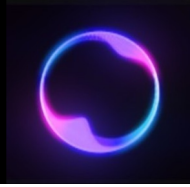
11. SGX攻撃編③

Ao Sakurai

2024年度セキュリティキャンプ全国大会
S3 – TEEビルド&スクラップゼミ

本セクションの目標

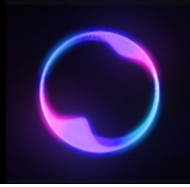


- SGXに対する過渡的実行攻撃の内、Foreshadowよりも後に発見された強力な攻撃として、「ZombieLoad」「LVI（Load Value Injection）」「Downfall」を紹介する

- 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

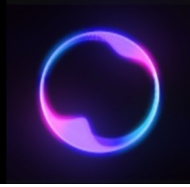


- フォールトの処理やページテーブルの内容の変更のような**複雑な操作**は、通常事前に定義された**マイクロコードルーチン**に対して**マイクロシーケンサ**を向けさせる
  - マイクロシーケンサ：命令処理に使用するマイクロ命令の組み合わせを決定する機構

- 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

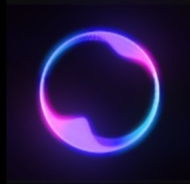
Intel TSX（1/2）



- **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

Intel TSX（2/2）



- 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

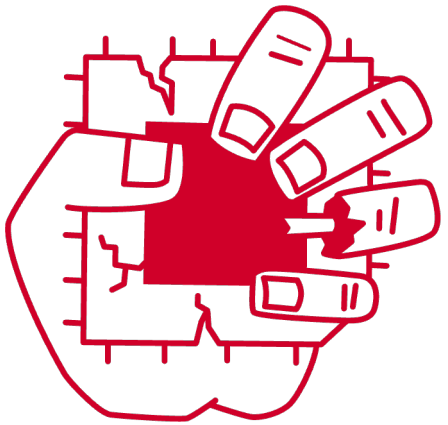
- TSXアボートが発動すると、実行自体を**トランザクション前**のアーキテクチャ状態に**ロールバック**し、トランザクション領域で実行された**全ての操作を破棄**する

- 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



ZombieLoad（2/2）

- **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**

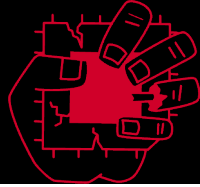
- Meltdownのように**アドレスで漏洩対象を選択する事は出来ない。**良くも悪くも、**直前にロードやストア**された**任意のデータの漏洩**であるため、この特性を上手く使ってやる必要がある

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）



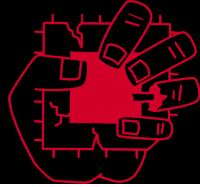
- マイクロコードアシストが発動されると、**命令パイプライン**を**フラッシュ**する**マシンクリア**が発生し、また**既に実行中の命令**についても**強制終了**させる

- 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**過渡的に処理が続行**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

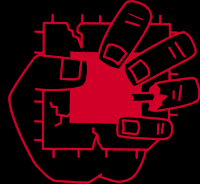
ZombieLoadにおける漏洩元（1/5）



- 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

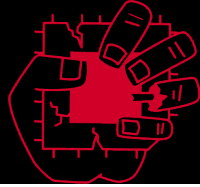
ZombieLoadにおける漏洩元（2/5）



- 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**

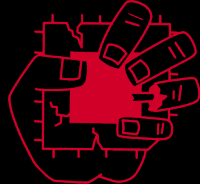
ZombieLoadにおける漏洩元（3/5）



- 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）

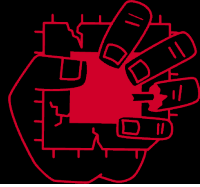


- トランザクション内部で書き込みを行う場合、**使用するデータ**が**書き込みセット内に存在**している必要がある
  - **書き込みセット**は**必ず**L1キャッシュ内に存在していなければならない

- 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

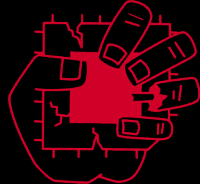
- **L1キャッシュ**は**LFB**よりも**CPUの中心に近い**位置に存在するため、**L1から提供**される限り**LFBへのアクセス**が**発生する事は無い**

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

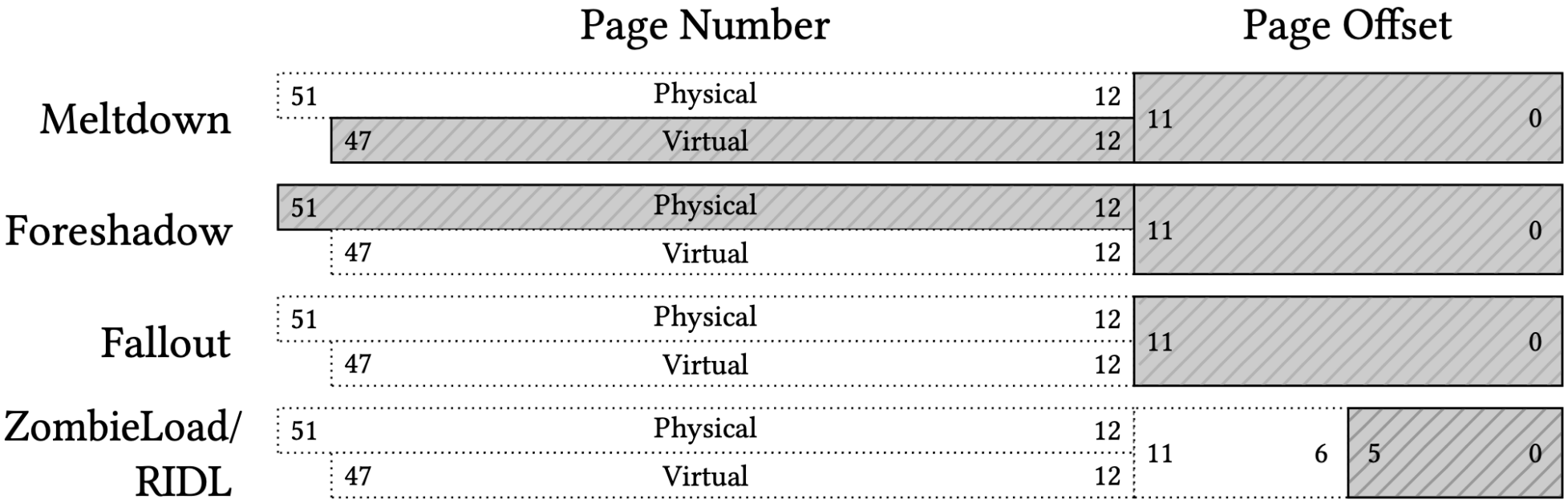
データサンプリング（1/5）



- 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）



データサンプリング（3/5）

- 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）

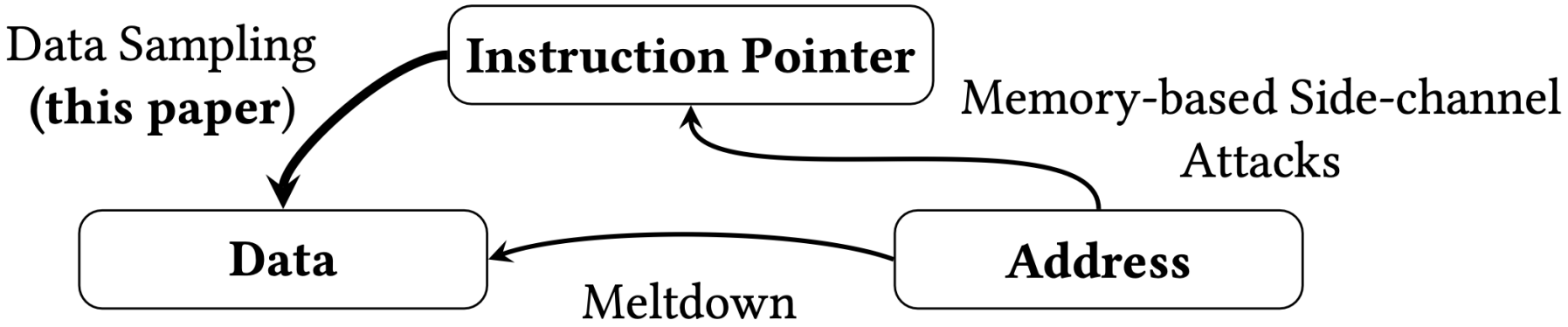
- 一方、ZombieLoadは**直前の処理**（**命令ポインタ**が現在指しているよりも前の命令）で使用された**データ**を、**LFB内**からバイト単位で**任意に指定して漏洩**させる事ができる

- 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**と呼ぶ事にしている。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）

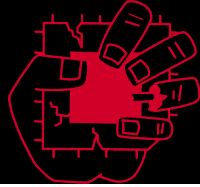


RIDLとの比較

- 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（バリアントについては後述）：

	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）
既知のバリアント数	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（バリアント2）

攻撃シナリオと攻撃モデル（1/3）



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。

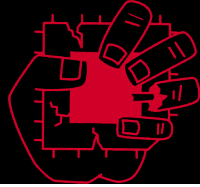
■ユーザ空間からの漏洩

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。

攻撃シナリオと攻撃モデル（2/3）



■Intel SGXからの漏洩

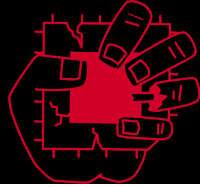
SGXのEnclave内部で実行されたロードとストアから漏洩させる事も可能。勿論、**Enclave内データを対象としていても同様**。
攻撃者は攻撃対象（Enclave）の**シブリングスレッド**で動作する。
SGXの脅威モデルの性質上、**特権攻撃者**を前提とする。

■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）

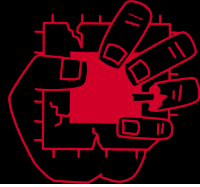
攻撃シナリオと攻撃モデル（3/3）



■ハイパーバイザからの漏洩

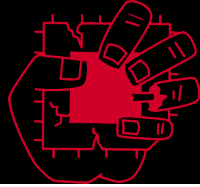
VM内の攻撃者が、**ハイパーバイザ**がロードやストアする値を漏洩させる事もできる。
こちらも**信頼不可能なVM内で実行**しているため、**特権コードの実行**を制限されない。

攻撃バリアント



- ZombieLoadには、具体的な攻撃アプローチとして**5つのバリアント**（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アント1～3

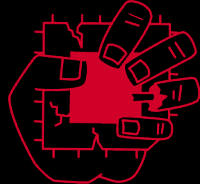
ZombieLoad v1 – カーネルマッピング（1/5）



- 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

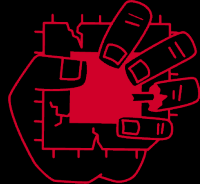
- この時、**カーネルページ**または**アクセス不能**な**仮想メモリページk**を用意し、**kも物理ページpを指す**ように仕向ける

ZombieLoad v1 – カーネルマッピング（2/5）



- この状態でv**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、このバリアント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

ZombieLoad v1 – カーネルマッピング（3/5）



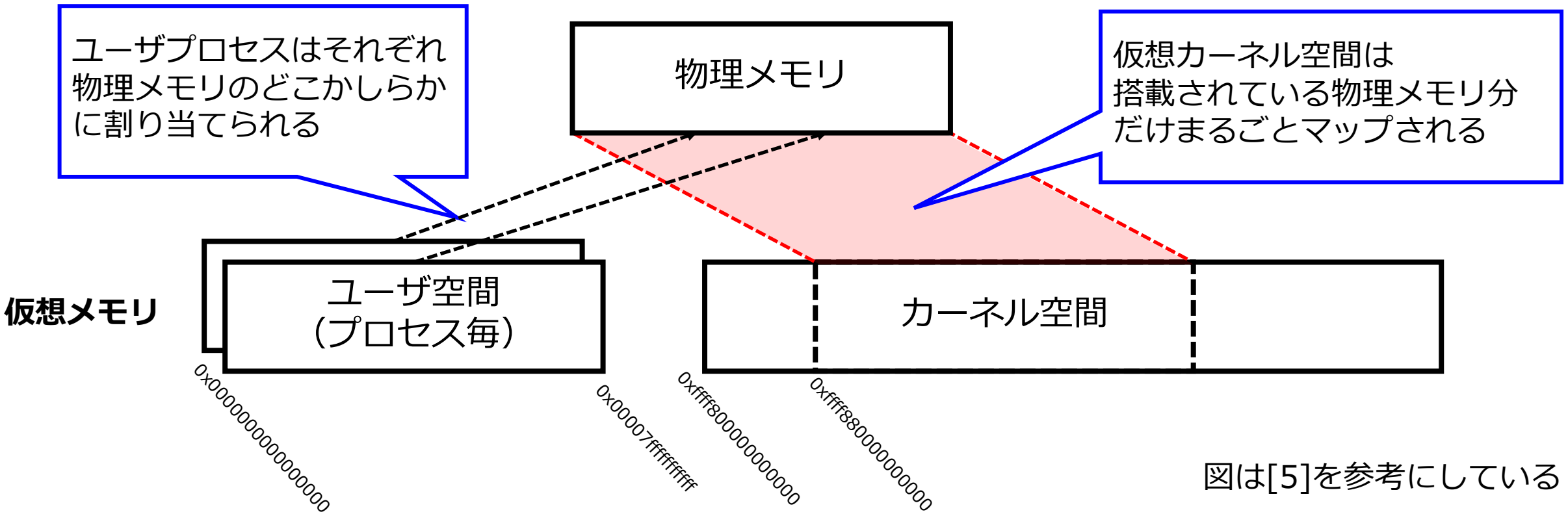
- カーネルは、**カーネルの仮想メモリ**が**特定の仮想アドレス（ベースアドレス）**から始まるようにし、かつ搭載メモリ分だけ物理メモリに**直接マップする**（Direct Physical Map; ダイレクト物理マップ）

