

10. SGX攻撃編③

Ao Sakurai

2023年度セキュリティキャンプ全国大会
L5 - TEEの活用と攻撃実践ゼミ

本セッションの目標



- SGXに対する攻撃の中でも最新の一角を担っている、「**Foreshadow** (L1 Terminal Fault)」「**LVI** (Load Value Injection)」「**ÆPIC Leak**」の3つについて解説する
- 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

Foreshadow (L1 Terminal Fault)

キャッシュ階層

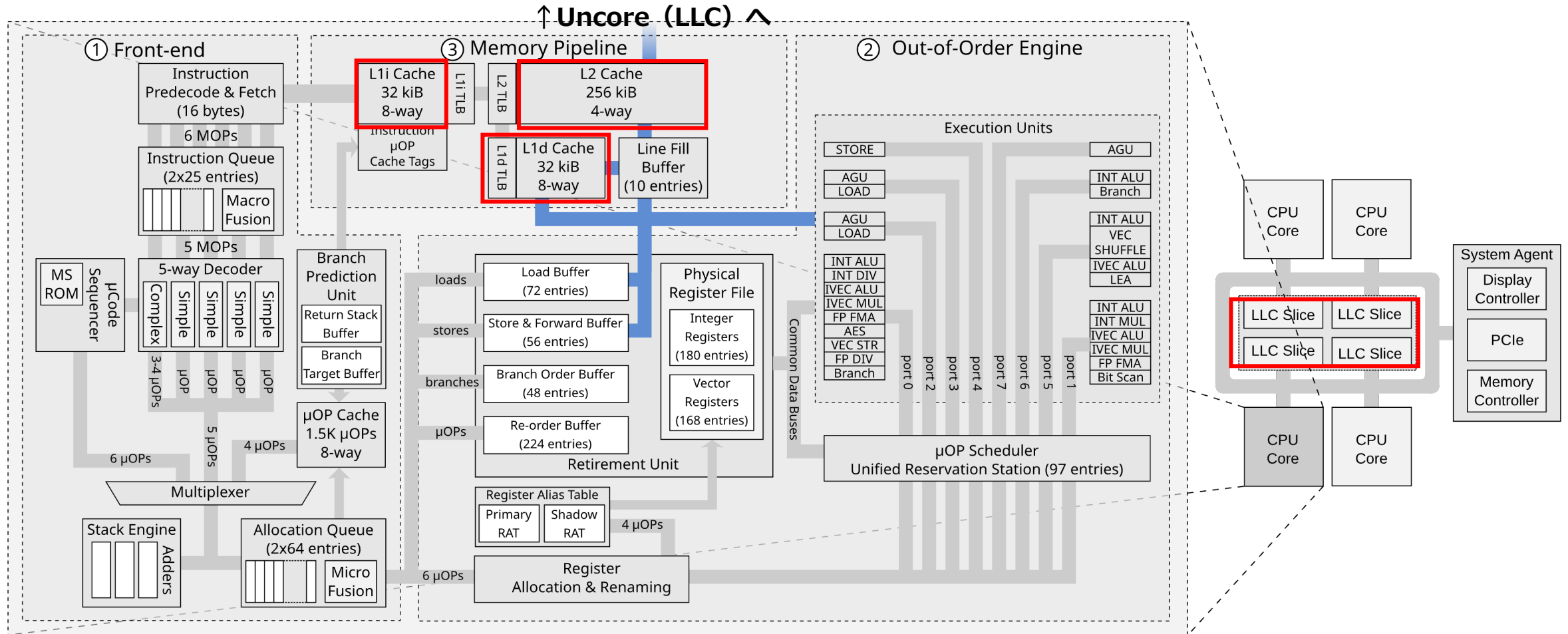


- 頻繁にアクセスするデータや命令を保持する役割を持つ、CPUパッケージ内に存在しメインメモリよりもアクセス時間の速い**キャッシュメモリ**は階層構造になっている
- 最近のCPUでは、CPUの主要部分に近い順に**L1, L2, L3キャッシュ**の3つのキャッシュにより成り立っている
- **L1キャッシュ**には、命令を保持しておく**L1Iキャッシュ**と、データを保持しておく**L1Dキャッシュ**が存在する

キャッシュ階層



- 以下の図はSkylake CPUの構造を示すものである ([2]より引用)





- **Enclave内のデータ**も、通常のメインメモリの値同様**キャッシュ**に**ストア**される
- ところで、通常**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
 - この仕様を**アボートページセマンティクス** (以下、**APS**) という



- しかし、APSが適用されるような命令でそれ以上アドレス解決の行われないページフォールト（=**ターミナルフォールト**）が発生すると、APSの適用前に**投機的実行**が発生する
- **フォールトの発生した命令**でアクセスした**アドレスに対応する値**がL1Dにキャッシュされていると、この**投機的実行**においてその値が**過渡的に使用**されてしまう
- 命令リタイアに伴う**投機的実行結果の棄却の前にこの情報依存の痕跡をキャッシュに残す**事で、**過渡的な値をキャッシュサイドチャンネル的に抽出**出来てしまう



- この仕様を悪用し、予め**Enclaveの秘密情報**をL1Dに**キャッシュ**させておき、そのアドレスにアクセスするEnclave外の命令で**ターミナルフォールト**を起こす事を考える
- すると後続の過渡的（投機的）実行で**秘密依存の痕跡**が**キャッシュに残る**ため、結果的に**秘密情報を抽出**できてしまう

Foreshadow攻撃 (4/4)



- この手段によりEnclave内の秘密情報を抽出する攻撃こそが**Foreshadow攻撃**である
 - **Meltdown型攻撃**の一種に分類される
- **L1D**に存在するデータが**ターミナルフォールト**に伴い漏洩する為、Intel公式では**L1 Terminal Fault** (L1TF) と呼ばれている
- 頑張れば**非root**でも**攻撃を成立**させられる



Foreshadow攻撃のごく簡単な攻撃例 (1/2)

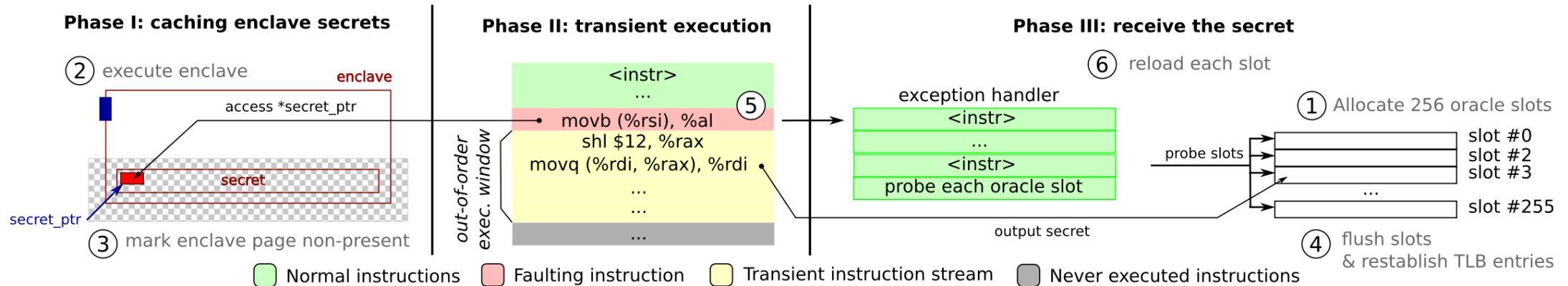


- ここではまず、Foreshadowの概念実証的な攻撃コードについて説明を行う
- Foreshadowは、主に以下の3フェーズにより構成される：
 - **フェーズI**：Enclave秘密情報のL1Dへのキャッシュ
 - **フェーズII**：ターミナルフォールトの誘発と過渡的実行
 - **フェーズIII**：秘密情報の抽出

Foreshadow攻撃のごく簡単な攻撃例 (2/2)



- Foreshadow攻撃のフロー概要図 ([1]より引用)





- 秘密情報の抽出を行うための**オラクルバッファ**と呼ばれる**監視用配列**を用意する
 - 秘密情報の抽出は**1バイト** (0x00~0xFFの**256通り**ある) **ずつ**行う
- オラクルバッファは、**1バイトずつ抽出**するために用いるため、1バイトが取り得る値の数 (256通り) に合わせて**256スロット**を有している
- **1スロットあたり4096バイト (=ページサイズ)** であるため、事実上のオラクルバッファのサイズは256×4096バイトである



- フェーズIIの過渡的実行により、**オラクルバッファの「秘密バイト×4096」のインデックス**にアクセスさせ、その**インデックスに格納されている値**を**キャッシュに残させる**

- **1スロットあたりのサイズがページサイズ**である理由は、**キャッシュラインプリフェッチャ**による誤作動で**正しくないスロットに対応する痕跡が残る**のを防止するためである



- フェーズIでは、**Enclave内**に存在する**秘密情報**を**L1Dにキャッシュ**させる
- 秘密情報の載るEPCページに対しENCLS命令であるEWBとELDUを繰り返す事で、攻撃対象の処理に関係なく、無理矢理L1Dに秘密情報をキャッシュさせる事が出来る
 - 後述の**ÆPIC Leak**攻撃でも使用される、**Enclave Shaking**と呼ばれる手法



- フェーズIIでは、**ターミナルフォールト**を発生させるためにアクセスする（秘密情報が載る）**Enclaveページへのアクセス権を剥奪**する
- Controlled-Channel攻撃の時と同様、これは**mprotect**関数で簡単に実現できる：

```
//PTEのPresentビットをクリアしアクセス不能にする  
mprotect( secret_ptr &~0xffff, 0x1000, PROT_NONE );
```



- 以下のコードを用いてForeshadowによる秘密情報の抽出を行う事を考える：

```
void foreshadow(uint8_t *oracle, uint8_t *secret_ptr)
{
    uint8_t v = *secret_ptr;
    v = v * 0x1000;
    uint64_t o = oracle[v];
}
```

- この攻撃コードは、**攻撃者が自前で用意し攻撃対象のマシンで実行**させられるため、Foreshadowは**攻撃対象のコードに依存しない**というメリットが存在する



- mprotectで**アクセス権を剥奪**した事により、3行目の
`uint8_t v = *secret_ptr;`で**ターミナルフォールト**が発生する
- ここで、**ターミナルフォールト**に伴う**過渡的実行**において、
4・5行目の**vの値**として、**L1D**から持ってきた**秘密バイト**が
過渡的に使用されてしまう
 - フェーズIでL1Dにこの秘密バイトをキャッシュしていないと、この過渡的なL1Dからの取得が発生しなくなる



- 5行目でオラクルバッファの**秘密バイト×4096**のインデックスにアクセスしているため、このインデックスのアドレスと格納されている値が**キャッシュ**される
- 命令リタイアにより**過渡的実行の値は棄却**されるが、既に**キャッシュに秘密依存の値が残っている**ので、**FLUSH+RELOAD**キャッシュサイドチャネル攻撃により**秘密バイトが抽出**できる
 - オラクルバッファの**全スロットにアクセスし、アクセス時間が短かったスロットのインデックスが秘密バイト**である



- 既にLEは**Deprecated**で**ほぼ形骸化している**が、論文での説明が一番充実しているのがLEへの攻撃であるため、LEに対する攻撃について説明する
 - ここでのLEはref-LEではなく、Intel公式により署名されている、昔ながらのLEである
- ENCLS命令の**EINIT**により**Enclaveを初期化**する前には、**LEからEINITTOKEN**構造体を受け取らなければならない
 - EINITTOKENには、起動しようとしているEnclaveのMRENCLAVEやMRSIGNER等が同梱されている



- LEは、以下の条件を満たすEnclaveに対し起動許可を与える実装になっている
 - 対象Enclaveが**デバッグモード**であるか、対象Enclaveの**MRSIGNER**が**Intel**によって**ホワイトリスト** (Intelのプライベート署名鍵で署名される) に登録されているかのどちらかである
 - 対象Enclaveが**PKへのアクセス権**のような、**特権的でIntelのみが使用可能な権限**を持っていない
- 起動を許可すると判断したら、LEはEINITTOKENの一部 (MRENCLAVE、MRSIGNER等) に対する**128bit AES/CMAC**を、**起動キー** (Launch Key) により算出しEINITTOKENに添付する
 - 起動キーは、**LAUNCHKEY属性**の付与されているEnclave (=LE) のみが**直接アクセス**する事が出来る



- LE自体のMRSIGNER値はプロセッサにハードコーディングされているため、**LE自体は起動許可処理を必要としない**
 - ref-LEでは、MSRのIA32_SGXPUBKEYHASH0~3にそのMRSIGNER値を書き込む事で同様に起動許可のスキップを行える
- LEからEINITTOKENを受け取ったら、**EINITは内部で起動キーを導出し、付与されたMACを検証する**
 - MACの検証に失敗した場合は起動はその時点で中止される
 - そもそも起動許可が与えられない場合、署名 (MAC) は付与されない[17]

