13. TEE.fail

Ao Sakurai

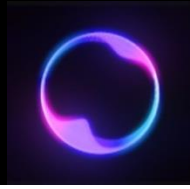
2026年度セキュリティキャンプ全国大会
L1 – 暗号・CVMビルド&スクラップゼミ

本資料の目標

- Scalable TEEに対する強力な物理攻撃の1つである、TEE.failについて説明する

TEE.fail

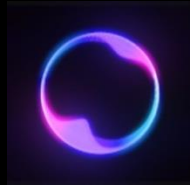
DDR5バスの概要



- Battering RAM[1]の方でも説明した通り、TDXをサポートする全てのCPUは**DDR5 RDIMMの使用を強制**している
  - DDR5 DIMM仕様は2020年に公開されている

- DDR5 RDIMMの1モジュールは288ピンを有し、さらにそのモジュールは内部で2つの**サブチャネル**に分割される
  - DDR4で説明したチャネルとは異なり、モジュール（基板）内部で分けるものである点に注意

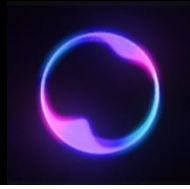
DDR5バスの概要



- 各サブチャネルにはデータ伝送用のピンが40本、そしてBattering RAMでも活躍したCA用ピンは、前述の通りDDR5では7本存在する
  - CAピンは、Battering RAMでも出てきたCAバス内のアドレス線等への接点となる金属ピンと考えれば良い

- さらに、DDR5のCAとデータは立ち下がりエッジと立ち上がりエッジのペアとしてエンコードされ、各サイクルで80bitのデータと14bitのCAが伝送される

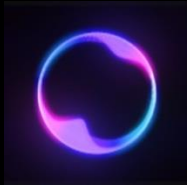
DDR5バスの概要



- メモリからの読み出しの際にはまず読み出し対象のサブチャネルを決定し、その後はDDR4と同様、行アクティベーションコマンドを発行する

- このコマンドが発行されると、以下の情報がCAピンにエンコード（書き込み）される：
  - 17bitの行アドレス
  - 3bitのチップID（ランク）
  - 5bitで表現されるバンクグループとバンクアドレス（本スライド101ページ目の図におけるBGとBA）

TEE.fail



- このように、サブチャネルの登場、CAバス幅の削減、コマンドの2サイクル化等、DDR5はDDR4に比べて大きくその構成が変わっている

- よって、Battering RAMでも議論の通り、DDR4向けのインターポーザは基本的にDDR5向けに転用できない

TEE.fail

- そのような中、DDR5向けの「**暗号文の読みに徹する**」**受動的インターポーザ**を作成し、それを用いて各種TEEに対する攻撃を仕掛ける**TEE.fail**が発表された
  - Battering RAMと異なり、能動的に接地を切り替えたりはしない

- 元々はDDR4向けにWireTapという同様の攻撃が同じ著者らから発表されていたが、それを改良した形となる

- インターポーザは**1000ドル**で作成可能で、Battering RAM程ではないがMembuster等に比べれば依然安価である

TEE.failの脅威モデル

- TEE恒例の脅威モデルに則り、攻撃者はrootレベル（**特権**）のアクセスを有するものとする
  - 特権を有するので、カスタムカーネル・ドライバのインストール、攻撃対象のユーザ空間設定やメモリマッピングも操作できる

- さらに、攻撃者はマシンに物理的にアクセス可能であり、かつ以下の事をできる程度の、**比較的初歩的な電子工作スキル**を有するものとする：
  - 単純な電気的組み立て作業（例：はんだ付け）
  - 攻撃対象マシンへの部品取り付け

- メモリ基板のPCB回路編集や、電子顕微鏡による検査のような**高度な攻撃能力は一切必要としない**

TEE.failの攻撃対象

- TEE.failでは、Scalable-SGX、Intel TDX、AMD SEV-SNPの全てを攻撃対象とし攻撃に成功させている
  - DDR5限定のTDXの攻撃に成功しているのは、WireTapやBattering RAMに対する大きな違いである

- SGX/TDXマシンの構成は以下の通り：
  - マザーボード：Supermicro X13SEI-F
  - CPU：Intel Xeon Silver 4509Y
  - BIOSバージョン：2.5a、CPUマイクロコードバージョン：0x2b000639
  - メインメモリ：16GB DDR5 RDIMM×8、4800MT/sで動作
  - ゲスト/ホストOS：TDX・Canonicalパッチ適用済みUbuntu 24.04、カーネルはバージョン6.8.2ベースのカスタムカーネル
  - Intel SGX/TDXバージョン2.25、DCAPバージョン1.22

TEE.failの攻撃対象

- AMD SEV-SNPマシンの構成は以下の通り：
  - マザーボード：ASRockRack BERGAMOD8-2L2T
  - CPU：AMD EPYC 9015
  - BIOSバージョン：10.02、CPUマイクロコードバージョン：0xb002116
  - メインメモリ：16GB DDR5 RDIMM×1、4800MT/sで動作
  - ゲスト/ホストOS：Ubuntu 24.04、カーネルはSEV-Stepに基づくパッチを当てたバージョン6.14.6ベースのもの
  - SEV-SNP FWバージョン：1.55.54

- TDXマシンではPCSに問い合わせた結果RAステータスが「**OK**」（**信頼可能**）であり、SEV-SNPマシンは**Ciphertext Hiding**機能が正常に有効化されている状況である

TEE.failインターポーザの構築

TEE.failインターポーザの構築

- TEE.failはBattering RAMとは異なり、TEEメモリ上の暗号文の読みに徹して、観測された**暗号文**から**サイドチャネル**的に**秘密を推測・特定**するタイプの受動的インターポーザ攻撃である

- よって、まずはDDR5メモリバスのトラフィックを観測するためのインターポーザを以下のステップで構築する
  - ステップ1：バス速度の低速化
  - ステップ2：メモリバスへの物理アクセス方法の確立
  - ステップ3：CPUのメモリ回路への電気的負荷の削減
  - ステップ4：インターポーザ部品の調達
  - ステップ5：インターポーザの構築本番
  - ステップ6：トラフィック観測用のロジアナのセットアップ

ステップ1：バス速度の低速化

- データ取得装置（インターポーザ）のコストは、読み取り対象の信号速度が**高ければ高いほど跳ね上がる**

- よって、可能な限り安価に構築できるように、まずは**メモリ速度**を可能な範囲での**下限まで落とす**事を考える

- 1つの方法としては、**BIOS設定**を変更する事で、このDDR5メモリの最低サポート速度である3200MT/s（クロック1.6GHz）にまで落とすという方法が有効である
  - 前述の通り、元々の速度は4800MT/s

ステップ1：バス速度の低速化

- または、装着されたDDR5 RDIMMのいずれか1つの**SPDデータを変更**する事でそのメモリ速度を3200MT/sに低下させる方法も有効である

- この場合、1つのDIMMの速度を下げると、それに合わせる形で他の全てのDIMMの速度も3200MT/sまで低下してくれる

- いずれの場合も、SGX、TDX、SEV-SNPといったTEEはマシンのメモリ速度を**TCB外と見なしている**点に注意
  - メモリ速度はTEEの信頼性要件に含まれないため、例えばSGX/TDXであれば、これを弄っても**RAステータスに負の影響は与えない**

ステップ2：メモリバス物理アクセスの確立

- 次は、マシンのメモリバストレース（要はメモリの配線）への物理アクセスを容易にする方法を確立する

- 物理アクセスを行うには配線を取り付けたいが、マシンのDIMMスロットやDIMMそのものに行うのは難しいため、スロットとDIMMの間に**DIMMライザー**（12ドル）を噛ませる
  - これにより、着脱容易な**ライザーのみを改造するだけで良くなる**

- ライザーは、DIMMコネクタメスとDIMMコネクタオスを備えた、DIMMとマザーボード間のパススルーとして機能するシンプルなPCBである
  - 本来は、パーツ取付の上で限られたスペース内で配置を工夫するため等に使われるもので、攻撃用とかでは一切ない

ステップ2：メモリバス物理アクセスの確立

- ライザーボードの具体例は以下の通り：

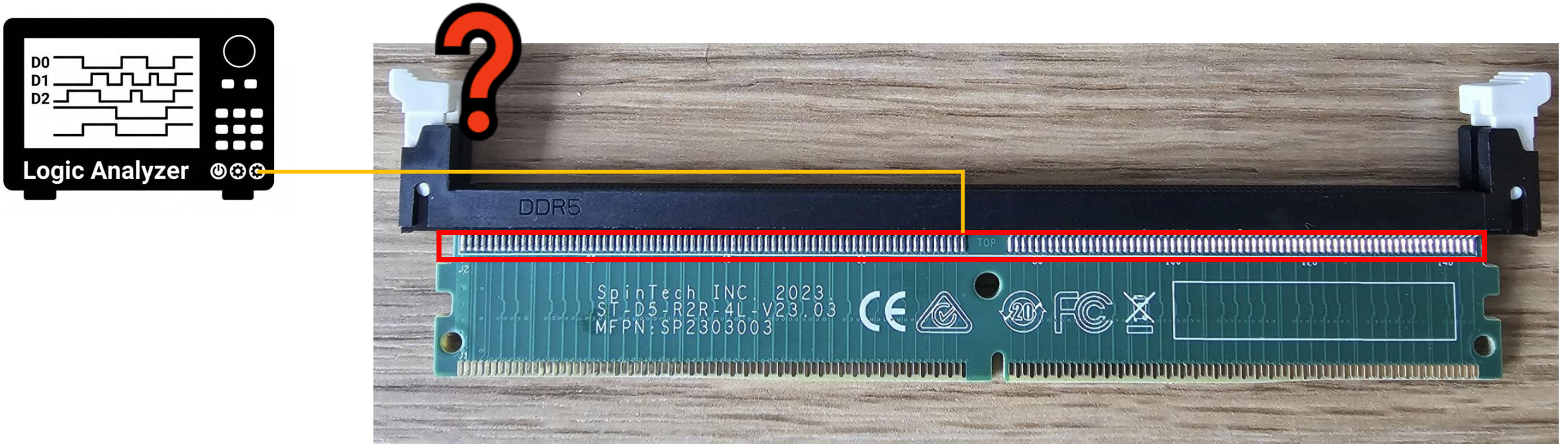
TEE.failにおけるDIMMライザー[2]。
DIMM（緑色の基板）が差さっている状態。

DDR4用ではあるがDIMMライザー（画像出典）。DIMMの反対側（黒部分）は、マザボ側DIMMスロットに差せるようになっている

差し方だけで見れば、任天堂DS用改造デバイスであるプロアクションリプレイは近い。こちらはパススルーどころか思いっきり改造処理を加えるが（画像出典）

ステップ3：CPUのメモリ回路への電気的負荷の軽減

- 愚直に考えると、ライザーにDIMMを差した状態で、ライザーのDIMM差込口の金属部分にはんだ付けした配線を直接ロジアナに繋げば、流れる電気信号を傍受・観測できるように思える



赤枠内の銀色のライザー金属部分にワイヤーをはんだ付けし、それを直接ロジアナに繋いでしまえば、愚直には流れる電気信号を観測できそうであるが…？

ステップ3：CPUのメモリ回路への電気的負荷の軽減

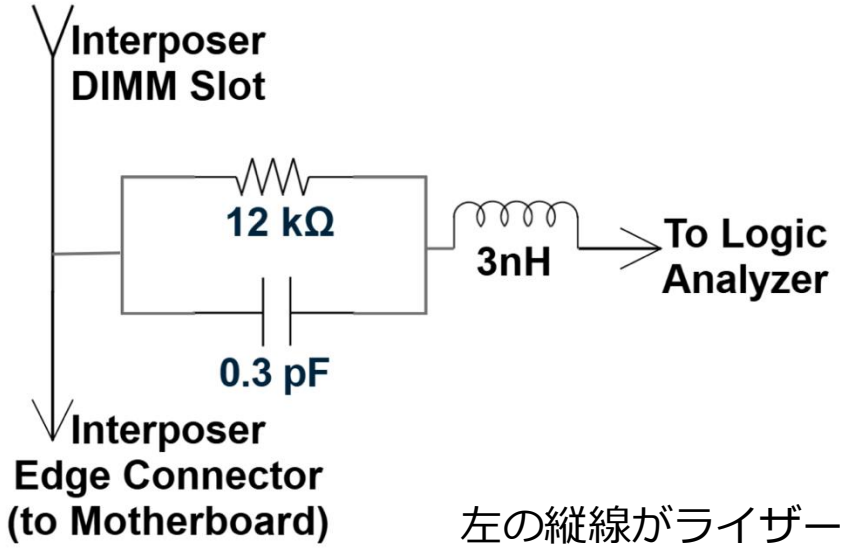
- しかしこれを実際に試してみると、CPUのメモリ回路に大きな電気的負荷がかかってしまい、**RAM認識エラー**により**マシンが起動しなくなる**事が判明した

- この電気的負荷の削減のため、既製の（攻撃とは無関係な）高周波バス向け設計のプローブである、Keysight SoftTouch プローブをリバースエンジニアリング（分解・参照）する
  - プローブは、デジタル回路から電気信号を読み取る端子つきケーブルと考えておくと良い