- 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）

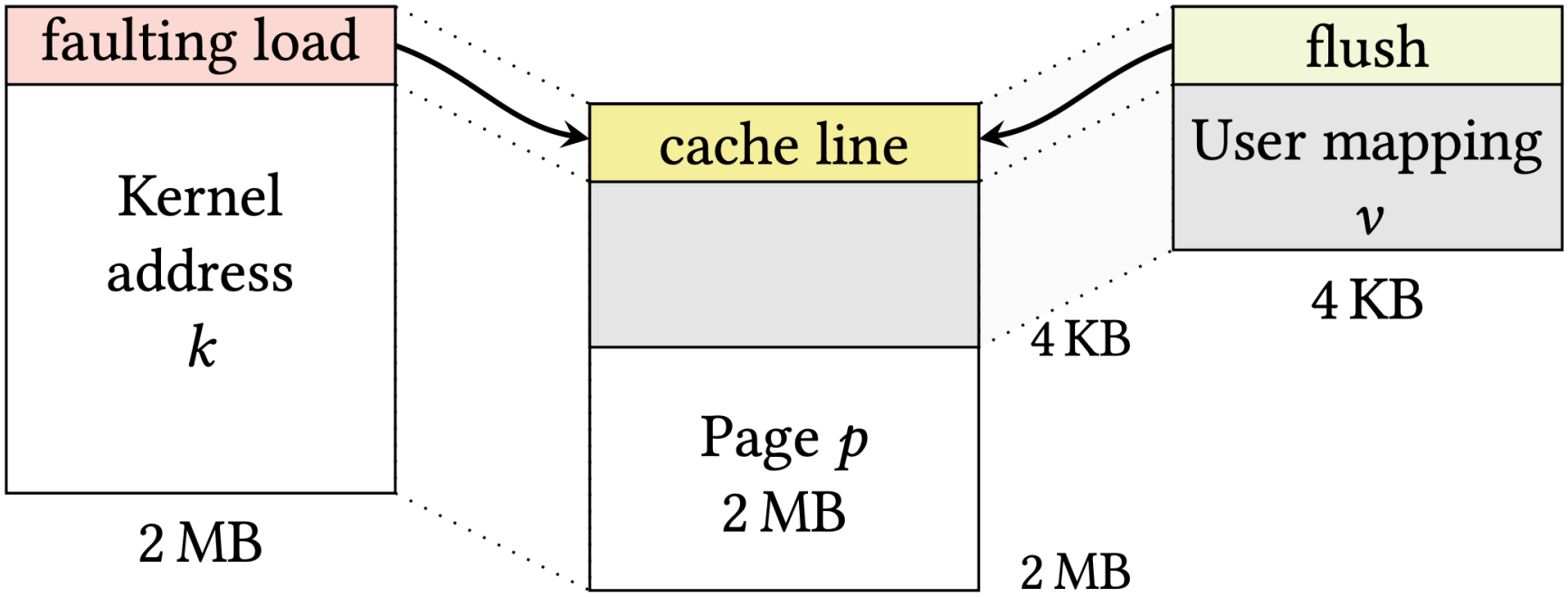
- ユーザ空間メモリは**プロセスごと**に**物理メモリにマップ**される
  一方で、カーネルメモリは**物理メモリ全体に丸ごとマップ**されるため、
  ユーザページとカーネルページは**重複**し得る
  - PTEのU/Sビット等により、アクセス違反が発生しないよう管理している



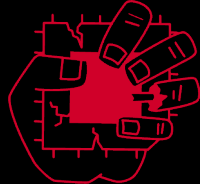
図は[5]を参考にしている

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）



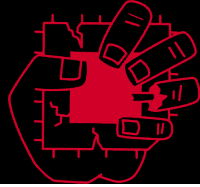
ZombieLoad v2 – Intel TSX（1/2）



- 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

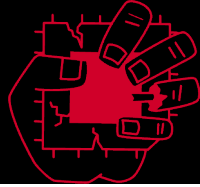
ZombieLoad v2 – Intel TSX（2/2）



- 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

- このバリアント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

ZombieLoad v3 – 特殊なページテーブルウォーク（1/2）



- 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

ZombieLoad v3 – 特殊なページテーブルウォーク（2/2）

- 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

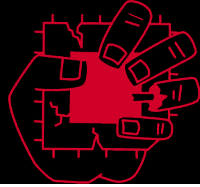
- この状態でv_2にアクセスを行う事で、**マイクロコードアシスト**を**誘発**し、かつキャッシュが確実にクリアされているのでLFBからの**ゾンビロード**を**誘発する事ができる**

ZombieLoad v1～3を実行できる環境

- バリアント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)	● ●	◐ ◐	● ●

ZombieLoad v4 – アボートページセマンティクス（1/3）



- **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

ZombieLoad v4 – アボートページセマンティクス（2/3）

- 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

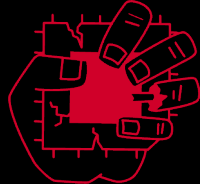
- 論文における攻撃の動作確認実験では、**TSXトランザクション内**で前述のようなvにアクセスして**ゾンビロードを発動**させている
  - Foreshadowでも使われている手法だが、TSXトランザクション内で実行すると、OSのフォールト処理に伴う、キャッシュやその中間経路であるLFBの**汚染を抑制**する事ができる

ZombieLoad v4 – アボートページセマンティクス（3/3）

- このバリアント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由来のものでなくても構わない**という点に注意

ZombieLoad v5 – キャッシュ不可メモリ



- バリアント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**

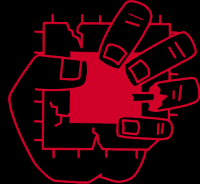
- キャッシュ不可であるようなページにアクセスすると、**ページミスハンドラ**が**マイクロコードアシスト**を**発動させる**挙動を利用している
  - **ページミスハンドラ**：TLB（PTEのキャッシュのようなもの）に対象の仮想アドレスが存在しない場合に、ページテーブルを参照して解決を図るページウォークを行う機構[7]

パフォーマンス評価（1/2）

- 様々なCPUのマシンにおける、バリアント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）

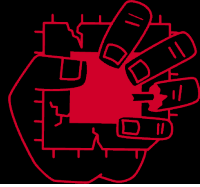
Setup	CPU (Stepping)	μ-arch.	Variant 1	2	3
Lab	Core i7-3630QM (E1)	Ivy Bridge	✓	-	✓
Lab	Core i7-6700K (R0)	Skylake-S	✓	✓	✓
Lab	Core i5-7300U (H0)	Kaby Lake	✓	✓	✓
Lab	Core i7-7700 (B0)	Kaby Lake	✓	✓	✓
Lab	Core i7-8650U (Y0)	Kaby Lake-R	✓	✓	✓
Lab	Core i7-8565U (W0)	Whiskey Lake	✗	-	✗
Lab	Core i7-8700K (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）



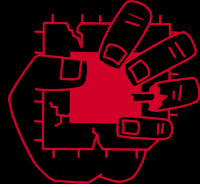
- バリアント1～3について、**Core i7-8650U**のマシンにおいて漏洩（伝送）速度を計測する実験を行っている
- **v1**の場合、v4のようにTSXで例外を抑制しながら実施した所、平均伝送速度は**5.30kB/s**、真陽性率**85.74%**であった
- **v2**の場合、平均伝送速度は**39.66kB/s**、真陽性率は**99.99%**だった
- **v3**は、**TSX無し**では平均伝送速度**0.08kB/s**で**52.7%**の真陽性率、**TSX有り**の場合平均伝送速度**7.73kB/s**で真陽性率**76.28%**であった
  - 論文中では、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）

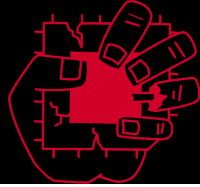


- **機密情報が格納**されているような状態で**CPU状態を固定**し、それに対して**攻撃を繰り返して逐次秘密情報を抽出**したい場合がある
  - ZombieLoadやForeshadowのように、1バイトずつしか漏洩させられない攻撃でCPUレジスタ状態から16バイトの鍵を抽出する、といった場合に必要

- 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）

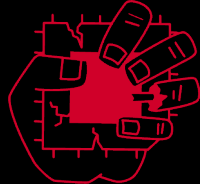


- 攻撃対象で使用する**キャッシュへのヒット**を**観測**した瞬間にZombieLoadを開始する等により、**興味のない処理**から漏洩する**ノイズ**を**大幅に制限**できる

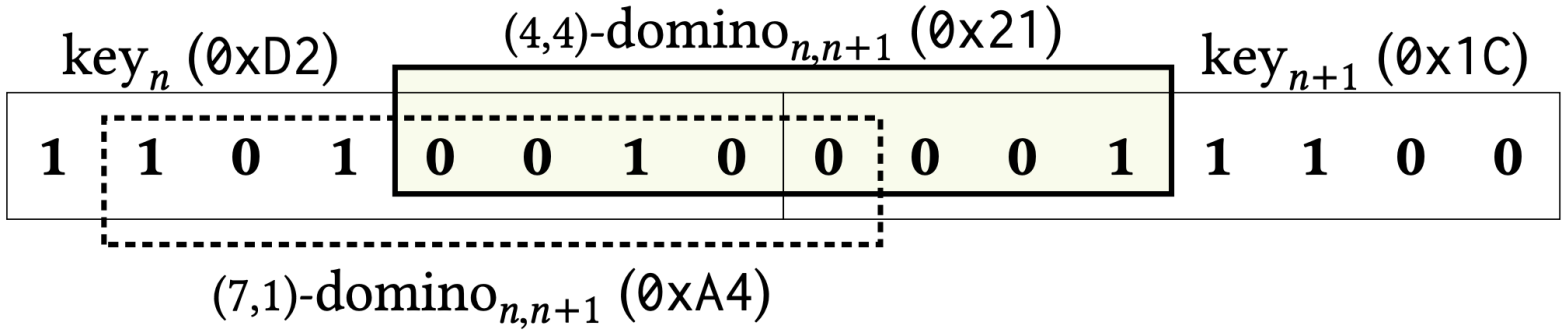
- しかし、それでも攻撃の裏でOSやプロセス等が行う**無関係なロード**からの値が漏洩してくる事もあり、これ自体を**抑制**しきる事は**不可能**に近い

- 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）

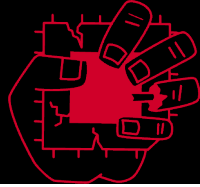


- 以下の図のように、**隣接する鍵それぞれ**（白実線矩形部）を漏洩させると共に、それらの**それぞれ一部**を含む**ドミノバイト**（薄緑矩形部）の漏洩と抽出も行う（図は[1]より引用）



- 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

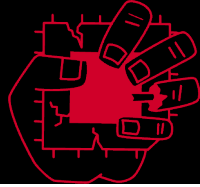
SGXシーリング鍵の抽出（6/10）

- 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

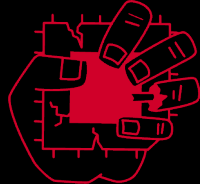
SGXシーリング鍵の抽出（7/10）



- 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

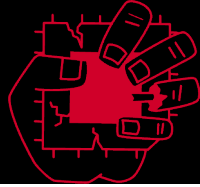
- よって、この独自のmemcpyを構成する機械語レベルでの**最後の命令**で**ゼロステップ処理**を行い、xmm0にSGX鍵を内包しているようなCPU状態を繰り返しSSAからロードする状況を作る

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**

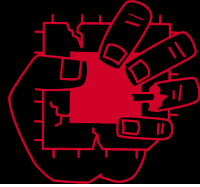
SGXシーリング鍵の抽出（9/10）



- 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

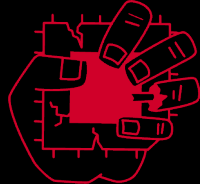
SGXシーリング鍵の抽出（10/10）



- **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

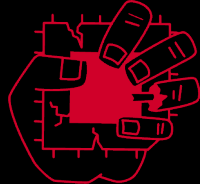
- この二者のVMはシブリングスレッド上にそれぞれ存在するとする

VM間秘密チャネル（2/7）

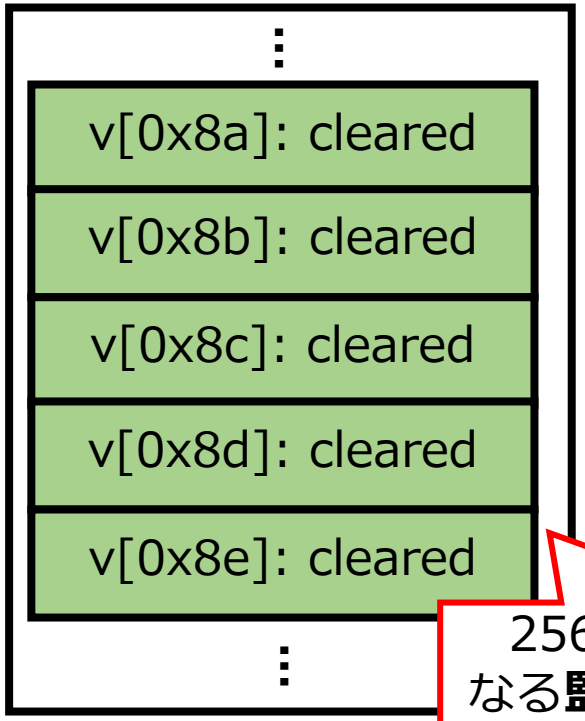
- **送信側**（**攻撃対象VM**）では、**漏洩させたい値**を**複数のメモリアドレス**に展開し、それぞれから**繰り返しロード**する
  - これにより、**複数のLFBエントリ**（CPUモデルによって**エントリ数は10か12個**）に漏洩させたい値が持ち込まれる確率が上がり、**ZombieLoad**により**漏洩する確率**も**上がる**

