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



## ■ L1キャッシュ

キャッシュ階層の中でも最もCPUの中心に近いキャッシュ。データを保持するL1Dと、命令を保持するL1Iが存在する。

## ■ ラインフィルバッファ (LFB)

L1Dに対して、他のキャッシュやメインメモリとのインタフェースとして機能するバッファ。

L1Dがキャッシュミスした場合、このLFBを介してより上位のキャッシュやメモリからデータが供給される[7]。

LFBエントリ1つあたりのサイズは64Bで、エントリ数は10か12。

## ■ ロードポート (LP)

メモリやI/Oからのロードを行うポート。



## ■ストアバッファ (SB)

未処理のストアデータとアドレスを追跡 (保持) するバッファ。  
準備・状況が整ったら、インオーダーで実際にストアを行う。  
実際のストア処理の完了を待つ代わりにこのバッファに一時的に保持させておく事で、命令パイプライン自体やアウトオブオーダー実行の高速化を図る事が出来る。

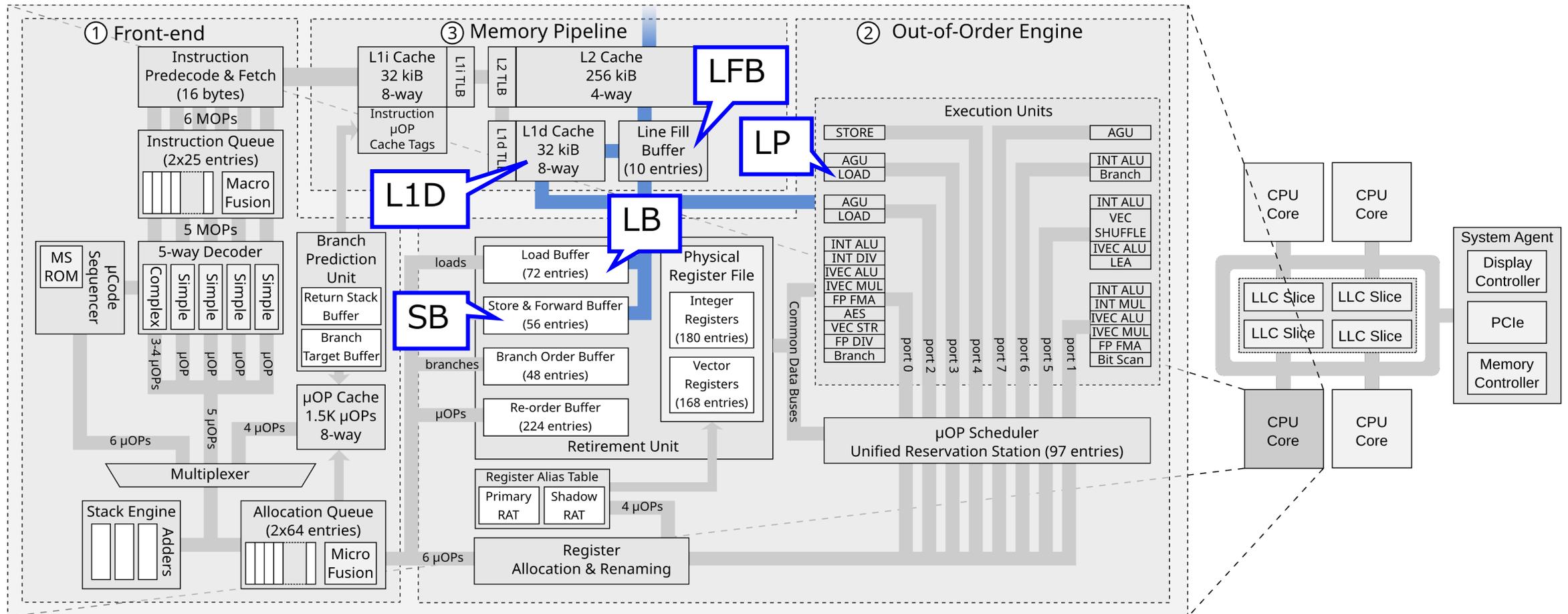
## ■ロードバッファ (LB)

SB同様、これから行うロードについてのエントリを保持する事で、最適化されたロード操作を行うためのバッファ。  
エントリに対応するロードが実際に行われると、エントリは解放されロード命令は命令リタイア (命令の最終的な完了) する