プローブの一例。各ケーブルが「プローブ」に相当し、
右上の部分でまとめられて「ポッド」を構成する（画像出典）

ステップ3：CPUのメモリ回路への電気的負荷の軽減

- このSoftTouchプローブをバラして見てみると、以下の回路図[2]のような抵抗・コンデンサ・インダクタからなる回路である、「**プローブ絶縁ネットワーク**」を構成している事が分かった
  - 恐らく一般的な用語ではない



左の縦線がライザーで、そこからロジアナに向けた配線（プローブ）を出している。
その際直接ロジアナに繋ぐのではなく、抵抗・コンデンサ・インダクタからなる
追加の回路を噛ませている事が分かる

ステップ4：インターポーザ部品の調達

- ここまでにより、ライザーを噛ませてプローブ絶縁ネットワークを挟んだ配線でロジアナに繋げば、DIMMの電気信号を観測できるであろう事（インターポーザの設計）が確立できた

- よって、次はこのインターポーザを実際に構築すべく、各種部品の調達を行っていく

- 論文では、必要な部品の回収元としてKeysight（旧：Agilent）のN4252Aというトランジションプローブアダプタに目をつけている
  - [12]によれば、ロジアナとプローブとの間に装着する、規格の差を吸収するための変換装置のようである

ステップ4：インターポーザ部品の調達

- N4252Aをバラして基板を見てみると102チャネルが搭載されており、その全てが前述の**プローブ絶縁ネットワークを備えている**ため、これを取り出して再利用する事を考える
  - おまけにN4252A自体は中古市場で40ドルと安い

- N4252Aのデータシートは入手できなかったが、このパーツに記載されている内容から、2008-2017年にかけてIntelが採用していたQPIプロトコルのデバッグに使われていたらしい
  - 実際、自分で検索してみてもこのパーツに関する情報は極端に少ない

ステップ5：インターポーザの構築本番

- まず、N4252AのPCBをヒートガンで加熱し**コンデンサと抵抗のペアを回収**する
  - インダクタはどこに存在するのかは論文の記述からは不明。後述のN4834内でそれ相当の機能がカバーされている可能性が考えられる
- 次に、はんだごてでそれをStep2で説明した**ライザーに接続**する
- これにより**DIMM側のインターポーザの工作は完了**したため、次は**ロジアナに繋ぐ側の工作**を行う

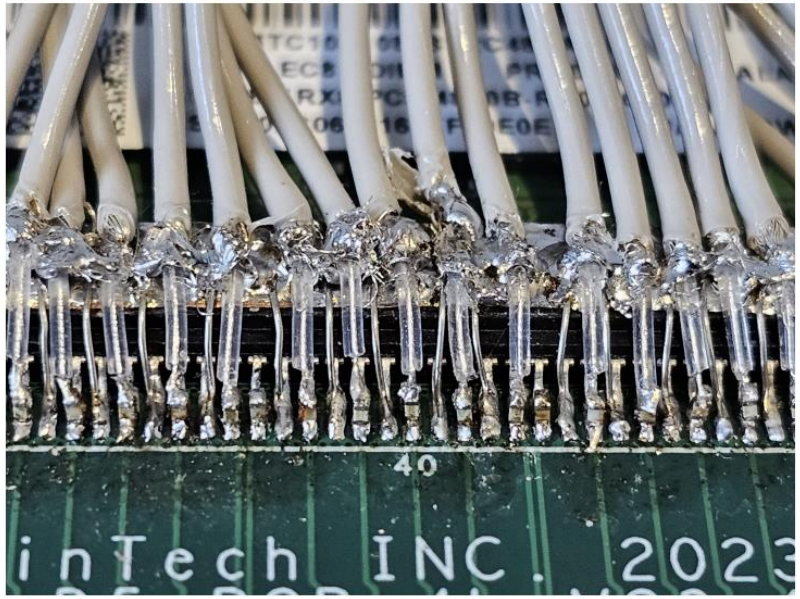
ステップ5：インターポーザの構築本番

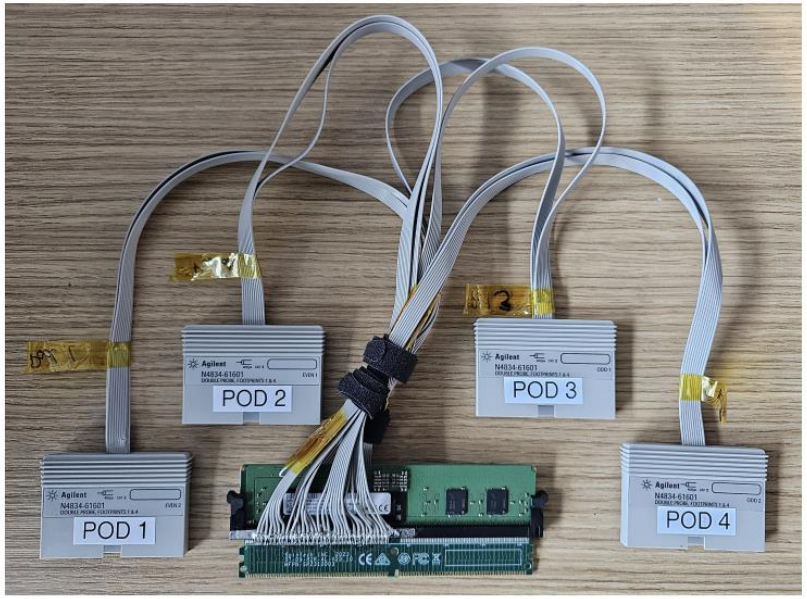
- ロジアナ側接続のため、Keysight N4834 SoftTouchプローブのSoftTouchコネクタ（DIMM方向側の端子）を切断する
  - これを合計4つのN4834に対して実施する。以下同様

- このプローブは90ピンポッドであり、つまり上記の切断により1つあたり90本の配線が出る事になるので、**その配線を先程の抵抗-コンデンサペアの先に直接はんだ付け**する
  - ちなみにポッドとは、プローブをまとめたものらしい
  - このリンク先では、複数プローブをまとめて1つのヘッドにする事でより効率的な測定ができると記述している

ステップ5：インターポーザの構築本番

- これにより、プローブ絶縁ネットワークを備えた、高信頼性のメモリインターポーザの構築に成功し、攻撃対象マシンへの悪影響が出る事も無かった
  - さらに、標準的なKeysightの90ピンコネクタを使用したため、このプローブは多様なKeysight機器との互換性を有する





画像は[2]より引用。左はライザーに接着したの抵抗-コンデンサペアの様子。
右はさらにそこにポッドを装着し、ロジアナに接続できるようにした状態の写真。

ステップ6：ロジアナの選定

- 後は実際に観測に用いるKeysight社製ロジアナを選定すれば攻撃用のセットアップは完了する

- 攻撃対象のメモリクロックを3200MT/s（1.6GHzクロック）まで下げられる事を鑑み、1.6GHz信号の観測が可能な、Agilentの16962A取得カードを2枚用意する（各110ドル）
  - 取得カード：原文はAcquisition Card。要は実際に信号の観測を行うロジアナの心臓部

- これらのカードをKeysight 16902Aロジアナシャーシ（中古品550ドル）に設置する

ステップ6：ロジアナの選定

- このロジアナの内蔵コンピュータを、Pentium IIIベースのCPUから Lenovo ThinkCentre M920Q MFF i5-8500T（中古品150ドル）にアップグレードした

- 最後に、このHWに対応する最新版である、Agilentのロジック・プロトコルアナライザのアプリケーションバージョン5.9をインストールした
  - ロジアナとプロトコルアナライザ双方の機能についてのアプリケーションという事であると思われる

ステップ6：ロジアナの選定

- 最終的な構築の成果物と、それを攻撃対象マシンに接続した様子の写真は以下の通り[2]：

バストランザクションの観察

- インターポーザを構築しロジアナに接続した事で、
  以下のようにバストランザクションを観測できるようになった：

画像は[2]より引用。
上側は波形ビュー、
下側はリストビューでの
Readコマンドの様子。

バストランザクションの観察

- 観察すると、まず行アドレス、バンクアドレス、バンクグループを含む行アクティベーションが送信される
- 次に、しばらくして同じバンクアドレスとグループ、及び列アドレスを伴う読み出しが発行される
- 最後に、読み出し遅延の後、フルバーストの読み出しデータがデータライン上で観測される
  - フルバースト：規定で定められた最大長のバースト

Enclaveメモリと実行の制御

Enclaveメモリと実行の制御

- これにより読みに徹する受動的インターポーザが用意できたため、次はこれを用いてSGXに対して攻撃を仕掛けるべく下準備を行う

- まず、Scalable XeonマシンでSGXを有効化するには各メモリコントローラ（MC）の**最初のスロット全て**に**DIMMを装着**する必要がある

- 論文著者らのマシンでは結果的に8つの16GB DDR5 RDIMMを装着し、これはDDR5 RDIMMのサブチャネルも鑑みると合計で16チャネルサポートする事を意味する

Enclaveメモリと実行の制御

- しかし、構築したインターポーザはマシンのメモリにおいて
  単一チャネルのバストランザクションのみを**監視できる**

- よって、このマルチチャネル構成をこのインターポーザで
  観測できるようにするためのセットアップを以降で実施していく

物理アドレスマッピングの取得

- Battering RAMは特定のアドレス線の接地の影響を受けるアドレスの法則さえ分かれば良かったが、TEE.failにおいてはより厳しい条件が必須となる

- TEE.failでは、インターポーザが装着された**単一チャネル上**に**攻撃対象データ**を**確実かつ強制的に存在させる**必要がある

- これを行うには、まず**物理アドレス**と**DRAM物理位置**との**マッピング法則**を**完全に復元**する事が大前提である
  - 部分的な解析だけでは確実に攻撃対象データを影響範囲内に配置するには不十分である

物理アドレスマッピングの取得

- この特定のため、論文ではIntelが公開しているアドレスデコードインタフェースを元に、デコードされたコンポネントから実際のマッピング関数を復元するアプローチを採用した
  - 先行研究では行バッファ競合というものを利用するアプローチを採るものもあるが、特にDDR5ではノイズが多い上に競合の誘発自体が格段に難しいらしい

- より具体的には、まずIntelがLinuxに公開している、物理アドレスから（部分的に）物理位置への変換を行う**ADXL**というメモリアドレス変換インタフェースに着目する

物理アドレスマッピングの取得

- このデコードインタフェースをsysfs経由でユーザ空間に公開し、以下のような各物理アドレスのDRAM物理位置を決定する要素を取得できるようにする：
  - メモリコントローラID
  - チャネルID
  - バンクグループとバンクアドレス
  - 行アドレス

- これを用いて次ページ以降に説明するように、攻撃対象マシンにおける物理アドレスからDRAM物理位置への実際のマッピング関数を復元する

物理アドレスマッピングの取得

- まず、既知の有効な物理アドレスaddrから開始し、前ページで説明した公開インタフェースを通す事でDRAM物理位置を取得する

- その後、物理アドレスビットのあるペア(i_0, i_1)ごとに、addr内のi_0番目とi_1番目のビットを入れ替え、その結果得られる新しいアドレスをデコードし結果を記録する
  - つまり、物理アドレスビットの内**2つを入れ替え**てどのように**DRAM物理位置に影響を与えるか**を**総当たり的に確かめる**

物理アドレスマッピングの取得

- どのアドレスがどの部分に影響を与えるかを把握したら、ガウス消去法（掃き出し法）を用いて、どのビットが線形関数を形成するかを決定する

- これはつまり、**出力ビット**が**全ての入力ビットをXORした結果であるか**どうかを各ビットについて見ていくという事である

- このように、XORのみで関係が表せる出力ビットを**線形出力ビット**と呼ぶ
  - 例：$bit_{out1} = bit_{in1} \oplus bit_{in2} \oplus bit_{in3} \oplus \cdots$

物理アドレスマッピングの取得

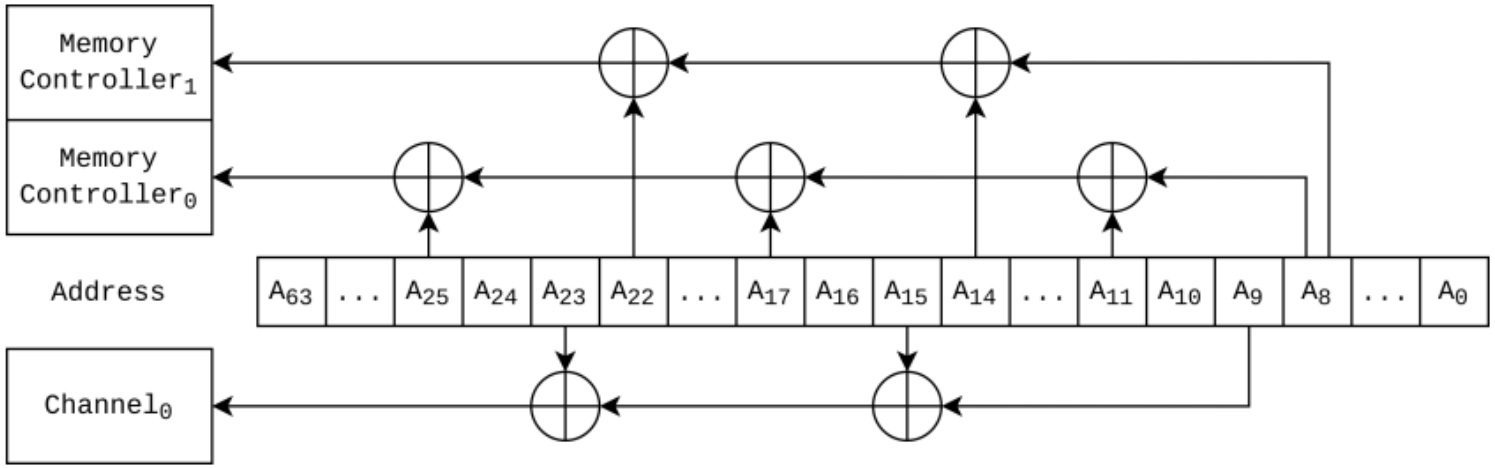
- 一方で、この線形出力ビットを特定するための線形方程式系が出力ビットに矛盾しているような場合は、それらを**非線形出力ビット**と見なす