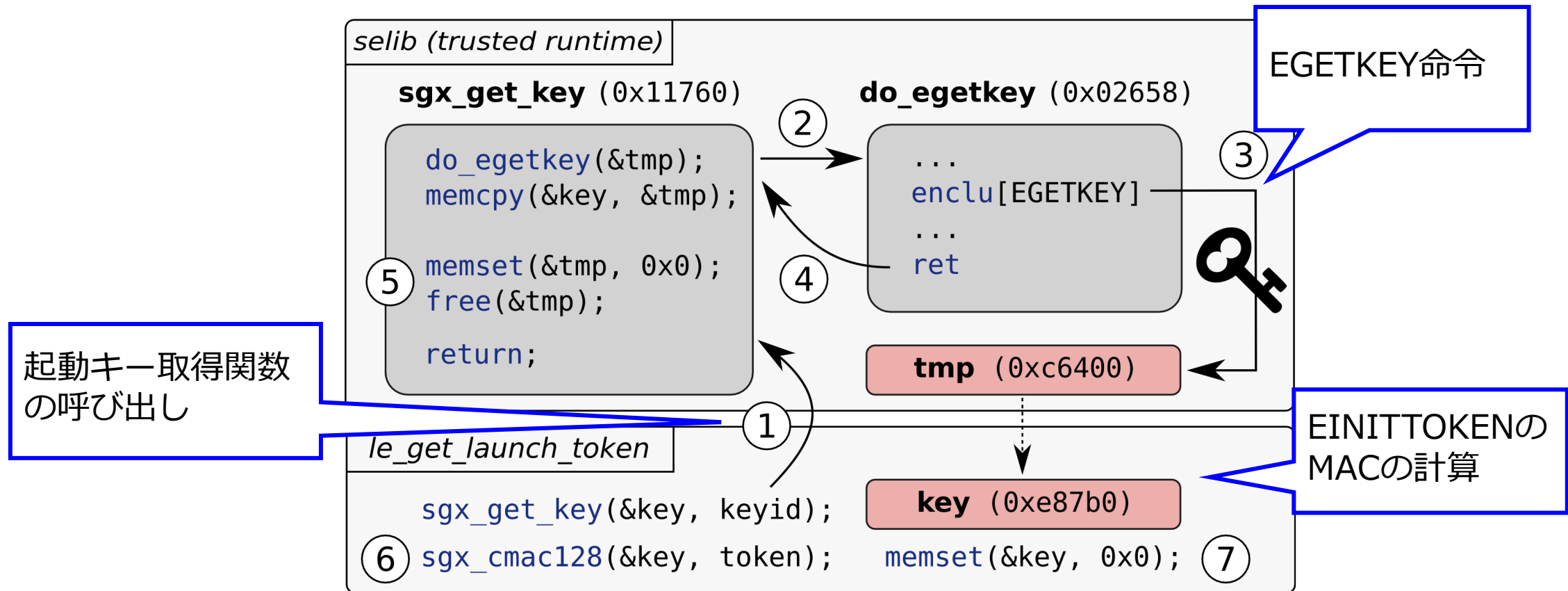


- もしForeshadow攻撃により**起動キーを漏洩**できた場合、完全にLEを迂回して**EINITTOKENを偽造**する事が出来てしまう
 - 本来**起動許可されていないEnclaveを起動**したり、**PKへのアクセス権を不正に付与**してEnclaveを起動させたり出来てしまう
- ただし、レポートキーの導出時と似たような話として、乱数要素を鍵に与える**KeyIDがEGETKEYごと**に**変わる**ため、**1回**のLEの処理で**鍵全体を抽出**する必要がある
 - よって、従来型のSGXに対するサイドチャネル攻撃のように、同じ処理を何度も繰り返して鍵全体を復元するというアプローチは取れない

Foreshadow攻撃例 – LEへの攻撃 (5/10)



- 以下の図は、対象Enclaveの同一性情報等を渡し、起動キーによるMACが付与されたEINITTOKENを取得するためのLEの関数の動作概要を示したものである ([1]より引用) :





- 前ページの図の内、起動キーを格納するバッファは**tmp**バッファと**key**バッファの2つが存在する
- 論文では、Foreshadowの攻撃性能を実証するために、より短命な**tmp**バッファから**起動キーを漏洩**させるケースを考えている
- 攻撃を行うにあたり、Controlled-Channel攻撃で登場した**ページフォールトシーケンス**に基づく判断や、**SGX-Step**という（マシン語レベルでの）**シングルステップ実行フレームワーク**を活用している



- オフライン（事前調査）フェーズでは、LEをSGX-Stepでシングルステップし、同一命令における**AEXを連発**させて**SSA**から**CPU内部の状態全体をダンプ**する（詳細は割愛）
- この攻撃手法を**ゼロステップ処理**と呼び、Foreshadow以外の様々なSGX攻撃においても頻繁に使用される
- このオフラインフェーズにより、攻撃対象である**tmpのアドレス**及び**関心のあるコードの位置**を**決定的に把握**できる



- **オンライン** (攻撃本番) フェーズでは、まず前述の図の③と④の間 (EGETKEY発行とdo_egetkeyからのリターンの間) でEnclaveに**割り込み**を加える
- **sgx_get_key**と**do_egetkey**で**交互にコードページのアクセス権を剥奪**するプログラムを実行する事で、ページフォールト回数から**do_egetkey**命令の**リターン位置を確実に特定**できる
 - オフラインフェーズの解析に基づき、**13回**ページフォールトが発生すると**確実にここに到達している事を保証する事が出来る**



- この時点で**起動キー**は直近で使用されている関係上**L1Dキャッシュ**に**存在**しているため、**リターン**における**ターミナルフォールト**に伴う**過渡的実行**から、**Foreshadowによる起動キーの抽出**に**成功**している
- 実際に抽出した**起動キー**を用いて、**LEを迂回する形でEINITTOKEN**を**偽造し不正に起動させる事に成功**している
 - LEでEGETKEYにより起動キーの生成を行わない限り、KeyIDはひたすら同一のままである為、**鍵の鮮度 (Freshness) 検証を排除**できる



- ただし、これをやった所で基本的には**Intelの利権の塊**である
ライセンス管理を迂回できるだけであるため、**SGX自体のセキュリティに及ぼす影響は小さい**
 - LEが散々批判され、遂にはDeprecatedになった所以でもある
- ただし、**PKへのアクセス権を持つユーザEnclaveを起動**させる事が出来てしまうという点については**懸念すべき**である
 - Plundervoltの解説でも述べた通り、実際にはPKの導出にはMRSIGNERが必要であり、これを得るにはIntelのプライベート署名鍵が必要になるため、実質的には**PKの導出は不可能に近い**



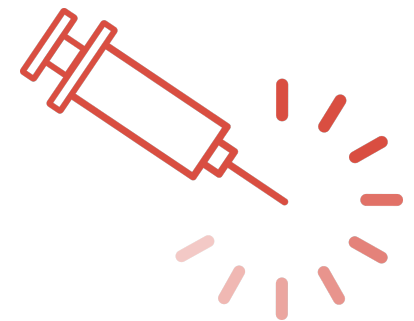
- 同様にして、QEにおけるEGETKEYに対してForeshadow攻撃を仕掛ける事で、**レポートキー**や**PSK**の抽出にも成功している
 - PSKが抽出できるとAttestationキーをアンシーリング出来るため、**Attestationキーの抽出にも成功**している
- **SGX Fail**の論文でも、**PowerDVD**に対し**Foreshadow攻撃**を仕掛け**Attestationキーを抽出**する事で攻撃を実現させたのはSGX Failのセクションで述べた通り
- **レポートキー**や**Attestationキーの漏洩**によって発生する**影響とその対策**についてもSGX Failのセクションで解説済み

Load Value Injection (LVI)

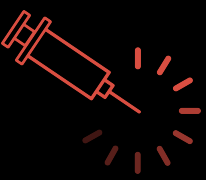
Load Value Injection (1/4)



- 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拳動**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

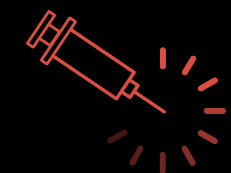


Load Value Injection (2/4)





- 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)

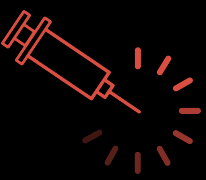


- 本来**秘密情報の漏洩**を行う**Meltdown挙動**をロード命令への**値の注入に転用**しているため、**逆Meltdown攻撃**とも、あるいはMeltdownを**Spectre的な注入に繋げる**という意味で二者の**融合型攻撃**とも表現する事が出来る

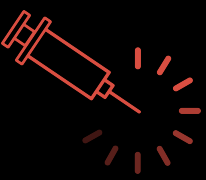
Methodology		Leakage 	Injection 
Prediction history	PHT	BranchScope [15], Bluethunder [24]	Spectre-PHT [38]
	BTB	SBPA [1], BranchShadow [40]	Spectre-BTB [38]
	RSB	Hyper-Channel [8]	Spectre-RSB [39, 44]
	STL	—	Spectre-STL [23]
Program data	L1D	Meltdown [42]	LVI-NULL
	L1D	Foreshadow [61]	LVI-L1D
	FPU	LazyFP [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[6]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



■L1Dキャッシュ

Foreshadowで説明した通り。

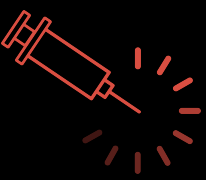
■ラインフィルバッファ (LFB)

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[7]

■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)

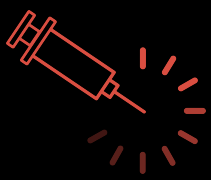
未処理のストアデータとアドレスを追跡 (保持) するバッファ。
準備・状況が整ったら、インオーダーで実際にストアを行う。

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。**ストア操作による汚染が可能である。**

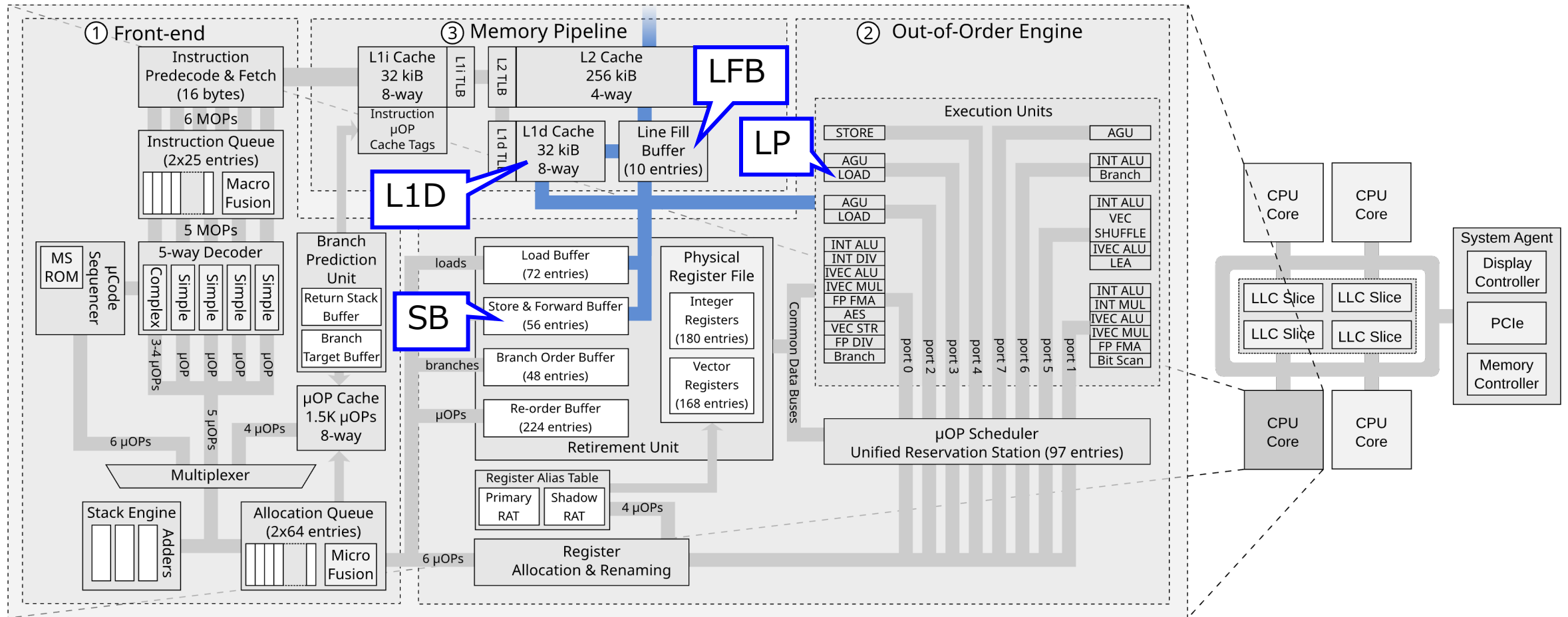
■浮動小数点演算処理装置 (FPU)

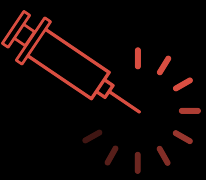
その名の通り、浮動小数点演算を専門に行う、CPUに内蔵されているユニット。

LVIに悪用できる μ -Archバッファ (3/3)

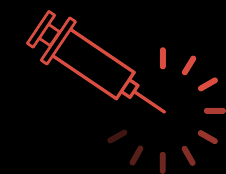


- CPU構造を示す図を再掲する ([2]より引用)





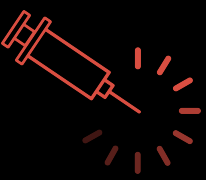
- **攻撃対象のドメイン内**（例：ユーザ空間であるEnclave）で攻撃のほとんどが**完結**するという特徴がある
 - よって、Foreshadow等と異なり、**攻撃対象のコード**において**LVI**を**発生**させる前提であり、Foreshadowのように攻撃者が**自前のコード**を用いて**攻撃を発動**させる事は**原則想定**されていない
- よって、**コンテキストスイッチ**（例：ユーザ→カーネルへの切り替え）時に**μ-Archバッファをフラッシュ**するような、**従来のMeltdown型攻撃への対策**は**LVIには効かない**



- **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

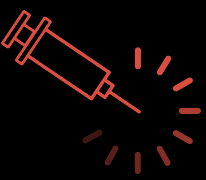
(*)英語でmemory disambiguation. 詳細は省略

- 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

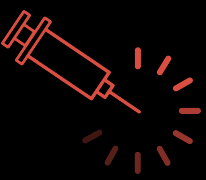


- また、Spectreについてはコード内で**過渡的実行攻撃が発生しそうな部分にlfence命令を挿入する事で対策**する方法も有名である
 - **lfence命令**：この命令が挿入された場所より**後の命令が、lfence命令よりも先にアウトオブオーダー実行される事を防ぐための命令**
- **LVIでも後続の命令を過渡的に実行する事を抑止するためにlfenceが有効であるが、挿入すべき対象があまりにも多く（例：ret命令）、全て対応しているとかなりのオーバーヘッドとなる**
- しかも、性質上**軽減策によるlfenceの導入が困難なロード命令も存在するため、完全な対応は不可能に近い**