# 様々なμ-Archバツプア (3/3)



- CPU構造を示す図を再掲する ([2]より引用)



# 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** : 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（排他処理的）**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

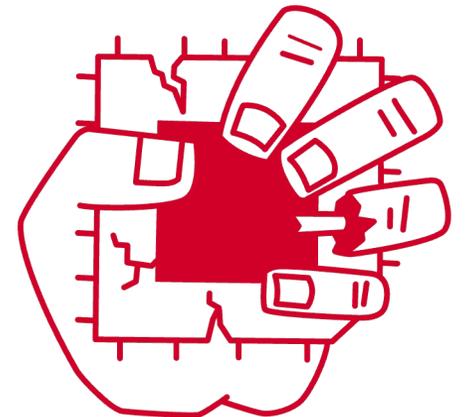


- トランザクション処理中に何らかの問題が発生した場合、アトミック性を保つために**TSXアボート**を発動させる
- TSXアボートが発動すると、実行自体を**トランザクション前**のアーキテクチャ状態に**ロールバック**し、トランザクション領域で実行された**全ての操作を破棄**する
- TSXアボートは、トランザクション内で変更されたアドレスから別の論理CPUから読み書きする事による**競合的なメモリ操作**等、様々な問題によって誘発される

# ZombieLoad (1/2)



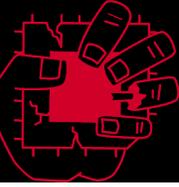
- ロード操作時に**フォールト**や**マイクロコードアシスト**が発生した際に、**過渡的実行ウィンドウ**においてLFBに存在する古い値が漏洩する事が観測された
  - LFBは**全ての論理CPUによって使用**され、プロセスや権限の**区別を行わない**点にも注意
- このようなLFBからの漏洩を発生させるロード (**ゾンビロード**) を悪用し、過去に使用された**秘密情報を漏洩**させる**過渡的実行攻撃**が **ZombieLoad**である



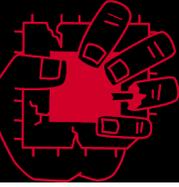


- **L1DとL2以上のメモリ階層とのインタフェースとして機能するLFBからの漏洩**であるため、攻撃の直前では原則として**キャッシュのクリア**を行い、**LFB経由でメインメモリからフェッチ**させる
  - これにより、フェッチに伴い**LFBにもそのデータが残留**する
- Meltdownのように**アドレスで漏洩対象を選択する事は出来ない**。良くも悪くも、**直前にロードやストアされた任意のデータの漏洩**であるため、この特性を上手く使ってやる必要がある

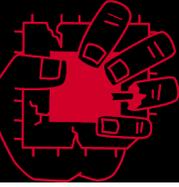
# ZombieLoadの原因



- 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**



- 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נק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



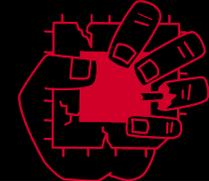
- この楽観的なマッチングにより、あるロードで例外が発生すると、**直前のロードやストアで有効だったような誤ったLFBエントリで過渡的に処理が継続され、その値の漏洩が発生する**
  - 本来もう使われてはならない古い値を使用する事になるため、これは **Use-After-Free脆弱性** である
- LFBは、前述の通り**論理CPU (ハイパースレッド) 間で競合的に使用される**事がIntelにより文書化されている
- よって、古いLFBエントリは**並行するハイパースレッド**からも**漏洩を行う**可能性がある
  - ちなみに、並行するハイパースレッドの事を**シブリングスレッ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

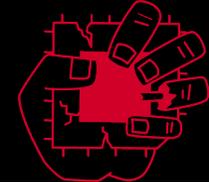


- このように、**LFBへのアクセスを封じた状況**でも、ZombieLoadによる**データの漏洩が確認**された
  - 数kB/sというかなりの漏洩レートが観測されている
  - LFBからの漏洩を間接的に示すMEM\_LOAD\_RETIRED.FB\_HITやMEM\_LOAD\_RETIRED.FB\_HITパフォーマンスカウンタは増えていない
  - 一方、MEM\_LOAD\_RETIRED.L1\_HITは数千回を記録している
- よって、少なくとも**漏洩はLFBからだけではない**事は分かるが、具体的にどこから漏洩しているかは不明
  - 紛らわしいが、上記の漏洩が**L1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



