Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution

引言

信任的困难：现代操作系统和应用程序的复杂性使得信任变得困难。这些系统的代码量往往达到数百万行，任何单一的漏洞都可能导致整个安全保障的崩溃。这表明，随着软件复杂性的增加，确保其安全性变得愈加困难。

受信执行环境（TEE）的发展：为了应对上述挑战，近期的研究和行业努力开发了受信执行环境（TEE）。TEE提供了一种替代的、非层次化的保护模型，用于隔离应用程序的“enclaves”（安全区）。这种模型旨在增强安全性，确保不同的enclaves之间可以相互不信任。

SGX的发布：2013年，英特尔宣布了其软件保护扩展（SGX），这是一种硬件强制的受信执行环境（TEE）技术。SGX允许在标准的x86处理器上实现隔离和证明机制，从而增强了安全性。

安全保障的承诺：SGX承诺提供强大的安全保障，这使得越来越多的行业参与者开始采用这一技术。SGX的设计旨在确保在不可信的环境中安全执行敏感计算。

Meltdown攻击：Meltdown是一种针对传统层次保护域的攻击。它的目标是从用户空间读取内核内存。传统的计算机系统通常采用层次化的安全模型，其中用户空间和内核空间是分开的，内核空间具有更高的权限。Meltdown利用了这一模型中的漏洞，使攻击者能够绕过这些保护，访问内核内存。

Foreshadow攻击：与Meltdown不同，Foreshadow考虑了一种非常不同的攻击模型。Foreshadow的攻击目标不是读取内核内存，而是破坏现代Intel SGX（Software Guard Extensions）技术所提供的内存保护机制，特别是针对"内地址空间"中的保护域。SGX允许在不可信的环境中安全地执行代码，并提供了对敏感数据的保护。

内地址空间保护域：Foreshadow攻击的重点在于它能够绕过SGX的保护机制，这些机制并不受最近部署的内核页表隔离（KPTI）防御措施的保护。KPTI是一种旨在防止Meltdown攻击的防御策略，但它并不适用于SGX的内存保护。

Foreshadow能够利用现代处理器中的微架构漏洞，影响到更复杂的安全模型，显示出其潜在的危害性。

本文解释了Foreshadow如何需要一种新的利用方法，并表明我们的基本攻击可以完全由一个没有特权的对手发起，而无需对受害者机器进行根访问。然而，鉴于SGX独特的特权攻击者模型，我们还提供了一组可选的内核级优化技术，以进一步降低根对手的噪声。

所有先前已知的针对英特尔SGX的攻击都依赖于来自任何侧通道的特定于应用程序的信息泄漏[30,39,45,51,57,58,60]或软件漏洞[38,59]。英特尔自己也认为自己很安全，然而，Foreshadow驳斥了这一论点，因为它仅依赖于基本的英特尔x86 CPU行为，不利用任何软件漏洞，甚至不需要了解受害者enclaves的源代码。我们率先从英特尔经过严格审查的关键架构发布中提取长期平台发布和认证密钥，并引用飞地，果断地废除了新加坡交易所的安全目标，从而证明了这一点。

本文贡献：

1. 我们通过展示瞬态执行CPU漏洞也适用于SGX的非终止中断页面语义，加深了对这些漏洞的理解。
2. 我们提出了新的利用方法，允许无特权的纯软件攻击者可靠地提取驻留在受保护内存位置或CPU寄存器中的enclave秘密。
3. 我们通过对照实验评估了Foreshadow攻击的有效性和带宽。
4. 我们从英特尔的架构enclave中提取完整的加密密钥，并演示如何（i）绕过enclave启动控制；以及（ii）伪造本地和远程证明，完全破坏远程计算的机密性和完整性保证。

背景

SGX介绍

存储隔离：

SGX确保**只有受信的代码（即在enclave内运行的代码）可以访问enclave私有内存**，而不受信的系统软件（如操作系统）则负责管理enclave的内存。这种设计意味着，尽管操作系统可以分配、驱逐和映射enclave的内存页面，但它无法直接访问这些私有内存。

SGX启用的**CPU会验证不受信的地址翻译过程。**这意味着，当CPU需要访问enclave的内存时，它会检查操作系统提供的地址映射是否有效。如果在访问过程中遇到不合法的内存映射（例如，恶意或错误的映射），CPU会发出页面错误信号，阻止访问。