LVIの攻撃力 (4/4)

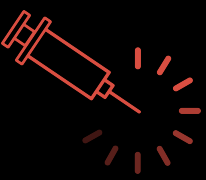


- 結局の所、**ロード命令**において**フォールト**や**アシスト**が発生した際に**過渡的実行が発生しない**ようCPUのシリコンレベルで**対策**をしない限り、**LVIの根絶は限りなく無理**に等しい
- 一方で、**Meltdown挙動**を発生させ、それにより注入された値を**後続の過渡的命令に使わせる**という**極めて難易度の高い攻撃**であり、**攻撃の実用性は低い**と言わざるを得ない
- このように、攻撃力というよりは**攻撃可能範囲が極めて広い**という点が、**LVIの厄介な性質**である

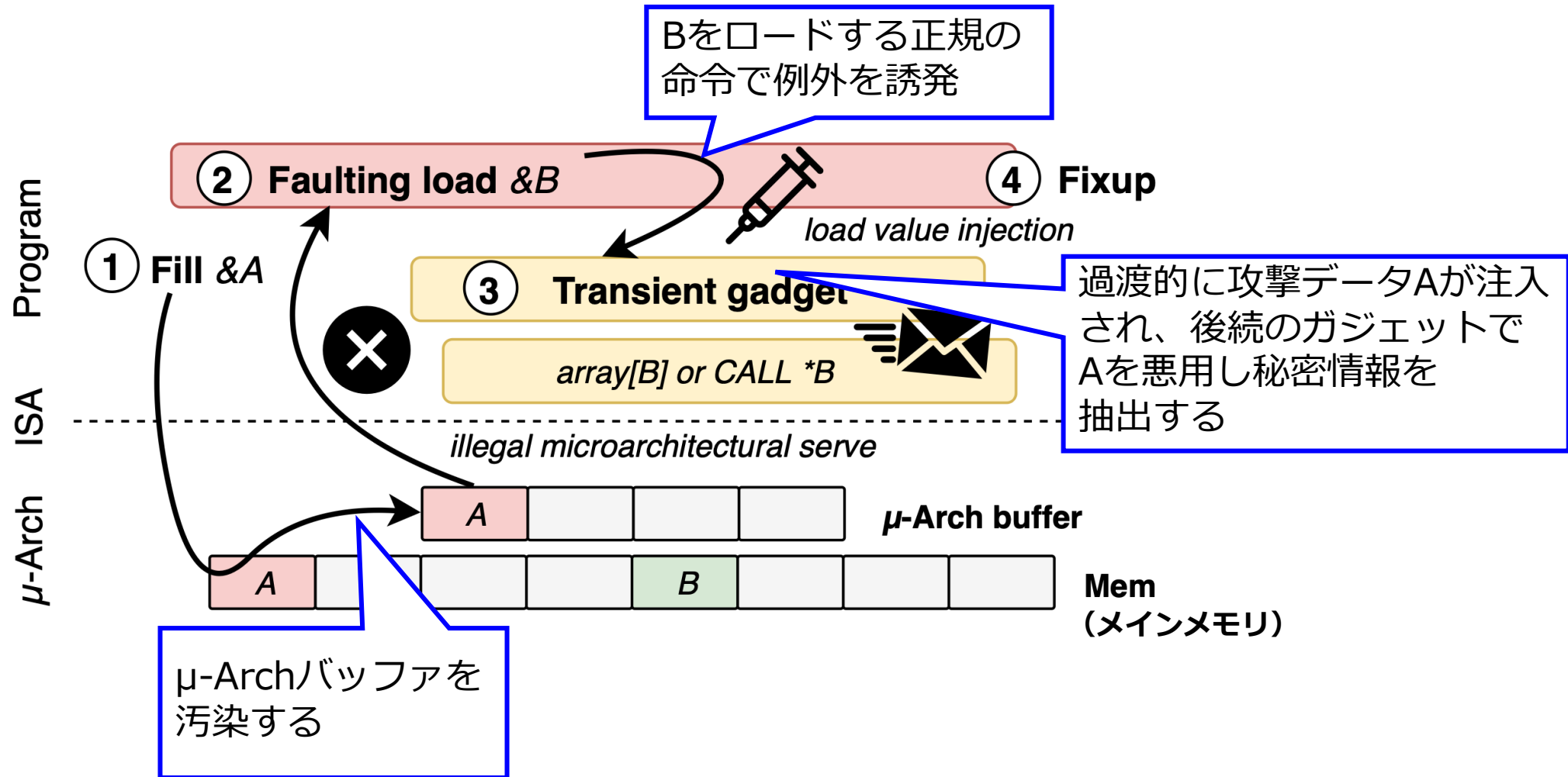


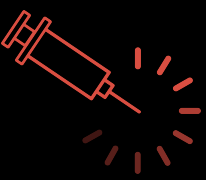
- Foreshadowの時と同様、まずはLVIの概念実証的な攻撃コードについて説明を行う
- LVIは、主に以下の3フェーズにより構成される：
 - **フェーズI (P1)** : μ -Archバッファの汚染
 - **フェーズII (P2)** : ロードにおけるフォールト/アシストの誘発
 - **フェーズIII (P3)** : ガジェット (攻撃に利用可能なコード) ベースでの秘密情報の転送 (漏洩)

LVIのPoC攻撃 (2/3)



- LVIの概要図は以下の通り (図は[6]より引用)

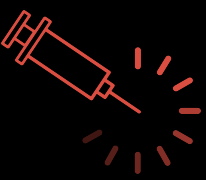




- 攻撃対象マシンで動作する以下のコードに対してLVI攻撃を行い、秘密情報を抽出する事を考える：

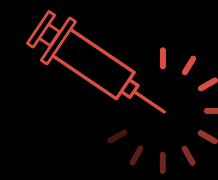
```
void call_victim(size_t untrusted_arg) {  
    *arg_copy = untrusted_arg;  
    array[**trusted_ptr * 4096];  
}
```

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

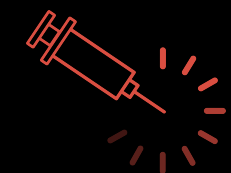
```
uint8_t *test_st = new uint8_t[sizeof(test_struct_t)]();  
uint8_t **trusted_ptr = &test_st;
```

参照
*trusted_ptr

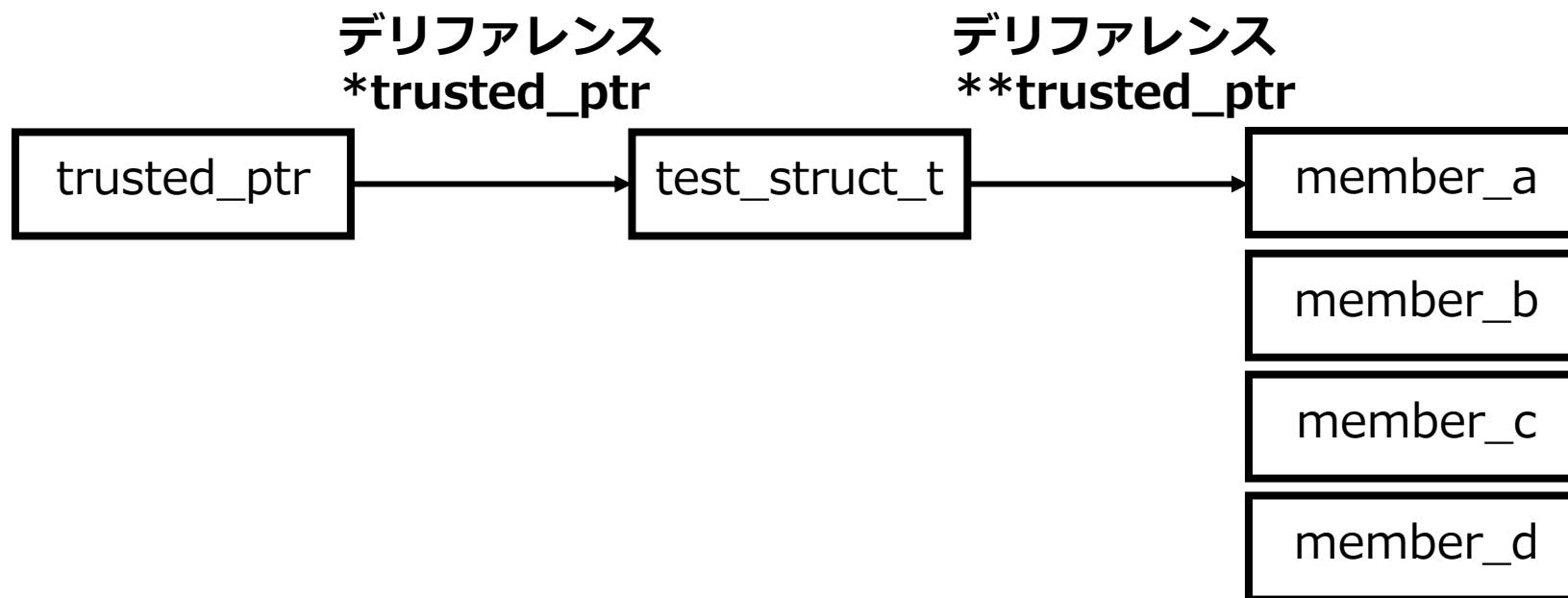
参照
**trusted_ptr

```
typedef struct {  
    uint8_t member_a;  
    uint8_t member_b;  
    uint8_t member_c;  
    uint8_t member_d;  
} test_struct_t;
```

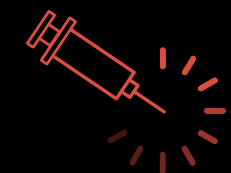
LVIのPoC攻撃 - P2ガジェット (2/4)



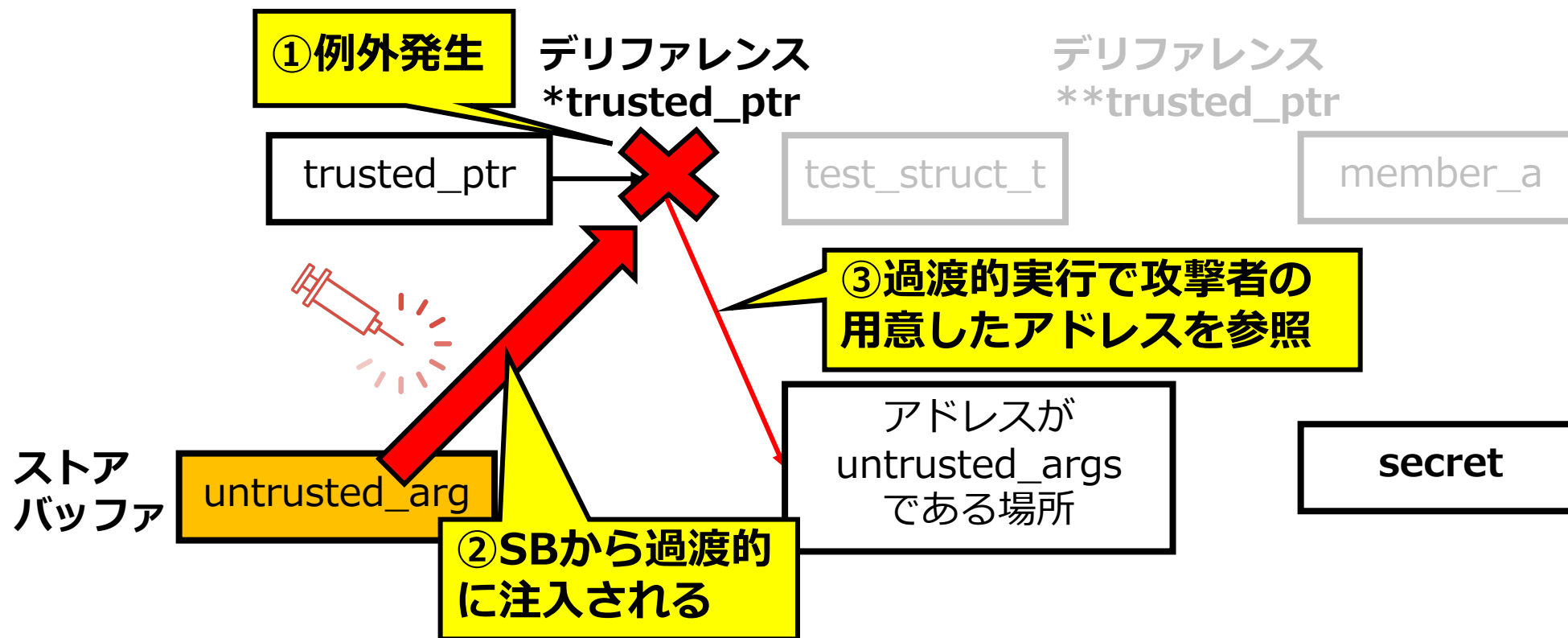
- 二重ポインタtrusted_ptrについて、*trusted_ptrのように1段階の**参照先取得 (デリファレンス)** を行くと、本来は構造体の先頭アドレスが手に入る



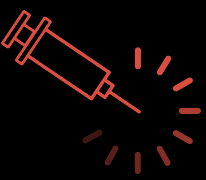
LVIのPoC攻撃 - P2ガジェット (3/4)