- 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)



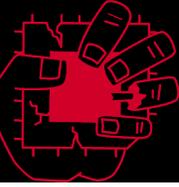
- 様々なMeltdown型攻撃において、漏洩対象のどこまでを攻撃者が明示的に指定できるかを表した図 ([1]より引用)

	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	5
	47	Virtual	12			



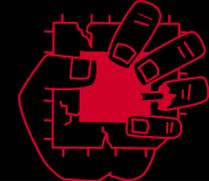
- **ところで、従来のメモリベースのサイドチャネル攻撃は、メモリアクセス位置を特定し、実行時間等からその時点での処理で使用されている秘密情報を漏洩させる**
  - 早い話、「この実行時間でこのアクセスパターンならこの値がこの処理で使われているに違いない」といった推測が可能
  - **メモリアクセス位置と実行中の処理（命令ポイントにより管理される）の実行時間を照らし合わせて秘密情報を推測するため、メモリアクセス位置と命令ポイントを関連付けるような攻撃である**
- **一方、Meltdownは本来アクセスしてはならないアドレスに過渡的にアクセスする事で、その値をサイドチャネル的に観測する**
  - 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)

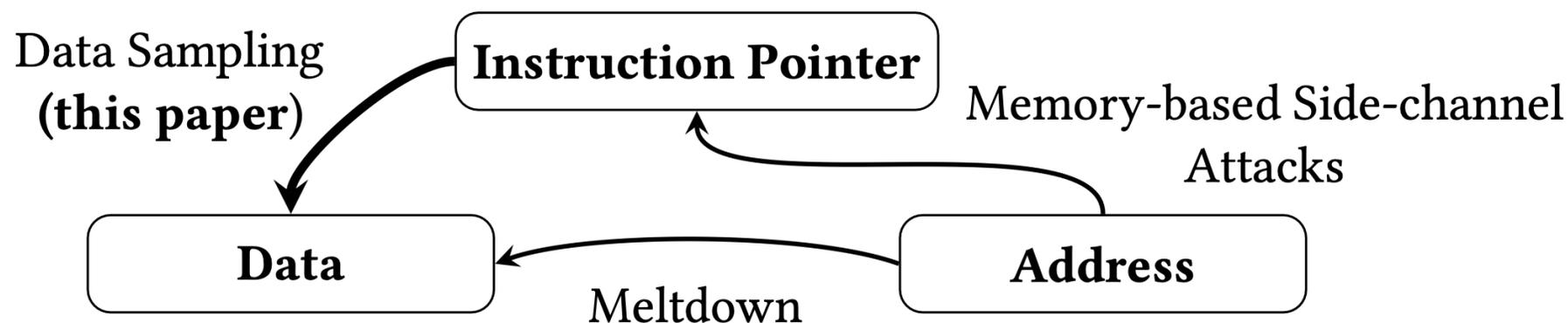


- 一方、ZombieLoadは**直前の処理**（**命令ポインタ**が現在指しているよりも前の命令）で使用された**データ**を、**LFB内**からバイト単位で**任意に指定して漏洩**させる事ができる
- よって、言わばZombieLoadは**命令ポインタ**（直前の処理）と**データ**を**関連付ける**ような攻撃と言える
- そして、このような攻撃の事を**データサンプリング攻撃**と呼ぶ事に行っている。ZombieLoad、RIDL、Falloutは全て**μ-Arch**上でこれを行うため、**MDS攻撃**（**Microarchitectural Data Sampling**）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

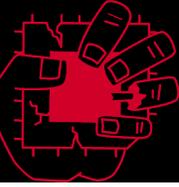
# データサンプリング (5/5)



- 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。



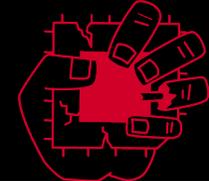
## ■ Intel SGXからの漏洩

**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。

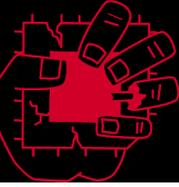
※PTE : ページテーブルエントリ (Page Table Entry)