- 非線形出力ビットについては、まずその入力アドレスビットの全ての可能な組み合わせに対応するアドレスをデコードし、この出力ビットの真理値表を直接構築する
  - 要は総当たり的に力技で真理値表を構築する

- そこから真理値表の最小化（例：カルノー図等）を実施し、その非線形出力ビットの論理式を導出する
  - 例：$bit_{out1} = (bit_{in4} \wedge bit_{in9}) \oplus bit_{in10}$

物理アドレスマッピングの取得

- これを一通り実施すると、以下のようなカオス極まりない**マッピング関数の論理式を特定**できる
  - 以下の図[2]はマッピング関数の一例で、上側が線形出力ビットについての図、下側が非線形出力ビットの論理式。Intelがこう設計したという話なので、この式の構成の意図等を結果から汲み取る必要はない



$$\mathbf{Row}_{11} \leftarrow (A_{31} \vee \neg A_{32}) \wedge (\neg A_{31} \vee A_{32}) \wedge (A_{33} \vee A_{34} \vee A_{35} \vee A_{36} \vee A_{37}) \wedge (\neg A_{36} \vee \neg A_{37}) \wedge (\neg A_{35} \vee \neg A_{37}) \wedge (\neg A_{34} \vee \neg A_{37}) \wedge (\neg A_{33} \vee \neg A_{37}) \wedge (A_{31} \vee A_{33} \vee A_{34} \vee A_{35} \vee A_{36})$$

SGXページ割当の制御

- 物理アドレスからDRAM物理位置へのマッピングを特定できたため、次はSGXのEPC内の関心対象の仮想アドレスがインターポーザから観測できるように仕向ける必要がある

- SGXは物理メモリ割当においてOSカーネルに依存するため、**監視可能な特定のDIMMチャネル**に**ページを割り当てる**ように**カーネルを修正**してしまえば良い

- このため、カーネルのSGXドライバを修正し、割当についてグローバルカーネルメモリに格納されるパラメータとして**仮想アドレスと物理アドレスのペア**を受け入れるようにする
  - 各ペアは**希望するマッピング**（対応する物理アドレスに固定する仮想アドレスというペア）である

SGXページ割当の制御

- Enclaveが初期化されカーネルがメモリ割り当てを試みる際、割り当てようとしている仮想アドレスが前述の入力で渡された**固定したい仮想アドレスであるか**を確かめる

- 仮想アドレスが上記の固定アドレス**ではない**場合、改造SGXドライバは対象物理アドレス以外の最初のページを返す
  - つまりは**どうでも良い無関係のアドレス同士でマッピング**させる

- 仮想アドレスが固定アドレス**である**場合、事前にカーネルに渡された希望マッピングを検索し、**対応する希望物理アドレスに確実に割り当てられる**ようにする

SGXページ割当の制御

- これを用いる事で、関心のある仮想アドレスを目的の物理アドレスに**正確に配置**する事が可能となる

- これはつまり**インターポーザを介して観測可能なDRAM物理位置**に**目当ての仮想アドレスを対応**させられるようになった事を意味する

SGX Enclave実行の制御

- 関心のある仮想アドレスを目的のDRAM物理位置に固定できるようになったため、次は**データ取得をロジアナと同期**させる事を**可能にする方法**で**Enclaveの実行を制御**する

- これを実現するには、**Enclave制御チャネル**（Enclave Control Channel）と**キャッシュスラッシング**（Cache Thrashing）の2つの手法を利用する
  - 制御チャネルはこれまた以前のスライドで説明したあの**Controlled-Channel Attacksと同じ文脈**である

Enclave制御チャネル

- データ取得のためにロジアナのトリガーを有効化（**観測開始**）するには、**関心のある特定のポイント**で**Enclave実行を一時停止**できなければならない

- SGXはroot権限を以てしても本番Enclaveにブレークポイントを打てないため、代わりに**Controlled-Channel Attacks**を用いて本番Enclaveの実行フローを制御する
  - 本番（製品版）Enclave：デバッグ版Enclaveでない、デバッグ用の機能が封鎖されるモードでビルドされたEnclave

Enclave制御チャネル

- より具体的には、まず対象プログラムをptraceでトレースし、Enclaveのページが初期化されるまで待機する

- 次に、実行を中断したコードページの権限を変更し、**読み取りと書き込みを禁止**する
  - 例えばmprotectでPROT_NONE権限にする等

- 攻撃対象Enclaveが**このコードページを実行**しようとすると、トレーサーが通知を受けて**ページの権限を復元できる**
  - 同時に、Enclaveの現在実行中のコードパスを把握しているため、**関心のあるアドレスのデータを取得**するよう**ロジアナをプログラム**する

Enclave制御チャネル

- この仕組みを用いる事で、**Enclaveの実行を追跡**しインターポーザを利用したいタイミングで**いつでも中断**できる

- この手法は特に関数呼び出し境界における中断に効果的である
  - 何故ならば、このような時Enclaveはしばしば別のコードページにジャンプするため
  - 逆に言えば、同じページ内で完結すると困るのでControlled-Channel Attacksの論文ではページフォールトシーケンスを提案していた（が、TEE.failではそこまでは恐らくしていない）

キャッシュスラッシング

- CPUキャッシュは、同一アドレスへのDIMMアクセスを繰り返す必要性を排除するためにデータを保持しメモリアクセスレイテンシを削減するものである

- これはデータ読み書き操作をCPU内部で完結させてしまうため、TEE.failのようにメインメモリを観測する攻撃を実行する上では困った事になる
  - さらに、SGX EnclaveではEnclaveキャッシュメモリのフラッシュや、TLB汚染攻撃防止のためか個別ページのUncachable（キャッシュ不可能）化もできない

- これを解決するために**キャッシュスラッシング**という手法を用いる

キャッシュスラッシング

- まず、**LLC**（ラストレベルキャッシュ）の**4倍**の大きさのメモリ領域を確保しランダムデータで埋める
  - LLCは、現代では一般的にL3キャッシュに等しい
- 次に、プログラムをEnclaveの**シブリングコアに移動**させ、プログラムが**攻撃対象Enclaveと全てのキャッシュを共有**するようにする
  - **シブリングコア**：ハイパースレッドは1物理コアあたり2つの論理コアを実現するが、片方の論理コアから見たもう片方の論理コアの事をこのように呼ぶ。「兄弟コア」の意
- SGXの古いEPID-RA時代はハイパースレッド有効化自体がRAのステータスに悪影響を及ぼす事があったが、2026年現在TDXマシンでは必ずしもその限りでないのを筆者が実機で確認済み

キャッシュスラッシング

- まず、**LLC**（ラストレベルキャッシュ）の**4倍**の大きさのメモリ領域を確保しランダムデータで埋める
  - LLCは、現代では一般的にL3キャッシュに等しい
- 次に、プログラムをEnclaveの**シブリングコアに移動**させ、プログラムが**攻撃対象Enclaveと全てのキャッシュを共有**するようにする
  - **シブリングコア**：ハイパースレッドは1物理コアあたり2つの論理コアを実現するが、片方の論理コアから見たもう片方の論理コアの事をこのように呼ぶ。「兄弟コア」の意
- SGXの古いEPID-RA時代はハイパースレッド有効化自体がRAのステータスに悪影響を及ぼす事があったが、2026年現在TDXマシンでは必ずしもその限りでないのを筆者が実機で確認済み

キャッシュスラッシング

- これにより、攻撃対象Enclaveのデータに対応する**キャッシュが全て追い出され続け**、データアクセスの際には**必ずメインメモリとやり取りせざるを得なくなる**

- これにより、攻撃対象Enclaveがマシンのメモリから**関心のあるEnclave変数を読み込んだ瞬間**に**ロジアナをトリガー**できる

Intel TEE Attestationに対する攻撃

Intel TEE Attestationに対する攻撃

- 攻撃の足がかりとして、まずはScalable TEEマシンの用いるメモリ暗号化が真に決定性（決定論的）であるかを実証的に検証する

- その後、インターポーザを用いてDCAP-RAの**PCKを抽出**し、任意のSGX及びTDXの**Quoteを偽造**できるようにする
  - これにより、SGX・TDXのエコシステムが**完全に侵害される**事になる

決定論的暗号化の検証

- 決定論的暗号化が実際に使用されている事の確認のため、まずは特定の仮想アドレスに書き込みと読み取りを繰り返し行うSGX Enclaveを作成する
- そして実際にこのEnclaveを動かし、固定仮想アドレスへの読み書きを繰り返させる
- 各ステップ後、ロジアナを用いて読み取りデータ（暗号文）をキャプチャする

決定論的暗号化の検証

- 具体的には以下のようなプログラムをEnclaveで実行する[2]：
  - 最初に固定値0x00、次に0xFF、最後に再び0x00をそれぞれ読み書きする

```
1   void ecall_experiment() {
2       memset(global_memory, 0x00, burst_size);
3       uncached_read(global_memory);
4       wait_for_logic_analyzer_collection();
5
6       memset(global_memory, 0xFF, burst_size);
7       uncached_read(global_memory);
8       wait_for_logic_analyzer_collection();
9
10      memset(global_memory, 0x00, burst_size);
11      uncached_read(global_memory);
12      wait_for_logic_analyzer_collection();
13  }
```

決定論的暗号化の検証

- 結果として、第1及び第3の読み取りに対応する暗号文は同一であった一方、第2の読み取りに対応する暗号文は異なったため、やはり暗号化が決定論的である事が確認された

NK ADDR	ROW 0-3	ROW 4-17	READ DATA	READ CHECK BITS	
3	F	3FFF	E55A 8366 C7EE 747A	B4BE	Read 1
3	F	3FFF	A373 7E8F 831E 8C27	7450	
3	F	3FFF	5F1F D06F 8E38 7463	117C	
3	F	3FFF	DF2A DF31 C88E 076C	39E1	
3	F	3FFF	90F9 1904 2AFB 7C41	308A	
3	F	3FFF	FD88 26E2 30B1 877E	F53C	
3	F	3FFF	96F4 C6A7 33F2 0092	FCAC	
3	F	3FFF	B777 0C42 68A0 7339	607D	
3	F	3FFF	FF03 606F EF7F 44C3	B7E8	Read 2
3	F	3FFF	C2E4 088C 7AF6 86DB	3B2E	
3	F	3FFF	3755 3225 DBEB A510	FDF2	
3	F	3FFF	DB57 9FB3 706C 02D6	E437	
3	F	3FFF	62E0 542C FB87 4A8A	7252	
3	F	3FFF	9DF9 488E 19C7 DB0F	A239	
3	F	3FFF	500B 50AC B2B6 35A7	852D	
3	F	3FFF	7EA7 6057 1AF6 EDDE	55CE	
3	F	3FFF	E55A 8366 C7EE 747A	B4BE	Read 3
3	F	3FFF	A373 7E8F 831E 8C27	7450	
3	F	3FFF	5F1F D06F 8E38 7463	117C	
3	F	3FFF	DF2A DF31 C88E 076C	39E1	
3	F	3FFF	90F9 1904 2AFB 7C41	308A	
3	F	3FFF	FD88 26E2 30B1 877E	F53C	
3	F	3FFF	96F4 C6A7 33F2 0092	FCAC	
3	F	3FFF	B777 0C42 68A0 7339	607D	

画像は[2]より引用

仮想アドレスと物理アドレスの影響

- 次に、メモリの**仮想アドレスが暗号文に影響を与えない**事を念の為**確認**する

- この確認のため、先程のテストと同じ物理アドレスを指す別の仮想アドレスでEnclaveを実行し固定値0xFFの書き込みテストを実施した

- 結果、以前に観測されたものと同一の暗号文が生成され、**仮想アドレスが暗号文に影響しない**事が**確認**できた

仮想アドレスと物理アドレスの影響

- 同じく物理アドレスの暗号文への影響の確認のため、元の仮想アドレスでEnclaveを実行しつつ、観測可能な物理アドレスを異なる値に固定した

- 結果、暗号化されたメモリ内容が異なる事を確認し、**物理アドレスが暗号文に影響を与える**事が**確認**できた

SGX Enclaveの測定値の影響

- 最後に、念の為Enclaveの測定値が暗号文に影響しない事を確認する

- このため、機能を変えないままコードを変更し、Enclaveへの署名鍵を変える事で、EnclaveのMRENCLAVEとMRSIGNERが異なる値に変わるようにする
  - それぞれ動作定義測定値（MRTD相当）、Enclave作成（署名）者測定値