- **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）



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();
```

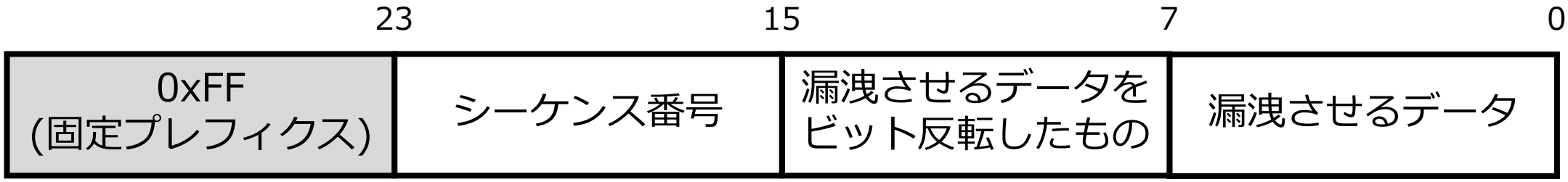
攻撃対象コード

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

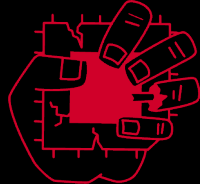
VM間秘密チャネル（3/7）

- 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

VM間秘密チャネル（7/7）

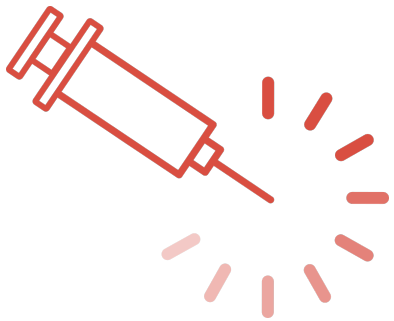
- 論文では、Core i7-8650Uを搭載する**ローカルマシン**上の**QEMU KVM**で動作する**2つのVM間**での転送と、**あるパブリッククラウド**の**同一ベアメタル上のVM間**での**漏洩と転送を行う実験**を実施している
  - どのパブリッククラウドであるかは当該クラウド事業者により口封じされているらしい
- 前者のシナリオでは、**TSXによる例外抑制**を用いながらの**ZombieLoad v1**を用いて**最大26.8kbps**の**伝送レート**を記録している
- クラウドのシナリオでは、そのクラウドでTSXを使用できなかったため、**TSX無し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

Load Value Injection（LVI）

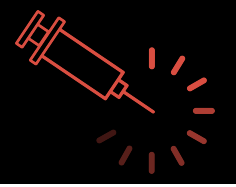
Load Value Injection（1/4）

- ForeshadowはMeltdown型の攻撃であり、**μ-Archバッファ**である**L1D**から**過渡的実行**において**秘密情報**を**漏洩**させていた
  - その後、過渡的実行の結果が棄却されても後から観測できるように、キャッシュに痕跡を残しサイドチャネル攻撃で抽出する

- これに対し、**過渡的実行**においてMeltdown**挙動**によりμ-Arch**バッファ**から**流れ出たデータ**を、その**過渡的命令**に対する**注入**に使う攻撃が**Load Value Injection（LVI）**である



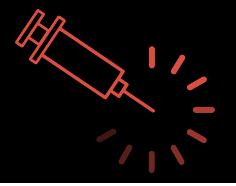
Load Value Injection（2/4）



- 名前の通り、**ロード命令**（あるいはロードマイクロ命令）において**フォールト**または**マイクロコードアシスト**を発生させ、それに伴うその**ロード命令の過渡的実行**に対し**μ-Archバッファ**から値を**Meltdown的に注入**（インジェクション）する攻撃
  - μ-Archバッファは予め攻撃に使う値で**汚染**（ポイズニング）しておく
  - フォールトとしてはページフォールト等が挙げられる
  - **マイクロコードアシスト**：普通は滅多に発生しないような状況に遭遇した際に、マイクロコードがそれを対処する動作

- 広義にはPlundervolt同様**故障注入攻撃**の**一種**でもあり、実際にそのように振る舞う事も出来る

Load Value Injection（3/4）

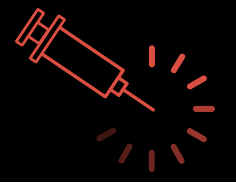


- 本来**秘密情報の漏洩**を行う**Meltdown挙動**をロード命令への**値の注入に転用**しているため、**逆Meltdown攻撃**とも、あるいはMeltdownを**Spectre的な注入に繋げる**という意味で二者の**融合型攻撃**とも表現する事が出来る

Methodology / μ-Arch Buffer		Leakage	Injection
Prediction history	PHT	BranchScope [15], Bluethunder [24]	Spectre-PHT [38]
	BTB	SBPA [1], BranchShadow [40]	Spectre-BTB [38]
	RSB	Hyper-Channel [8]	Spectre-RSB [39, 44]
	STL	—	Spectre-STL [23]
Program data	L1D	Meltdown [42]	LVI-NULL
	L1D	Foreshadow [61]	LVI-L1D
	FPU	LazyFP [57]	LVI-FPU
	SB	Fallout [9]	LVI-SB
	LFB/LP	ZombieLoad [53], RIDL [67]	LVI-LFB/LP

LVIは、μ-Archバッファからの値の注入という新しい攻撃を実現するものである（図は[8]より引用）

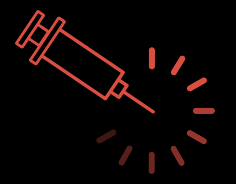
Load Value Injection（4/4）



- μ-Archバッファを**秘密情報の存在するアドレス**で汚染し、過渡的実行でそのアドレスを注入し直接**キャッシュ**に**秘密依存の痕跡を残す**LVI手法を**ユニバーサルリードガジェット法**と呼ぶ

- 一方、μ-Archバッファを攻撃者の選んだ**攻撃コードのアドレス**で汚染し、**ret等**における**フォールトロード**に伴う過渡的実行でそのアドレスを注入し**攻撃コードに誘導**するLVI手法を**制御フローリダイレクトガジェット法**と呼ぶ
  - 誘導後に攻撃コードで秘密情報を漏洩させるような処理を行う

LVIに悪用できるμ-Archバッファ（1/3）



■L1Dキャッシュ

Foreshadowで説明した通り。

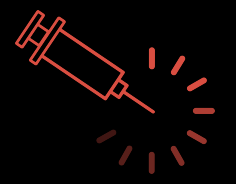
■ラインフィルバッファ（LFB）

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[9]

■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。

LVIに悪用できるμ-Archバッファ（2/3）



■ストアバッファ（SB）

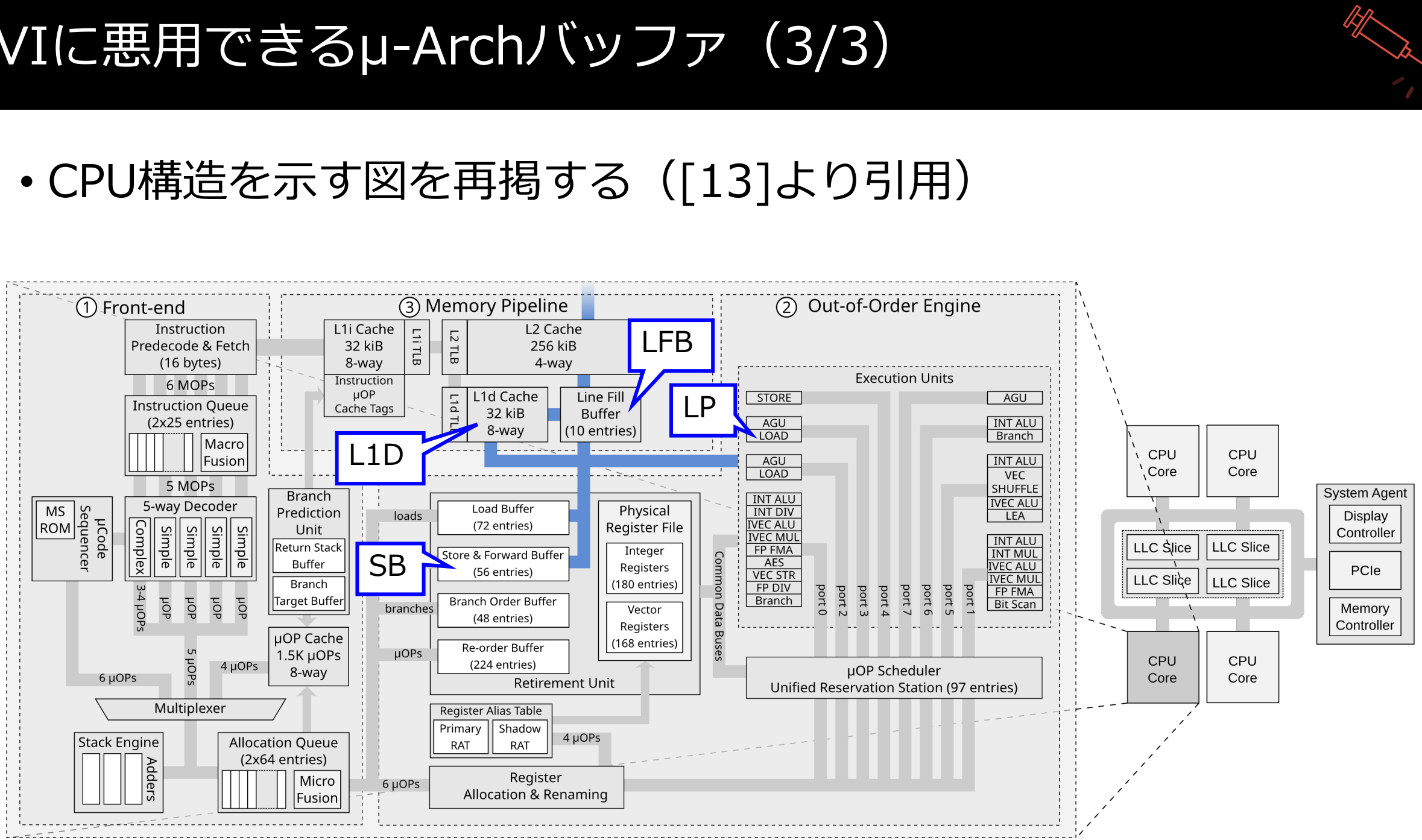
未処理のストアデータとアドレスを追跡（保持）するバッファ。
準備・状況が整ったら、インオーダーで実際にストアを行う。

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。**ストア操作による汚染が可能である。**

■浮動小数点演算処理装置（FPU）

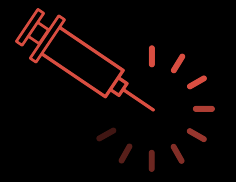
その名の通り、浮動小数点演算を専門に行う、CPUに内蔵されているユニット。

LVIに悪用できるμ-Archバッファ（3/3）



- CPU構造を示す図を再掲する（[13]より引用）

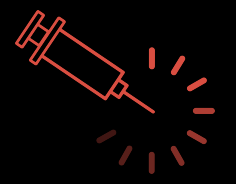
LVIの攻撃力（1/4）



- **攻撃対象のドメイン内**（例：ユーザ空間であるEnclave）で攻撃のほとんどが**完結**するという特徴がある
  - よって、Foreshadow等と異なり、**攻撃対象のコード**において**LVIを発生**させる前提であり、Foreshadowのように攻撃者が**自前のコードを用いて攻撃を発動**させる事は**原則想定されていない**

- よって、**コンテキストスイッチ**（例：ユーザ→カーネルへの切り替え）時に**μ-Archバッファをフラッシュ**するような、**従来のMeltdown型攻撃への対策**は**LVIには効かない**

LVIの攻撃力（2/4）

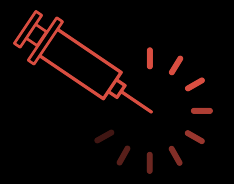


- **Spectre攻撃**に対する**対策**としては、**メモリ曖昧性解消機能**(*)の無効化（Spectre-STLの場合）や、**分岐予測器**に**汚染に対する耐性を付与**するような対策が取られている

  (*)英語でmemory disambiguation. 詳細は省略

- しかし、LVIは**Meltdown的な挙動**により**μ-Archバッファから漏洩**した値を後続の命令に対して**直接注入**するため、**分岐予測関係を補強**する上記のような**Spectre対策**は**根本的に無意味**である

LVIの攻撃力（3/4）

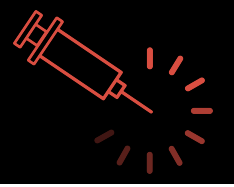


- また、Spectreについてはコード内で**過渡的実行攻撃が発生しそうな部分**に**lfence命令**を**挿入する事で対策**する方法も有名である
  - **lfence命令**：この命令が挿入された場所より**後の命令**が、**lfence命令よりも先**に**アウトオブオーダー実行**される事を**防ぐ**ための命令

- **LVI**でも後続の命令を過渡的に実行する事を抑止するために**lfenceが有効**であるが、挿入すべき**対象があまりにも多く**（例：ret命令）、全て対応していると**かなりのオーバーヘッド**となる

- しかも、性質上**軽減策**による**lfenceの導入が困難なロード命令**も存在するため、**完全な対応**は**不可能に近い**

LVIの攻撃力（4/4）

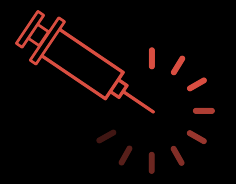


- 結局の所、**ロード命令**において**フォールト**や**アシスト**が発生した際に**過渡的実行が発生しない**よう**CPUのシリコンレベルで対策**をしない限り、**LVIの根絶**は**限りなく無理**に等しい

- 一方で、**Meltdown挙動を発生**させ、それにより注入された値を**後続**の**過渡的命令に使わせる**という**極めて難易度の高い攻撃**であり、攻撃の**実用性は低い**と言わざるを得ない

- このように、攻撃力というよりは**攻撃可能範囲が極めて広い**という点が、**LVIの厄介な性質**である

LVIのPoC攻撃（1/3）

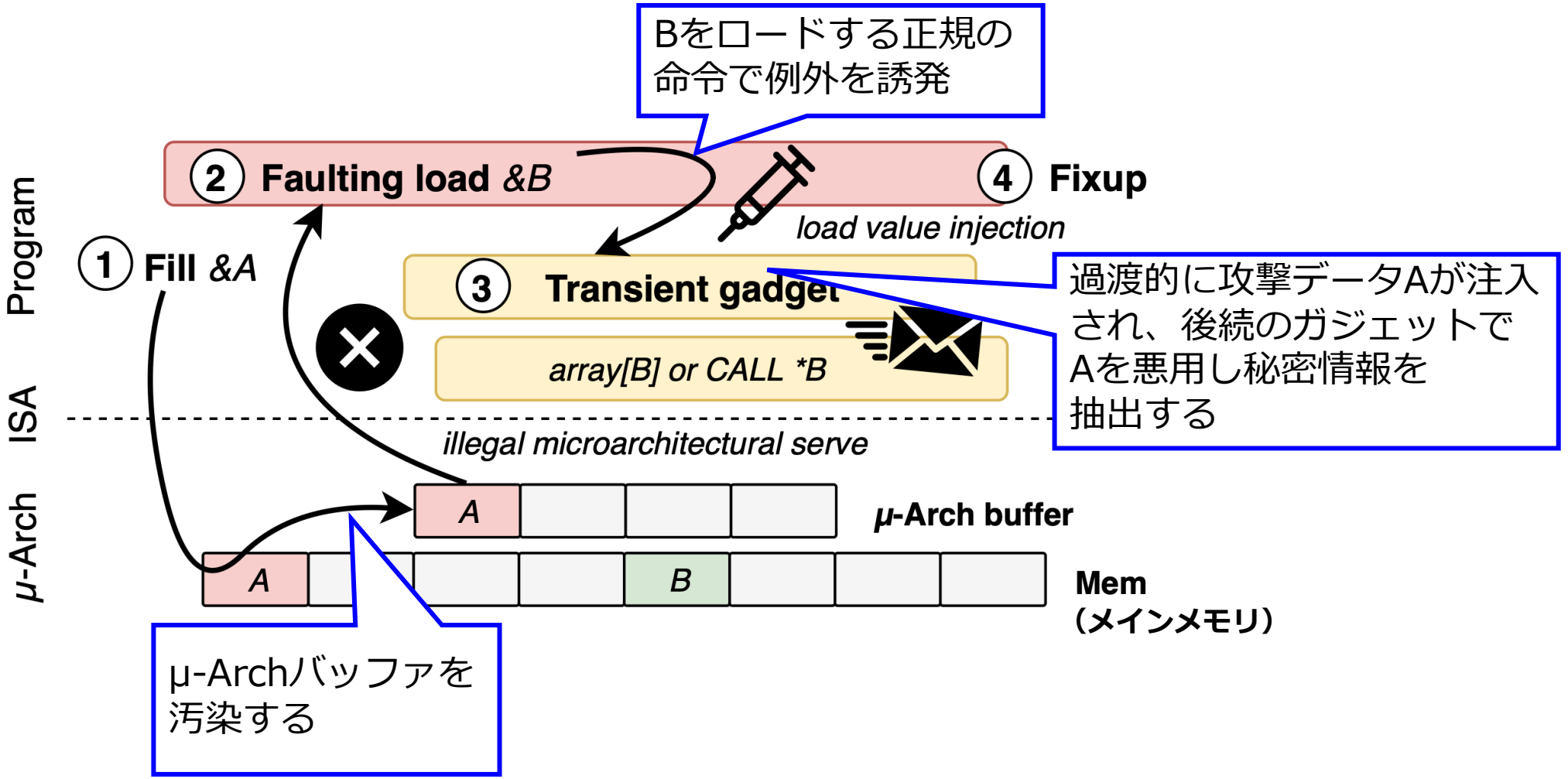


- Foreshadowの時と同様、まずはLVIの概念実証的な攻撃コードについて説明を行う

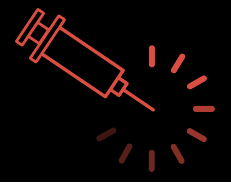
- LVIは、主に以下の3フェーズにより構成される：
  - **フェーズI（P1）**：μ-Archバッファの汚染
  - **フェーズII（P2）**：ロードにおけるフォールト/アシストの誘発
  - **フェーズIII（P3）**：ガジェット（攻撃に利用可能なコード）ベースでの秘密情報の転送（漏洩）

LVIのPoC攻撃（2/3）

- LVIの概要図は以下の通り（図は[8]より引用）



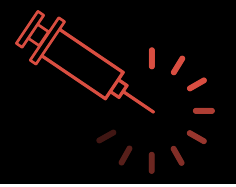
LVIのPoC攻撃（3/3）



- 攻撃対象マシンで動作する以下のコードに対してLVI攻撃を行い、秘密情報を抽出する事を考える：