在处理地址翻译时，CPU会将后续的地址翻译结果缓存到翻译后备缓冲区（TLB）中，以提高效率。当enclave被进入或退出时，TLB会被刷新。这意味着，**任何与enclave相关的地址映射都会被清除**，以防止不必要的泄露或错误访问。

如果有尝试**从enclave外部直接访问私有页面的行为，SGX会采取措施来阻止这种访问。**这种情况下，读取操作会返回-1，而写入操作则会被忽略。这种机制确保了enclave的私有数据不会被外部代码访问，从而保护了数据的机密性和完整性。

enclaves只能通过几个预定义的入口点进入。eenter和eexit指令在不受信任的主机应用程序和飞地之间传递控制权。

在enclave初始化时，SGX处理器会对enclave的初始代码和数据进行哈希计算，生成一个称为MRENCLAVE的安全哈希值。这个哈希值是enclave内容的唯一表示，任何对enclave内容的修改都会导致哈希值的变化。除了MRENCLAVE，enclave还具有一个基于开发者身份的标识，称为MRSIGNER。MRSIGNER包含了enclave开发者的公钥哈希和版本信息。这一标识用于验证enclave的来源和版本，确保其来自可信的开发者。在enclave被创建后，处理器会将MRENCLAVE和MRSIGNER的测量结果存储在一个安全的位置，这个位置对软件（包括enclave内部的代码）是不可访问的。这种设计确保了enclave的初始测量结果是不可伪造的，任何试图篡改enclave内容的行为都会被检测到。

具体而言，Intel 提供 （i） 一个启动 Enclave，用于决定哪些其他 Enclave 可以在平台上运行， （ii） 一个预置 Enclave，用于最初提供长期平台认证密钥，以及 （iii） 一个引用 Enclave，它使用非对称平台认证密钥为远程利益相关者签署本地认证报告

攻击模型和目标

我们的基本攻击要求 enclave 密钥驻留在 L1 数据缓存中。我们展示了在执行受害者安全区时，当机密被引入 L1 数据缓存时，非特权攻击者如何抢先或同时提取机密。对于根对手，我们还提供了一项创新技术，该技术利用 SGX 的分页指令将任意安全区内存预取到 L1 数据缓存中，甚至不需要受害者安全区的配合。

微架构组织

“Instruction Pipeline”是现代处理器设计中的一个关键概念，它指的是将指令执行过程分解为多个阶段，以提高处理器的吞吐量和效率。通过将指令的执行过程分为多个阶段，处理器可以在同一时间处理多条指令，从而实现并行处理。

基本概念：在指令流水线中，指令的执行被分为多个阶段，每个阶段负责指令执行过程中的一个特定任务。常见的阶段包括：

- **取指阶段（Fetch）**：从内存中获取指令。
- **译码阶段（Decode）**：将获取的指令解码为微操作（μops）。
- **执行阶段（Execute）**：执行指令所需的运算。
- **访存阶段（Memory Access）**：访问内存以读取或写入数据。
- **写回阶段（Write Back）**：将结果写回寄存器或内存。

并行处理：通过将指令执行分为多个阶段，处理器可以在每个时钟周期内同时处理多条指令。例如，当第一条指令在执行阶段时，第二条指令可以在译码阶段，第三条指令可以在取指阶段。这种并行处理显著提高了处理器的效率和吞吐量。

流水线深度：流水线的深度指的是指令在流水线中经过的阶段数量。更深的流水线可以在每个时钟周期内处理更多的指令，但也可能导致更高的延迟和复杂性。

1. Out-of-Order Execution

基本概念

乱序执行是一种允许处理器在不遵循程序原始指令顺序的情况下执行指令的技术。处理器会根据指令的可用性和资源的可用性动态调度指令，以提高执行效率。

工作原理

- **指令解码**：处理器首先将指令解码为微操作（μops）。
- **调度**：解码后的μops被放入一个调度队列，处理器根据可用的执行单元和操作数的可用性来安排执行顺序。
- **执行**：一旦所需的操作数可用，μops就会被发送到执行单元进行处理，而不必等待前面的指令完成。
- **结果写回**：执行结果会在适当的时候写回寄存器或内存，确保最终的程序状态与原始指令顺序一致。

优势

- **提高资源利用率**：通过并行执行多个指令，处理器可以更有效地利用其执行单元。
- **减少等待时间**：指令不必按顺序执行，从而减少因数据依赖而导致的等待时间。