- この新しいEnclaveを以前と同じ物理アドレスに固定して実行しても、このアドレスへの同一の平文の書き込みは**元のEnclaveと同一の暗号文を生成**する事が確認できた

SGX Enclaveの測定値の影響

- これによりシステム上で動作する**全てのSGX Enclaveが同一のメモリ暗号化鍵を共有**し、個別のTME-MK鍵はTDXのTDにのみ適用される事を確認できた

- Intel SGXは、そのマシンにおける全てのEnclaveの暗号化に単一のTME鍵を使用するという話は知られていたが、これを実験的に示した事になる

DCAP-RAへの攻撃

DCAP-RAへの攻撃

- DCAP-RAの**PCK**は、SGXとTDXの両方のAKを認証する鍵であるため、PCKを抽出させるだけで双方のRAプロセスを侵害し、**任意のAKを認証**できるようになってしまう

- これに着目し、**TEE.failではこのPCKを抽出**する事で、Quoteを偽造しIntelのTEE保証を侵害する事を考える

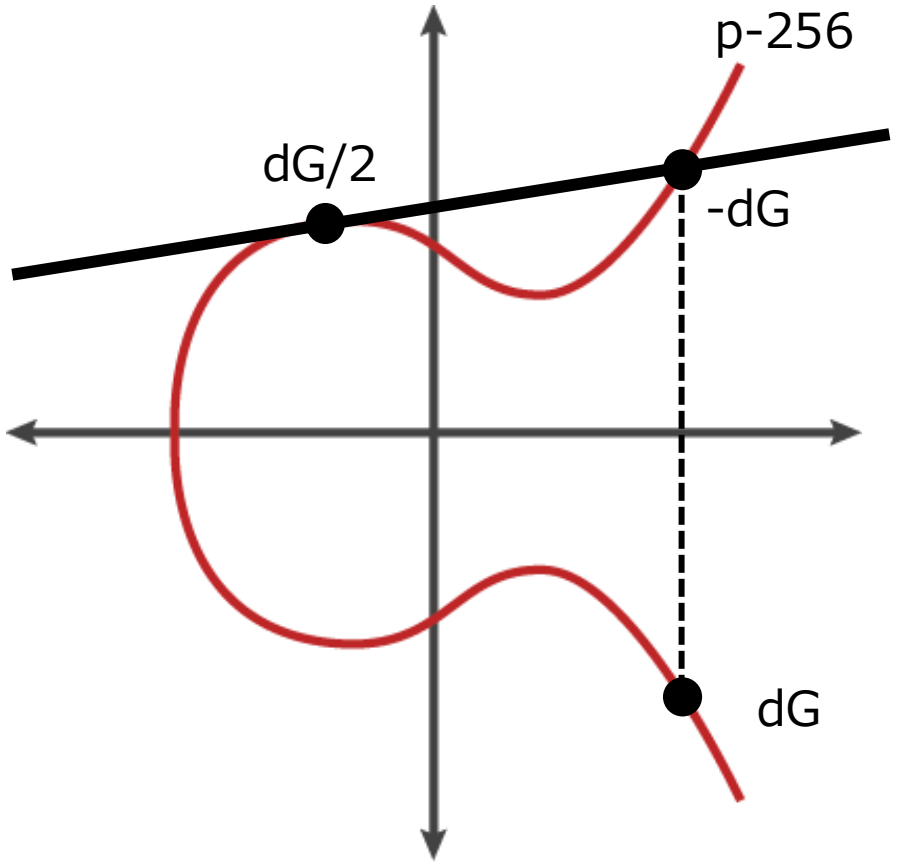
ECDSAの鍵生成

- デジタル署名の署名・検証のために、DCAP-RAでは**p-256曲線上のECDSAを使用**する

- p-256上の位数nの加法群に対する群生成元Gが与えられた場合、秘密鍵として$d \leq n$であるような乱数を選択し、p-256上の**点のスカラー倍算**を用いて公開鍵$Q = [d]G$を計算する
  - この秘密鍵と公開鍵のペアが鍵生成の結果である

ECDSAの鍵生成

- …と書くとまるで意味不明だが、要はLAのスライドで説明したこれである：



- 楕円曲線: p-256
  ベースポイント:G
  加算回数: d
  Q: dG
- dがnまで行くと無限遠点（0）に戻りまた1からはじまる
- 楕円曲線上の加算の一手法が点のスカラー倍算

※極めて雑な説明

ECDSAの署名

- メッセージmに署名するには、まずメッセージを$\log_2 n$ビットにハッシュ化し、その結果として整数zを得る
  - p-256ではこのハッシュは256bitとなる

- 署名者は次に、ランダムなnonceである$k < n$を選び、以下のようにr, sを計算しそのペア(r, s)を署名として出力する：
  - $(x, y) = [k]G$
  - $r = x \bmod n$
  - $s = k^{-1}(z + r \cdot d) \bmod n$
  - これらは次ページの検証式を逆算して得られた（設計された）式と言える

ECDSAの署名検証

- 署名の検証では、まず検証者はメッセージmからハッシュ値zを計算し、以下の2つを導出する：
  - $u = z \cdot s^{-1} \bmod n$
  - $v = r \cdot s^{-1} \bmod n$

- その後、検証者は$(x', y') = [u]G + [v]Q$を計算し、最終的に$r = x'$となる事を確認する
  - $s = k^{-1}(z + rd)$なので$ks = z + rd$、つまり$k = (z + rd)s^{-1}$
  - これを前提に$[u]G + [v]Q = (zs^{-1} + drs^{-1})G = (z + rd)s^{-1}G = kG$
  - $[u]G + [v]Q$（結果的にkGになるが検証者はk自体は知らない点に注意）のx座標（つまりxそのもの）を$\bmod n$し、その結果がrになれば正当な署名であると検証できる

IPP Cryptoの点のスカラー倍算

- ECDSAの実装のため、PCEは**Intel IPP Crypto**と呼ばれる
  カスタム暗号ライブラリを用いる
  - **IPP Crypto**: Integrated Performance Primitives Cryptography

- 完全に余談だが、SGXに対する攻撃の一種であるPlundervoltの
  論文でもそのRSA-OAEPが標的となっていた

IPP Cryptoの点のスカラー倍算

- 以下にIPP Cryptoライブラリにおける点のスカラー倍算アルゴリズムの簡略化された擬似コードを示す：
  - ちなみに今後ちょくちょくこのアルゴリズムの各行に言及するが、どうも論文側では行数の記述がところどころズレているようである

Require: A scalar k with Booth encoded digits $k_0 \cdots k_n$ valued $-16 \leq s_i \leq 16$ and a elliptic curve point G
Ensure: $R = [k]G$

1: $T[0] = 0$
2: $T[1] = G$
3: **for** $i = 2$ **to** 16 **do**
4: $\quad T[i] = T[i-1] + G$
5: $R = T[k_n]$ ▷ done in constant-time
6: **for** $i = n - 1$ **to** 0 **do**
7: $\quad R = 2^5 R$ ▷ using 5 point doubling operations
8: $\quad H = T[|k_i|]$ ▷ done in constant-time
9: $\quad$ **if** $k_i < 0$ **then** $H = -H$ ▷ done in constant-time
10: $\quad R = R + H$
11: **return** R

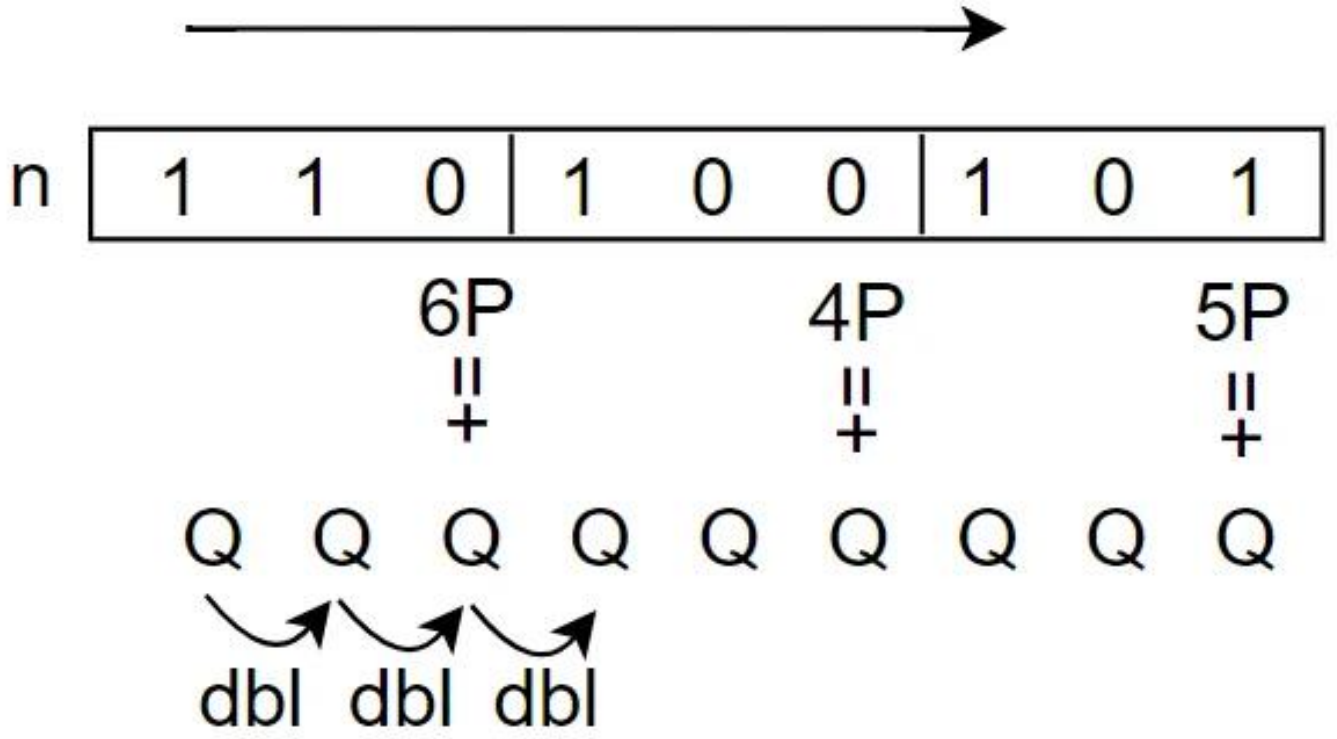
IPP Cryptoの点のスカラー倍算

- 入力は、ウィンドウ法を適用かつブース符号化した任意のスカラー値kと任意の点P、出力は別の点$R = [k]P$である
  - 全てのテーブル検索と点加算が定数時間で実行されるため、メモリアクセスパターンを介した情報漏洩は発生しない

- ウィンドウ法：kをwビットずつ区切り、wビットの範囲の数字それぞれでPを倍化した点を計算しておく事で、kを1ビットずつ見るより計算を高速化する手法
  - 例えば$w = 3$で区切ると各ウィンドウは0以上7になり（3bitであるため）、$0, P, 2P, \ldots, 7P$を計算しておいてそれらを使いながら加算を行っていく[3]
  - 逆にkを1ビットずつ見て1なら追加で倍化、0なら何もしないを繰り返すのがバイナリ法という手法である

IPP Cryptoの点のスカラー倍算

- 参考文献[3]より、$w = 3$である場合のウィンドウ法のイメージ図を引用する：
  - ビット毎に強制的に2倍にしているのは、2進数であるが故にビットシフトに伴い値を2倍にする必要があるため



IPP Cryptoの点のスカラー倍算

- 次に、例えばまた$w = 3$の時、$15P$を計算するには$7P + 7P + P$と分けて計算しなければならない

- しかし、もし$16P - P = 15P$とできれば、これは大幅な効率化を実現できる可能性がある
  - $16P$のような2のべき乗の処理は前述のビットシフト毎の倍化演算で付随的に得られるので使い回せる可能性が高い

- これを実現するための技術がブース符号化というものである
  - このブログ記事の説明がかなり分かりやすい[4]

IPP Cryptoの点のスカラー倍算

- IPP Cryptoに戻ると、このアルゴリズムはまず、事前計算（precomputation）テーブルであるTを初期化する
  - これにより、全ての$i = 0, \ldots, 16$に対し$T[i] = [i] \cdot G$となる
  - この$T[i]$の内容は、前掲の擬似コード行1-4を見ての通り、スカラー値kに依存しない
- その後出力であるRに対する最初の処理として、kの最上位桁k_nに対応する$T[k_n]$をRに代入する（擬似コードの行5）
  - このkは先程ECDSAの説明で登場した、$s = k^{-1}(x + rd) \bmod n$であるような値である事に留意
- ただし、k_nとはウィンドウ法でウィンドウ単位に分割され、かつブース符号により正負の表現が許されているという点に注意
  - この場合のアルゴリズムではウィンドウサイズは5bitと論文に書かれている