```
void call_victim(size_t untrusted_arg) {
    *arg_copy = untrusted_arg;
    array[**trusted_ptr * 4096];
}
```

LVIのPoC攻撃 – P1ガジェット

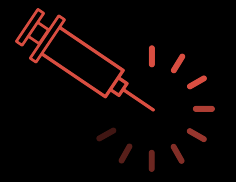


- 2行目の*arg_copy = untrusted_arg;により、64ビット（=ポインタのサイズ）の**信頼不可能な値（攻撃に使用する値）**を**信頼可能なメモリ**（Enclave内のスタック等）に**コピー**している

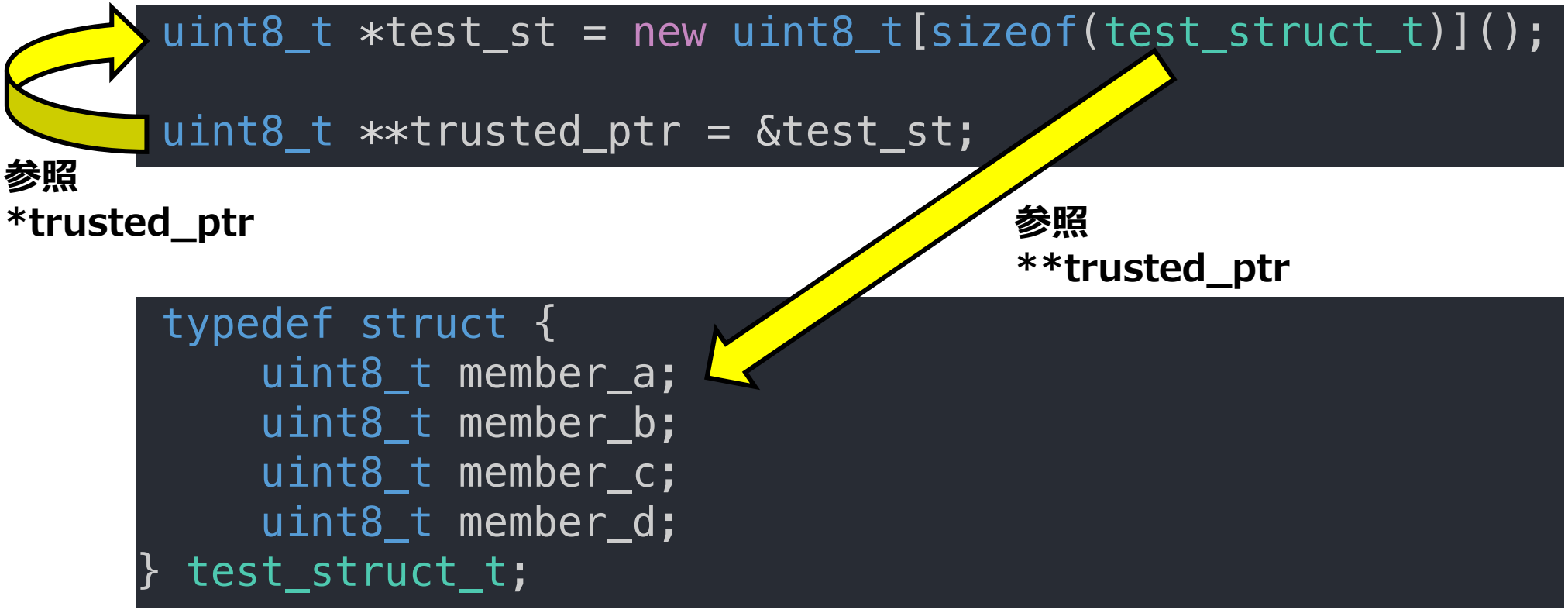
- この動作により**メモリへのストアが発生**するため、**ストアバッファ**を攻撃に使用する（=**注入する**）**値**で**汚染**できる

- よって、この2行目のコードはμ-Archバッファを汚染するための**P1ガジェット**である事になる
  - arg_copyはコピーによるSBの汚染を発生させるためだけに使ったので、これ以降は登場しない

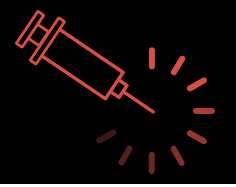
LVIのPoC攻撃 – P2ガジェット（1/4）



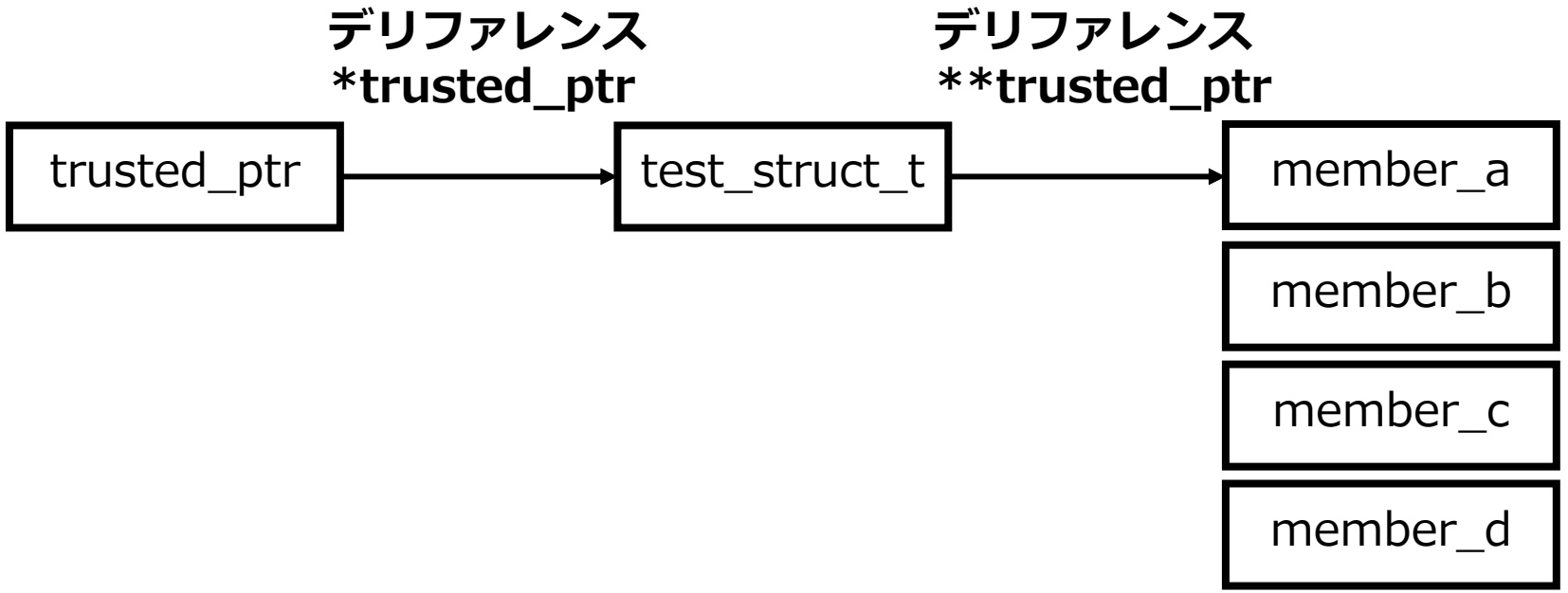
- 3行目の**trusted_ptrは**二重ポインタ**であり、例えば（動的に確保した）構造体へのポインタ等が具体例として挙げられる



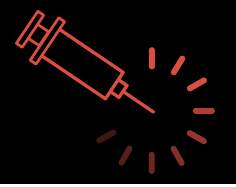
LVIのPoC攻撃 – P2ガジェット（2/4）



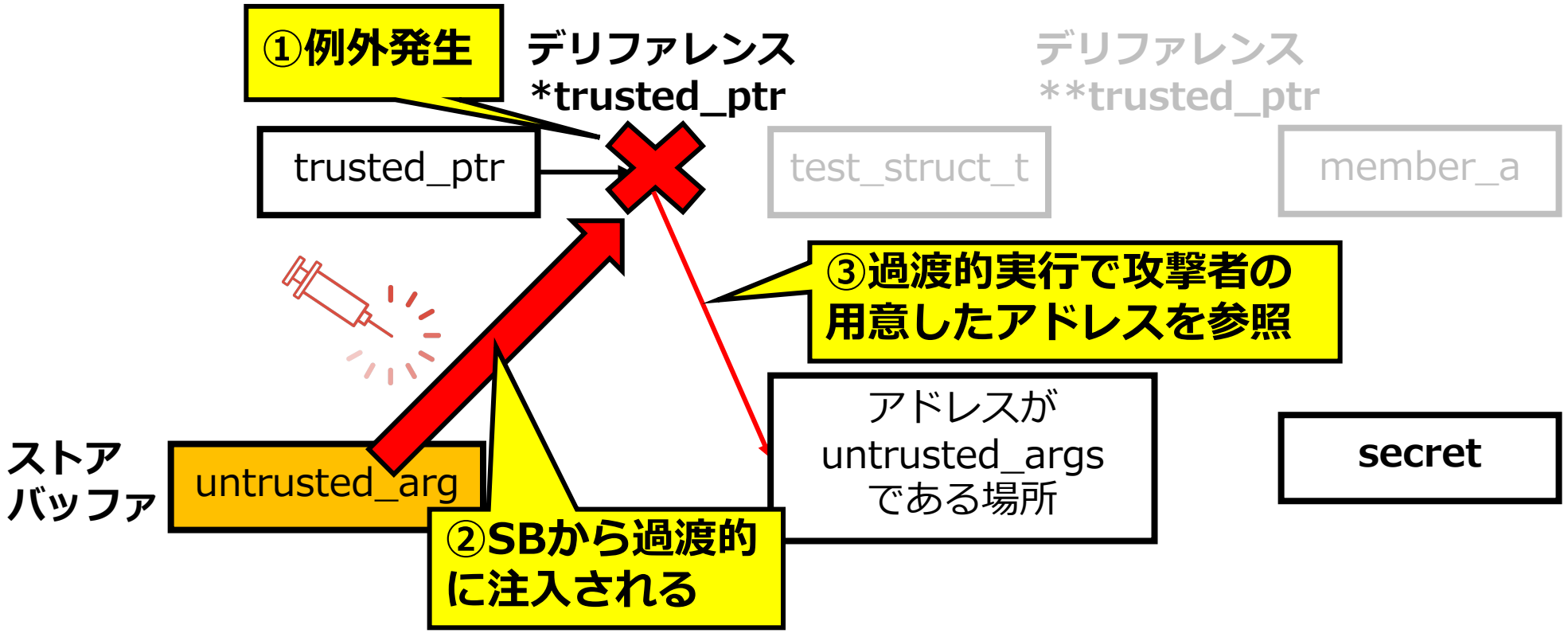
- 二重ポインタtrusted_ptrについて、*trusted_ptrのように1段階の**参照先取得**（**デリファレンス**）を行うと、本来は構造体の先頭アドレスが手に入る



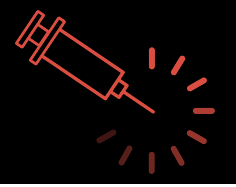
LVIのPoC攻撃 – P2ガジェット（3/4）



- ここで、1段階のデリファレンスである***trusted_ptr**において**フォールト**または**アシスト**を発生させる。すると、それに伴う**投機的実行**において、**SBから流れ込んできた値**がデリファレンスにおける参照先として**後続の過渡的命令**で**過渡的に使用**されてしまう



LVIのPoC攻撃 – P2ガジェット（4/4）

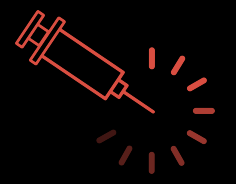


- この1段階目のデリファレンスでフォールトまたはアシストを発生させて過渡的実行を誘発できるため、**このデリファレンス**は**P2ガジェット**であると見なす事が出来る

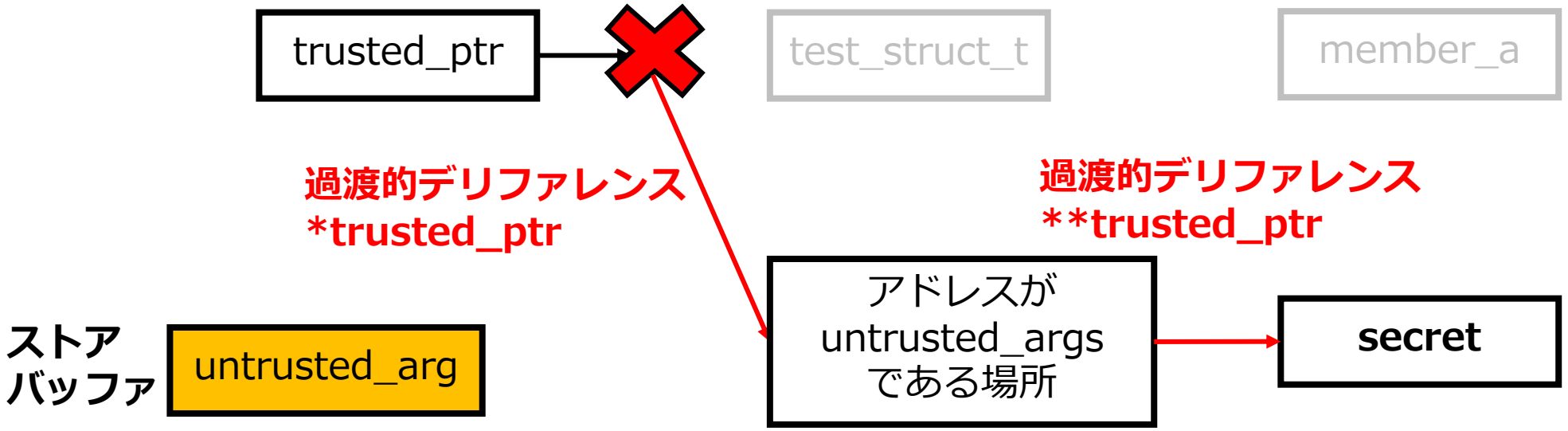
- フォールトまたはアシストを発生させるには、何度か登場している**mprotect**を用いたり、**ページテーブルエントリ（PTE）を直接改竄**する等の様々な手法が利用可能である

- ストアバッファからの値の注入を誘発しているので、このPoC攻撃は後述の**LVI-SB**というバリアントに属する事が分かる

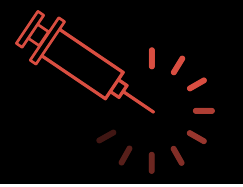
LVIのPoC攻撃 – P3ガジェット（1/2）



- **2段階目のデリファレンス**は、注入された**untrusted_argsをベースに実施**されるため、untrusted_argsの場所に存在する**秘密情報をフェッチ**してしまう
  - ちなみに、過渡的実行が始まり、アーキテクチャの処理が追いついて棄却されるまでの**攻撃可能時間**を**過渡的実行ウィンドウ**（Transient Window）という



LVIのPoC攻撃 – P3ガジェット（2/2）

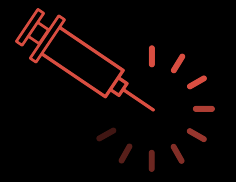


- 攻撃対象コードの3行目において、 **監視用配列array**に対して array[**trusted_ptr * 4096];のように**秘密バイト依存**の**ページアクセス**を行っている

- ここまで来れば**Foreshadowの時と同様**、過渡的実行が棄却された後に**キャッシュに痕跡が残る**ため、FLUSH+RELOAD**キャッシュサイドチャネル攻撃**で**秘密バイトを抽出**できてしまう

- このPoC攻撃では、2段階目のデリファレンスから監視用配列への秘密依存のアクセスまでが**P3ガジェット**である

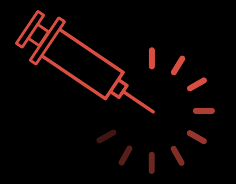
LVIのバリアント



- ロード命令に対する注入をどこから行うかに応じて、LVIには
  いくつかの**バリアント**（変種・種類）が存在する

- 本ゼミでは、原論文に倣い**以下のLVIバリアント**について解説する：
  - **LVI-L1D**
  - **LVI-SB**
  - **LVI-NULL**

LVI-L1D（1/2）



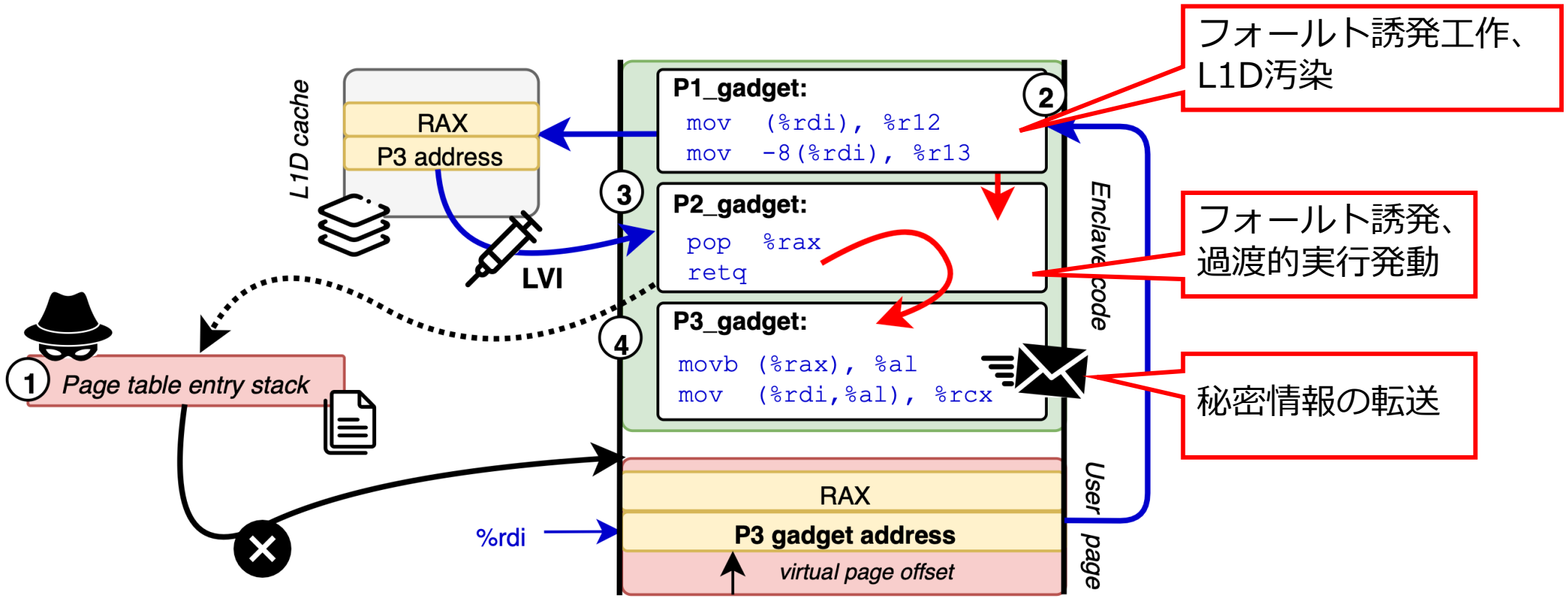
- その名の通り**L1D**から**値の注入**を行うLVIバリアント

- L1Dからの漏洩を注入に転用する事から、LVI-L1Dは**逆Foreshadow攻撃**であると捉える事も出来る
  - 恐らくこの世に存在するSGX攻撃の中でも頂点に君臨するレベルの複雑度

- Foreshadowに対する対策が適用されている環境では、LVI-L1Dによる攻撃を成立させる事は出来ない