2.Speculative Execution

基本概念

推测执行是一种技术，处理器在确定某些条件（如分支指令的结果）之前，提前执行可能的指令。这种技术旨在减少因控制冒险（如分支指令）导致的性能损失。

工作原理

- 分支预测：处理器使用分支预测算法来猜测分支指令的结果（即下一条要执行的指令）。
- 提前执行：基于预测结果，处理器会提前执行预测路径上的指令。
- 结果验证：一旦分支结果确定，如果预测正确，提前执行的结果将被保留；如果预测错误，处理器会丢弃这些结果，并回滚到正确的执行路径。

优势

- 减少延迟：通过提前执行，处理器可以在分支指令的延迟期间继续执行其他指令，从而提高整体性能。
- 提高指令吞吐量：推测执行可以增加有效执行的指令数量，尤其是在存在大量分支的程序中。

瞬态执行攻击

1. Spectre攻击

- **分支预测机制**：现代CPU使用分支预测来提高执行效率。分支预测器根据历史执行路径预测程序的控制流，以提前加载可能需要执行的指令。
- **欺骗受害者保护域**：Spectre攻击通过“毒化”共享的分支预测资源，诱使受害程序（即受害者）执行不在其预期执行路径中的指令。这意味着攻击者可以操控程序的执行，使其执行一些本不应执行的指令。

- **瞬态执行**：在这种情况下，受害者程序可能会访问一些内存位置，这些位置是受害者有权限访问的，但攻击者没有权限。攻击者利用这种瞬态执行的特性，能够在不直接访问受害者内存的情况下，获取敏感信息。

2. Meltdown攻击

- **更严重的缺陷**：Meltdown攻击利用了现代Intel处理器中的一个更根本的缺陷，即在处理器的乱序执行过程中，存在一个小的时间窗口，在这个窗口内，未经授权的内存访问结果仍然可用。
- **乱序执行**：现代处理器通常会乱序执行指令，以提高性能。这意味着处理器可以在等待某些操作完成时，先执行其他指令。
- **竞争条件**：Meltdown攻击利用了这种乱序执行的竞争条件，允许攻击者暂时执行访问未经授权内存位置的指令。虽然处理器最终会检测到这些未经授权的访问并回滚（撤销）这些指令的效果，但在回滚之前，攻击者可以利用微架构状态（如CPU缓存）来提取敏感信息。

Foreshadow攻击

与前者相比，Foreshadow 针对在不受信任的上下文中运行的enclaves。

在攻击开始之前，攻击者的非信任enclave主机应用程序需要分配一个“oracle buffer”（oracle缓冲区），该缓冲区包含256个槽，每个槽的大小为4 KiB。这一设计是为了避免由于意外激活处理器的缓存行预取器而导致的误报（false positives）

攻击第一阶段

enclave的秘密数据不在L1缓存中，攻击者无法通过瞬态执行来提取这些数据。为了成功提取enclave中的秘密数据，必须首先将这些秘密数据加载到L1缓存中。这是Foreshadow攻击的一个关键前提。

L2缓存的计算能力：尽管Foreshadow攻击可以在L2缓存中暂时计算内核数据，但对于驻留在L2缓存中的enclave秘密数据，攻击者无法成功读取。这进一步强调了L1缓存的重要性。

攻击第二阶段

我们通过撤销对希望读取的 enclave 页的所有访问权限来进行操作。具体来说，使用 `mprotect` 系统调用清除了相应页面表项中的“存在”位，这样对该页面的任何访问最终都会导致错误。这意味着对该页面的任何访问最终都会导致一个错误（fault），因为操作系统会检测到该页面不再存在。

之前的 Meltdown 攻击主要集中在读取内核内存页面，而 Intel 对投机执行漏洞的分析明确指出，恶意数据缓存加载仅适用于页面表中标记为仅限监督的内存区域，而不适用于标记为不存在的内存。这表明，Intel 的分析认为，只有在特定的内存区域（即内核空间）中，攻击者才能利用投机执行来读取数据。然而，研究人员的发现与 Intel 的分析不一致。他们的研究表明，即使是标记为不存在的内存区域（即 enclave 内存），也可以通过特定的攻击手段（如使用 `mprotect` 清除“存在”位）来进行数据读取。这意味着攻击者可以绕过 Intel 的安全假设，从而在 enclave 内存中提取秘密数据，而不仅仅是内核内存。