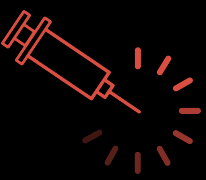
- ここで、1段階のデリファレンスである*trusted_ptrにおいて
フォールトまたはアシストを発生させる。すると、それに伴う
投機的実行において、**SBから流れ込んできた値**がデリファレンスに
おける参照先として**後続の過渡的命令**で**過渡的に使用**されてしまう



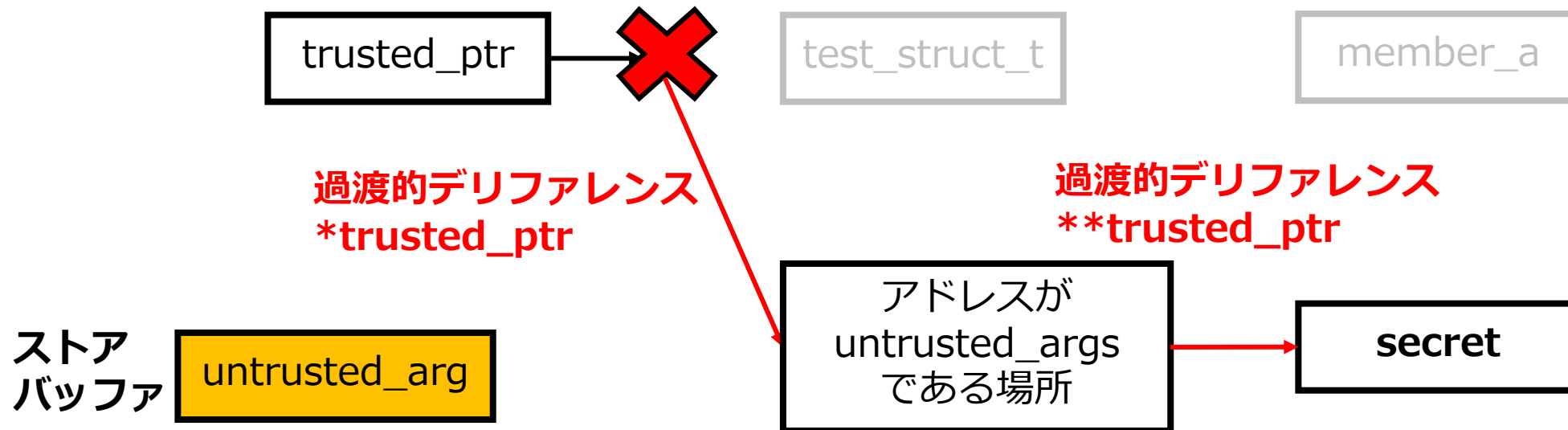
LVIのPoC攻撃 - P2ガジェット (4/4)



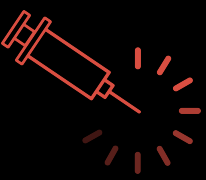
- この1段階目のデリファレンスでフォールトまたはアシストを発生させて過渡的実行を誘発できるため、**このデリファレンスはP2ガジェット**であると見なす事が出来る
- フォールトまたはアシストを発生させるには、何度か登場している **mprotect** を用いたり、**ページテーブルエントリ (PTE)** を **直接改竄** する等の様々な手法が利用可能である
- ストアバッファからの値の注入を誘発しているので、このPoC攻撃は後述の **LVI-SB** というバリエーションに属する事が分かる



- 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

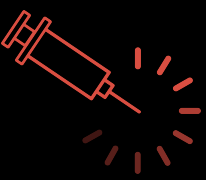


LVIのPoC攻撃 - P3ガジェット (2/2)



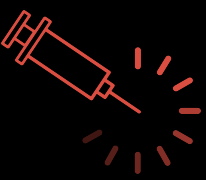
- 攻撃対象コードの3行目において、**監視用配列array**に対して `array[**trusted_ptr * 4096];` のように**秘密バイト依存のページアクセス**を行っている
- ここまで来れば**Foreshadowの時と同様**、過渡的実行が棄却された後に**キャッシュに痕跡が残る**ため、FLUSH+RELOAD**キャッシュサイドチャンネル攻撃**で**秘密バイトを抽出**できてしまう
- このPoC攻撃では、2段階目のデリファレンスから監視用配列への秘密依存のアクセスまでが**P3ガジェット**である

LVIのバリエーション



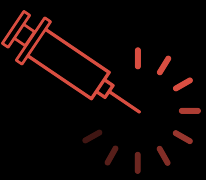
- ロード命令に対する注入をどこから行うかに応じて、LVIにはいくつかの**バリエーション**（変種・種類）が存在する

- 本ゼミでは、原論文に倣い**以下のLVIバリエーション**について解説する：
 - **LVI-L1D**
 - **LVI-SB**
 - **LVI-NULL**

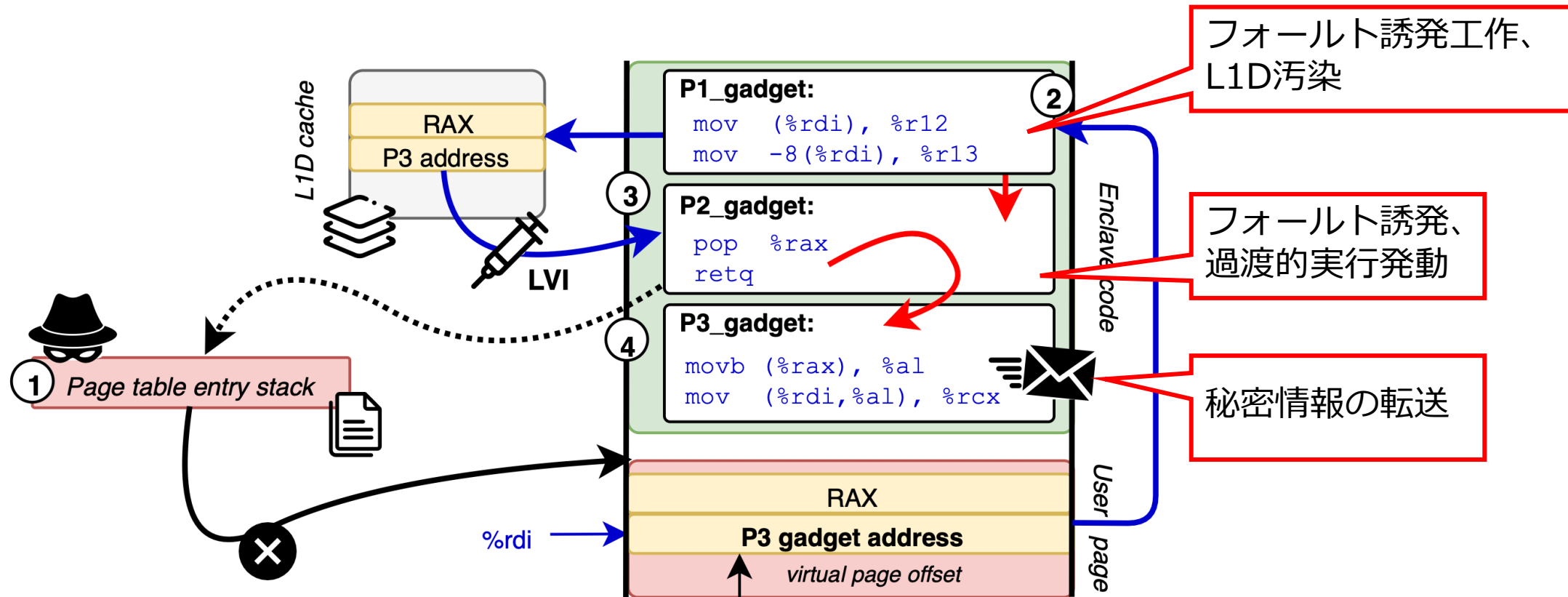


- その名の通り**L1D**から**値の注入**を行うLVIバリエーション
- L1Dからの漏洩を注入に転用する事から、LVI-L1Dは**逆Foreshadow攻撃**であると捉える事も出来る
 - 恐らくこの世に存在するSGX攻撃の中でも頂点に君臨するレベルの複雑度
- Foreshadowに対する対策が適用されている環境では、LVI-L1Dによる攻撃を成立させる事は出来ない

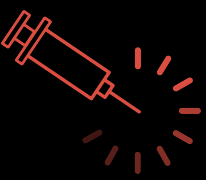
LVI-L1D (2/2)



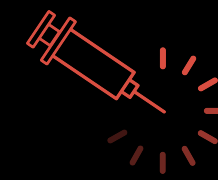
- LVI-L1Dの概要図は以下の通り ([6]より引用) :



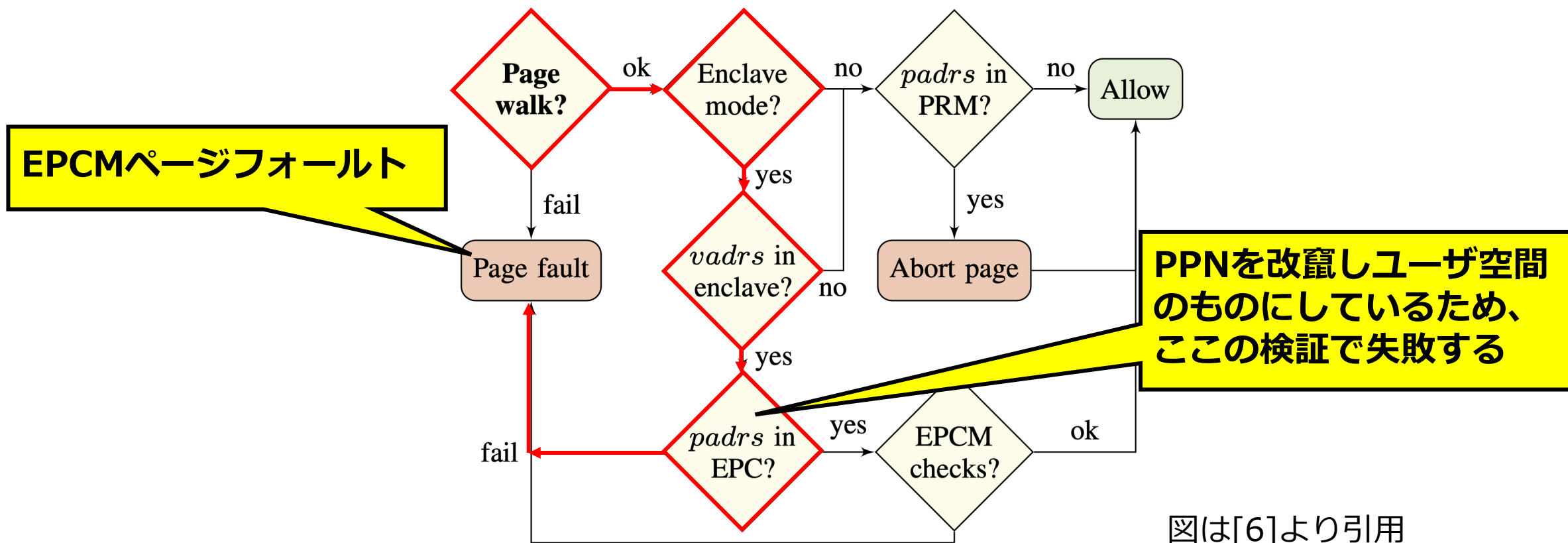
アセンブリはAT&T形式 ([命令] <ソース> <宛先>) で書かれているので注意



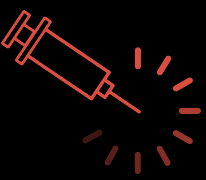
- ページフォールトシーケンス手法やSGX-Stepを用いる事で、Enclaveの実行を**P1ガジェットの直前**まで正確に進める
- その後、**P2_gadget**の**retq**命令がロードする**スタックページ**のPTE内の**PPN (物理ページ番号)**を、**P3ガジェットのアドレス値**を格納する**ユーザ空間上のページ** (以下、**ページU**と呼ぶ) のものに**改竄**する (前図①)
 - アドレス値はuintptr_tをイメージすると分かりやすい



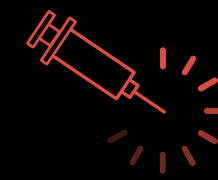
- このPPNの改竄により、P2におけるretqの実行時に、Enclaveの**正常な範囲の外**へ飛ぼうとした事による、**EPCMページフォールト**と呼ばれる特殊な**ターミナルフォールト**が発生する



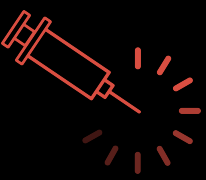
図は[6]より引用



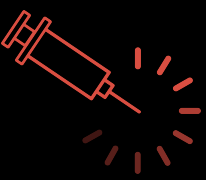
- その後、攻撃対象のEnclave内に存在する、例えばout属性のOCALL引数 (=攻撃者の用意した値で、P3ガジェットのアドレス値) をコピーするような**P1ガジェット**により、ページU上の**攻撃用の値**を**L1Dにキャッシュ**させる (概要図②)
- 前述の図においては、ページUの**保持する値** (=P3ガジェットの**アドレス値**) を%r12及び%r13レジスタにmovする事で**L1Dへのキャッシュ**を行っている



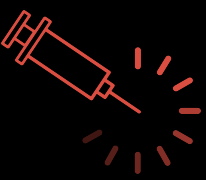
- P2ではまず、P2の載っている**Enclave内関数**からの**戻り値** (前図のシナリオでは**これが秘密情報**である) をスタックから **%rax**に**ポップ**している
 - 前図のP1・P2ガジェットは、別のEnclave内関数から呼び出されているEnclave内関数であるのが前提である
- P1でのPPN改竄の仕込みにより、**P2_gadget**の**retq**が実行されると、**Enclaveの仮想ページ**に対し**Enclave外のページUの物理ページ**が割り当てられているため、**EPCMページフォールト**が発生する (前図③)
 - ここでEPCMページフォールトに伴う**過渡的実行が発生**する