IPP Cryptoの点のスカラー倍算

- つまり、k_nというのは単一ビットではなく、ウィンドウ法の単位である5ビットである（例：$k_n = 11011$）

- さらにブース符号化されているので、$0\mathrm{b}11011$を27とするのではなく、$27 = 32 - 5$なので-5と表現する

- よって、この例の場合は先程のRの初期化は$R = -T[5]$となる

IPP Cryptoの点のスカラー倍算

- 次はそれより1つ下の桁について処理するが、ウィンドウサイズ単位でビットシフトするので、今回の場合は2^5をRに乗算する

- 次に、k_iの絶対値をインデックスとしたTの値をHとする。つまりは$H = T[|k_i|]$を計算する
  - もし$k_i = 10001$の場合、これはブース符号化表現で-15になるので、絶対値を取り$H = T[15]$を計算する

- かつ、もし上記の例のようにk_iが負数であった場合には、上記計算後にHを正負反転（$H = -H$）させる（行9）

IPP Cryptoの点のスカラー倍算

- そして、RにこのHを足す事でそれを新しいRとする、つまりは$R = R + H$とし、再び前ページの最初の処理に戻ってさらに1つ下の桁について同様に処理する（ループ）

- 各桁について計算が終わったら、結果のRを返す

- 一見ややこしいが、要は事前計算した値（T）を使って桁を考慮しつつ**各ウィンドウ毎に計算し合算**すれば、それが**ちゃんとkGになるのでRを簡単に計算できる**という**スマートな仕組み**である

PCKの抽出

- ところで、このkという値は$s = k^{-1}(z + rd) \bmod n$であり、つまりは$ks = z + rd \bmod n$、即ち$rd = ks - z \bmod n$、よって$d = (ks - z)r^{-1}$となる

- ここで、この式に登場する各パラメータの公開状況について整理する：
  - d: 秘密鍵なので当然**非公開**
  - k: **Enclave（PCE）内で扱われるため非公開**
  - s: 署名なので公開
  - r: 署名なので公開
  - z: メッセージから計算可能

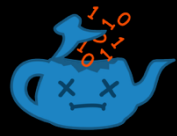
PCKの抽出

- これはつまり、何らかの方法でkを特定できてしまえば、前述の式から秘密鍵dを計算できてしまう事になる

- この秘密鍵dとは、PCEにおけるPCKによるAKへの署名処理においては**PCK秘密鍵**を意味する

- よって、TEE.failではこのkをインターポーザ攻撃で特定し、そこからPCK秘密鍵を逆算する事を考える

PCKの抽出



- 前述のアルゴリズムの8行目を見ると、Hは直接k_iをインデックスとした事前計算テーブルの値$T[k_i]$が楕円曲線上の座標として入っている

- 事前計算テーブルTはブース符号化されたk_iの絶対値をインデックスとしたものに対してのみ計算されているため、16通りしか存在しない
  - $-16 \leq k_i \leq 16$であるため、絶対値を取ると1～16の16通り
  - 0は点の加算処理で使わないので無視していると思われる

- そして、アルゴリズムの9行目ではk_iの正負に応じて正負反転されるため、結果として**Hは32通り存在**する

暗号文マッピングの構築

- Tの取り得る値の集合、ひいては**Hの値の集合**は公開情報から計算可能であるため、**攻撃者は把握済み**である
  - k_iは5bitは既定の情報、かつ曲線Gは判明しているので、一通りの$k_i G$を計算する事で簡単に算出可能である

- kの抽出のため、まず既知の平文をターゲット仮想アドレスに書き込むだけのSGX Enclaveを構築する
  - この仮想アドレスは、インターポーザで観測可能な物理アドレスaddrに固定する

暗号文マッピングの構築

- このaddrに、Hの取り得る全ての値を1つずつ書き込み・読み出しし、対応する暗号文サンプルを収集する
  - SGX Enclaveは全て単一のTME鍵で暗号化されるため、物理アドレスさえ合わせれば同じ平文でこの収集したデータが再現される点に留意

- その後、観測された**各暗号文**を、対応する**平文H**及び対応するk_iと紐づける
  - Hとk_iの関係は公開情報なので既知、そしてHを書き込んだ結果の暗号文をHと紐づければこの3つの関連付けが完了する

- このマッピングは**マシン再起動後にTME鍵が変更されるまで有効**である

kの復元

- 前述の制御チャネルとキャッシュスラッシングを利用したEnclave実行制御を用いてPCEを実行し、QE3 Reportの認証をトリガーする
  - これにより、PCEはQE3のAKに対しECDSA署名を行う

- 署名処理中に**Hの物理アドレス**を**固定アドレスaddrに固定**する事で、インターポーザを用いて**Hの暗号化された値を観測可能**にする

- ECDSA署名処理で前述のアルゴリズムに入ったら、6行目の**各ループ反復**でEnclave制御チャネルにて**実行を一時停止**する

kの復元

- その状態でロジアナをaddrへのメモリアクセスでトリガーするように設定し、10行目におけるHの読み取りで返されるデータを記録する

- これを繰り返し**Hの全暗号文**から先程**事前に取得したマッピング**に基づいて**各Hを復元**し、さらにそこから**各k_iを復元**する

- あとは**各k_iを連結しkを復元して抽出完了**。また、このPCEの処理結果である署名(r, s)も控えておく
  - ちなみに、k_iをブルートフォース的に特定しkを復元しようとしても、結局大体kのビット数、つまりはP-256では256ビット相当のブルートフォースに帰着するので、これは現実的ではない

PCKの復元とQuoteの偽造

- 前述した攻撃対象SGX/TDXマシンで実際にこの攻撃を行い、抽出したkと署名(r, s)から、前述の方程式を用いて**PCK秘密鍵を復元**した
  - この際、暗号文マッピングに約2分、PCEの実行とECDSA署名追跡に13分を要した

- あとはこのPCK秘密鍵でTEEに守られてすらいない**偽造AKに署名**し、この**偽造AKで署名したQuote**を同じく**TEE外で生成**する事で、**任意の内容のQuoteを偽造**しVerifierを騙せる
  - PCKが漏洩しているとはVerifierは知る由もないため、この偽造Quoteを検証しても、そのPCKのマシンのRAステータスがOKである限り**OKと判定**されてしまう

PCKの復元とQuoteの偽造

- TDXはRAにおいてSGXに依存し、かつQE3 AKもTDQE AKも同じPCKで認証されるため、この攻撃により**SGXもTDXも RAを破綻**させられる事になる

- TEE.failの論文執筆時点で、これはTDXのRAを完全に破る初めての事例である

BUILDERNETへの攻撃

BUILDERNETへの攻撃

- TDXとSGXのRAを偽造できるようになったため、これを悪用しまずは**BUILDERNET**というSWを攻撃する
  - いわゆるWeb3系のSWの1つ
  - 最終的な目標は、BUILDERNET SWをTDX外で実行しつつRAを通過させセキュリティ保証を破る事である

- これにより、悪意あるオペレータはBUILDERNETの**構成秘密**を抽出できるだけでなく、任意のブロックの構築やフロントランの気づかれず実行する事ができるようになる
  - 構成秘密は原文ではConfiguration Secrets。BUILDERNETにおいてTDXにより守らなければならない各種情報の事と思われる
  - フロントランについては後述

BUILDERNETの概要

- BUILDERNET：Ethereum向けのブロック構築オペレータのネットワーク

- Ethereum：分散型スマートコントラクトを構築するためのブロックチェーンプラットフォーム
  - **スマートコントラクト**：**ブロックチェーン**上での**処理・取引**を行うに当たって**自動実行されるプログラム**。例えばスマートコントラクトによる**検証の結果**、**特定の条件**を満たし**取引が許可された場合のみ**実際に**自動的に取引が行われる**、等の運用が出来る

- 要するにEthereumというブロックチェーンのブロックを構築するために使えるインフラの1つがBUILDERNETというイメージ

BUILDERNETの概要

- Ethereumでは、トランザクションはブロックにコミットされる
  - **トランザクション**：そのブロックチェーン上における**取引履歴**の事。
    ブロックの中身そのものでもある
  - **ブロック**：固定最大サイズのトランザクションのバッチ

- 12秒毎にランダムに選ばれたEthereumバリデータノードが
  次のブロックの内容を構築し、その後他のノードにより検証される

BUILDERNETの概要

- この構築の際、コンストラクタ（ブロック構築者）はブロックに含めるトランザクションだけでなく、その順序についても決定する

- トランザクション同士は互いに影響し合うため、順序次第で結果の値段等の価値が変動してしまう
  - 例えば高額なやり取りが先に来れば、その後のEthereumの相場に影響を及ぼし後続の金額も変わるであろう
  - かつ、この価値はバリデータが請求できる報酬に直結するため、バリデータとしても可能な限り最大化したい

- あるブロックにおける抽出可能な最大の価値の事をMEVと呼ぶ
  - MEV：最大抽出可能価値
  - このMEVに応じたバリデータへの報酬をMEV報酬と呼ぶ

BUILDERNETの概要

- 多くのバリデータは、MEV報酬の最大化のためにMEV-Boostと呼ばれるミドルウェアを実行し、分散型ブロック構築市場からブロックを調達している
  - ブロックの構築をバリデータ自身でなく市場の「ブロックビルダー」に任せ、バリデータは最終的にそれを承認するイメージ

- 市場のブロックビルダーはブロックを構築・提案するが、この時彼らはMEVを最大化するアルゴリズムを実行し、構築に利用できる広範なトランザクション情報源にアクセスしている
  - これにより構築・提案されたブロックを最終的にバリデータ（コンストラクタ）が承認し反映させる

BUILDERNETの概要

- BUILDERNETは、このMEV-Boost市場に貢献するビルダー集団の1つであり、Ethereumブロックの90%以上を構築している

- このBUILDERNETのビルダーは、公平性・機密性・証明可能なMEV再分配の実現のための**TDX TD上で動作**する

- BUILDERNETは例えば2025/5だけ19982ブロックを構築し、200万ドル相当のMEVを生み出している
  - 同月に68499（750万ドル相当）のブロックを構築したBeaver-Buildネットワークもビルダーを BUILDERNETに移行する過程にある（論文執筆当時）

BUILDERNETのセキュリティ保証

- BUILDERNETは、TEEにより全てのオペレータ（ビルダー）が正しいビルダーソフトウェアを実行している事を保証する
  - これは、ブロックがオープンで検証可能なアルゴリズムとSWによって公正に構築されている事を証明する事を目的としている

- さらに、TDX VM内で動作するビルダーサービス内に、機密Attested-TLSチャネルを構築するために**Attestationが使用**される
  - Attested-TLS：例えばReport DataにTLS証明書を同梱する等して、そのTLSセッションを特定のTEEインスタンスにバインドする手法

BUILDERNETのセキュリティ保証

- このAttested-TLSチャネルは、トランザクションデータと構成秘密の共有に使用され、その過程でこれらの機密性が保護されたままである事を保証する
  - 特にトランザクションデータの保護はフロントランニング攻撃の回避のために必要である

- **フロントランニング攻撃**：悪意のある当事者が保留中のトランザクション情報を入手し、より高い手数料を伴う新たなトランザクションを構築する事で、自身のトランザクションが先に実行されるようにする攻撃
  - 例えば、大口注文の前に自分の買付注文をこの攻撃でねじ込めば、大口注文後相場が上がった後にすぐ売却して利益が得られる

BUILDERNETのセキュリティ保証

- フロントランニングはユーザに重大な影響を与え、システムの公平性と完全性を失うだけでなく、手数料の高騰を招く
- この被害の回避のため、ユーザはBUILDERNET等の信頼可能な当事者とのみ共有されるプライベートプールにトランザクションを送信できる
- 2023年には、悪意あるMEV-Boostバリデータがフロントランニングで Ethereumから2500万ドルを搾取している
  - また、2021年の研究ではフロントランニングによる利益は既に3400万米ドルに達すると推定されている

BUILDERNETのアーキテクチャ

- BUILDERNETは個々のノードオペレータで構成され、これらは中央のBuilderHubサービスと通信する
  - BuilderHubサービスは、ノードのレジストリ維持と、設定及び鍵情報のプロビジョニングを担当する
- BUILDERNETでは、オペレータとして参加するにはFlashBotsによる手動承認が必要である
  - FlashBotsはMEVの健全化に貢献する研究開発組織
- ただ、適切なAttestationが提供されればあらゆる主体がノードオペレータになれるパーミッションレス型システムを目指している

BUILDERNETのアーキテクチャ

- TDX VMとして、BUILDERNETノードはそのVMのネットワーク内で内部通信を行う多くのサービスを実行する
  - これにはブロックビルダー、トランザクションプール、Ethereumクライアントが含まれる
  - 外部リクエストはTLSとRA（つまりAttested-TLS）により保護される

- カスタムプロキシは、外部公開されるVMポートに対してAttested-TLS終端を提供し、信頼可能サービスがAttestedなリクエストを発行できるようにするために使用される
  - これはBUILDERNETにおいてTDXの機能を直接利用する唯一のコンポネントであり、**他のコンポネント**は**TEE内で実行されているか否かを区別しない**

BUILDERNETのAttestation

- 論文執筆時点では、BUILDERNETはMicrosoft Azure向けのTDXのためのRAのみをサポートしている
- Azureでは、測定の実行とQuoteの生成にvTPMを使用する
  - TD QuoteはMRTDまでの測定だけを行い、RTMRは0クリアされている。MRTDはvTPM自体を測定し、かつReport DataにはvTPM AK公開鍵のハッシュ値を同梱する
  - vTPMはRTMR相当の測定を行い、vTPM Quoteとしてその測定結果を出力できる。これはTD Quoteで認証されているAK公開鍵で検証可能
- Report Dataに加えたいデータの追加やQuoteの取得も、AzureではvTPMに対するリクエストを経由して実施する
  - 具体的にはvTPMのNV Indexに対し読み書きを行う

BUILDERNETのAttestation

- BUILDERNETでは、Quoteの使い回しを防ぐため、Report Dataにリモート側が用意するnonceを同梱する
  - これは**ランタイムデータ**というものの一部として存在する以下のuser-dataに書き込まれ、ハッシュ計算の材料の1つとして使用される[5]

JSON Field	Description
keys	An array of keys in JWK format. Expected `kid`: `HCLAkPub` (vTPM AK public), `HCLEkPub` (vTPM EK public).
vm-configuration	Selective Azure confidential VM configuration.
user-data	64-byte data (HEX string) read from `0x01400002` NV index (Report Data).