Foreshadow 显式重新建立每个 oracle 槽的 TLB 条目。此外，我们需要确保 oracle 槽条目不在处理器的缓存中。我们通过对所有 256 个 oracle 槽发出 `clflush` 指令来同时实现这两个要求。

最后，我们执行了在 Listing 1 中显示的瞬态指令序列。我们提供了与 Listing 2 中的等效 C 代码逐行翻译。当使用指向 oracle 缓冲区和 `secret_ptr` 的指针调用时，秘密值在第 5 行被读取。由于我们确保将 enclave 页标记为不存在，因此 SGX 的中止页面语义不再适用，最终会发出错误。然而，第 6-7 行的瞬态指令仍将被执行，并计算出 oracle 缓冲区中槽 v 的密钥相关位置，然后从内存中获取它。

```
1 foreshadow:
2   # %rdi: oracle
3   # %rsi: secret_ptr
4
5   movb (%rsi), %al
6   shl $12, %rax
7   movq (%rdi, %rax), %rdi
8   retq
```

Listing 1: x86 assembly.

```
1 void foreshadow(
2     uint8_t *oracle,
3     uint8_t *secret_ptr)
4 {
5   uint8_t v = *secret_ptr;
6   v = v * 0x1000;
7   uint64_t o = oracle[v];
8 }
```

Listing 2: C code.

攻击第三阶段

1.在这一阶段，处理器意识到它不应该已经投机性地执行了某些指令。这意味着在执行过程中，处理器会丢弃未提交的寄存器更改，并发出一个页面错误（page fault）。这个页面错误是因为之前的操作试图访问一个被标记为不存在的页面。

2.当页面错误发生时，操作系统会捕获这个错误，并将控制权转移到攻击者的用户级异常处理程序。这个处理程序是攻击者预先设置的，用于处理错误并继续执行攻击。

3.在用户级异常处理程序中，攻击者会仔细测量重新加载每个 oracle 槽的时间。这些 oracle 槽是用来存储与秘密数据相关的信息。通过测量这些时间，攻击者可以推断出 enclave 中的秘密字节。

在这一阶段，攻击者可以通过分析缓存的状态来确定秘密值。这种方法利用了缓存的时间特性，即访问缓存中的数据比访问主内存要快得多。

如何读取完整的缓存行

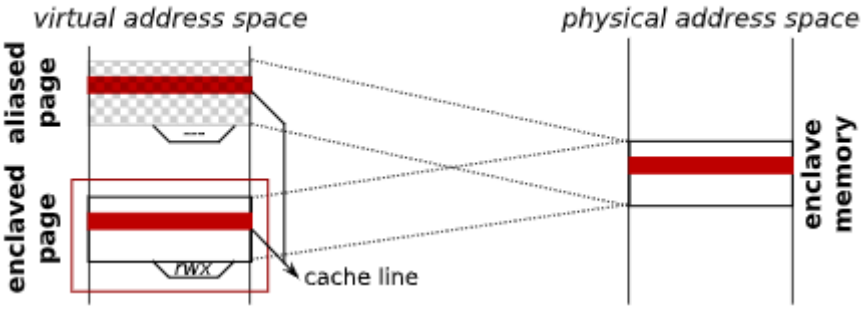
我们特别发现，重复的上下文切换和内核代码执行可能会无意中从 L1 缓存中逐出 enclave 密钥。因此，我们认为，最大限度地减少缓存污染对于从单个 enclaved 执行中成功提取更大的密钥至关重要。

1.建立 aliasing 页面

aliased page 定义：页面别名是指在内存管理中，多个虚拟地址映射到同一物理内存页面的情况。这意味着不同的虚拟地址可以指向相同的物理内存内容。

利用页面别名：攻击者可以通过创建一个别名页面（aliased page）来绕过SGX限制。具体来说，攻击者可以使用 `mprotect` 系统调用来清除 enclave 页面在页面表中的“存在”位，从而使得该页面在逻辑上被标记为不存在。然后，攻击者可以创建一个新的虚拟地址映射到同一物理页面，但这个新的映射不受 enclave 的保护。通常在 SGX 中，当未授权代码访问 enclave 内存时，通常会触发中止页面（abort page）语义，这意味着访问会被拒绝，秘密数据不会被泄露。然而，通过使用页面别名，攻击者可以避免这种中止行为，因为新的别名页面并不被 enclave 自身引用。