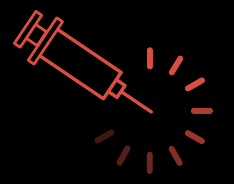
LVI-L1D（2/2）

- LVI-L1Dの概要図は以下の通り（[8]より引用）：



アセンブリはAT&T形式（[命令] <ソース> <宛先>）で書かれているので注意

LVI-L1D – P1ガジェット（1/3）

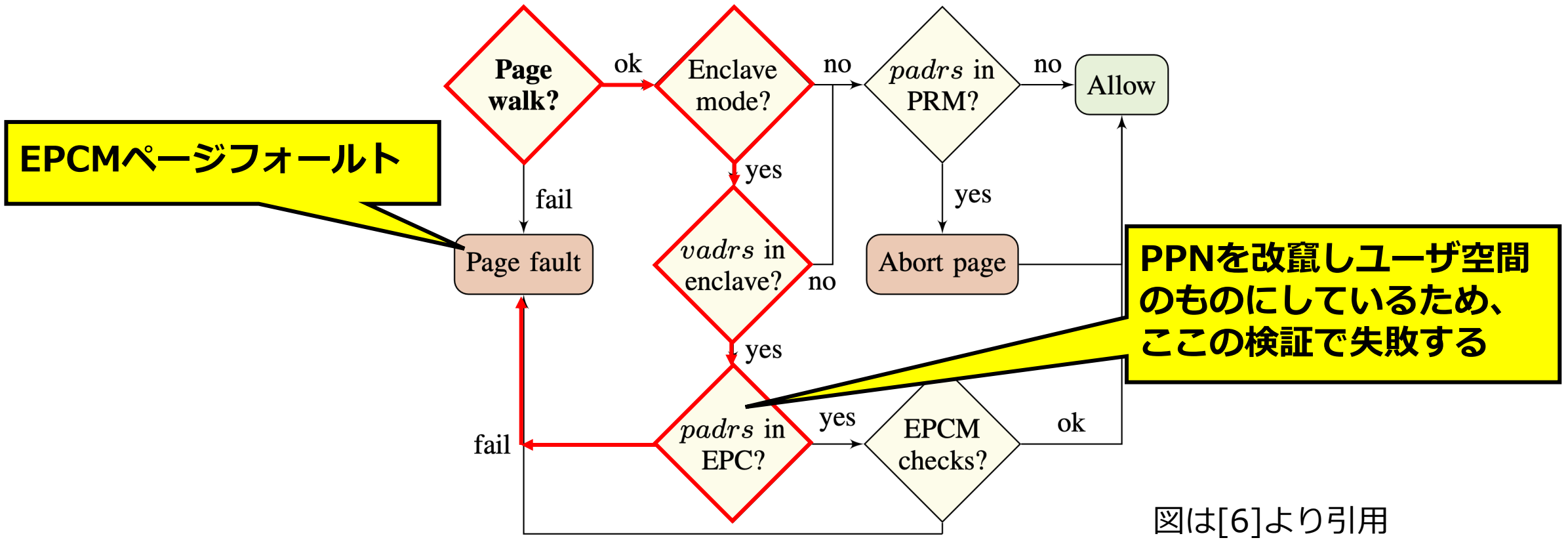


- ページフォールトシーケンス手法やSGX-Stepを用いる事で、Enclaveの実行を**P1ガジェットの直前**まで正確に進める

- その後、**P2_gadgetのretq命令**がロードする**スタックページ**のPTE内の**PPN（物理ページ番号）**を、**P3ガジェットのアドレス値**を格納する**ユーザ空間上のページ**（以下、**ページU**と呼ぶ）のものに**改竄**する（前図①）
  - アドレス値はuintptr_tをイメージすると分かりやすい

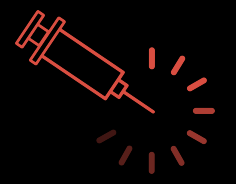
LVI-L1D – P1ガジェット（2/3）

- このPPNの改竄により、P2における**retq**の実行時に、Enclaveの**正常な範囲の外**へ飛ぼうとした事による、**EPCMページフォールト**と呼ばれる特殊な**ターミナルフォールト**が発生する



図は[6]より引用

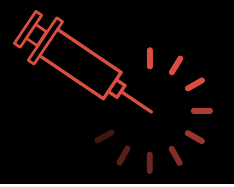
LVI-L1D – P1ガジェット（3/3）



- その後、攻撃対象のEnclave内に存在する、例えば**out属性**の**OCALL引数**（=**攻撃者の用意した値で、P3ガジェットのアドレス値**）をコピーするような**P1ガジェット**により、**ページU**上の**攻撃用の値**を**L1Dにキャッシュ**させる（概要図②）

- 前述の図においては、**ページUの保持する値**（=**P3ガジェットのアドレス値**）を%r12及び%r13レジスタにmovする事で**L1Dへのキャッシュ**を行っている

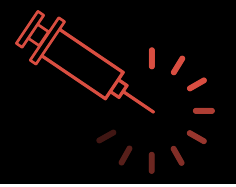
LVI-L1D – P2ガジェット（1/2）



- P2ではまず、P2の載っている**Enclave内関数**からの**戻り値**（前図のシナリオでは**これが秘密情報**である）をスタックから**%raxにポップ**している
  - 前図のP1・P2ガジェットは、別のEnclave内関数から呼び出されているEnclave内関数であるのが前提である

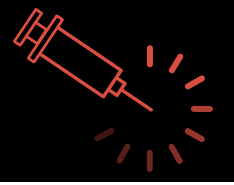
- P1でのPPN改竄の仕込みにより、**P2_gadgetのretq**が実行されると、**Enclaveの仮想ページ**に対し**Enclave外のページUの物理ページ**が割り当てられているため、**EPCMページフォールト**が発生する（前図③）
  - ここでEPCMページフォールトに伴う**過渡的実行が発生**する

LVI-L1D – P2ガジェット（2/2）



- しかし、これに伴う**過渡的実行**において**ページUのアドレス**をベースに**L1Dへの問い合わせ**が行われ、**P1**で**ページUはキャッシュ済み**であるため、**過渡的なretqのジャンプ先**としてページUの値（＝**P3ガジェットのアドレス**）が**使用**されてしまう

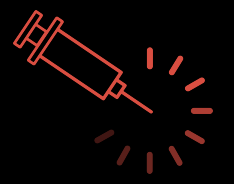
LVI-L1D – P3ガジェット（1/2）



- P2により**同一Enclave内**であれば**どこにでも過渡的に制御フローを転送**できるため、**Enclave内に存在**する、**攻撃者の選択した P3ガジェット**に**転送**させる事が出来る
  - P3ガジェットの位置の指定は、ページUが保持する、P3ガジェットのアドレス値により行う

- 後はそのP3ガジェットを用いて**キャッシュに痕跡を残し**、過渡的実行リタイア後に**キャッシュサイドチャネル攻撃**で**秘密バイトを観測**出来てしまう

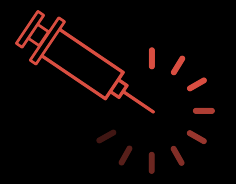
LVI-L1D – P3ガジェット（2/2）



- 前図のP3ガジェット（④）では、**秘密バイトである戻り値をP2でポップ**して**格納**した**%rax**を、**%rdiに対するインデックス**として使用し、そのアクセスにより**秘密バイトの痕跡を残している**

- 元々%rdiには**P3ガジェットのアドレス値**が格納されていたが、P3フェーズに入ると%rdiに**何が入っていたかは関係ない**
  - 言わばここでは%rdiを**再利用している**だけであり、**監視用配列として使用**しているだけで元々何が入っていたかは既に**この時点で無意味**である

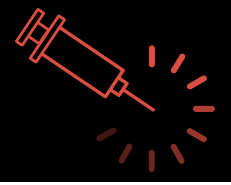
LVI-SB（1/3）



- 前述のLVIのPoC攻撃も属しているバリアントのLVI攻撃であり、**ストアバッファ**からの**値の注入**を悪用する攻撃

- Fallout攻撃[10]の論文により、**SB**に対する**Meltdown挙動**を取る場合、**ロード命令**における**ページオフセット**（対象アドレス下位12bit）が**最近の未処理のストア**のそれと**一致**しなければならない事が判明している

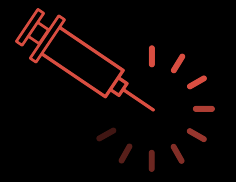
LVI-SB（2/3）



- 攻撃対象として、**Edger8r**が生成するエッジコードの以下の例を取り上げる：