表は[5]より引用

プライベートトランザクションと秘密への攻撃

- このBUILDERNETのセキュリティ保証を、前述の**Quote偽造能力で破綻させる**

- これにより、機密トランザクションの受信、構成秘密の取得、完全性チェックの回避が可能となってしまう

攻撃のセットアップ

- まず、無害なホストとして機能するBuilderHubインスタンス v0.2.1をセットアップする

- このインスタンスを初期化し、Azure TDXインスタンスに対応する、公式により公開されているTEE測定値のみを許可するように設定する
  - 具体的には、BUILDERNETイメージv1.4.0を使用するためその測定値の許可を行う
  - つまり、BuilderHubがBUILDERNETのビルダーノードをRAするという事である

攻撃のセットアップ

- 次に、別の非TDXマシンで改変したリバースプロキシを実行し、Attestationリクエストがこのマシンに来るようにする
  - 偽造Attestationを行うため、この改変リバースプロキシはこれのバージョン0.1.7のプロキシコードを基に実装する

- そして、この非TDXマシン上でローカルのトランザクションプロキシとビルダーインスタンスを起動する
  - 前述の通りこれらのコンポネントはTEE外で実行されているかを判別しないため、そのままデプロイできる

Attestationチェーンの偽造

- まず、ここで公開されている、信頼可能なBUILDERNETの一連の測定値とAzure TEE Quote（TPM QuoteとTD Quote）、つまりはリファレンスデータを事前に取得しておく

- 次に、Attestedプロキシのコードを改変し、以下の手順で偽造Attestation情報を生成する：
  1. Azure vTPMと通信するコードをスタブ化し、代わりにTPM AKとして機能する独自のRSAキーペアを作成する
  2. 事前に取得しておいたリファレンスデータを元にTPMランタイムデータを直接構築し、埋め込まれたAK公開鍵を上書きする
  3. 新しいランタイムデータをハッシュ化し、これを同梱する偽造TD Reportを生成する
  4. リファレンスTPM測定値を新しいTPM AKで署名してTPM Quoteを構築し、リファレンスデータからTPMイベントログをそのまま含める

Attestationチェーンの偽造

- この偽造Attestationデータは、プロキシサーバを起動し、BUILDERNETのドキュメント推奨の変更されていなBUILDERNETコードである、**attested-get CLIを利用して検証**する
  - attested-getが厳密にBuilderHubの内部検証ロジックと同一かは論文からは不明だが、少なくとも両者ともに同様の処理はすると思われる

- その結果、これらのデータは**有効**であり、**信頼可能な公開ノードと同一の測定値を返す**事が確認された

登録と機密データの取得

- この偽造Attestationフローにより、不正にRAを通過させBUILDERNETノードを登録する

- そして、Attested-TLSで保護されたBuilderHubエンドポイントにアクセスするため、プロキシをクライアントとして起動する
  - 勿論これは起動しているのは非TDXマシンである

- 起動後は認証情報を登録し、他のBUILDERNETノードからトランザクションを受信するためのリクエストを送信する
  - その結果、保護されたエンドポイントから**秘密と構成設定情報**、及び**アクティブなBUILDERNETオペレータのリスト**を**取得**できる

登録と機密データの取得

- これらの手順の完了により、悪意のあるノードオペレータは**以下の構成秘密や情報にアクセス**できるようになる：
  - Ethereumアカウントキー
  - リレー（異種チェーン間でのやり取りの事か）提出用署名鍵
  - ブロック入札サービスへのアクセス権
  - プライベートトランザクションアーカイブへの、アクセス情報を含む秘密へのアクセス

- Ethereumキーは構築済みブロックの署名とバリデータへの支払いに使用され、この鍵が管理するウォレットには**20万ドル相当のETH**が論文執筆当時で含まれていた
  - EthereumキーがBuilderHubが持つBUILDERNET全体共有の単一鍵なのか、ビルダーごとに違うのかは論文からは不明

登録と機密データの取得

- また、注文フローを閲覧できる能力を得ているため、悪意のあるオペレータは**プライベートトランザクション**を**フロントラン可能**となる

- かつ、偽造RAにより**完全性を主張**できてしまうため、攻撃者は信頼を維持しつつ**悪意のある活動を否定しながら利益を得る**事ができてしまう

TD及びGPUへの攻撃

TD及びGPUへの攻撃

- 次に、TDXに依存する**DSTACK**を基盤とするアプリケーションへの攻撃を行う事を考える

- **DSTACK**：コンテナ化されたアプリケーションをTEEにデプロイする事を可能にするSDK。既に多くのプロジェクトにおいてCVMバックエンドとして採用されている

- DSTACK SDKは、ユーザがTDX HW上でホストされているCVM（TD）にデプロイするためのAPIを公開する

TD及びGPUへの攻撃

- そこで、ユーザアプリケーションが攻撃者の制御するDSTACKノードに割り当てられるか、あるいは完全に不正なDSTACKクラウドにデプロイされるという一般的な脅威モデルを考える
  - これは本来、RAさえ正常にできる状態であれば未然に回避できる

- 正当なDSTACK API呼び出しと同等の出力を生成する実装を構築し、TDX外部でユーザが要求した信頼可能な環境を模倣する事で**TEEのセキュリティ保証を迂回**する
  - RAを偽造できるためユーザはこれに気づけず、TD外で動作するため当然秘密等も外部に筒抜けである

TD及びGPUへの攻撃

- 具体的なシナリオの1つ目として、**TD外で動作**する改変されたDSTACK PythonSDK呼び出しがユーザを欺きクラウドプロバイダのJupyter NotebookがTEE内で動いていると信じ込ませる
  - これにより、Notebookの内容を平文で閲覧可能となってしまう

- 2つ目に、独立したデバイスからの**NCCのAttestation**を、**あたかも自身のものかのように悪用する**手法を実証する
  - NCC：NVIDIA Confidential Computing
  - これにより、CCが提供する多くの保証を無効化してしまう

- 具体的には、信頼可能領域内での動作を保証するためにIntelとNVIDIAのAttestationを提供するLLMフロントエンドを破壊し、**TDXとNCCの外側にインスタンスを起動**させる

DSTACK攻撃手法

- まず、前述のQuote偽造能力を用いて**Attestation呼び出しを傍受・偽造**する**悪意のあるDSTACK API**を構築する

- 手始めに、TDX内で無害なDSTACKインスタンスを初期化し、正当なインスタンスの基準（Ground Truth）として設定する
  - このインスタンスから**ベースライン測定値（ランタイム測定値以外の測定値）を取得**し、攻撃の次のステップのために保存する

- 別途、TDXの外でDSTACKソフトウェアを起動し、**偽造Quoteを生成するように改変**する

DSTACK攻撃手法

- このDSTACK起動時、一連のイベントと関連ハッシュ（例：ファイルシステムのハッシュ）を記録する

- これらのイベントハッシュを用いて偽造TD Reportに入れる4つのRTMRを算出する

- これらのRTMRと先程のベースライン測定値から偽造TD Reportを作成し、さらに**偽造TD Quoteを作成**する
  - これは抽出した正当なPCKで認証されたAKで署名されているため、悪意があるにも関わらずそれを**RAで検知する事はできない**
  - DSTACKはCVMデプロイメントプラットフォームであるため、次は**これに基づいて構築された複数の製品のセキュリティ保証を破る**事を考える

「非機密VM」の構築

- まず、PHALA NETWORK等の、CVM内で動作するJupyter Notebookを提供するプロバイダに着目する

- DSTACK経由でホストされたNotebookは、Attestedな環境内で機密Pythonワークフローを実行する事をユーザに可能にする

- ユーザがNotebookに直接接続した場合、以下のいずれかの方法でNotebookのPythonランタイム内でTEEの状態をさらに検証し、TD Quoteを取得する事ができる：
  - DSTACK Python SDKを利用する
  - DSTACKのRPC APIを直接クエリする

「非機密VM」の構築

- いずれの方法でも、再生攻撃対策のためReportのカスタムデータフィールドを含める事が可能である
  - Quote取得後、ユーザはそれを自身で検証する必要がある

- 攻撃の上で、まずは前述の非CVM上で動作する**悪意のあるAPI**を作成する
  - このAPIは、動作している**基盤となるHWに関係なく有効な偽造TD Quoteを返す**ようにする

- 次に、DSTACKのネイティブRPCエンドポイントとして偽装し、**このAPIをユーザに公開**する
  - DSTACK Python SDKもQuote取得にはこのAPIを使うため、いずれのQuote取得方法でもこの改竄されたAPIを使う事になる点に留意

「非機密VM」の構築

- 最後に、プロバイダが指定した方法でJupyter Notebookを公開しユーザが接続できるようにする

- すると、ユーザが利用可能なDSTACKツールは、この改竄API から返される**偽造TD Quoteは正当なものであると評価**し、**RAを信頼可能であると判断**する
  - 再三の通り、抽出した正当なPCKで署名しているため

- 実際にはこのNotebookは**TDXの保護範囲外で動作**しているため、**セキュリティやプライバシは一切保証されない**

GPUリレー攻撃

- 次は、DSTACKを基盤とする別製品として、PHALA NETWORKとNEAR AIのプライベート機械学習SDKに着目する
  - この中でも特に**VLLM-PROXY**というものに着目する

- これはプライバシ保護型LLMフロントエンドであり、TDX内部で実行している事を証明する機能を有する

- このAttestationと測定値は、フロントエンドが特定のバックエンドと構成で動作している事を保証する事を目的としている
  - これにより、ユーザクエリのGPU内における機密性等、望ましいセキュリティ要件を強制できる

GPUリレー攻撃

- 次は、DSTACKを基盤とする別製品として、PHALA NETWORKとNEAR AIのプライベート機械学習SDKに着目する
  - この中でも特に**VLLM-PROXY**というものに着目する

- これはプライバシ保護型LLMフロントエンドであり、TDX内部で実行している事を証明する機能を有する

- このAttestationと測定値は、フロントエンドが特定のバックエンドと構成で動作している事を保証する事を目的としている
  - これにより、ユーザクエリのGPU内における機密性等、望ましいセキュリティ要件を強制できる

GPUリレー攻撃

- このSDKは、H100等の**NCC対応GPUを搭載したCVM上で動作**する事を想定している
  - NCCは、PCIeトラフィックを暗号化し、CVMの信頼ドメインにGPUも包含するように拡張する事ができる
- NCCは、CPUが真正なNCC GPUと直接インタフェースしている事を証明する**GPU Attestation**機能も提供する
- ただし、このGPU Attestation**はGPU上で実行する内容について特定する測定値は取り扱わない**
  - 基本的にCVM側の命令に基づきGPUでは実行されるため、**CVM側のRAで縛っておくべき**という話である

GPUリレー攻撃のセットアップ

- 攻撃の手始めに、ローカルの非TEEな攻撃用デバイスにRTX 3060を装着する
  - 当然こんなコンシューマ向けGPUにNCC機能は存在しない
- 次に、TDX外部にVLLM-PROXYを設定し、TDXまたはNCCとインタフェースする全ての機能をオーバーライドする
  - この時、GPU Attestation呼び出しを外部のH100搭載レンタルサーバに転送するように変更する
- ユーザがこの悪意あるVLLM-PROXYサーバにRAを要求すると、**偽造TD Quoteを生成**し、そしてレンタルサーバからオンデマンドで**GPU AttestationのEvidence（AR）を取得**する

GPUリレー攻撃のセットアップ

- このレンタルサーバ上のTDX VMはNCC Attestationを誠実に実行して結果を返却する
  - このレンタルサーバ上では攻撃専用コードは一切実行されず、その後データが保持される事も無い

- NCC Attestationは特定のCVMの同一性に紐づける事はできないため、悪意のあるVLLM-PROXYは**TDXとNCCの両方のAttestationを正常に通過させてしまう**

- NCCは、RPがRAでその正しさを検証する**CVMがAttestationする事で縛る**ため、その**真ん中のCVM（TD）のRAが破綻**する事で、**いくらでもNCC Attestationをザルにできてしまう**のが**根本原因**

GPUリレー攻撃の結果

- 実際に攻撃するため、VLLM-PROXYエンドユーザSWを設定し、サーバからARを取得させる

- このSWはNCC Attestationを正常に検証し、TD Quoteを検証するため、AUTOMATAのオンチェーンDCAP検証サービスの利用を推奨している
  - このAutomataもTDXに依存しているため、論文では補遺においてどう悪用しているかを説明しているが、本スライドでは省略