通过这种方式，攻击者能够在不触发 SGX 的安全机制的情况下，访问 enclave 内存中的数据。这使得攻击者能够在 Phase I 中设置别名页面，并在后续阶段利用这些别名页面来提取秘密数据。



2.故障抑制

攻击过程中发生页面错误，导致密钥数据从缓存中被驱逐。

攻击者可以利用现代 Intel 处理器中的事务性同步扩展（Transactional Synchronization Extensions, TSX）。TSX 允许程序在事务中执行一系列指令，如果在事务执行期间发生页面错误，处理器会立即调用用户级的事务中止处理程序，而不是将控制权转移到内核。这种机制可以减少上下文切换的次数，从而降低缓存污染的风险。

3.Keeping Secrets Warm (Root)

在攻击过程中，攻击者需要从 L1 缓存中提取 enclave 的秘密数据。然而，缓存是有限的，随着其他数据的访问，秘密数据可能会被驱逐（evicted）出缓存。这种情况会导致攻击者无法成功提取所需的秘密。

文中给了两种方法：

使用特权指令：攻击者可以利用特权指令（如 `wbinvd`）来清空整个 CPU 缓存层次结构。这一操作会在执行 enclave 之前释放缓存空间，从而为 enclave 中的秘密数据提供更多的缓存空间，减少其他非 enclave 访问对缓存的影响。

循环执行攻击：对于未特权的攻击者，可以通过在 Phase III 中将瞬态执行（Phase II）过程放入紧密循环中来增加对缓存的访问频率。这种方法可以提高对 oracle 槽的访问，从而增加成功提取秘密的机会。也就是说，通过在每次重新加载 Oracle 槽之前临时访问 enclave 密钥，我们可以确保保存密钥数据的缓存行保持"热"状态，并且不太可能被 CPU 最近最少使用的缓存替换策略驱逐。

4.隔离内核

逻辑处理器之间共享，因此一个内核上的缓存活动可能会无意中逐出其同级内核上的 enclave 密钥。为了限制此类影响，根对手可以将受害者 enclave 进程固定到特定内核，并将中断尽可能多地卸载到另一个物理内核。

5.处理零偏见

我们发现处理器在遇到异常时会将未经授权的内存读取结果归零。当这种 Null 发生在阶段 II 中的瞬态无序指令可以对真实密钥进行操作之前时，这将显示为在阶段 III 中读取全零值。为了抵消这种零偏差，Foreshadow 在阶段 III 中接收到0x00时多次重试瞬态执行阶段 II，然后果断地得出秘密字节确实为零的结论。

由于 Foreshadow 的瞬态执行阶段严重依赖于 L1 缓存中的 enclave 数据，因此从从 L1 缓存中驱逐秘密缓存行的那一刻起，我们始终会收到 0x00 字节。因此，处理器的清零机制还使我们能够可靠地检测目标安全区数据是否仍存在于 L1 缓存中。也就是说，继续进行 Foreshadow 缓存行提取是否仍然有意义。

加快提取秘密

为了提高 Foreshadow 攻击的时间分辨率，攻击者可以在秘密数据被加载到 L1 缓存后，异步退出 enclave。这意味着攻击者可以在秘密数据被覆盖或驱逐之前，及时提取这些数据。

攻击者可以利用 SGX-Step 这一先进的 enclave 执行控制框架，最大化时间分辨率。这种结合使得攻击者能够在每条指令执行后泄露内存操作数，从而更精确地捕获秘密数据。

即使是未特权的攻击者，也可以通过 `mprotect` 系统调用接口，以较粗粒度的 4 KiB 页面故障粒度暂停 enclave。这种能力使得未特权攻击者能够在 enclave 执行过程中，可靠地检查被抢占的受害者 enclave 的私有 CPU 寄存器内容。通过上述方法，攻击者能够开发出一种新技术，使得未特权的 Foreshadow 攻击者能够可靠地检查 enclave 中的私有数据。

单步执行中断

攻击者依赖于最近发布的开源框架 SGX-Step。这个框架允许攻击者逐条指令地中断受害者的 enclave 执行，从而实现更精确的控制。SGX-Step 附带一个 Linux 内核驱动，能够为本地的高级可编程中断控制器（APIC）设备建立方便的用户空间虚拟内存映射。这种映射使得用户空间的程序能够直接与硬件中断控制器进行交互。通过直接从用户空间配置 APIC 定时器，SGX-Step 实现了非常精确的单步执行技术。这种方法消除了内核上下文切换带来的噪声，确保了中断能够在 enclave 执行的第一条指令后可靠到达。它允许攻击者在 enclave 执行的关键时刻捕获和分析数据。