## ■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** [7][9]
  - 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** [6]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

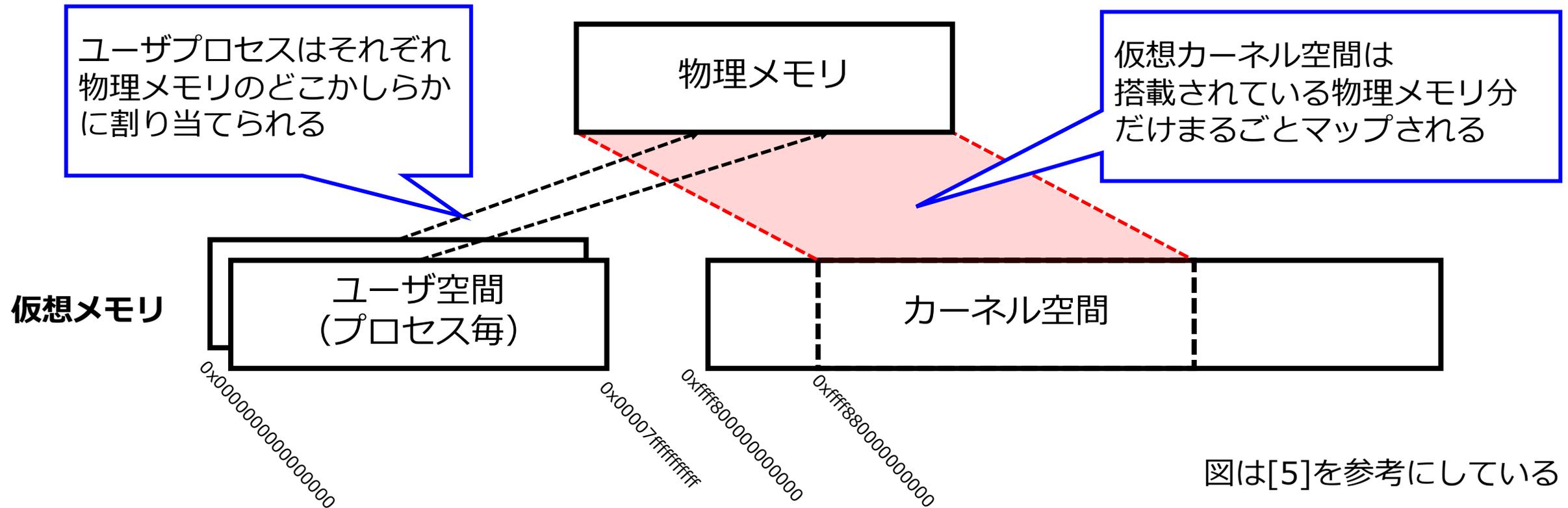


- カーネルは、**カーネルの仮想メモリが特定の仮想アドレス（ベースアドレス）から始まるようにし、かつ搭載メモリ分だけ物理メモリに直接マップする**（Direct Physical Map; ダイレクト物理マップ）
- このカーネルの**ベースアドレスを基準**とし、物理アドレス $p$ を**オフセット的に使用する事で、仮想カーネルアドレスである $k$ を攻撃者が取得**する事ができる
- 物理アドレス $p$ は、**`/proc/pageinfo`**から取得する（要特権）、**PTEditor**を用いる（非特権で可能[4]）、別の**サイドチャネル攻撃で奪取**する等により入手する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