- 結果的に、ユーザは自身のGPU呼び出しが**NCC GPUを備えたTDX VM内で実行**されていると**誤って暗号学的に確信**させられる
  - が、実際には**TDでもなければRTX3060搭載**という滅茶苦茶な環境で動作している

GPUリレー攻撃の結果

- この攻撃を用いれば、プロバイダは単一のH100を多重化し、例えば数百のような数多くのインスタンス分のAttestationを提供可能となる

- 同様に、悪意のあるLLMホストは機密性を主張しつつも、密かに全てのLLMリクエストをロギングする事ができる

- H100は1機だけでも数百万円するが、スポットで借りたりすると1時間2.5ドルだったりとかなり安いので、**古いHWを使いつつもNCCを使っているとユーザに信じ込ませる事ができてしまう**

Enclave内秘密情報の直接抽出

Enclave内秘密情報の直接抽出

- これまでの攻撃では、抽出したPCKとそれで認証したAKで偽造TD Quoteを生成し、それによりRAを破綻させて攻撃するというアプローチを取っていた

- 次は、実際にインターポーザ攻撃により**SGX Enclave内の秘密情報を直接抽出**してセキュリティ保証を破綻させる

- 具体的には**SECRET Network**というブロックチェーンで使用されるEnclaveを攻撃して秘密鍵を抽出し、このチェーンが保証する機密性を完全に破綻させる
  - ちなみに、このSECRET Networkは同一著者らによるSGX.Failにおいても過去に一度SGX攻撃で破壊されている

SECRET Networkの概要

- **SECRET Network**：スマートコントラクトの実行をSGXの**Enclave内で行い、トランザクションも暗号化**する事を特長としている**ブロックチェーン**の1つ。略称はSECRET

- このSECRETは、 Enclave内で動作するように適応されたSGXベースのスマートコントラクト実行層と、独立したコンセンサス層で構成される

secret network

画像出典：https://learn.bybit.com/defi/what-is-secret-network/

SECRET Networkの概要

- スマートコントラクトにメッセージを送信するため、ユーザは既存バリデータノードから受け取った**マスター公開鍵**と自身のECDHKE秘密鍵から**共有秘密**を導出する
  - そしてそれによる暗号文をトランザクションに含める
- このマスター公開鍵は、**コンセンサスシード**というSECRETにおける信頼の根源となる鍵から導出される**マスター秘密鍵**に対応している
  - ~~SGX.Failの時のように~~侵害された場合にローリングできるようにするため、SECRETは初期コンセンサスシードと現在のコンセンサスシードの双方を維持する
- マスター秘密鍵は**ネットワーク全体に複製**されSGX Enclave内で**シーリング**（SGX HW鍵を用いての暗号化をしての保存）する

バリデータノードの登録

- **バリデータ**：SECRET Network上のオペレータ。トランザクションを実行し、各トランザクション後に新たなコンセンサス状態を検証する役割を持つ

- トランザクションを実行するには、バリデータは暗号化された**トランザクションを復号**できる必要があり、このためには**コンセンサスシードを受け取る必要**がある

バリデータノードの登録

- 新規バリデータノードは、SECRET Networkに登録するために
  RAを用いる必要がある

- まず、新規ノードはCurve25519のECDHKEで使用するための
  一時的なキーペアを生成する

- 一方で、既存バリデータノードはECDHKE鍵として**マスター公開鍵**
  及びマスター秘密鍵を用いる

バリデータノードの登録

- その後、新規ノードとSECRET参加済み既存バリデータノードが、DHにより各自のECDHKE秘密鍵と相手のECDHKE公開鍵から先程も言及した**共有秘密を導出**する

- この共有秘密は、新規ノードのECDHKE公開鍵を追加データとして用いた、128bit AES-SIVによる**コンセンサスシードの暗号化及び復号に使用**される

バリデータノードの登録

- 次に、新規ノードはブロックチェーンへの認証に使用される**ARを作成**する
  - この共有ARは**SGX DCAP Quote**とその**コラテラル**を**組み合わせたもの**で、Report Dataの先頭32バイトに新規ノードの公開鍵が含まれる

- ネットワークに参加するため、新規ノードは自身のウォレットの送信者アドレスとARを含むトランザクションをブロックチェーンに対してブロードキャストする

バリデータノードの登録

- ネットワーク内の既存ノードはこのトランザクションを監視し、自身のEnclaveを用いてARの検証を実施する

- 検証に成功した場合、その既存ノードは**コンセンサスシード**を**アンシーリング**し、**ECDHKEで導出した共有秘密**で**それらを暗号化**する
  - このコンセンサスシードは、恐らく前述の初期シードと現在のシードの双方を指していると思われる

- そして、既存ノードは連結された暗号文をencrypted_seedとしてトランザクションを更新する
  - 恐らく、初期シードの暗号文と現在シードの暗号文の連結

バリデータノードの登録

- そして、新規ノードは自身のECDHKE公開鍵でブロックチェーンにencrypted_seedを問い合わせ暗号文を取得する
- これを**ECDH共有秘密で復号**し、コンセンサスシードを取得する
- 最後に、機密性を確保するためにこれらのコンセンサスシードをEnclave内でシーリングする

バリデータノードの登録

- ここまでで登場した各鍵を整理すると以下の通り：

鍵名	役割	何で保護されるか
コンセンサスシード	マスター秘密鍵やマスター公開鍵の導出元であり、SECRETにおける信頼の根源	新規ノードへの転送時はECDHKE共有秘密。各ノードでの保存時はSGXシーリング
マスター秘密鍵	既存バリデータノード側のECDHKE秘密鍵として機能。ネットワーク全体に複製される。新規ノード追加のECDHKEの度に使い回される	SGXシーリング
マスター公開鍵	既存バリデータノード側のECDHKE公開鍵として機能。新規ノード追加のECDHKEの度に使い回される	-
新規ノード ECDHKE秘密鍵	新規ノードの一時的なECDHKE公開鍵	新規ノードのEnclave内で保持
新規ノード ECDHKE公開鍵	新規ノードの一時的なECDHKE秘密鍵	-
ECDHKE共有秘密	上記既存ノードのマスター鍵ペアと新規ノードECDHKE鍵ペアとで交渉されたECDHKE共有秘密	既存ノード・新規ノードそれぞれのEnclave内で保持
encrypted_seed	既存ノードから新規ノードに受け渡す際の暗号化されたコンセンサスシード	ECDHKE共有秘密

ECDHへの直接攻撃

- もし登録手続き中の**ECDHKE共有秘密を暴ければ**、それを用いて**コンセンサスシードを取り出し**、SECRETのセキュリティ保証を**完全に破綻させる**事が可能となる

- よって、この**共有秘密をインターポーザにより抽出**する事を考える
  - これはAEだけでなく通常のEnclaveもTEE.failの影響を受ける事を意味するため、Enclave開発者にとっても大きな問題である

- セットアップとして、前述のSGX/TDXマシンにて、Pulsar-3テストネットに参加するSECRETバリデータをセットアップする
  - そして、変更を加えていないSECRETバイナリとEnclave（v1.17.1）を実行する

ECDHへの直接攻撃

- SECRETのECDHアルゴリズムは、**マスター公開鍵**と**新規ノードECDHKE秘密鍵**に対してスカラー倍算を実行する

- より具体的には、SECRETのEnclaveは**モンゴメリラダー法**を用いてこの乗算を実行する
  - Montgomery Ladder Algorithm

- このアルゴリズムは、イテレーションごとに2bitのウィンドウ（前ビットと現ビット）を用いて秘密鍵をスライドさせていく

ECDHへの直接攻撃

- このアルゴリズムの実装の簡易的なコードを以下に示す[2]：

```
1  let mut bits = scalar.bits_le().rev();
2  let mut prev_bit = bits.next().unwrap();
3  for cur_bit in bits {
4    let choice = prev_bit ^ cur_bit;
5    conditional_swap(x0, x1, choice);
6    differential_add_and_double(x0, x1, aff_u);
7    prev_bit = cur_bit;
8  }
```

ECDHへの直接攻撃

- モンゴメリラダー法について説明すると、例えば$k = 0\mathrm{b}1101$であるようなkGを計算する事を考える

- モンゴメリラダー法では、常にR_0, R_1（SECRETが採用しているアルゴリズムではその各x座標のx_0, x_1）の2点を持つ

- この2点は、常に$R_1 = R_0 + G$、つまりは加算単位であるベースポイント分だけR_1が先に進んでいる状態という関係を維持している
  - これはモンゴメリラダー法がそう定義しているものである。最初は例えば$R_0 = 0P, R_1 = 1P$から始め、以降の計算でもこの関係性及び正しさが破綻しないように計算していく

ECDHへの直接攻撃

- kのビットを最上位から右方向に読み進めていく

- もしそのイテレーションで見ているビットが0である場合、2進数上でのビットシフトのみを考慮し、単純に点を2倍にするだけで良い。つまり以下のような処理を行えば良い：
  - $R_{0_{new}} = 2R_{0_{old}}$（単純に$R_0$を2倍）
  - $R_{1_{new}} = R_{0_{old}} + \mathrm{R}_{1\mathrm{old}}$（元々の$R_0$と$R_1$の加算結果を代入）

- $R_{1_{old}}$は$R_{0_{old}}$よりも$1G$だけ多いため、これを足すと$2R_{0_{old}} + 1$と等しくなる。これはR_0とR_1の不変条件である$R_1 = R_0 + G$を維持できている

ECDHへの直接攻撃

- 見ているビットが1である場合、ビットシフトの2倍に加えて+1が欲しいので、以下のような処理を行えば良い：
  - $R_0 = R_{0_{old}} + R_{1_{old}}$（元々の$R_0$と$R_1$の加算結果を代入）
  - $R_1 = 2R_{1_{old}}$（単純にR_1を2倍）

- $R_{1_{old}}$は$R_{0_{old}}$よりも$1G$だけ多いため、これを足すと$2R_{0_{old}} + 1$に等しくなる。一方、元々のR_1は$R_0 + 1$であるため、これを2倍にすると$2R_0 + 2$となる
  - これはやはり不変条件である$R_1 = R_0 + G$を維持できている

ECDHへの直接攻撃

- 重要なのは、ビットに応じて操作対象の点が違うというだけで、いずれも片方を単に2倍にする、そしてもう片方は2点間で足すという操作はいずれのビットの場合も同じである

- であれば、ビットが1の場合にのみ最初に2点をスワップしてしまえば、後はいずれのビットの場合も後続の処理は共通化（片方を2倍、もう片方を2点の足し算）できる

- これは結果としてスワップ、加算、倍算という共通の手続きをいずれの場合においても必ず実施するため、サイドチャネル攻撃で予測が困難である**定数時間実装**になっている

ECDHへの直接攻撃

- 最終的に、kの全ビット分を計算しきった後のR_0を最終的なスカラー倍算結果とする
  - R_1はあくまでも計算上の補助的な意味しか持たない

- 例えば$k = 0\mathrm{b}101$の場合、$R_0 = 0G$は$1G \rightarrow 2G \rightarrow 5G$となり、これは2進数101の10進数表現5に等しい

ECDHへの直接攻撃

- まず、コードは「選択値」（Choice）を計算する
  - これはウィンドウ内の2つのビットのXOR値である（4行目）
  - これに基づき、2つのビットx_0, x_1をスワップすべきか否かを決定する

- 前述の説明を見ると、R_0とR_1の意味（入れるべき式）をスワップする必要があるのは、ビットシフトの前後でビットが違う時のみ。よって、違う（XORが1）場合のみスワップするという仕組み

ECDHへの直接攻撃

- ここで、もしこのx_0が**各イテレーションの条件付きスワップ前後で値が変わったか**を**観測**したとする

- もし変わっていない場合はスワップ不要であるという事であるため、選択値は0である事になる
  - 変わった場合は逆に選択値は1である

- このように各イテレーションごとの選択値を推測し、**x_0の初期状態0から選択値に従い式を実行**していけば、最終的な点、つまりは**SECRETのECDHKE共有秘密が計算**できてしまう

ECDHへの直接攻撃

- よって、TEE.failではこのx_0を各イテレーションにてインターポーザで観測する事を考える

- 条件付きスワップ前後のx_0の観測のためにはEnclaveの実行を適切な位置で停止させる必要があるため、ここでも**Controlled-Channel Attacks**を用いる

- 具体的には、条件付きスワップ前後にEnclaveページをアンマップし、ページフォールト発生までEnclave実行を進め、発生後にページを再マップし正しい位置に到達させる

ECDHへの直接攻撃

- x_0の正確なアドレスを取得するため、SECRETのEnclaveをデバッグモードでビルドし、sgx-gdbを接続して実行する
  - さらに仮想アドレス空間が各実行間で一貫するようにASLRを無効化する

- 次に、各ループ反復（イテレーション）における条件付きスワップ前後の暗号文を取得するために**DRAMトラフィックを監視**する
  - 例の通り制御チャネル+キャッシュスラッシング手法を用いる