转储CPU寄存器内容，页面错误驱动

在 SGX 中，当 enclave 被中断时，处理器会将其寄存器内容安全地存储在预分配的 SSA（State Save Area）框架中。

Foreshadow 攻击者可以通过针对 SSA enclave 内存来提取私有的 CPU 寄存器内容。然而，为了成功提取，这些 SSA 框架的数据必须驻留在处理器的 L1 缓存中。整个 SSA 框架占用多个缓存行，其中仅通用寄存器区域就占据了 144 字节（约 2.25 个缓存行）。如果这些 SSA 缓存行在内核上下文切换或 Foreshadow 的提取过程中被意外驱逐，攻击者将无法成功提取所需的数据。

攻击者提出了一种创新的方法：通过撤销受害者 enclave 代码页面的执行权限，未特权的应用程序上下文可以在完成 `eresume` 后的第一条指令上引发页面错误（page fault）。在这个过程中，没有实际执行任何 enclave 指令，因此寄存器内容保持不变。

在恢复执行权限之前，Foreshadow 攻击者逐字节读取完整的 SSA 框架，迫使 enclave 在每次 SSA 缓存行被驱逐时进行零步操作（即读取所有零）。这种方法确保了攻击者能够准确地捕获寄存器的内容。

并发提取秘密

我们通过在进入受害者enclave之前将专用间谍线程固定在兄弟逻辑核心上来探索这种隐蔽的 Foreshadow 攻击模式。间谍线程在紧密循环中重复执行 Foreshadow，以尝试读取感兴趣的秘密。只要并发运行的 enclave 未将密钥引入 L1 缓存，间谍就会丢失 CPU 内部争用条件。这显示为始终读取零值。我们使用此观察来同步 spy 线程。只要读取零值，间谍就会继续暂时访问秘密的第一个字节。当 enclave 最终触及秘密时，并发间谍线程会立即提取该秘密。

这种方法不如前面的中断方法，受到带宽限制，且enclave持续执行可能导致秘密内容从缓存中被驱逐，但是我们仍需要考虑这一方法的防御对策。

读取未缓存的秘密

管理 Enclave 页面缓存 (EPC)：

- SGX 设计依赖于不受信任的系统软件来超额订阅有限的受保护物理内存 EPC 资源。为了实现这一点，不受信任的操作系统可以使用特权的 `ewb` 和 `eldu` SGX 指令，分别将加密和完整性保护的 4 KiB enclave 页面从 EPC 中复制出去，再加载回去。
- 当解密和验证一个加密的 enclave 页面时，`eldu` 指令会将整个页面以明文形式加载到 CPU 的 L1 缓存中。重要的是，实验验证表明，`eldu` 微代码实现不会在指令终止后驱逐该页面，确保页面的内容在缓存中保持不变。

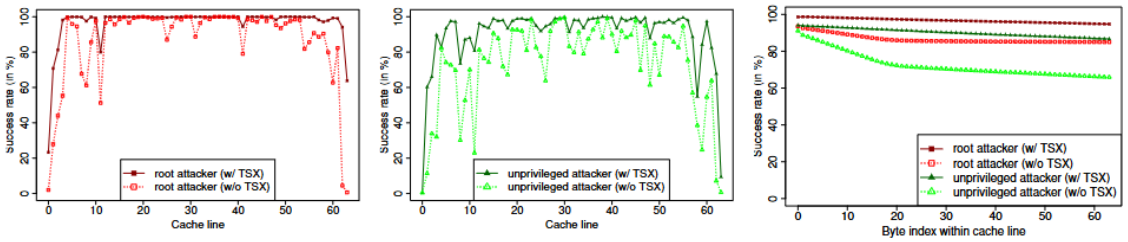
提取整个 enclave 内容：

- 攻击者的恶意内核驱动程序会遍历所有 enclave 页面（例如，通过检查 `/proc/pid/maps`），首先使用 `ewb` 将页面从 EPC 中驱逐，然后立即使用 `eldu` 将其加载回 L1 缓存。由于 `eldu` 加载页面后不会驱逐它，攻击者可以可靠地提取其内容。这个过程会对受害者 enclave 的每个页面重复进行。
- 通过上述 `eldu` 技术，攻击者可以在不需要受害者合作的情况下提取整个 enclave 的地址空间。尽管在 enclave 创建时，初始内存内容对攻击者是已知的，但秘密通常是在运行时生成或引入的（例如，通过密封或远程秘密提供）。因此，实际上，受害者 enclave 至少应该执行一次，攻击者可以依赖单步执行原语（如 SGX-Step）在 enclave 包含秘密且在被擦除之前精确暂停其执行。