# ZombieLoad v1 - カーネルマッピング (3/4)



- 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

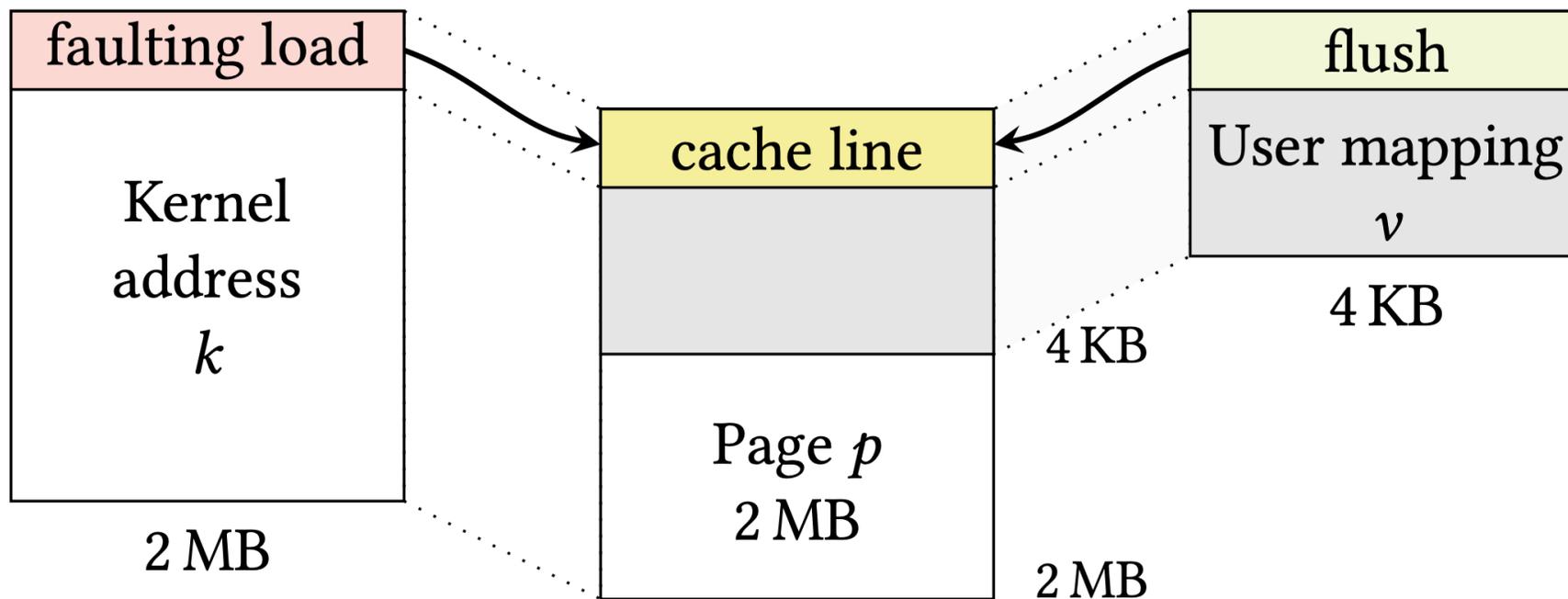


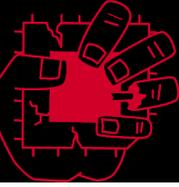
図は[5]を参考に行している

# ZombieLoad v1 - カーネルマッピング (4/4)



- 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

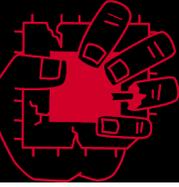


- ある1つの**物理ページ** $p$ を同時に指す、**2つの別々の仮想アドレス** $v, v_2$ が存在するとする
  - 共有メモリやメモリマップトファイルのように、2つの別々のプロセスから同じ位置のデータ（物理ページ）にアクセスする状況を考えると分かりやすい
- $v$ からは $p$ にアーキテクチャ的に**正当にアクセスできる**とする
- 一方、 $v_2$ についてはPTEの**Accessedビット**（Aビット）を**0**にする
  - **Aビット**：そのページが使用された場合に1になるPTE上のビット
  - **Linux**ではAビットのクリアには**特権**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**の状態では**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）：
  - **白丸**：実行可能、**白黒半分ずつの丸**：特定のHW構成で可能、**黒丸**：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**[8]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[11]

# 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)	$\mu$ -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	✗	✓	✗



- 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)

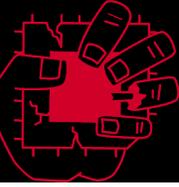


- **特権攻撃者**が、Enclave内で実行される**sgx\_get\_key**関数から、この関数で生成された**鍵**を**ZombieLoad**で**漏洩**させるシナリオについて考える
  - `sgx_get_key`は、主にEnclave内で使用される各種の重要な鍵を導出する**EGETKEY**というCPU命令のラッパー関数である
- `sgx_get_key`からの漏洩レートを図るベンチマーク用Enclaveからの漏洩の他、**自己署名QE**からの**PSKの漏洩**についても行っている
  - PSKについては[こちら](#)を参照
- 特権攻撃が可能であるため、Foreshadow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