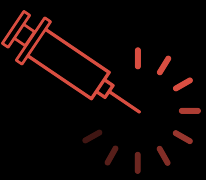
- しかし、これに伴う**過渡的実行**においてページUのアドレスをベースにL1Dへの問い合わせが行われ、P1でページUはキャッシュ済みであるため、**過渡的なretqのジャンプ先**としてページUの値(=**P3ガジェットのアドレス**)が**使用**されてしまう



- 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ンネル攻撃で秘密バイトを観測**出来てしまう

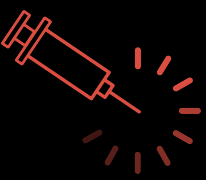


- 前図のP3ガジェット (④) では、**秘密バイトである戻り値をP2でポップして格納した%raxを、%rdiに対するインデックスとして使用し、そのアクセスにより秘密バイトの痕跡を残している**
- 元々%rdiには**P3ガジェットのアドレス値が格納されていたが、P3フェーズに入ると%rdiに何が入っていたかは関係ない**
 - 言わばここでは%rdiを**再利用している**だけであり、**監視用配列として使用**しているだけで元々何が入っていたかは既に**この時点で無意味**である



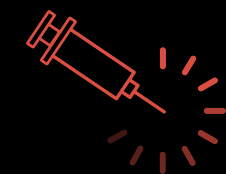
- 前述のLVIのPoC攻撃も属しているバリエーションのLVI攻撃であり、**ストアバッファからの値の注入**を悪用する攻撃

- Fallout攻撃[8]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

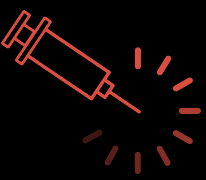


- 攻撃対象として、**Edger8r**が生成するエッジコードの以下の例を取り上げる：

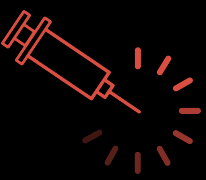
```
; %rbx: user-controlled argument ptr (outside enclave)
sgx_my_sum_bridge:
    ...
    call my_sum ; compute 0x10(%rbx) + 0x8(%rbx)
    mov %rax, (%rbx) ; P1: store sum to user address
    xor %eax, %eax
    pop %rbx
    ret ; P2: load from trusted stack
```



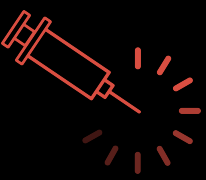
- **Enclave外のポインタ**（例：配列の先頭ポインタ）を**引数**として受け取り、Enclave内でそのポインタ先が含む**2つの値を加算**して引数経由で**リターン**する**ECALL関数**についての**エッジコード**である
- **加算結果**を**ストアバッファ**に格納させ、8行目の**ret命令**で**過渡的実行を発動**させる事で、**SBからの注入**により**リターン先**として攻撃者の選択する**P3ガジェットに飛ぶ**ように仕向ける攻撃を考える



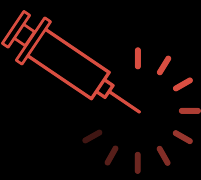
- 3行目の「...」部では、結果の返却にも用いる**引数ポインタ**（[in, out]属性をイメージすると分かりやすい）が**Enclave外に存在しているかを確認しているだけ**である
 - よって、**攻撃者が用意した引数の値がEdger8rによって阻害される事はない**
- 結果として、攻撃者はこの引数の**加算値**（=P2で**ret先**として**過渡的にSBから注入されるアドレス値**になる）をEnclave外で**任意に設定（改竄）**し、前述の**ページオフセット一致の制約を容易に満たす**事が出来る
- この性質を利用し、攻撃が上手く行くように引数を渡し、my_sum関数での加算結果をSBに格納させればP1は完了



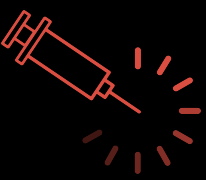
- 4行目のcall my_sumの後に**Enclaveを中断し、8行目のretで参照する事になるEnclaveスタックのページでのフォールトやアシストを誘発**する
 - 論文中では、ここではこのスタックページに対応するPTEのAccessedビットかスーパーバイザビットをクリアする事でこれを実現すると述べている
 - ただし、今まで通りPresentビットをクリアするのでは不可能であるとは書かれておらず、このケースに限ってPresentビットクリアが通用しない理由も無さそうであるため、Presentビットによる誘発も可能そうである
- 8行目で**実際にret命令が呼ばれると、フォールト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



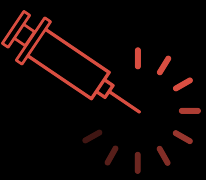
- あとはLVI-L1Dの時と同様、**任意のP3ガジェットに制御フローをリダイレクト**出来ているため、それを用いて**秘密情報の漏洩**を行える
- 制御フローリダイレクトに限らず、ユニバーサルリードガジェット法的にLVI-SB攻撃を行う事も勿論可能である



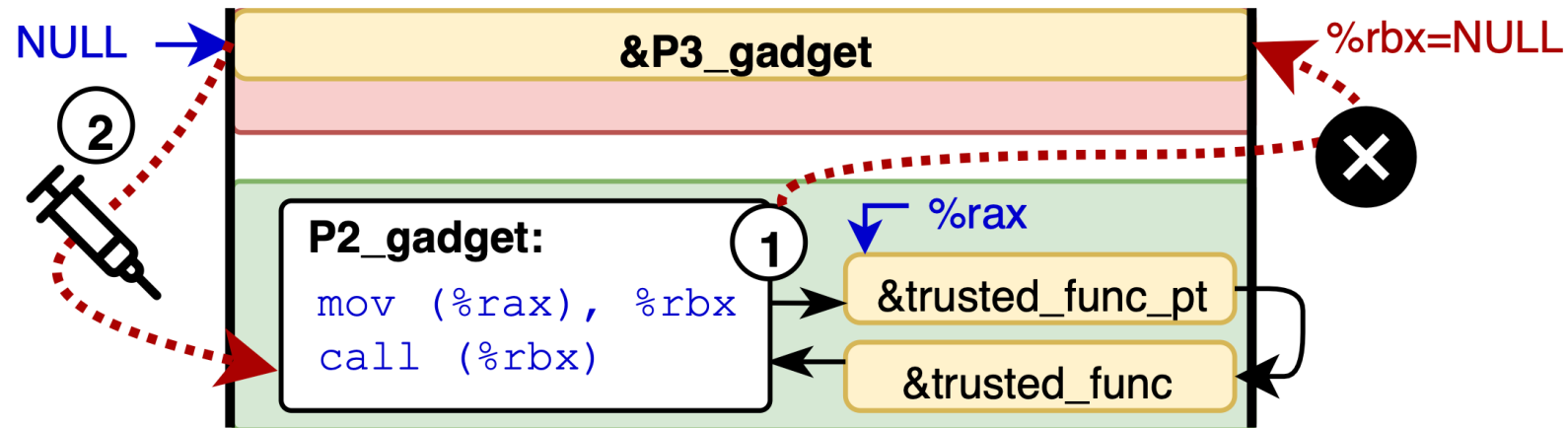
- Meltdownへの対策を行っている直近の世代のCPUでは、過渡的実行で**Meltdown的に漏洩する値をダミーの0x00に強制的に置き換える**事で、秘密情報が漏洩する事を防ぐようにしている
 - MSRのIA32_ARCH_CAPABILITIESアドレスのRDCL_NOビットが1であればMeltdown対策済みである
 - RDCLは**Rogue Data Cache Load**の略で、**Meltdownの正式名称**。NOは文字通り否定のnoの意
- しかし、**過渡的実行に0が注入される**という挙動自体を悪用するLVI攻撃も実行可能であり、これが**LVI-NULL**である

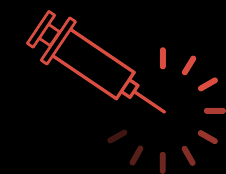


- OSや環境にもよるが、**root権限を持つ攻撃者は仮想アドレスNull (=アドレス0) に任意のメモリページをマッピング**する事が出来る
 - 手っ取り早い方法として、**PTE内に登録されている仮想アドレスを0**にしてしまえば良い
- よって、**Meltdown対策済みCPU**による、**過渡的実行**における**0値の転送**（注入）を悪用し、**仮想アドレス0**に用意した**P3ガジェット**に**制御を転送**する攻撃を例として考える

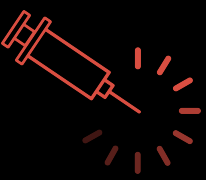


- ここでのLVI-NULL攻撃の概要図は以下の通り（[6]より引用）：

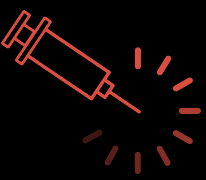




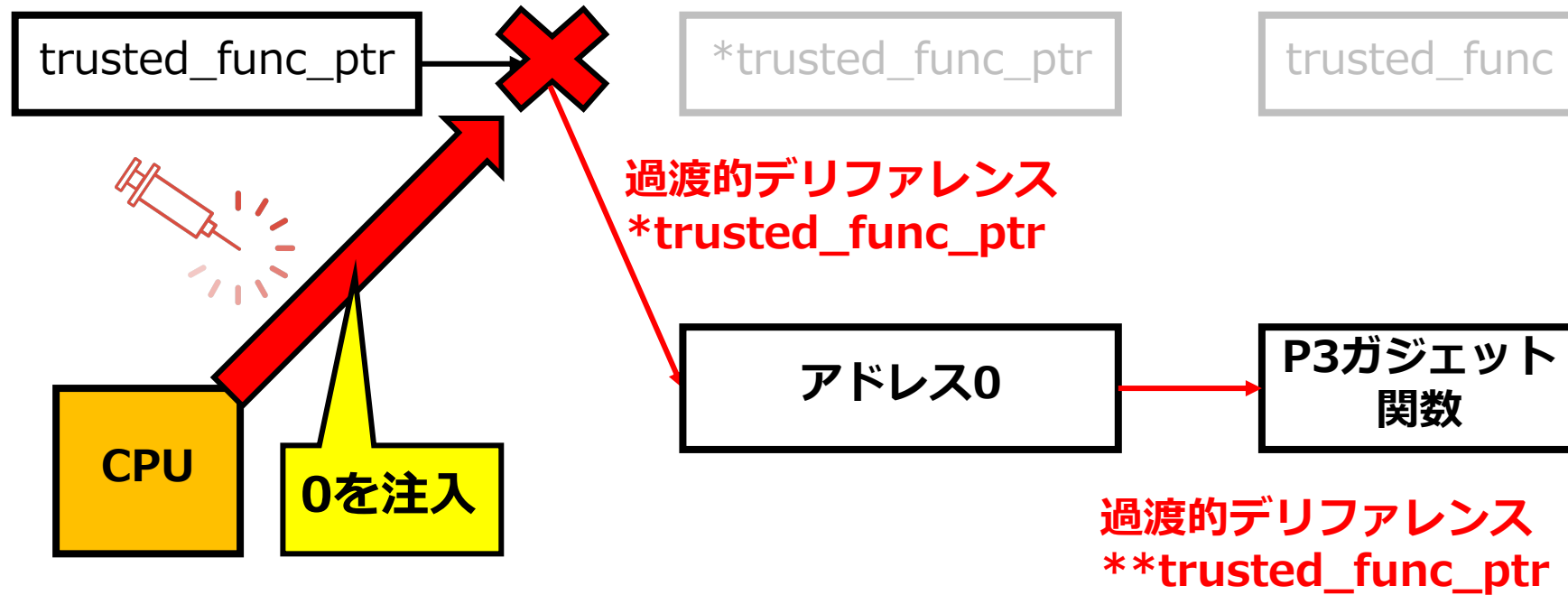
- このシナリオでは、**関数の二重ポインタ** (**ptrのようにデリファレンスする事で関数にアクセスできるようなポインタ) において**LVIを行う事**を考える
 - 関数の二重ポインタの具体的な実例として、動的に確保した構造体のようなヒープオブジェクトに含まれる関数ポインタが挙げられる
- 二重ポインタに対して攻撃するという意味では、**LVIのPoC攻撃**の説明で示した**LVI-SBのケース**と若干似ている

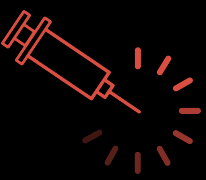


- この関数の二重ポインタの**第一段階のデリファレンス**でフォールトやアシストを誘発する事で、**過渡的実行を発生**させる
- この時、Meltdown耐性のあるCPUは過渡的実行に**ダミー値0**を転送し、結果的に**過渡的**な**第一段階のデリファレンス**で**アドレス0**を使用してしまう
- 結果として、**第2段階のデリファレンス**では**攻撃者が用意したアドレス0**をベースアドレスとした場所にある**不正な関数ポインタ**を**過渡的に取得**してしまい、それにより**P3ガジェットに制御転送**されてしまう



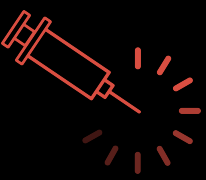
- このLVI-NULLの実行の様子を示した図は以下の通り：





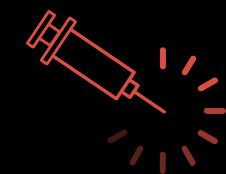
- ここで**1段階のみの関数ポインタ**を用いると、**過渡的な関数呼び出しが上手く行かないらしい**
 - 詳細は不明だが、一度過渡的にEnclave外ページをデリファレンスしてからその上のEnclave外関数を呼ぶ事は出来るが、直接Enclave外関数を呼ぶのは過渡的実行上でも不可能であると推測できる
 - デリファレンス先ページを実行不可能とマークする等して時間を稼ぎ、その間にアドレス0に再配置可能なEnclaveイメージをロードすれば1段階のみの関数ポインタでも攻撃が成立する可能性はある
- LVI-NULLの場合、他のバリエーションと比べても**過渡的実行ウィンドウが小さい**ため、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[9]