这种操作是为了排除了那些强制数据直接存储在内存中而故意避开 CPU 缓存层次结构的软件缓解策略。

Microbenchmark 评估

所有实验均在公开可用的现成 Intel x86 硬件上进行。我们使用了商用 Dell Optiplex 7040 台式机，配备 Skylake 四核 Intel i7-6700 CPU 和 32 KiB、8 路 L1 数据缓存。通过攻击一个特制的基准测试安全区来评估 Foreshadow 的有效性。



(a) Root attacker cache line extraction. (b) Unprivileged cache line extraction. (c) Intra-cache line degradation.

图(a) 显示了根攻击者模型中每个缓存行的成功率。总体而言，我们达到了 99.92% 的出色中位成功率（使用 TSX）。由于并非每台支持 SGX 的机器都支持 TSX，因此我们在不依赖 TSX 功能的情况下执行了相同的基准测试。这导致中位成功率下降了 2.59 个百分点 （97.32%）。

在目标页面的开头/结尾存储数据的缓存行（即缓存行 #0 和 #63）的平均成功率明显较低：有和没有 TSX 分别为 23.25/2.03% 和 63.78/0.63%。我们将这种影响归因于意外的 L1 缓存行驱逐

图(b) 显示了针对非特权攻击者的相同基准测试的结果。正如预期的那样，有和没有 TSX 的中位成功率分别合理下降到 96.82% 和 81.14%。虽然这些成功率略低，但它们清楚地表明，即使是更克制的用户级对手也可以以令人印象深刻的成功率成功攻击 SGX enclave。

图(c) 在缓存行中提取的每个字节，无意中从 L1 缓存中逐出密钥的可能性都会增加。图 4c 量化了这种缓存内行降级行为。对于根对手，成功提取缓存行中第一个字节的概率为 98.61%。但是，当提取缓存行的最后一个字节时，成功率已下降到 94.75%。尤其是 TSX 展会在这里发挥了重要作用。

攻击Intel架构的安全区

为了获得最大的可靠性，我们对 Intel 架构启动和引用 enclaves 的攻击都采用根对手模型，并应用所有优化技术。不过不需要单步中断和edlu预提取技术。

攻击Intel Launch Enclave

攻击背景

1. **SGX Enclave的创建**：
   - SGX enclaves是通过不可信的系统软件以多阶段的方式创建的。在enclave初始化之前，必须从Intel的Launch Enclave获取一个有效的EINITTOKEN。
2. **EINITTOKEN的内容**：
   - EINITTOKEN包含了目标enclave的身份信息（MRENCLAVE和MRSIGNER）、请求的特性和属性，以及一个随机的KEYID。
   - 这个token的完整性由消息认证码（MAC）保护。
3. **平台启动密钥**：
   - 安全性依赖于一个处理器级别的秘密，称为平台启动密钥。`einit`指令使用这个密钥来验证EINITTOKEN的正确性，并确保只初始化与token中身份和属性匹配的enclave

攻击方法

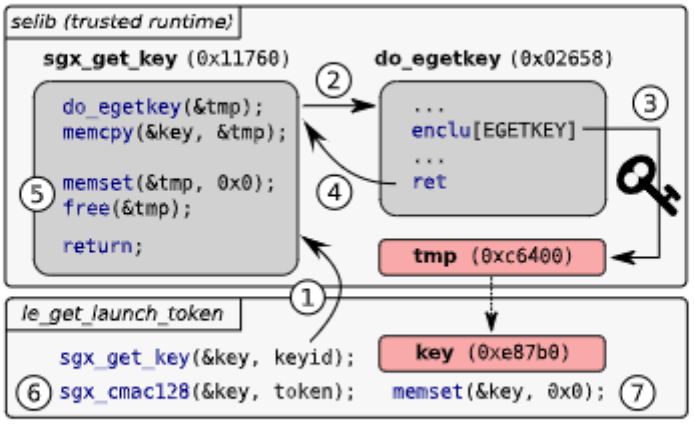


Figure 5: Key derivation in the SGX Launch Enclave.