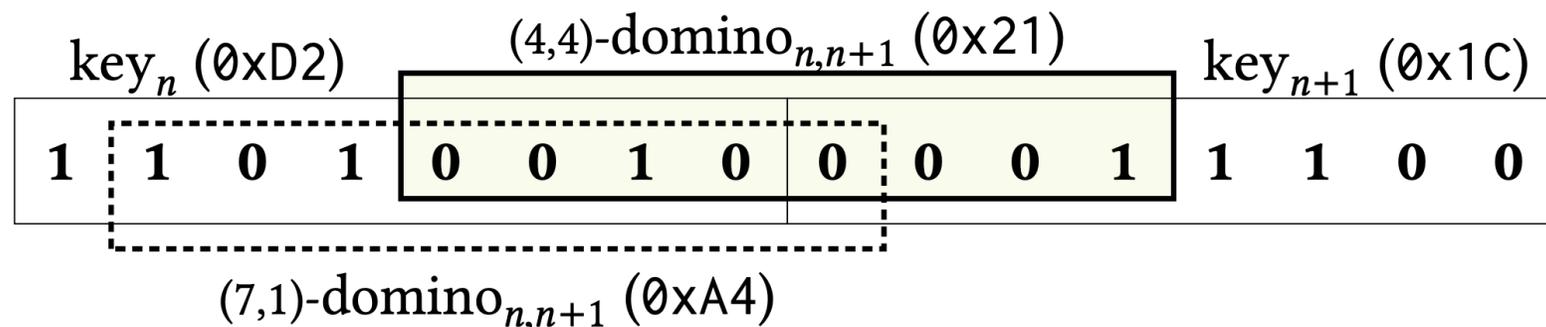


- 攻撃対象で使用する**キャッシュへのヒット**を観測した瞬間にZombieLoadを開始する等により、**興味のない処理**から漏洩する**ノイズを大幅に制限**できる
- しかし、それでも攻撃の裏でOSやプロセス等が行う**無関係なロード**からの値が漏洩してくる事もあり、これ自体を**抑制**しきる事は**不可能**に近い
- そこで、漏洩させたい鍵バイトだけを観測するだけでなく、**隣り合う鍵バイトのそれぞれ一部分を内包**するような**ドミノバイト**を観測する手法を実施する

# 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）

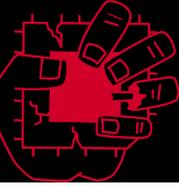


- 各鍵バイト位置で攻撃を繰り返し、最も漏洩の多いバイトが正しいと仮定するだけでなく、その候補にドミノバイトの内容が一致して初めて正解であると判断するようにする
  - 隣接する2つの鍵バイトと、それにまたがるバイトであるドミノバイトとの整合性が取れていれば、そこに誤りは無いだろうというロジック



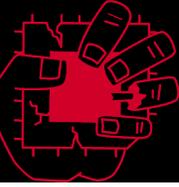
- ZombieLoadでは**LFBエントリ内の位置をバイトレベルで指定して漏洩させる事しか出来ないため、単なる鍵バイトの観測だけでは**  
**ドミノバイトを漏洩させる事は出来ない**
  - 例えば、**連続するLFBエントリのそれぞれ後ろ4ビットと前4ビットずつの漏洩、というビットレベルの指定は出来ない**
- そのため、過渡的実行により**AES鍵全体の内どの部分の漏洩も可能**であり、かつそれを過渡的に任意の処理に利用できるという前提が必要
  - この条件を満たせば、隣接する鍵のそれぞれ一部ずつからなる1バイト(=ドミノバイト)を**ロードまたはストア**しておき、直後に**ZombieLoadを発動**させてそれを**漏洩させる**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)



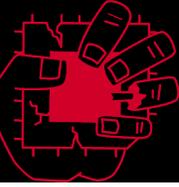
- `sgx_get_key`は、内部で独自の`intel_fast_memcpy`という関数を呼び出し、**ベクトルレジスタ**である`xmm0`を経由させる形で、**128bit単位のmove命令でコピー**している事が判明した
  - **ベクトルレジスタ**：SIMD命令等でよく使われる、そのアーキテクチャのビット数よりも大きいようなレジスタ。`xmm`レジスタは128bit長である
  - 余談だが、このベクトルレジスタからの過渡的漏洩が**Downfall**として数年後の2023年に報告されている
- よって、この独自の`memcpy`を構成する機械語レベルでの**最後の命令**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)