LVI攻撃例 – AES-NIへの攻撃 (1/8)



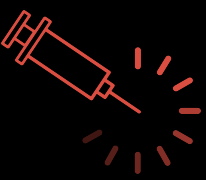
- LVIを利用したより実践的な攻撃として、**AES-NI**を利用するようなEnclaveに対し**LVI-NULL**攻撃を仕掛ける事で、**共通鍵を抽出**するような**故障注入攻撃**を行うケースを考える
 - **AES-NI** : AES暗号の暗号化及び復号の高速化を目的に実装されている、x86の拡張命令
- **既知暗号文攻撃** (暗号文のみを知った状態で秘密を解読する攻撃)のシナリオを前提とし、正しい暗号文を復号する処理に対して故障注入攻撃を行う事を考える
- 前述のLVI-NULLの例が**制御フローの改竄**を行っていたのに対し、この例はある意味**Plundervolt**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)

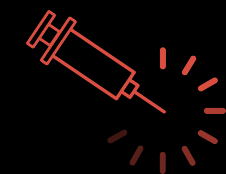


- AES暗号では、暗号文を**128bit** (16バイト) の**ブロック**という単位に**分割**し、さらに各ブロックを**1マス8bit** (1バイト) とした**4×4の行列**にする
- そして、この**4×4行列**に対し、ある**一連の処理**をAESの仕様で定められている**ラウンド数**分だけ繰り返す (**鍵伸長処理**)
 - 鍵長が**128bit**である場合はこのラウンド数は**10**である

LVI攻撃例 – AES-NIへの攻撃 (3/8)

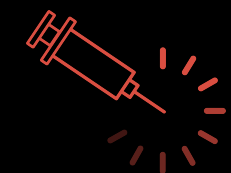


- 一連の処理とは、暗号化の場合は順に以下の通り：
 - **SubBytes** : S-Boxという置換表を参照するByte単位の置換
 - **ShiftRows** : 4×4 行列の n 行目を $(n-1)$ マスだけ左側にずらす。左端を超えるようなマスは右端に行くようにする (回転)
 - **MixColumns** : 列単位の処理。数式が長いので省くが、XORをベースとした演算 (CBCにおけるXOR処理とは別物)
 - **AddRoundKey** : state (処理中の 4×4 行列。上記の処理を適用した状態の途中段階の行列) と**ラウンド鍵**で、列ごとにXOR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



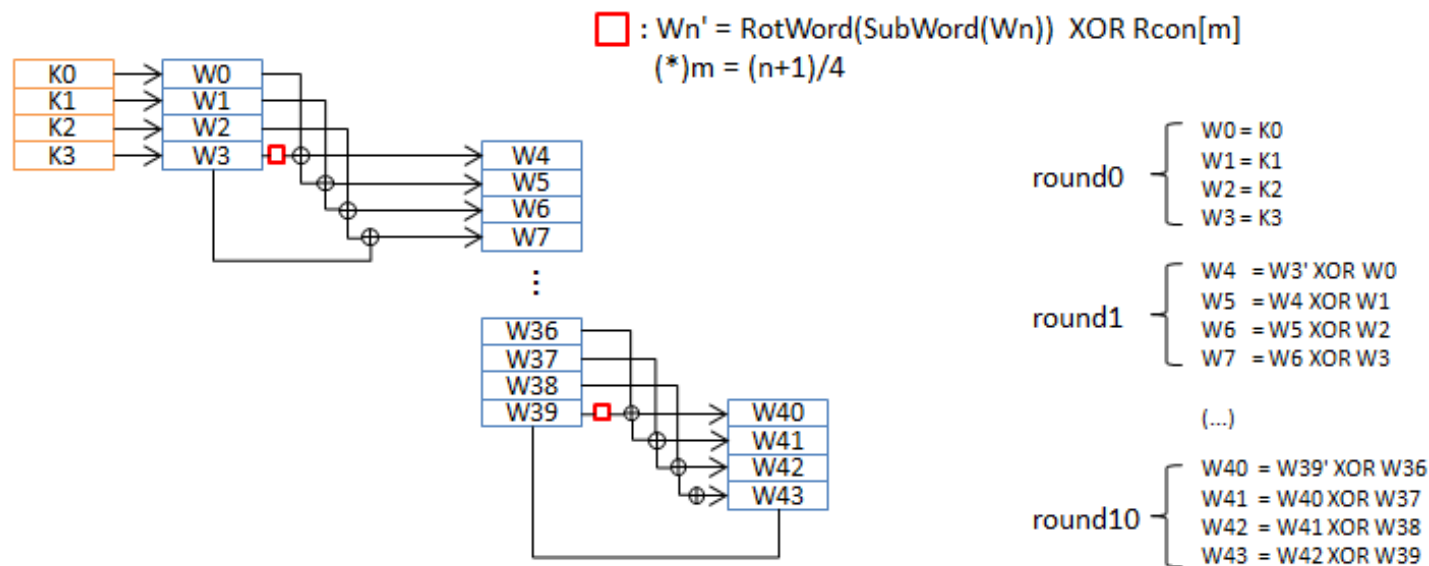
- 復号の場合は以下の通り：
 - **InvShiftRows** : 暗号化時のShiftRowsの右回転バージョン
 - **InvSubBytes** : Inverse S-Boxを参照した、SubBytesの逆置換
 - **AddRoundKey** : **暗号化時と同様**
 - **InvMixColumns** : これも列単位の処理で、数式が複雑なので詳細は割愛
- 最終ラウンドのみ、InvMixColumnsは実行されない

LVI攻撃例 – AES-NIへの攻撃 (5/8)

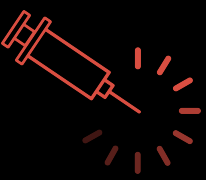


- ここで、ラウンド鍵はAESの共通鍵からラウンド分だけそれぞれ導出されるもので、以下のようにして導出される
(図は[10]より引用)

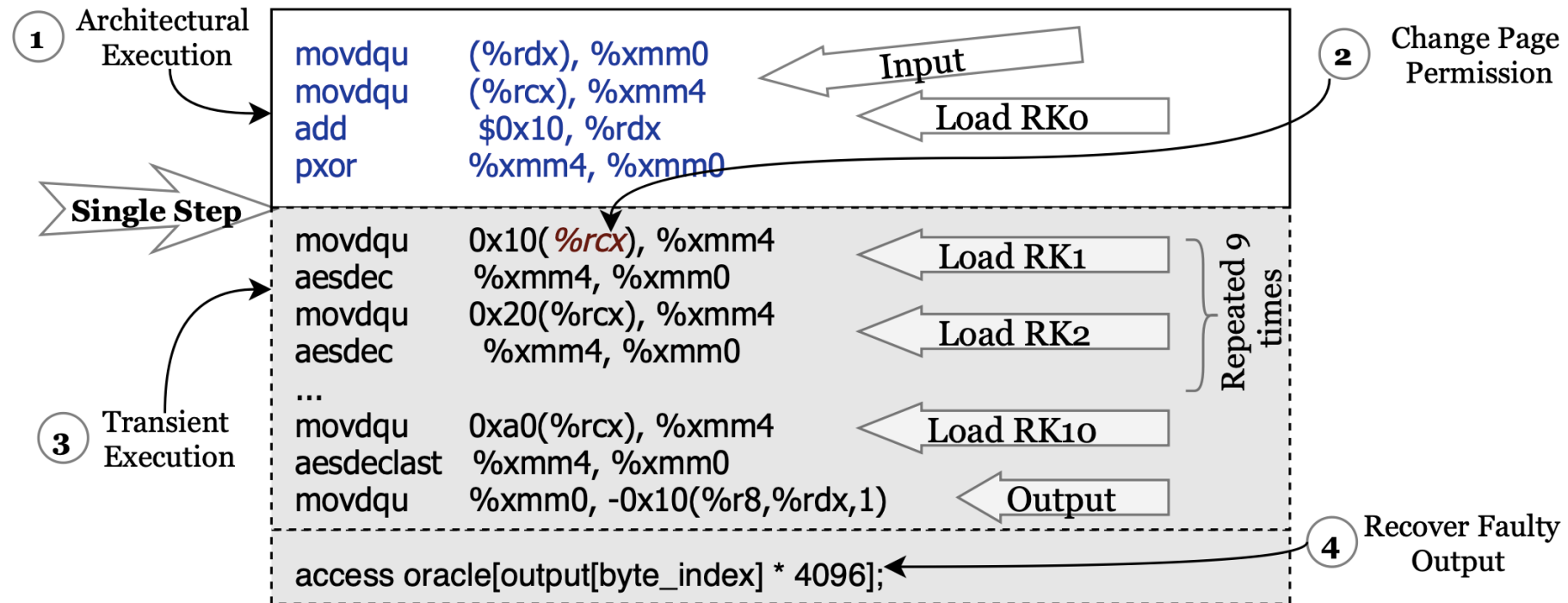
AES-128

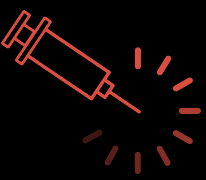


LVI攻撃例 – AES-NIへの攻撃 (6/8)

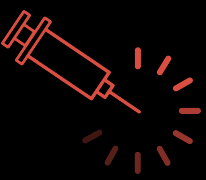


- AES-NIを利用するEnclaveに対するLVI-NULL攻撃の概要図は以下の通り (図は[6]より引用) :





- 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



- **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

ÆPIC Leak



- **APIC** : Advanced Programmable Interrupt Controllerの略で、割り込み処理の高度な制御を行う**割り込みコントローラ**
- **各CPUコア内部**に実装され、**CPU**に対し**配送**された**割り込み**を処理する**Local APIC**と、システムに**1つのみ**存在し、**外部デバイス**からの**割り込み**を適切な**CPU**に**転送**する**I/O APIC**がある
 - SGX-Stepも、Local APICのAPICタイマを用いてシングルステップ実行を実現している



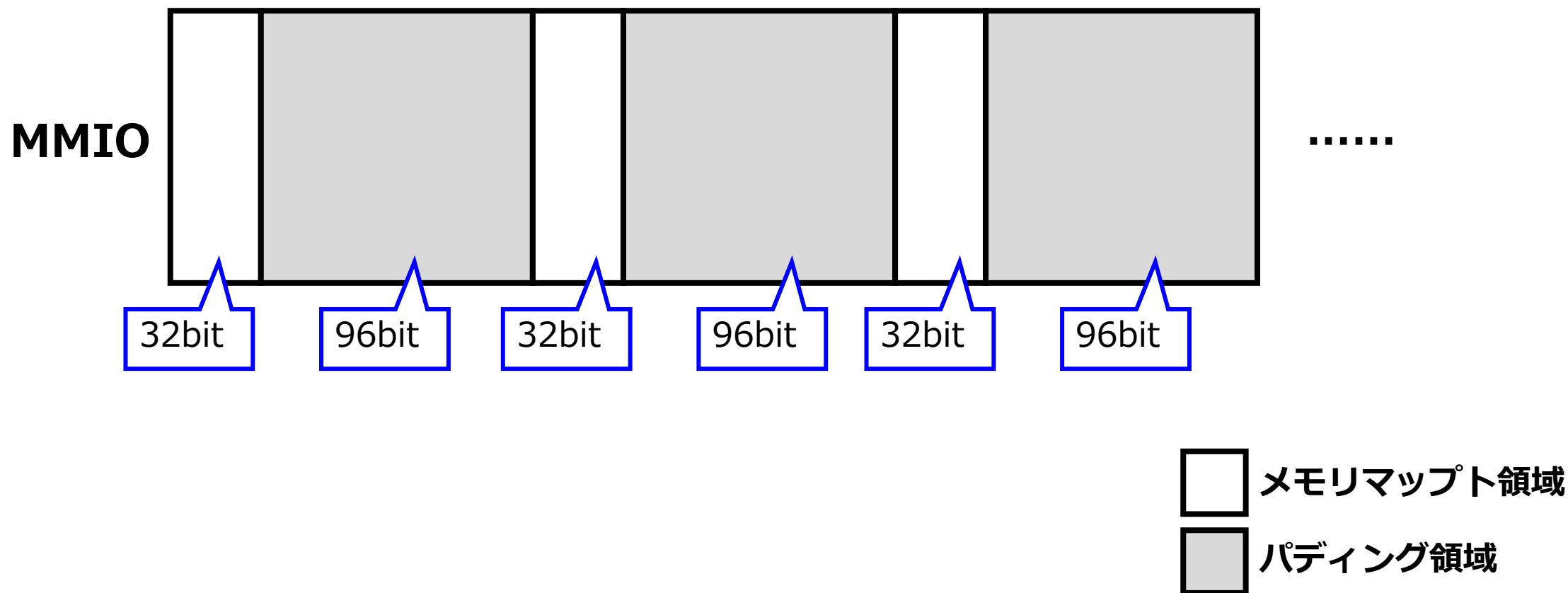
- 最近のAPICは、デフォルトでは**xAPIC**と呼ばれるモードで動作している。xAPICでは、Local APICとのやり取りを行うための**APICレジスタ**を、物理アドレス空間上の4kBの**メモリマップトI/O (MMIO)**の形で公開している
 - **MMIO**：メインメモリ上のある特定のアドレスにアクセスすると、対応する入出力機器とのやり取りが出来る仕組み
- より新しいモードとして、**x2APIC**というモードも存在する。割り込み配送の性能向上が図られており、またAPICレジスタには完全に**MSRを通してアクセスする (MMIOアクセスの廃止)**
- どちらのモードを用いるかは**OS (root権限)**によって**設定**できる



- MMIOのベースアドレスはデフォルトでは物理アドレス 0xFEE00000に設定されているが、MSRのIA32_APIC_BASEの値を変更する事でコアごとに対応するMMIOアドレスを変更できる
- APICレジスタは32bit、64bit、または256bitのいずれかの大きさの値を取り扱うが、**xAPICモード**における**メモリマップト領域**では**32bit単位に分割**してマッピングされ、さらに**128bit単位に整列**される