1.LE 首先生成一个 random，并调用 sgx_get_key 函数来获取 launch key。为此，受信任的 enclave 运行时会分配一个临时缓冲区，

2.然后调用一个小的 do_egetkey 程序集存根，

3.该存根执行 egetkey 指令以派生实际的启动密钥，

4.接下来，将临时缓冲区复制到调用方提供的缓冲区中，

5.被覆盖的 plus 在返回之前被解除分配。

6.LE 现在使用启动密钥计算所需 MAC ，

7.然后立即将密钥缓冲区归零。

在攻击的在线阶段，我们**中断上述步骤 3 和 4 之间**的受害者 enclave，并指示 Foreshadow 提取包含 128 位密钥的缓存行。

攻击结果：我们在实践中观察到 100% 的成功率;也就是说，我们的最终（在线）漏洞利用程序从单个 LE 运行中无噪音地提取完整的 128 位密钥.

攻击Intel Quoting Enclave

攻击背景

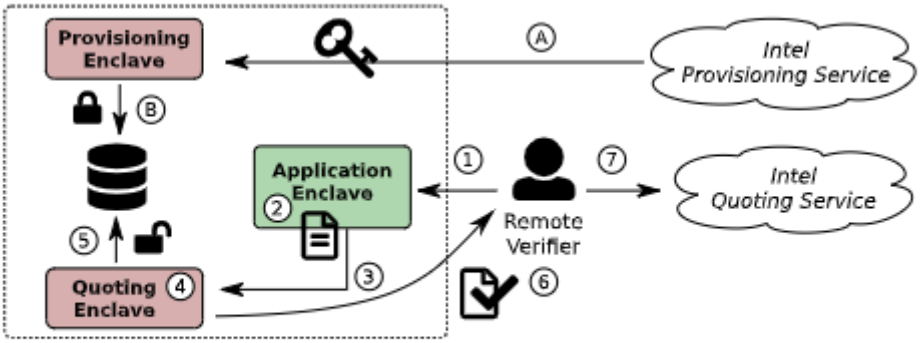


Figure 6: SGX Quoting Enclave for remote attestation.

A：在初始平台配置阶段 A 中，Intel 部署了一个专用的 Provisioning Enclave （ PE ），以从远程 Intel Provisioning Service 请求 EPID 私钥，从这里称为平台认证密钥。

B：收到认证密钥后，PE 会派生一个基于作者的配置签章密钥，以便将长期认证密钥安全地存储在不受信任的存储中。

1：为了成功进行 enclave 认证，远程验证程序会发出 1 个质询

2：然后 enclave 执行 ereport 指令，将质询绑定到其身份

3：不受信任的应用程序上下文现在将本地认证报告转发 到 QE

4：QE 派生其报告密钥以验证报告的完整性。

5：接下来，QE 派生配置签章密钥以解密从系统软件收到的平台认证密钥

6：QE 对本地认证报告进行签名以将其转换为报价

7：收到认证响应后，远程验证员最终将报价提交给 Intel 的认证服务，以使用 EPID 组公钥进行验证

攻击方法

与图 5 所示的 LE 攻击一样，我们的两个 QE 密钥提取漏洞都以sgx_get_key可信运行时函数为目标。我们再次构建了一个精心设计的页面错误状态机，以确定性地抢占 egetkey 调用和密钥缓冲区被覆盖之间的 QE 执行。同样在实践中实现了 100% 的成功率。

缓解策略

目前SGX的防御手段存在局限性。

我们预计 Foreshadow 将在未来英特尔处理器中通过基于硅的更改直接解决。SGX 设计 包括了 TCB 恢复的概念，在所有测量中都包含 CPU 安全版本号。因此，未来的微码更新原则上可以缓解现有支持 SGX 的处理器上的 Foreshadow。

Foreshadow 要求 Enclave 数据驻留在 L1 缓存中，我们设想了一种硬件-软件协同设计缓解策略。应保证有抵御foreshadow的安全区 （i） 超线程设置中的两个逻辑内核都在同一安全区内执行，并且 （ii） 在每个安全区退出事件时刷新 L1 缓存。