```
; %rbx: user-controlled argument ptr (outside enclave)
sgx_my_sum_bridge:
  ...
  call my_sum ; compute 0x10(%rbx) + 0x8(%rbx)
  mov %rax,(%rbx) ; P1: store sum to user address
  xor %eax,%eax
  pop %rbx
  ret ; P2: load from trusted stack
```

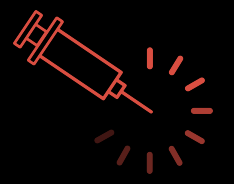
LVI-SB（3/3）



- **Enclave外**の**ポインタ**（例：配列の先頭ポインタ）を**引数**として受け取り、Enclave内でそのポインタ先が含む**2つの値を加算**して引数経由で**リターン**する**ECALL関数**についての**エッジコード**である

- **加算結果**を**ストアバッファ**に格納させ、8行目の**ret命令**で**過渡的実行を発動**させる事で、**SBからの注入**により**リターン先**として攻撃者の選択する**P3ガジェットに飛ぶ**ように仕向ける攻撃を考える

LVI-SB – P1ガジェット

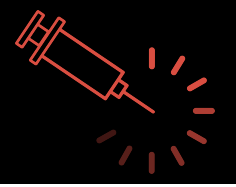


- 3行目の「...」部では、結果の返却にも用いる**引数ポインタ**（[in, out]属性をイメージすると分かりやすい）が**Enclave外に存在**しているかを**確認しているだけ**である
  - よって、**攻撃者が用意**した**引数の値**が**Edger8r**によって**阻害される事はない**

- 結果として、攻撃者はこの引数の**加算値**（=**P2**で**ret先**として**過渡的にSBから注入**される**アドレス値**になる）をEnclave外で**任意に設定（改竄）**し、前述の**ページオフセット一致の制約**を**容易に満たす**事が出来る

- この性質を利用し、攻撃が上手く行くように引数を渡し、my_sum関数での加算結果をSBに格納させればP1は完了

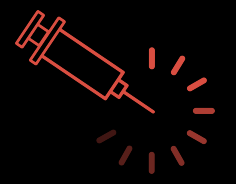
LVI-SB – P2ガジェット



- 4行目のcall my_sumの後に**Enclaveを中断**し、**8行目のret**で参照する事になる**Enclaveスタック**の**ページ**での**フォールトやアシストを誘発**する
  - 論文中では、ここではこのスタックページに対応するPTEのAccessedビットかスーパーバイザビットをクリアする事でこれを実現すると述べている
  - ただし、今まで通りPresentビットをクリアするのでは不可能であるとは書かれておらず、このケースに限ってPresentビットクリアが通用しない理由も無さそうであるため、Presentビットによる誘発も可能そうである

- 8行目で**実際にret命令が呼ばれる**と、**フォールト**または**アシスト**が**発生**し、過渡的実行でSBから**引数同士の加算値**（＝**P3ガジェットの位置**を指すように先程している）が**注入**される

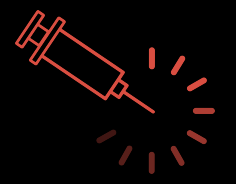
LVI-SB – P3ガジェット



- あとはLVI-L1Dの時と同様、**任意のP3ガジェット**に**制御フローをリダイレクト**出来ているため、それを用いて**秘密情報の漏洩**を行える

- 制御フローリダイレクトに限らず、ユニバーサルリードガジェット法的にLVI-SB攻撃を行う事も勿論可能である

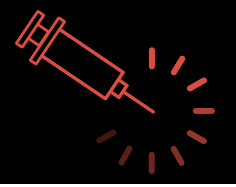
LVI-NULL（1/7）



- Meltdownへの対策を行っている直近の世代のCPUでは、過渡的実行で**Meltdown的に漏洩する値**を**ダミーの0x00**に**強制的に置き換える**事で、秘密情報が漏洩する事を防ぐようにしている
  - MSRのIA32_ARCH_CAPABILITIESアドレスのRDCL_NOビットが1であればMeltdown対策済みである
  - RDCLは**Rogue Data Cache Load**の略で、**Meltdownの正式名称**。NOは文字通り否定のnoの意

- しかし、**過渡的実行に0が注入される**という**挙動自体**を**悪用**するLVI攻撃も実行可能であり、これが**LVI-NULL**である

LVI-NULL（2/7）

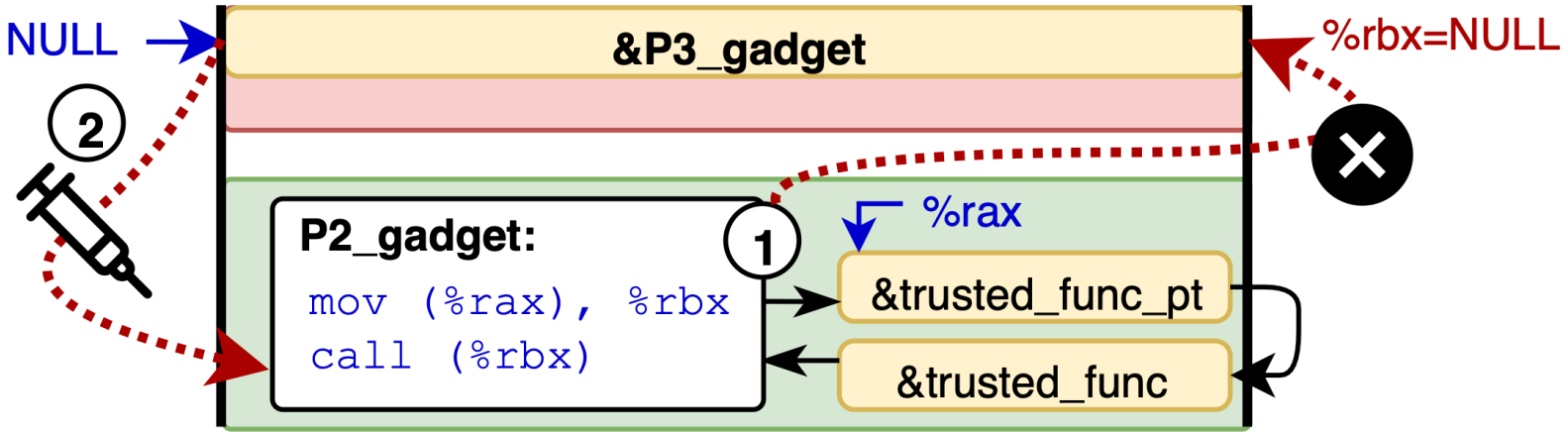


- OSや環境にもよるが、**root権限を持つ攻撃者**は**仮想アドレスNull**（=**アドレス0**）に**任意のメモリページ**を**マッピング**する事が出来る
  - 手っ取り早い方法として、**PTE内**に登録されている**仮想アドレスを0**にしてしまえば良い

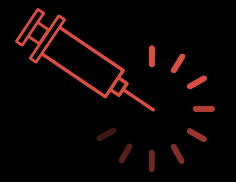
- よって、**Meltdown対策済みCPU**による、**過渡的実行**における**0値の転送**（注入）を悪用し、**仮想アドレス0に用意**した不正なポインタを通じて、**P3ガジェット**に**制御を転送**する攻撃を例として考える

LVI-NULL（3/7）

- ここでのLVI-NULL攻撃の概要図は以下の通り（[8]より引用）：



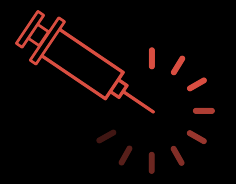
LVI-NULL（4/7）



- このシナリオでは、**関数の二重ポインタ**（**ptrのようにデリファレンスする事で関数にアクセスできるようなポインタ）において**LVIを行う**事を考える
  - 関数の二重ポインタの具体的な実例として、動的に確保した構造体のようなヒープオブジェクトに含まれる関数ポインタが挙げられる

- 二重ポインタに対して攻撃するという意味では、**LVIのPoC攻撃**の説明で示した**LVI-SBのケース**と**若干似ている**

LVI-NULL（5/7）



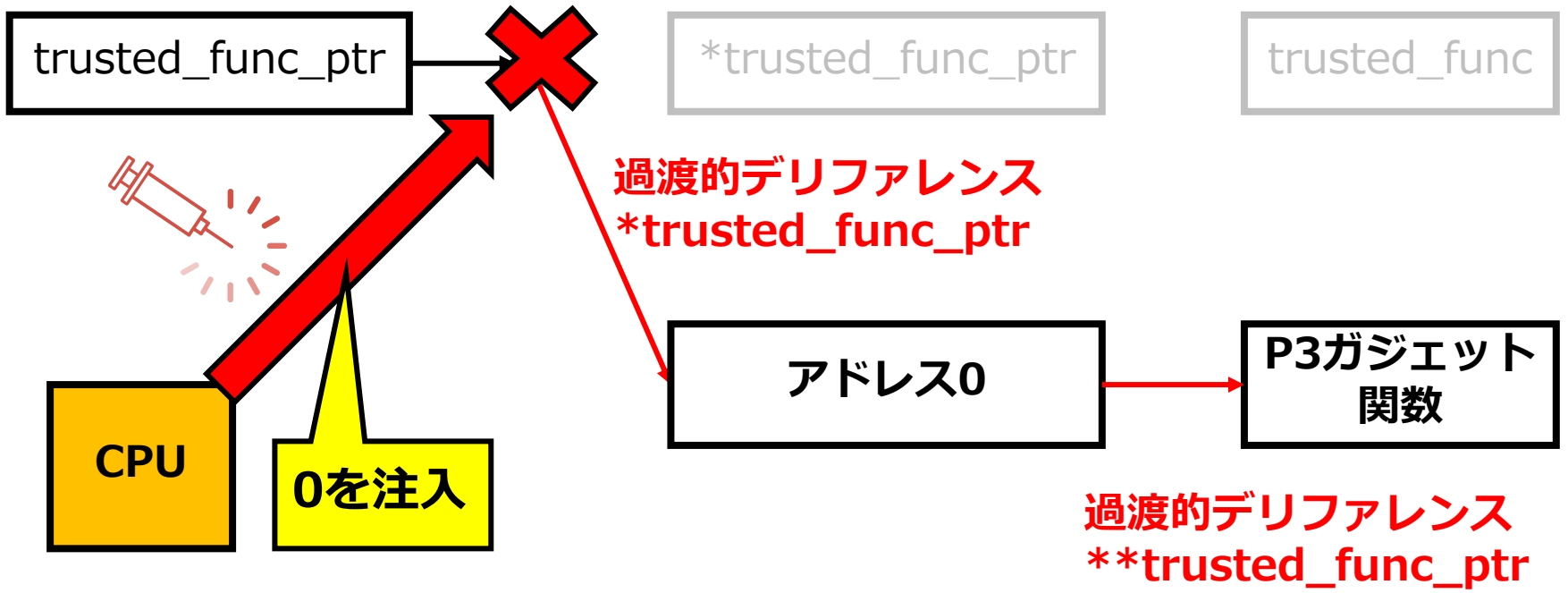
- この関数の二重ポインタの**第一段階のデリファレンス**でフォールトやアシストを誘発する事で、**過渡的実行を発生**させる

- この時、Meltdown耐性のあるCPUは過渡的実行に**ダミー値0**を転送し、結果的に**過渡的**な**第一段階**の**デリファレンス**で**アドレス0**を使用してしまう

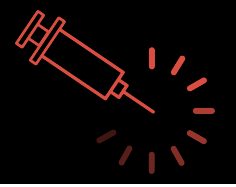
- 結果として、**第2段階のデリファレンス**では**攻撃者が用意**した**アドレス0**をベースアドレスとした場所にある**不正な関数ポインタ**を**過渡的に取得**してしまい、それにより**P3ガジェットに制御転送**されてしまう

LVI-NULL（6/7）

- このLVI-NULLの実行の様子を示した図は以下の通り：

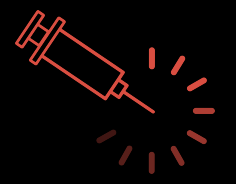


LVI-NULL（7/7）



- ここで**1段階のみの関数ポインタ**を用いると、**過渡的な関数呼び出し**が**上手く行かない**らしい
  - 詳細は不明だが、一度過渡的にEnclave外ページをデリファレンスしてからその上のEnclave外関数を呼ぶ事は出来るが、直接Enclave外関数を呼ぶのは過渡的実行上でも不可能であると推測できる
  - デリファレンス先ページを実行不可能とマークする等して時間を稼ぎ、その間にアドレス0に再配置可能なEnclaveイメージをロードすれば1段階のみの関数ポインタでも攻撃が成立する可能性はある
- LVI-NULLの場合、他のバリアントと比べても**過渡的実行ウィンドウ**が**小さい**ため、実際に**有効な攻撃**を**成功裏**に行うのは**極めて難しい**とIntelは主張している[11]

LVI攻撃例 – AES-NIへの攻撃（1/8）

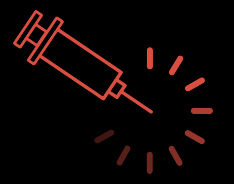


- LVIを利用したより実践的な攻撃として、**AES-NIを利用する**ようなEnclaveに対し**LVI-NULL**攻撃を仕掛ける事で、**共通鍵を抽出**するような**故障注入攻撃**を行うケースを考える