- **32bit**のメモリマップト領域を**128bit**単位に整列するために、**96bit**の**パディング領域**を挟む事によってこれを実現している



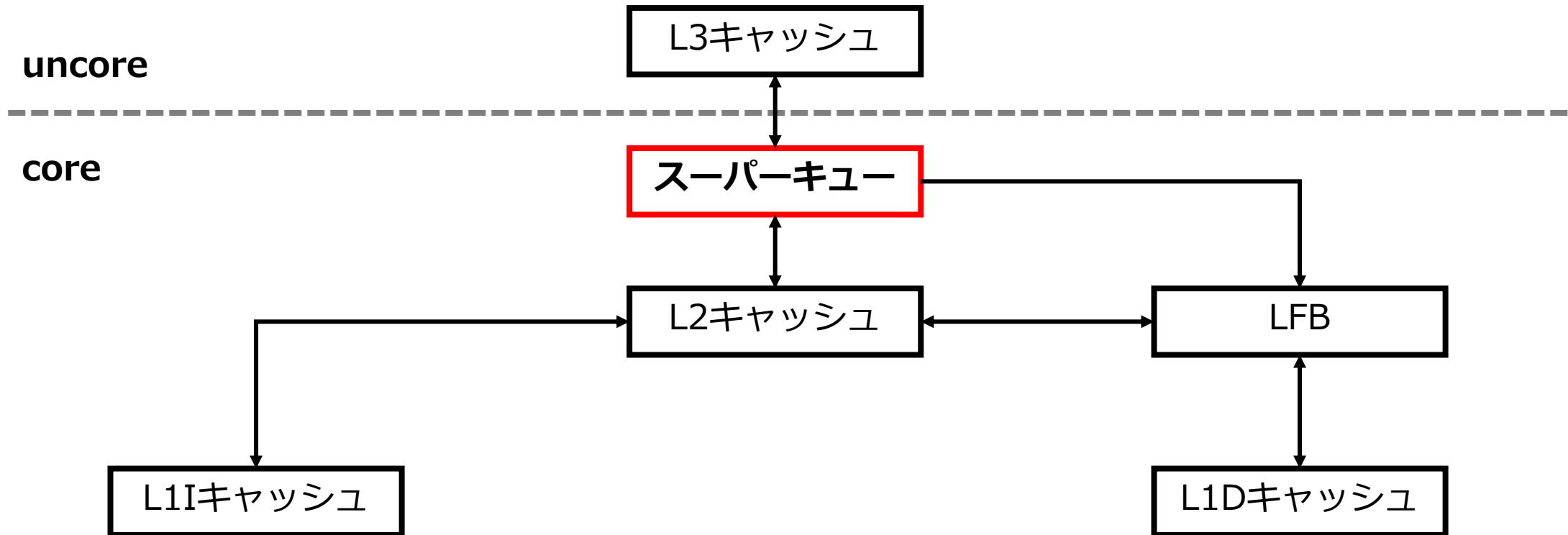


- このパディング領域は文字通りパディング用であり、それ以外に何らかのアーキテクチャ的な定義がなされていない領域である
- Intelは、このパディング領域へのアクセスは**未定義の動作**を引き起こす可能性があるため**行ってはならない**と言及している
 - 普通は、0x00または0xFFの読み出し、システムハング、そしてトリプルフォールト（例外ハンドラの失敗の対応にも失敗した際のフォールト）のいずれかが発生する
 - あくまでもメモリマップト領域である32bitの部分にのみアクセスしAPICとのやり取りを行うのが本来の想定

キャッシュ階層 (1/2)



- ForeshadowとLVIの説明では言及しなかったが、**L1D-L2間のLFB**に相当するものとして、**L2とL3の間にスーパーキュー (Superqueue)** という架け橋的なバッファが存在する

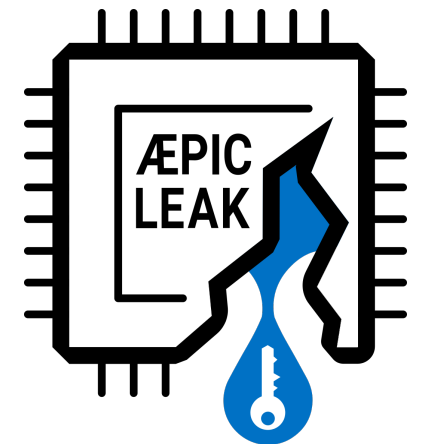




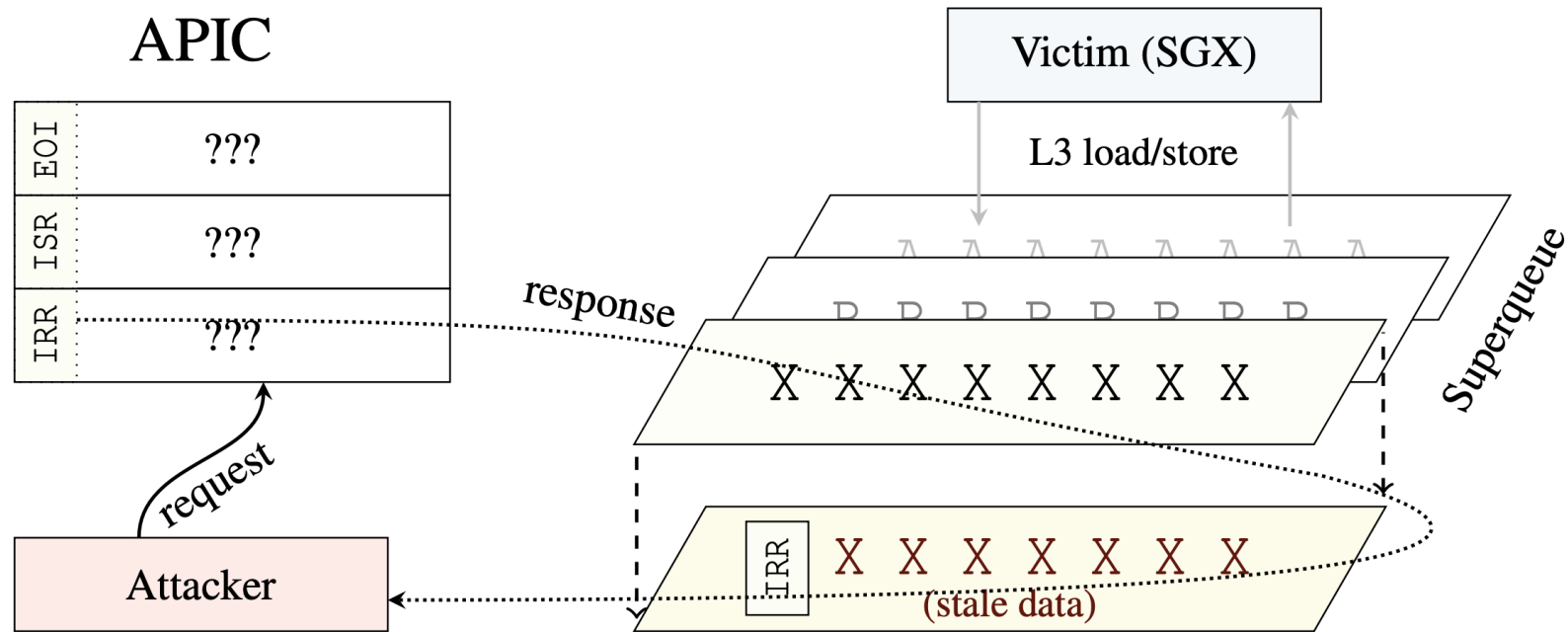
- **APIC**は、**L3キャッシュ**から**L2キャッシュ**に値を持ってくる際に、この**スーパーキュー**を使用する
- スーパーキューはキャッシュ間のデータの転送に用いられるものであるため、ユーザ空間、カーネル空間、そして**Enclaveのデータ**等、**あらゆる出処のデータ**が格納される可能性がある



- **Sunny Coveマイクロアーキテクチャ**を搭載するCPUにおいて、**xAPICのMMIOのパディング領域**を読み取ると、**スーパーキューに残留している値が漏洩する**バグがある事が判明した
 - Sunny Cove μ -Archを搭載するCPUとしては、第10~13世代Core シリーズCPUや、第3世代Xeon-SP CPUが挙げられる
- このバグを悪用し、**Enclave由来のスーパーキュー内の秘密情報**を**漏洩させる**攻撃が**ÆPIC Leak**である
 - 根本原因は、APICレジスタ (MMIO) のパディング部分が**正しく初期化されない**という**アーキテクチャ上のバグ**である
 - 現在発表されているSGX攻撃の中では最新に近い攻撃の1つ



- ÆPIC Leakの概要図 ([12]より引用)



- 具体的には、**パディング領域に1~4バイト単位**でアクセスを行うと**前述のような漏洩が発生**する
 - 8バイト以上単位でのアクセスでは必ず0xFFが返却される
- APICレジスタは複数存在するが、**アクセスするAPICレジスタとその内容に相関は存在しない**
- 一方、APICアドレスに対応する**キャッシュライン上のオフセット**は、**漏洩させるデータのキャッシュライン上のオフセットに一致**する
 - **キャッシュライン**：キャッシュの構成単位（通常**64バイト**）
 - 例えばAPICベースアドレスから0x10のオフセットの場所にアクセスすると、攻撃対象キャッシュラインの0x10のオフセットの値が漏洩するイメージ

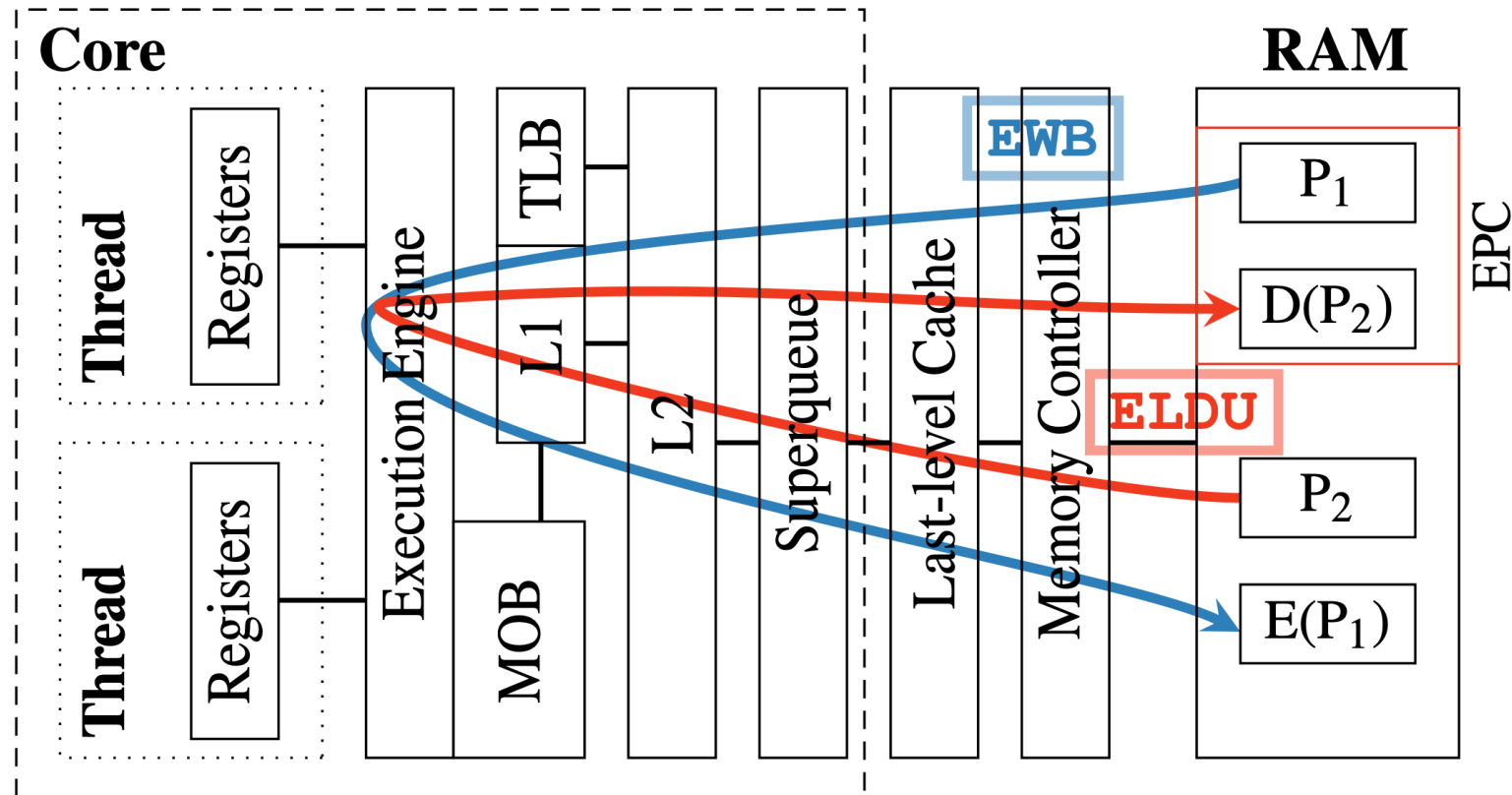
- 128bitの内パディング部分でない**32bitのメモリマップト領域**は正常に初期化されるため、**64バイトのキャッシュライン**であれば**後半48バイト**（後半**3/4**）のみが漏洩する
- さらに、**128の倍数**のアドレスから始まる**キャッシュライン**からのみ漏洩させられる
 - キャッシュラインは**通常2つペア**で転送されるため、**2つ目**のキャッシュライン（アドレス： $64+128 \times n$ ）がまず**転送**され、その後**1つ目**（アドレス： $128 \times n$ ）で**上書きする形になる**からであると考えられる
- 結果的に、ÆPIC Leakでは $\frac{96}{128} \times \frac{1}{2} = 0.375$ 、即ち**任意のページ**の**37.5%**を漏洩させる事が出来る