- 各ループ反復において前後で**同一の場合**は**選択値を0**、**異なる場合**は**1**と推定する

ECDHへの直接攻撃

- これを繰り返し、必要なトレースを全て収集するのに約90分を要し、最終的に256bitの内、最初の1bitと最後の3bitが欠落した**252bit**を抽出できた
  - 欠落の理由は論文補遺Cにあるが、アルゴリズム上の性質によるもののようである
- ただし、この欠落4bitはたかだか16通りしかないため、総当たりは容易である
- かつ、SECRETが暗号化に**認証付き暗号化方式**（AEAD）であるAES-SIVを用いているため、**正解の候補を入力すると正解であると識別する事ができる**

ECDHへの直接攻撃

- ちなみに、ASLRはあくまでも**仮想アドレスをランダム化**するものであるため、**物理アドレスには影響せず**、つまりは**デバッグ版でのこの実行で見たパターンを本番にも流用**できる

- 後は本番Enclaveでもインターポーザで上記パターンに基づきx_0の変化を観察し、選択値を算出し、秘密鍵の252bitを復元する

- 後は暗号化されたコンセンサスシードに対して欠落4bit分を総当たりで補いながら復号を試行し、**正解のコンセンサスシードを取り出せてしまう**
  - encrypted_seedを抽出した秘密鍵で復号しその中身のコンセンサスシードを取り出した形となる

ECDHへの直接攻撃

- SECRETの主な用途の1つはトランザクションのプライバシーを保護し、当事者がデジタル資産の管理権を非公開で移転できるようにする事である

- そして、初期及び現在のコンセンサスシードを抽出した事で、SECRET Network上の**あらゆるトランザクションを直接復号できる**事になる
  - これは同様にコンセンサスシードを漏洩させたSGX.Failでも実証済み

- これにより、AKを一切確認する事なく、**SECRETの機密性保証を完全に侵害**する事が可能になる

SEV-SNP CVMへの攻撃

SEV-SNP CVMへの攻撃

- Zen5アーキテクチャベースの EPYCサーバ上でのAMD SEV-SNPでは、前回スライドで説明した**Ciphertext Hiding**に対応している
  - これはCipherleaksのような暗号文を読み取っての攻撃を阻止するための機能である

- よって、TEE.failではCipherleaksに従い、OpenSSLのECDSA署名操作を実行する**SEV-SNP CVMを攻撃**する
  - Ciphertext Hidingがあるため、CipherleaksのようなSWベースの攻撃は通用しない点に注意
  - 攻撃対象マシンは本資料前半で説明済みのSEV-SNPマシンである

SEV-SNP CVMへの攻撃

- SEV-SNPは、前回も説明の通り128bit AES-XEXまたは256bit AES-XTSでメモリの暗号化を行う

- SEV-SNPはAMD SMEのメモリ暗号化エンジンでメモリが暗号化されるが、この暗号化もTDX同様、物理アドレスと平文に基づく**決定論的暗号化であると実験的に確認**できた
  - SME : Secure Memory Encryption

- SGX/TDXと異なり、SEV-SNPでは全MCチャネルを埋める必要が無いため、メインメモリを1枚差せば良く、SGXドライバに行ったようなページ**割り当て制御は不要**である

SEV-SNP CVMへの攻撃

- SGXにおけるEnclave制御チャネルと同様のページ粒度の制御チャネルプリミティブを導入するため、前述の通りこのマシンのホストカーネルに**SEV-Stepパッチ**を適用している

- また、SGXやTDXと異なり、SEV-SNPではMTRR（メモリタイプレンジレジスタ）を介して特定メモリページのキャッシュを無効化できるため、**キャッシュスラッシング手法も不要**である

OpenSSL ECDSA鍵復元攻撃

- SEV-SNP CVM内で動作する、OpenSSL暗号ライブラリにおける単一のECDSA署名操作からECDSA秘密鍵を復元する
  - OpenSSLのバージョンは3.0.13
  - これも定数時間実装であるため、決定論的暗号化を介したメモリ内の平文内容に関する情報漏洩によってのみ秘密鍵を復元できる

- 曲線secp256k1用のランダムなECDSA鍵を生成し、OpenSSLのEVP_DigestSign APIでメッセージに署名するシンプルなテストアプリケーションを作成し実行させる

OpenSSL ECDSA鍵復元攻撃

- ECDSA署名の一環として、OpenSSLもSECRET Network同様、点のスカラー倍算をモンゴメリラダー法で以下のように実施する[1]：

```
1  for (i = scalar_bits - 1; i >= 0; i--) {
2      current_bit = bit(scalar, i) ^ previous;
3      conditional_swap(p0, p1, bit);
4      differential_add_and_double(p0, p1);
5      previous ^= choice;
6  }
```

OpenSSL ECDSA鍵復元攻撃

- SECRET同様、OpenSSLも2bitのウィンドウをスライドさせ、条件付きスワップと加算・倍化を実施する

- 各ウィンドウについて、まずコードは前回の選択値と現在のスカラービットのXORを計算し、新たな選択値を算出する

- 次に、選択値に基づいて2つの点p_0, p_1に対して定数時間の条件付きスワップを実施し、その後両点に対しラダーステップ（SECRETでも実施した加算または倍算）を実施する

OpenSSL ECDSA鍵復元攻撃

- ここで、p_0とp_1がスワップされたかを判定する事で、選択値ビットのシーケンスを復元し、ECDSA nonceであるkを復元できる
  - これはPCK抽出攻撃で抽出したkと同じ立ち位置のもの。ここからECDSA署名鍵を復元できるのは前述の通り

- このため、インターポーザを用いてメモリ内の点p_0の内容を観測する
  - p_0が行3実行前後で変われば選択値ビットは1、そうでなければ0であるという事になる

OpenSSL ECDSA鍵復元攻撃

- SEV-Stepによるページ粒度制御チャネルにより、以下の2つのタイミングで実行を中断する：
  - 条件付きスワップが行3でp_0を読み取る際
  - 行4のadd_and_doubleラダーステップ実行中

- 各中断時にCVM再開前にロジアナを起動し、最初の読み取り時のメモリ内容（x_0の暗号文）を記録する

- 結果、1時間かけて514回メモリ読み取りと記録を行い、選択値ビットを100%の精度で復元してnonceを入手、そしてPCE攻撃の時と同じようにして数秒で**ECDSA鍵を復元**した

軽減策

軽減策

- Battering RAMの軽減策の議論とも重複する部分もあるが、改めてTEE.failの軽減策についても議論する

- まず最も大きな**原因**は、やはりScalable TEEがLegacy-SGXが有していたような、書き込みの度に更新される**HWベースのカウンタ値を失った事**である

- また、TEE.failでは（Battering RAMでも）あまり関係ないが、CPUパッケージ内で管理するマークルツリーによりHWレベルでメモリ完全性を保護する機能を失ってもいる
  - ちなみに、Legacy-SGXのMEEではマークルツリーでカウンタ値の完全性を保護している

軽減策

- しかし、マークルツリーの使用は大きなオーバーヘッドを伴ったため、EPCサイズがどんなに大きくても512**MB**、一般的には128MB（の内ユーザが使えるのは**96MB**）に制限されていた
- よって、保証を犠牲にせずにスケーラブルなメモリ暗号化を実現するにはまだまだ**研究が必要**である
- ちなみに**MK-TMEエンジンの内部動作を変えるマイクロコードアップデート**を**発行する事は不可能**とIntelに明言されたらしい
- よって、このインターポーザ攻撃は近い将来においても有効な攻撃ベクトルであり続けると予想できる

軽減策

- 論文の攻撃構成ではメモリ周波数を4800MT/sから3200MT/sに低下させたが、これは逆に**メモリ周波数を向上**させれば**攻撃のコスト・難易度を上げられる**事を意味する

- しかし近い将来により高速なインターポーザ構成を入手可能になるだろうと考えると、**いたちごっこでしかなく本質的ではない**

軽減策

- 決定論的暗号化が問題であるため、例えばメモリの128bitブロックごとに十分なエントロピーを持たせる事も考えられる

- つまり、例えば64bitのデータ→ランダムな初期値を持つ64bitのカウンタ→64bitのデータ→…を繰り返す**カスタムメモリレイアウト**を導入するイメージである
  - これを導入する事で、TEE.failやBattering RAMのようにメモリ暗号化のTweak値が決定論的である事が原因である攻撃はかなり対策できる

軽減策

- しかし、このレイアウトを**CVMのOSカーネル**や**Intel署名のPCE**等**全てに適用**するのは**明らかに現実的ではない**

- また、物理的なメモリロールバック攻撃のような**物理的完全性攻撃**は**これでは対策できない**

軽減策

- あるいは、クラウドプロバイダのような**強力な警備下でマシンを管理する組織**にのみ**TEEのデプロイを許可**する仕組みが考えられる
  - 元々**クラウドを仮想敵としている**TEEとしては**皮肉な事実**である

- このアプローチとしては、2026年7月現在、以下のような技術が提案されている：
  - Intel POE（Platform Ownership Endorsements）
  - DCEA（Data Center Execution Assurance）
  - Proof of Cloud Alliance

- それが不可能な場合は**秘密分散**の併用も考えられるが、オーバーヘッドが極めて重いため、TEEのメリットを台無しにしてしまう
  - ~~というかそれなら最初から秘密分散を使えば良い~~

軽減策

- 最後に考えられる軽減策として、**Scalable TEEのRAメカニズム**に、位置情報や所属クラウドを検証できる**仕組みを取り入れる**という方法も考えられる

- さらに厳密にやるのであれば、加えて物理的に安全な場所に存在する事が確認されている特定のCPUインスタンスをユーザがホワイトリスト化してそれのみを許可する方法も考えられる

軽減策

- ただし、いずれもTEEベンダ管理のRAプロトコルの変更が必要なため、**ベンダへの依存性が強い**
  - その意味では前述の3つの技術はある程度依存性を避けられている

- また、**クラウド的にはインフラ構成を知られたくない**ため、このような追加の対策には**消極的**になる可能性もある

ちなみに

- TEE.failの作った偽造Attestationボット（抽出したPCKを使用）を受けて、Intelが珍しく明示的にPCK証明書の失効についてアナウンスを出している
  - ~~ボットが使用したPCKを徹底的に潰す意気込みが文面から滲み出ており、流石のIntelも大分怒っているように見える~~

Recently Intel issued a security announcement in response to academic research, known as TEE.fail, that attacked Intel SGX-based attestation. As part of their work on TEE.fail, the authors implemented an automated bot to create fake attestations with a genuine Intel SGX attestation key. Intel has been working to limit any potential impact of the quotes generated from the TEE.fail bot by updating the Intel SGX/TDX Certificate Revocation Lists (CRLs) materials. Note that the zip file with sample verification tool provided with the bot uses a cached CRL that will continue to pass if used (the CRL in the zip file has not been refreshed).

A verifier using the updated CRL available from Intel's services should now cause an Intel SGX or Intel TDX quote generated by the TEE.fail bot to fail attestation verification.

Action Requested for Intel SGX / TDX Verifiers:

Ensure your PCK CRL has been refreshed / updated after November 9, 2 pm Pacific Standard Time from the applicable Intel Service endpoint version your solution currently leverages:

- https://api.trustedservices.intel.com/sgx/certification/v4/pckcrl?ca=platform
- https://api.trustedservices.intel.com/sgx/certification/v3/pckcrl?ca=platform
- https://api.trustedservices.intel.com/sgx/certification/v2/pckcrl?ca=platform

Should you suspect a genuine attestation key is being used maliciously, you can always report this to Intel through Intel PSIRT.

Thank You,

Intel SGX & TDX Services Team

ちなみに

- 実はDDR4ベースのBattering RAMはともかく、DDR5対応のTEE.failですら**TDへの直接攻撃はできていない**
  - TDへの攻撃は全てPCEから抽出したPCKを悪用しているだけである
  - これはTDが何らかの特別な耐性を有しているのか、たまたま著者らが~~サボった~~スキップしただけなのかは不明

- Legacy-SGXのMEEのマークルツリーが対策できるような、**物理的なメモリロールバック攻撃**は2026/7現在**その実際の実行は報告されていない**
  - つまり、TEE.failやBattering RAMとも異なるまた毛色の違う物理攻撃が近い内に出てきてもおかしくはない

まとめ

- Scalable TEEに対する強力な物理攻撃の1つであるTEE.failについて詳細に解説した

謝辞

- 本攻撃の調査は、株式会社Datachainによる金銭的協力の下実施したものです。この場を借りて感謝を申し上げます。

参考文献

[1]”Battering RAM: Low-Cost Interposer Attacks on Confidential Computing via Dynamic Memory Aliasing”, Jesse De Meulemeester et al., https://batteringram.eu/batteringram.pdf

[2]”TEE.fail: Breaking Trusted Execution Environments via DDR5 Memory Bus Interposition”, Jalen Chuang et al., https://tee.fail/files/paper.pdf

[3]”楕円曲線暗号のための数学4（ウィンドウ法）”, 光成滋生, https://zenn.dev/herumi/articles/ecc-mul-window

[4]”符号なし乗算器の作り方の勉強と11ビット乗算器の設計”, 2026/6/27閲覧, https://lpha-z.hatenablog.com/entry/2023/07/16/231500

[5]”Confidential VM Guest Attestation Design Detail”, 2026/6/29閲覧, https://learn.microsoft.com/en-us/azure/confidential-computing/guest-attestation-confidential-virtual-machines-design