  - **AES-NI**：AES暗号の暗号化及び復号の高速化を目的に実装されている、x86の拡張命令

- **既知暗号文攻撃**（暗号文のみを知った状態で秘密を解読する攻撃）のシナリオを前提とし、正しい暗号文を復号する処理に対して故障注入攻撃を行う事を考える

- 前述のLVI-NULLの例が**制御フローの改竄**を行っていたのに対し、この例はある意味**Plundervoltに近い故障注入攻撃**を行う点でかなり毛色が異なる

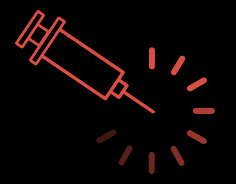
LVI攻撃例 – AES-NIへの攻撃（2/8）



- AES暗号では、暗号文を**128bit**（16バイト）の**ブロック**という単位に**分割**し、さらに各ブロックを**1マス8bit**（1バイト）とした**4×4の行列**にする

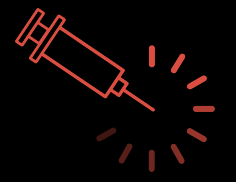
- そして、この**4×4行列**に対し、ある**一連の処理**をAESの仕様で定められている**ラウンド数**分だけ繰り返す（**鍵伸長処理**）
  - 鍵長が**128bit**である場合はこのラウンド数は**10**である

LVI攻撃例 – AES-NIへの攻撃（3/8）



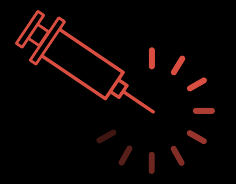
- 一連の処理とは、暗号化の場合は順に以下の通り：
  - **SubBytes**：S-Boxという置換表を参照するByte単位の置換
  - **ShiftRows**：4×4行列のn行目を(n-1)マスだけ左側にずらす。左端を超えるようなマスは右端に行くようにする（回転）
  - **MixColumns**：列単位の処理。数式が長いので省くが、XORをベースとした演算（CBCにおけるXOR処理とは別物）
  - **AddRoundKey**：state（処理中の4×4行列。上記の処理を適用した状態の途中段階の行列）と**ラウンド鍵**で、列ごとにXORを取る
- 最終ラウンドのみ、MixColumnsは実行されない

LVI攻撃例 – AES-NIへの攻撃（4/8）

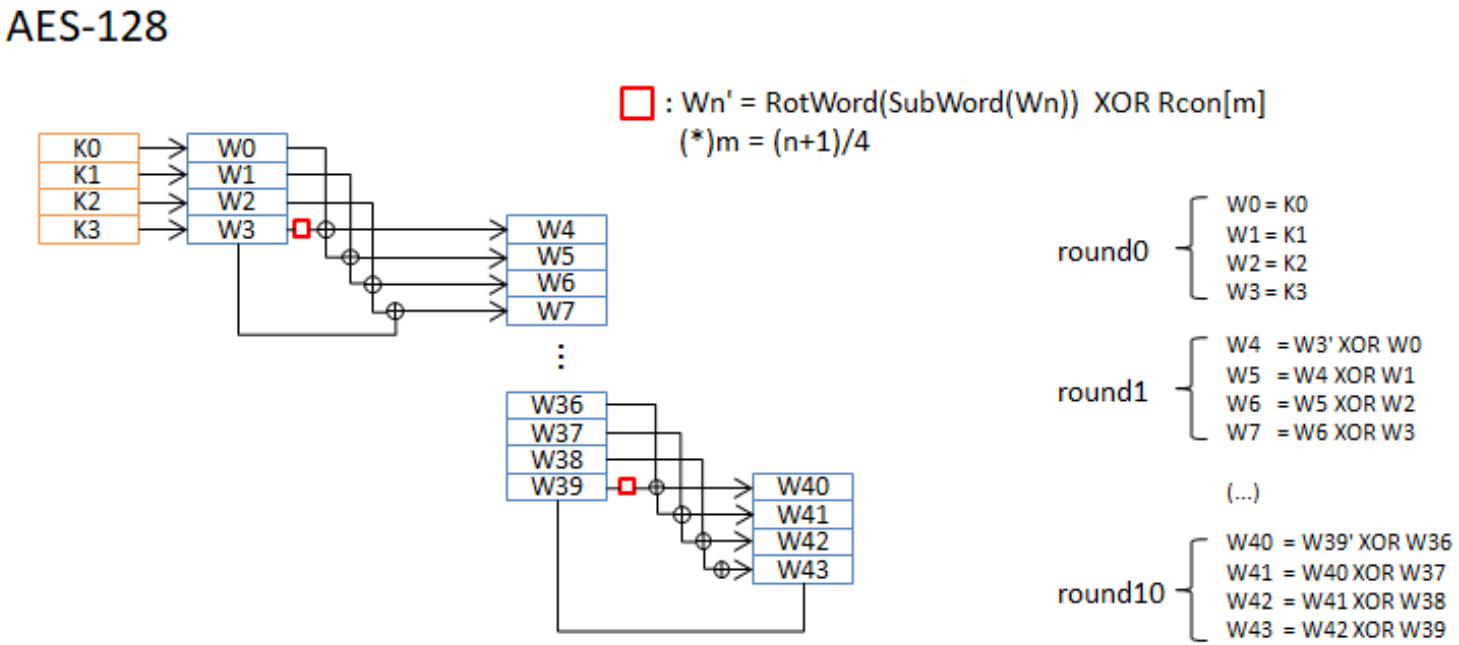


- 復号の場合は以下の通り：
  - **InvShiftRows**：暗号化時のShiftRowsの右回転バージョン
  - **InvSubBytes**：Inverse S-Boxを参照した、SubBytesの逆置換
  - **AddRoundKey**：**暗号化時と同様**
  - **InvMixColumns**：これも列単位の処理で、数式が複雑なので詳細は割愛
- 最終ラウンドのみ、InvMixColumnsは実行されない

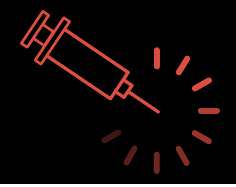
LVI攻撃例 – AES-NIへの攻撃（5/8）



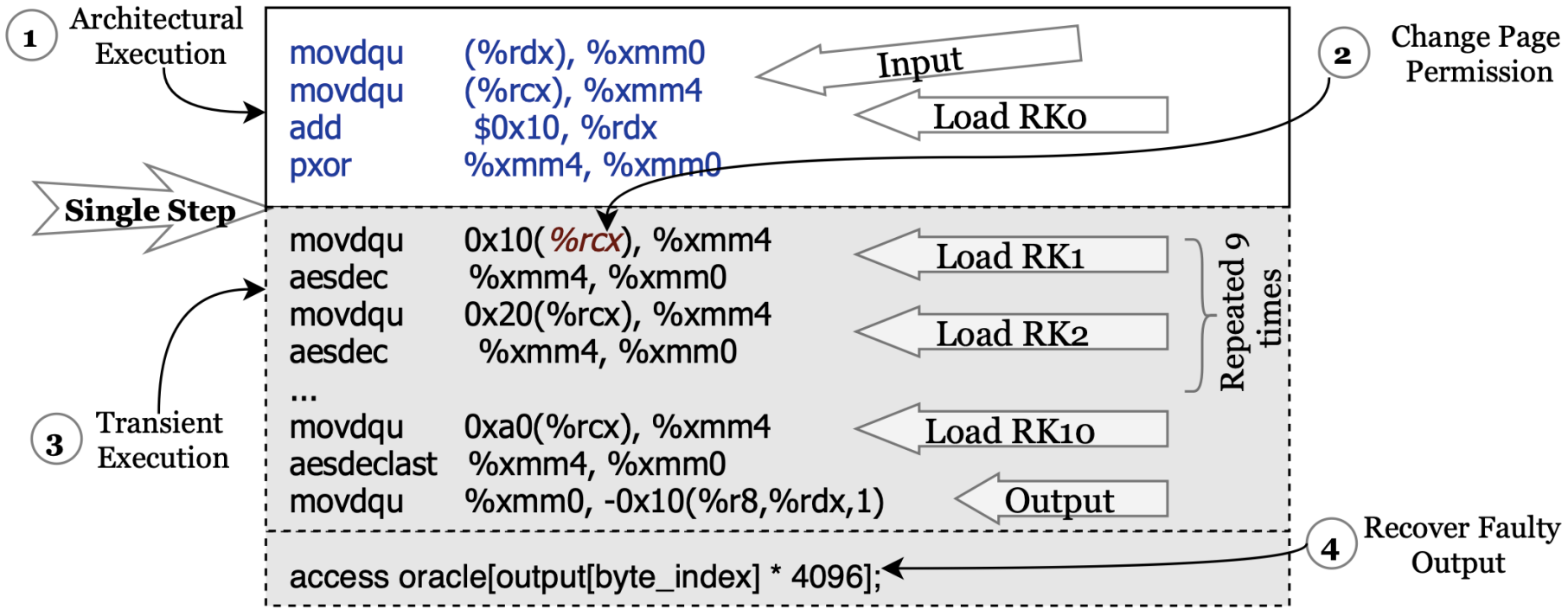
- ここで、ラウンド鍵はAESの共通鍵からラウンド分だけそれぞれ導出されるもので、以下のようにして導出される
  （図は[12]より引用）



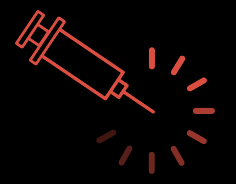
LVI攻撃例 – AES-NIへの攻撃（6/8）



- AES-NIを利用するEnclaveに対するLVI-NULL攻撃の概要図は以下の通り（図は[8]より引用）：



LVI攻撃例 – AES-NIへの攻撃（7/8）

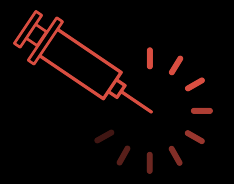


- まず、SGX-Stepを使用して**最初（第0）のラウンドを実行した後**に攻撃対象のEnclaveに**正確に割り込む**

- その後、**ラウンド鍵が格納**された**メモリページのアクセス権**を**剥奪**してから**Enclaveを再開**する

- 続くラウンド処理で**ラウンド鍵へのアクセス**により**フォールトが発生**するため、最初以外のラウンドではMeltdown対策挙動により**オールゼロのラウンド鍵**が**過渡的に使用（LVI-NULL）**されてしまう

LVI攻撃例 – AES-NIへの攻撃（8/8）

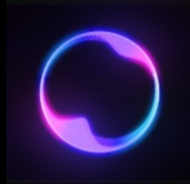


- **第0ラウンド鍵以外**の**ラウンド鍵**として**全ラウンド**にて**オールゼロの故障した鍵**で**復号**された、**故障した平文**を**キャッシュ**に残す

- 「暗号文⊕第0ラウンド鍵⊕オールゼロ鍵＝故障した平文」であるため、XORの性質より、
  「故障した平文⊕オールゼロ鍵＝**暗号文⊕第0ラウンド鍵**」となる
  - **暗号文は既知**であるため、これにより**第0ラウンド鍵が抽出**できる

- **第0ラウンド鍵**は前図の通り**共通鍵そのもの**であるため、これにより**目当ての共通鍵を抽出**できてしまう
  - 実際には単純なXORだけでなくAES特有の処理が挟まるため、AESの逆処理を行う関数を用意して共通鍵の抽出を行う

まとめ

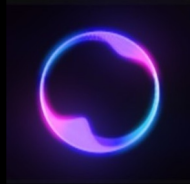


- SGX攻撃の中でも最先端の一角を担っている、Foreshadow、LVI、ÆPIC Leakの3つの攻撃についてある程度詳細に解説した

- これらの攻撃を実践するのは非常に難易度が高いため、その実践は完全にSGXを極めるのであればおすすめする

Downfall

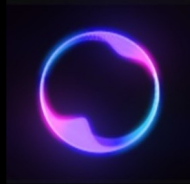
コンテキストスイッチ



- **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）

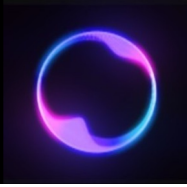


- **SIMD**：Single Instruction Multiple Dataの略。**同じ操作**を**異なるデータ**で**並列に実行**するような処理
  - 例：8個のデータを、単一のSIMD命令で並列で一気にビット反転する

- 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

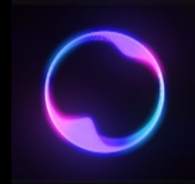
SIMDとベクトルレジスタ（2/3）



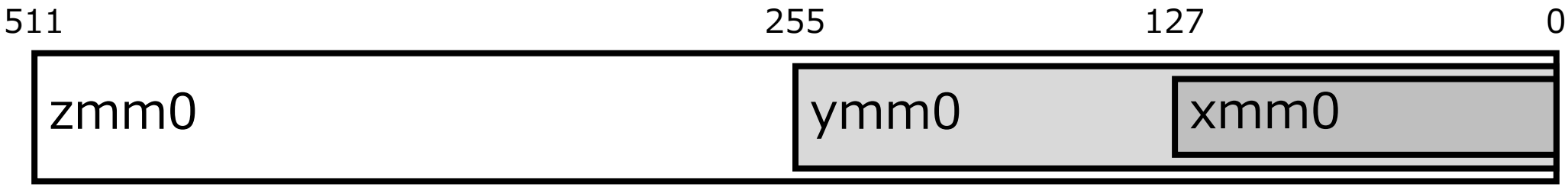
- 64ビットレジスタにおけるraxのように、ベクトルレジスタにもレジスタ名が付与されている

- 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：xmm0）

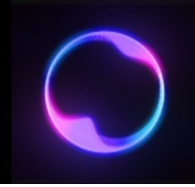
SIMDとベクトルレジスタ（3/3）



- 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）

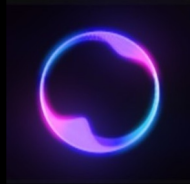


- 主に**メモリの各所に分散して存在**している非連続なデータを**効率的に収集しロード**する命令
  - **%rsi**：ベースアドレス
  - **%xmm2**：収集してロードするデータの位置を指定する、ベースアドレスに対するインデックスを格納するインデックス配列
  - **%xmm1**：マスクレジスタ。ここで0であるようなビット位置のデータは、インデックス配列に格納されていても収集を行わず無視する

    （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）

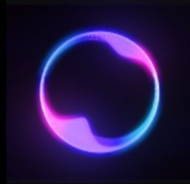


- 前ページの例では、32bitの値（**dword**）4つを収集して**128bitの**<u>**ベクトルレジスタ**</u>である**xmm3レジスタに格納**する

- 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）



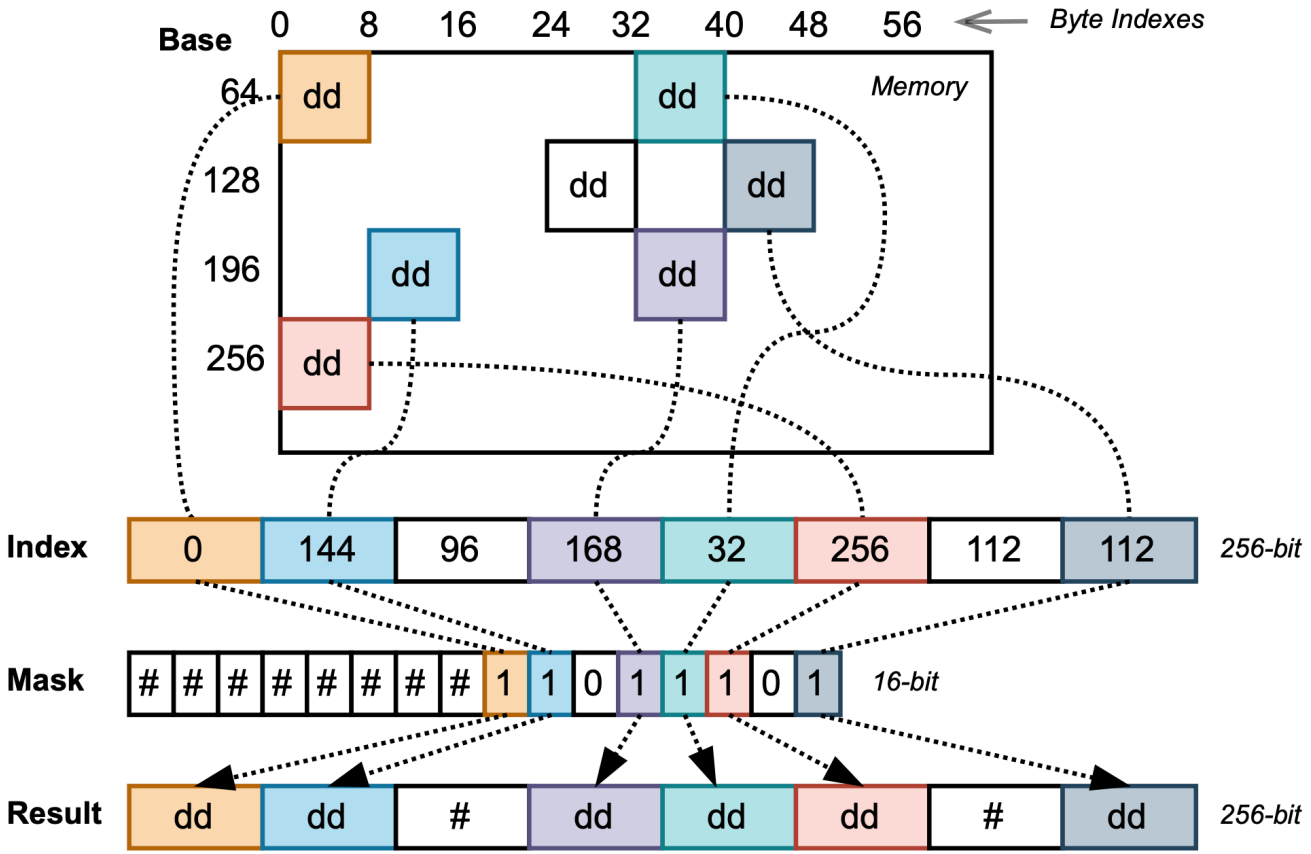