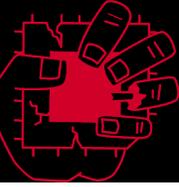


- この状態でZombieLoadにより鍵を漏洩させる攻撃を、まずは前述の**ベンチマーク用Enclave**に対しCore i7-7700のマシン上で実行した
  - レポートキーを漏洩させ正しいかをチェックする事で成功確率を検証している
- 結果、100回中30回、つまり**30%の確率**で鍵候補の中に**全鍵バイトが内包**されている状態が確認された
  - 実際に**完全な鍵を回復**できたのはその内**3%**の試行だけであり、割合としてはかなり低いと言わざるを得ない
- 残り70%では全鍵バイトは観測できず、内31%では**平均して10バイト**含まれており、残り39%では**一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SGXSDK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

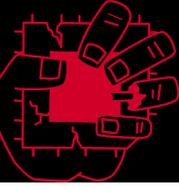


- **PSKはMRSIGNERポリシ**、つまり**Enclave署名鍵に依存する**シーリング鍵であるため、この**実験用QEから漏洩したPSK**では、Intel公式の**PvE** (QEと署名鍵が同じ) で**正規のPSKにより**シーリングされた**Attestationキーはアンシーリング出来ない**
- 論文中ではIntelが署名した**公式QEからの漏洩は行っていないが**、もしできてしまうと漏洩したPSKで**正規のAttestationキーが暴かれ**、**RAのセキュリティ保証が崩壊**してしまう

# VM間秘密チャネル (1/7)

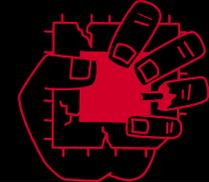


- **秘密チャネル**：本来データ転送のために用意されているものではない要素を用いて構築された、秘密裏にデータ転送を行う通信路
  - 英名：Covert Channel
  - 過渡的実行攻撃でのキャッシュサイドチャネル攻撃におけるキャッシュも、まさに**過渡的領域**から**命令リタイア**後への**秘密チャネル**である
- **攻撃対象VMの秘密情報**を漏洩させて**攻撃者のVMに転送する秘密チャネル**を、**ZombieLoad**によって**実現し攻撃を実行**する
  - プロセス間、カーネル、SGX、ハイパーバイザといった他のシナリオでも可能ではあるが、ここではVM間攻撃を考える
- 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