- Foreshadow攻撃でも使用されていた手法であるが、**秘密情報が載るEPCページをEWBとELDUでページ退避&ロードを繰り返す**事で、強制的に秘密情報の載るページの内容を**キャッシュ階層（スーパーキュー含む）**に**持ち込む**事が出来る
- Foreshadow攻撃の論文ではこの手法に対する名前をつけていないが、ÆPIC Leakの論文で**Enclave Shaking**と名付けられている

ÆPIC Leakの補助手法 (2/3)



- Enclave Shakingの概要図 ([12]より引用)。Enclave Shakingにより、秘密情報の載るページの内容がキャッシュ階層に浸透する様子を示している



- 別の補助手法として、ある**攻撃対象**の載るものと**並行して実行**されている、**同一物理コア内の論理コア** (=ハイパースレッド)、即ち**兄弟スレッド** (Sibling Thread) を**悪用**する手法がある
- **攻撃対象に並行して実行**される**兄弟スレッド**にて、攻撃対象と**無関係なページ**の**ページオフセット x** に**連続的にアクセス**すると、**攻撃対象ページ**の**オフセット x** の値が**漏洩**する可能性を上げられる
- 論文では、この手法を**Cache Line Freezing**と呼んでいる



- ÆPIC Leakの攻撃例として、SGX Failのセクションでも取り上げた **SECRET Networkのコンセンサスシードの抽出**を行う攻撃を取り上げる
 - 当該セクションで説明した通り、SGX Failの論文で実験として実際に行われた攻撃である
- 早い話、コンセンサスシードを暗号化しているSGX_FILEのAES鍵 (**IPFSL鍵**) に対して**ÆPIC Leak**を仕掛け、IPFSL鍵を抽出し**コンセンサスシードを復号**する事を考える

(復習) SECRETにおけるコンセンサスシード (1/4)



- SECRETでは、前述の様々な保護機能で使用する鍵は、全て親玉（**マスター秘密鍵**）である**コンセンサスシード**から導出される
- スマートコントラクトへのメッセージ送信時は、ユーザは**コンセンサスシード**から導出した**公開鍵**を用いて**暗号文**を**トランザクション**に含める
 - 一方、**秘密鍵**は**MRSIGNERポリシーのシーリングデータ**として、チェーン全体に複製（デプロイ）される
- また、口座残高等の現在の状態を暗号化する鍵もまた**コンセンサスシード**から導出される

(復習) SECRETにおけるコンセンサスシード (2/4)



- 当然、この**コンセンサスシード**が**万が一にも漏洩**すると、SECRETが売りにしている**あらゆる保護機能が無力化**される
- さらに、このコンセンサスシードは原則として**永続的かつ不変**のものであるため、もし漏洩すると**極めて面倒な事になる**
 - **ブロックチェーンそのものを分裂**させなければならない**ハードフォーク**でのみ対応する事が出来る



- コンセンサスシードは、独自に用意した鍵により **Intel Protected File System Library** (以下、IPFSL) を用いて128bit AES/GCMで暗号化され**Enclave外にストア**される
- **IPFSL : SGX_FILE**型という**Enclave内**で直接扱えるようなFILE構造体に基づき、**暗号化した状態でのファイル入出力** (sgx_fopen等) を提供する機能
 - Enclave内から**直接Enclave外に保存**できる、**暗号化鍵を指定**できる等の部分でシーリングと異なる
 - 本ゼミでは使用していないが、includeとリンクさえ行えば普通に**デフォルトのSGXSDKで利用可能**

(復習) SECRETにおけるコンセンサスシード (4/4)



- IPFSLでコンセンサスシードを暗号化する際に使用した鍵は
シーリングでストアされる
 - 鍵自体はAESで使う普通の128bitの共通鍵
- SGX Failでは、この**IPFSL用の鍵**がEnclave内で**コンセンサスシードの暗号化及び復号**に**使用されている最中**に**攻撃を仕掛け**、IPFSL用鍵を抽出する手法を選択した
 - **IPFSL鍵が漏れる**と簡単に**コンセンサスシードが復号**出来てしまい、**SECRET上のあらゆる秘密情報**が**究極的には暴かれてしまう**

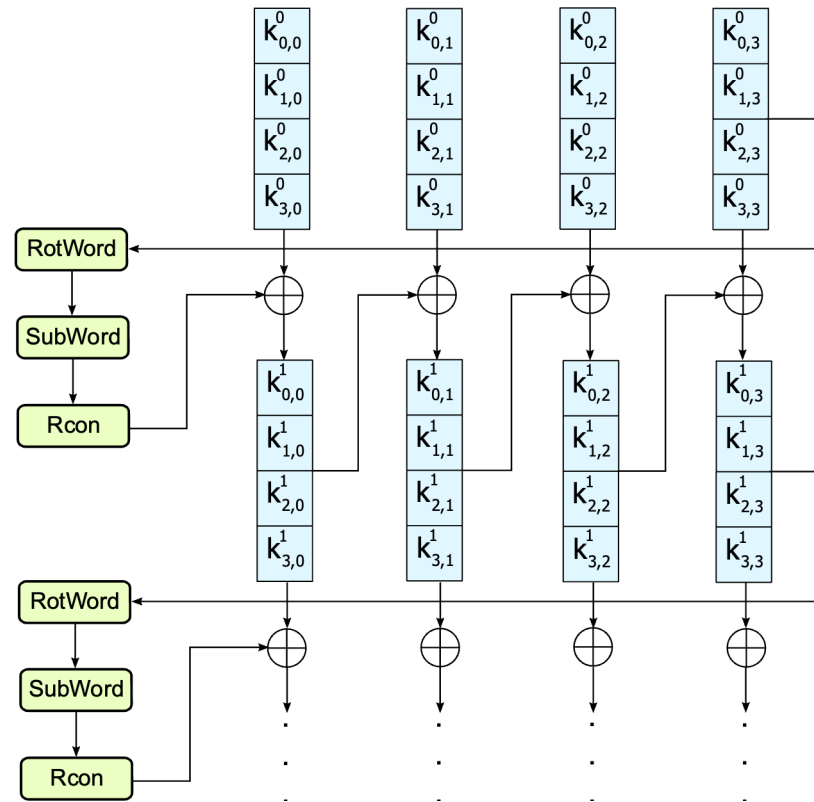


- まず、Controlled-Channel攻撃の**ページフォールトシーケンス**手法を利用し、IPFSLがAES鍵により**コンセンサスシードを復号**する関数の所まで**実行を進める**
 - 厳密には、復号にあたり鍵伸長処理を行う
k0_aes_DeckKeyExpansion_NI()関数の実行まで進める
- この状態で**Enclaveを中断**し、**ÆPIC Leak攻撃**を実行する



- ÆPIC Leakを仕掛けると、**N**ラウンド目と**N+1**ラウンド目のラウンド鍵の、それぞれ**後半3ワード**が**復元**できる
 - 1ワード：サイズは4バイトで、AESブロックである4×4行列の1行または1列に相当する。ワード換算では128bitは4ワードになる
 - 先頭1ワードが抽出できないのは、前述の37.5%のリーク率が原因である

- ここで、AESの鍵伸長では、**N+1**ラウンド鍵の**第2ワード**を導出するために、**N+1**ラウンド鍵の**先頭ワード**と**N**ラウンド鍵の**第2ワード**とでXORを取っている (図は[14]より引用)



- ここで、AESの鍵伸長では、**N+1**ラウンド鍵の**第2ワード**を導出するために、**N+1**ラウンド鍵の**先頭ワード**と**N**ラウンド鍵の**第2ワード**とでXORを取っている (図は[14]より引用)

- つまり、 n ラウンド鍵の第 i ワードを W_i^n と表現すると、上記は

$$W_2^{N+1} = W_1^{N+1} \oplus W_2^N$$

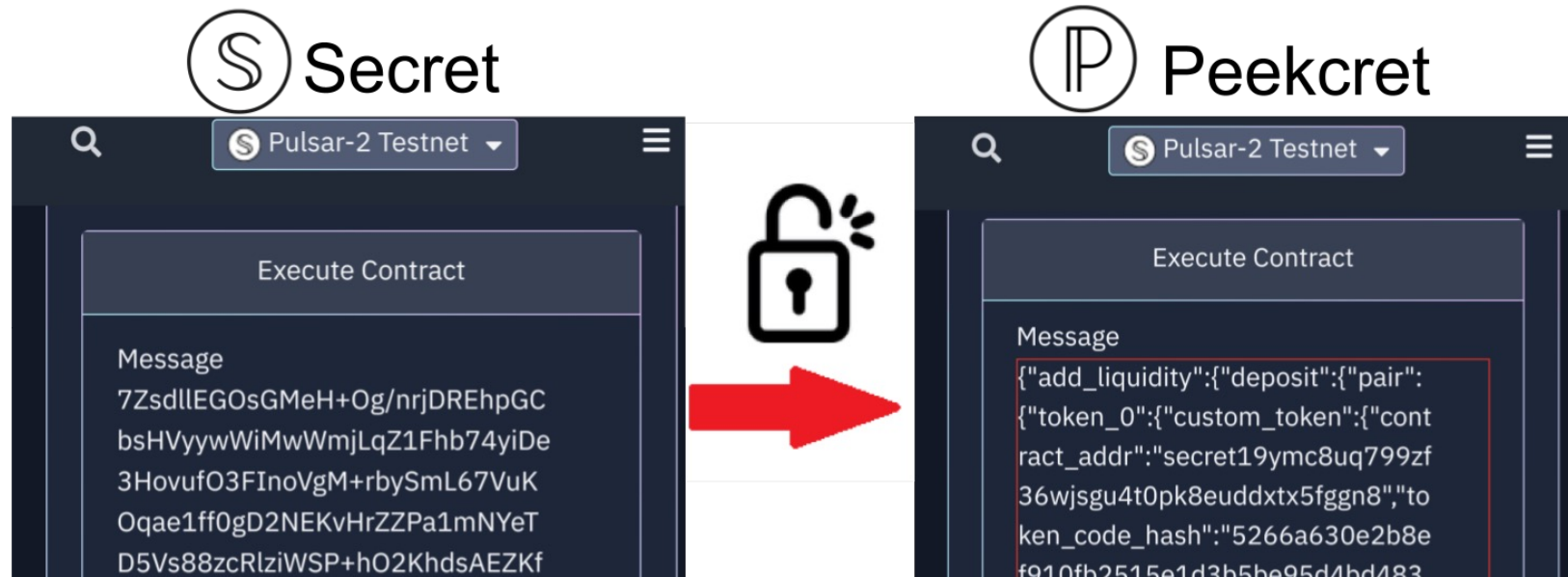
となり、XORの対称性により

$$W_1^{N+1} = W_2^N \oplus W_2^{N+1}$$

が成立する

- これは即ち、**N**ラウンド鍵と**N+1**ラウンド鍵のそれぞれ**第2ワード同士をXOR**すれば、**N+1**ラウンド目の**先頭ワードが復元**でき、**N+1**ラウンド鍵が**完全に抽出**できる事になる
- 使用する**AES関数の仕様** (S-BoxやRcon等) は**公開情報**であるため、あるラウンド鍵から**1つ前のラウンド鍵を導出する材料**はこれで**全て揃う**事になる
- どのラウンド鍵を抽出したかは分からないため、128bit AESのラウンド数である**10通りのブルートフォース**を行い、最終的に**IPFSLのAES鍵を復元しコンセンサスシードを復号**できてしまう

- 解読したコンセンサスシードを使用し、以下のように**SECRETのブロックのランザクションの復号に成功**している
(図は[16]より引用)





- 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

参考文献 (1/3)



- [1]“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>
- [2]“MDS: Microarchitectural Data Sampling”, 2023/7/20閲覧, <https://mdsattacks.com/>
- [3]“Intel SGX Explained”, Victor Costan & Srinivas Devadas, <https://eprint.iacr.org/2016/086.pdf>
- [4]“Overview on Signing and Whitelisting for Intel® Software Guard Extension (Intel® SGX) Enclaves”, Intel,
<https://www.intel.com/content/dam/develop/external/us/en/documents/overview-signing-whitelisting-intel-sgx-enclaves-737361.pdf>
- [5]“SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control”, Jo Van Bulck et al., <https://jovanbulck.github.io/files/systex17-sgxstep.pdf>
- [6]“LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection”, Jo Van Bulck et al., <https://lviattack.eu/lvi.pdf>

参考文献 (2/3)



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

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

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

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

[11]“Local APICについて - 睡分不足”, 2023/7/25閲覧, <https://mmi.hatenablog.com/entry/2017/03/27/202656>

[12]“ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture”, Pietro Borrello et al., <https://aepicleak.com/aepicleak.pdf>

参考文献 (3/3)



[13]“What is the semantics for Super Queue and Line Fill buffers?”, 2023/7/25閱覽,
<https://stackoverflow.com/questions/45783251/what-is-the-semantics-for-super-queue-and-line-fill-buffers>

[14]“AES key schedule - Wikipedia”, 2023/7/27閱覽,
https://en.wikipedia.org/wiki/AES_key_schedule

[15]“SoK: SGX.Fail: How Stuff Gets eXposed, by Stephan van Schaik et al.,
<https://sgx.fail/files/sgx.fail.pdf>

[16]“How Stuff Gets eXposed”, 2023/7/27閱覽, <https://sgx.fail/>

[17]“Questions about launch_token and EINITTOKEN”, Intel,
<https://community.intel.com/t5/Intel-Software-Guard-Extensions/Questions-about-launch-token-and-EINITTOKEN/td-p/1094870> (魚拓: <https://archive.is/vp6hL>)