- このGather命令でメモリの各所に散らばったデータを効率的に**収集**して**ベクトルレジスタにロード**した上で、**他のSIMD命令**を実行すると**効率的にSIMD処理を行う事ができる**

- 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

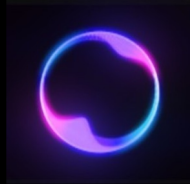
```
vpgatherdd 0(%rsi, %zmm2, 1), %zmm3{%k1} // AVX-512
```

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[14]より引用）



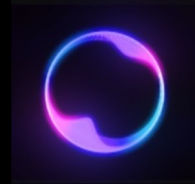
マイクロアーキテクチャ（μ-Arch）



- **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：

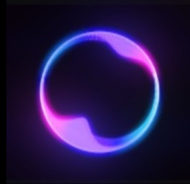
➢**0であるマスクビット**に対応するメモリ位置からは、そもそもロード処理自体行わずに**処理から排除**する。

➢**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？

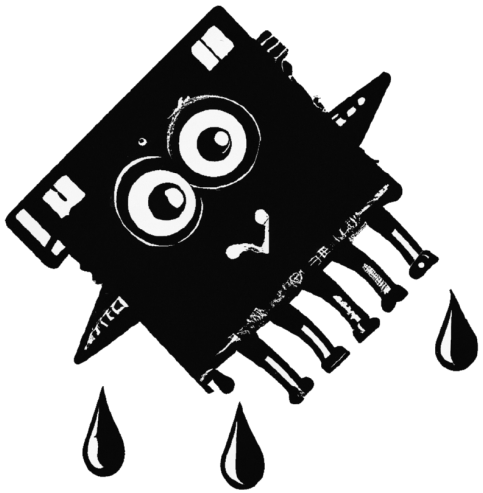
- **投機的に実行**して駄目ならアーキテクチャ状態（CPUやメモリの実際の状態）への反映時（**命令リタイア時**）に**破棄**、という挙動は、**過渡的領域**において**何かしらの脆弱性**を抱えているのでは？
  - 過渡的実行中にキャッシュ等に秘密情報に依存する値の痕跡を残す攻撃はもはや恒例である

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[14]



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

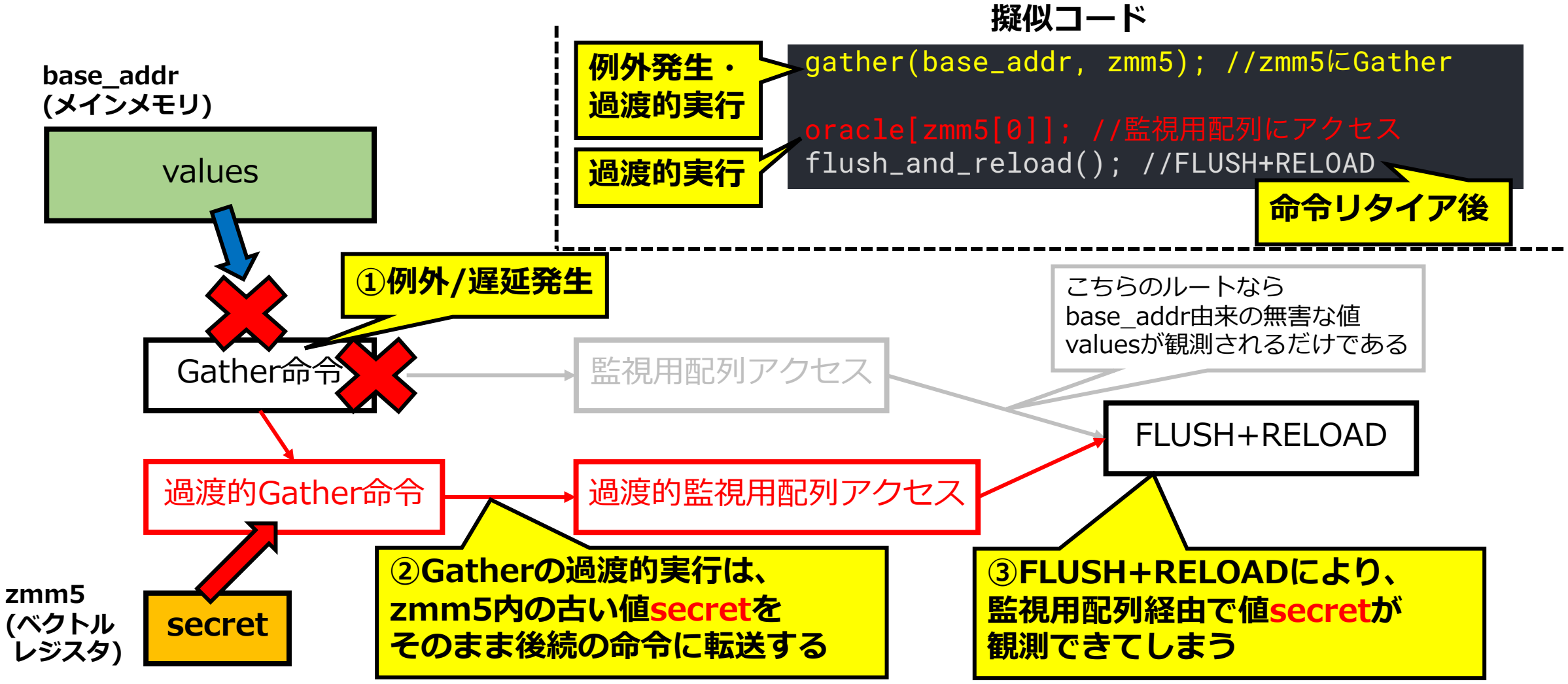
GDSのPoC実装（1/7）

- 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：

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[17]。

■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**

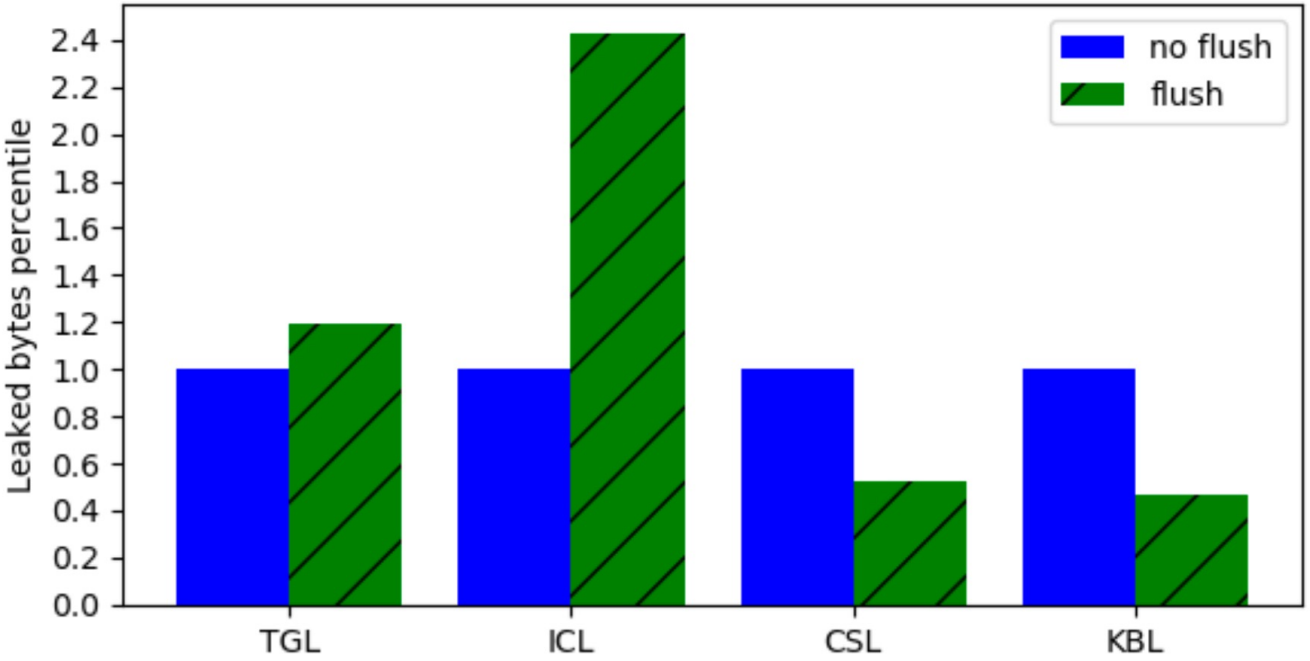
漏洩元の推測（1/4）

- 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[14]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[15]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）

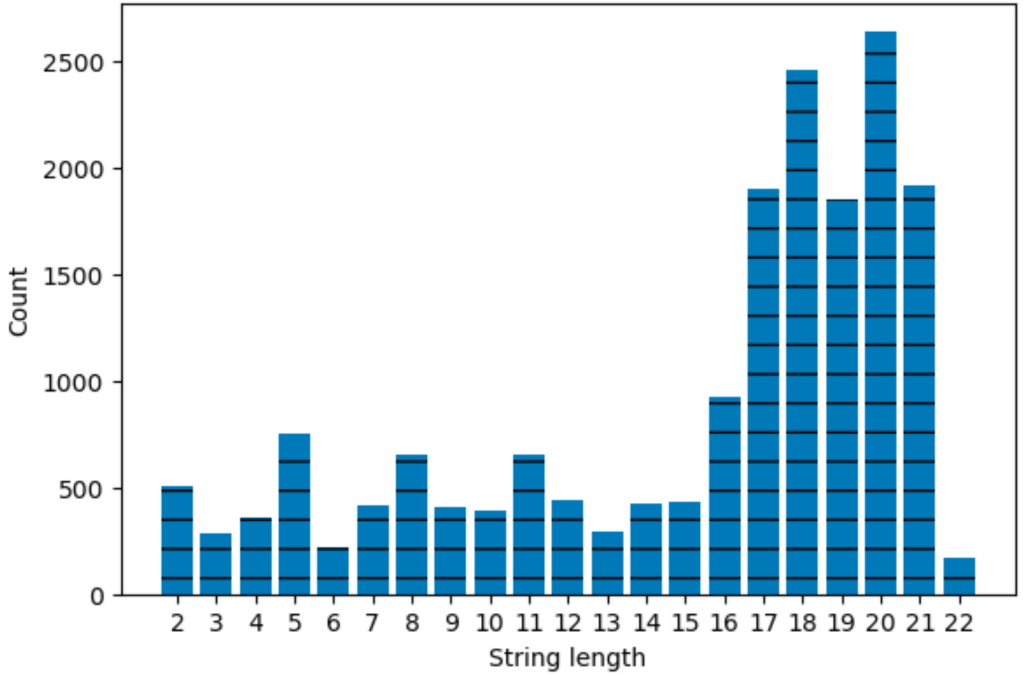
- 理論的には最大32バイト同時に漏洩させる事ができそうだが、実際にはどの程度の同時漏洩が可能であるのかを実験的に確認する

- 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[14]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

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, %rsi, %rcx * 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[18]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[19][20]。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）

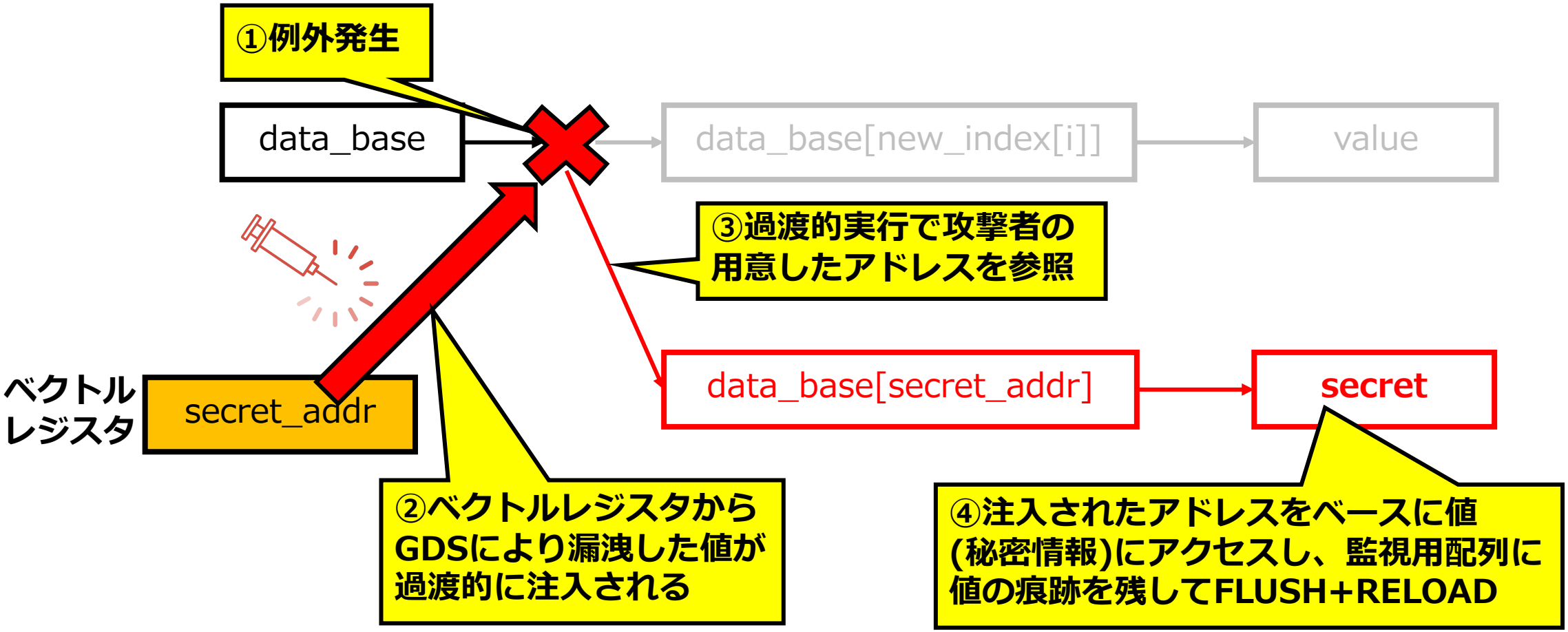
- GDSは**Meltdown型**の過渡的実行攻撃であるため、LVI同様ベクトルレジスタから漏洩した値を後続の命令への**注入に転用**できる事が推測できる

- 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[21]**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_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

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**[23]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

参考文献（1/4）

[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

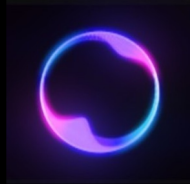
[4]”ZombieLoad Attack”, Daniel Gruss et al., https://gruss.cc/files/zombieload_36c3.pdf

[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（2/4）



[8]”LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection”, Jo Van Bulck et al., https://lviattack.eu/lvi.pdf

[9]”ZombieLoad: Cross-Privilege-Boundary Data Sampling”, Michael Schwarz et al., https://zombieloadattack.com/zombieload.pdf

[10]”Fallout: Leaking Data on Meltdown-resistant CPUs”, Claudio Canella et al., https://mdsattacks.com/files/fallout.pdf

[11]”Load Value Injection”, Intel, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/load-value-injection.html

[12]”AESを理解する”, Qiita, 2023/7/25閲覧, https://qiita.com/tobira-code/items/152befa86bd515f67241

[13]”MDS: Microarchitectural Data Sampling”, 2023/7/20閲覧, https://mdsattacks.com/

参考文献（3/4）

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

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

[16]”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）

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

[18]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（4/4）

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

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

[21]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

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

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