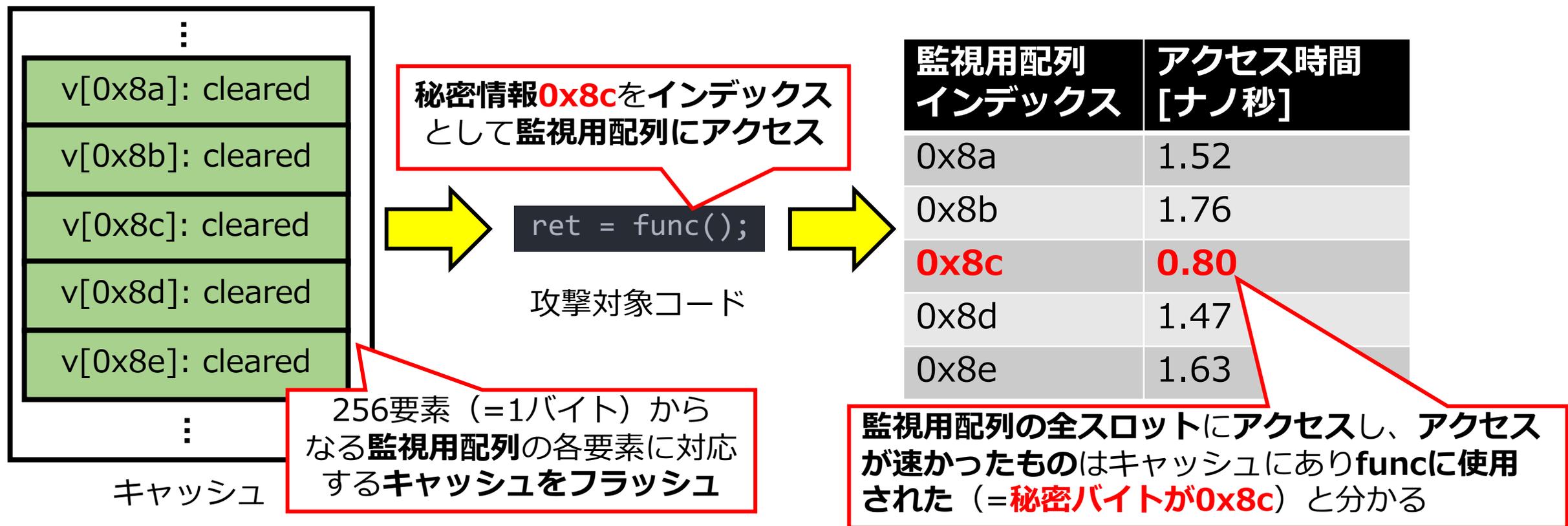


- **送信側 (攻撃対象VM)** では、**漏洩させたい値を複数のメモリアドレスに展開し、それぞれから繰り返しロードする**
  - これにより、**複数のLFBエントリ** (CPUモデルによってエントリ数は**10か12個**) に漏洩させたい値が持ち込まれる確率が上がり、**ZombieLoad** により**漏洩する確率も上がる**
- **受信側 (攻撃側VM)** では**ZombieLoadを実行**し、送信側がロードした値を漏洩させる。漏洩させた値は監視用配列の**キャッシュに痕跡を残し、FLUSH+RELOAD**で過渡的実行終了後に**観測する**
  - Foreshadow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** (合計256要素) は**ページサイズ間隔**で配置する

# (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[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)



# VM間秘密チャネル (3/7)

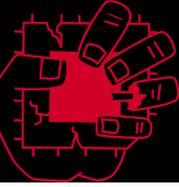


- 例に漏れずZombieLoadは**無関係な処理**に由来する**ノイズ**も**漏洩させる**ため、以下の図に示すような**32bitのパケット**を漏洩・転送させる事でそのような**ノイズの排除**を試みる



- 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**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



- 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



- 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
- 右から4バイト目については、特に用途が無いので固定で0xFFにしているようである



- 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が、Azure以外使うことはあまりないのでAzureな気がする
- 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

# 本セクションのまとめ



- ZombieLoadは、直前の処理で使用された値をLFBから漏洩させる、MDS攻撃の一種である
- SGXだけでなく、プロセス間、カーネル、VMM等の様々な境界をまたいで漏洩を行う事ができる
- ZombieLoad v2に関しては、当時MDS耐性があるとされていたCPU上でも漏洩を行う事ができてしまう
- ZombieLoadの漏洩元にLFBが含まれている事は確かだが、LFBだけではない事が実験的に示唆されている

# 参考文献 (1/2)



[1] "ZombieLoad: Cross-Privilege-Boundary Data Sampling", Michael Schwarz et al., <https://zombieloadattack.com/zombieload.pdf>

[2] "MDS: Microarchitectural Data Sampling", 2023/7/20閲覧, <https://mdsattacks.com/>

[3] "Linuxカーネル4.1のメモリレイアウト (ドラフト)", 2023/9/27閲覧, <https://kernhack.hatenablog.com/entry/2019/05/10/205220>

[4] "PTEditor", GitHub, <https://github.com/misc0110/PTEditor>

[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] "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>

[7] "Rapid Prototyping for Microarchitectural Attacks", Catherine Easdon et al., <https://www.usenix.org/system/files/sec22-easdon.pdf>

## 参考文献 (2/2)



[8]“ページング入門”, 2023/09/28閲覧, <https://os.phil-opp.com/ja/paging-introduction/#peziteburunoxing-shi>

[9]“ZombieLoad Attack”, Daniel Gruss et al., [https://gruss.cc/files/zombieload\\_36c3.pdf](https://gruss.cc/files/zombieload_36c3.pdf)

[10]“FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”, Jo Van Bulck et al., <https://foreshadowattack.eu/foreshadow.pdf>

[11]“Intel SGX Explained”, Victor Costan & Srinivas Devadas, <https://eprint.iacr.org/2016/086.pdf>