

针对 AES 查表法最后一轮加密的 L3 缓存攻击

陆 垚 陈开颜 王寅龙 尚倩伊

陆军工程大学 石家庄 050000

(294102746@qq.com)

摘 要 文中对 Cache 最新的攻击方法进行了研究,在配置 Intel i5-4590 四核心、3.3GHz CPU 处理器的机器上,对 Linux 系统虚拟环境下 Bouncy Castle JDK1.0 库中的 AES 快速加密法—AESFastEngine.java 进行 flush+flush 计时攻击。在加密进程持续执行时,使用 flush+flush 方法遍历共享主存地址来检测活动地址集(S 盒地址),然后找到 S 盒偏移位,对 S 盒偏移位中的表项进行监控,从密文数据中筛选对应 flush+flush 时间较短的密文值,再利用 S 盒中的表项值恢复最后一轮密钥值,即通过确定监测 S 盒中固定范围的表项的使用情况来恢复最后一轮加密使用的密钥值,这种方法需要大量的已知密文,并且能够精确地计算出 S 盒的偏移和最后一轮的密钥值。

关键词: AES 查表法; Rijndael 算法; flush+flush 攻击; cache 计时攻击; S 盒偏移位

中图法分类号 TP309.7

L3 Cache Attack Against Last Round of Encryption AES Table Lookup Method

LU Yao, CHEN Kai-yan, WANG Yin-long and SHANG Qian-yi

Army Engineering University of PLA, Shijiazhuang 050000, China

Abstract According to the research status of Cache Side-Channel attacks, on machines equipped with Intel i5-4590 four-core, 3.3GHz CPU processor, flush + flush timing attack is carried out on AES fast encryption method (AESFastEngine.java) of Bouncy Castle JDK1.0 library in Linux system virtual environment. When the encryption process continues to execute, flush + flush method is used to traverse the shared main memory address to detect the active address set (s-box address), and then the S-box offsets is found to monitor table entries in the s-box offset. Select ciphertext value corresponding to shorter flush + flush time from all ciphertexts, and restore the last round key value with the table entry value of S box, that is, the key value used in the last round can be restored by determining the usage of entries in S-box. This method needs a large number of known ciphertext, and can accurately calculate the offsets of S-box and the last round key values.

Keywords AES table look-up method, Rijndael algorithm, flush+flush attack, Cache timing attack, S-box offsets

1 AES 快速实现法——查表法

1.1 AES 算法

分组密码研究至今已有近 50 年的历史,高级加密标准 AES 已取代 DES 成为主要的加密算法。在 1997 年 4 月,美国国家标准技术研究所(NIST)成立 AES 工作组,共征集了 15 个 AES 候选算法,最终 Rijndael 算法胜出,并成为美国高级加密标准(AES)主要的优化算法之一。文献[1-2]中,比利时设计者 Joan Daemen 和 Vincent Rijmen 对 Rijndael 算法进行了详细的分析,提出了快速有效的软件实现优化方法,给出了 T 表和逆 T 表、S 盒、逆 S 盒的设计,通过查找预先构造的数据表,可以有效地提高算法的执行速度。该方法是将 Rijndael 算法的查找表 S-box 代入轮操作中进行数学推导,对轮加密中的 4 个变换进行合并,并用 4 个 256 项的 T 查找表实现轮加密,这种优化算法在 OpenSSL, Bouncy Castle 等多种密码学库包中得到了广泛运用。

1.2 AES 查表法的实现原理

查表法是将字节替换层、行移位层和列混淆层合并为查

找表^[3]。以本次实验的攻击目标 Bouncy Castle JDK1.0 库中的 AES 快速加密法^[4]为例, T 表由 256 项 32 bits 字组成(共 8 个 T 表),前 9 轮加密操作需查找 4 个表(T0—T3),轮解密操作查找 4 个表(Tinv0—Tinv3)。对于 S 盒和逆向 S 盒(Si 盒)两个表,在最后一轮加密将用到 S 盒,在最后一轮解密用到 Si 盒,其大小都是 256 字节,由 256 项 8 bits 的字节构成,8 个 T 表可以编码在加密函数中,也可以只保存 S 盒。执行加密时,在内存中进行 T 表扩展,其扩展方法由式(1)给出。

行移位: $A_0 \rightarrow A_0, A_5 \rightarrow A_1, A_{10} \rightarrow A_2, A_{15} \rightarrow A_3, A_4 \rightarrow A_4, A_9 \rightarrow A_5, A_{14} \rightarrow A_6, A_3 \rightarrow A_7, \dots, A_{11} \rightarrow A_{15}$ 。

字节替换层: $B_0 = S(A_0), B_1 = S(A_5), B_2$ 和 B_3 同理。

列混淆层:

$$\begin{pmatrix} C_0 & C_4 & C_8 & C_{12} \\ C_1 & C_5 & C_9 & C_{13} \\ C_2 & C_6 & C_{10} & C_{14} \\ C_3 & C_7 & C_{11} & C_{15} \end{pmatrix} \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} B_0 & B_4 & B_3 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{pmatrix}$$

密钥加层将其按列向量拆开,这里只给出第一列:

$$\begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{pmatrix} = \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} \oplus K_{j_0} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} \oplus K_{j_0}$$
$$= \begin{pmatrix} 02 \\ 01 \\ 01 \\ 03 \end{pmatrix} S(A_0) \oplus \begin{pmatrix} 03 \\ 02 \\ 01 \\ 01 \end{pmatrix} S(A_5) \oplus \begin{pmatrix} 01 \\ 03 \\ 02 \\ 01 \end{pmatrix} S(A_{10}) \oplus \begin{pmatrix} 01 \\ 01 \\ 03 \\ 02 \end{pmatrix} S(A_{15}) \oplus K_{j_0}$$

在合成 T 表后,轮加密简化为:

$$\begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{pmatrix} = (T0(A_0) \oplus T1(A_5) \oplus T2(A_{10}) \oplus T3(A_{15})) \oplus K_{j_0} \tag{1}$$

其中, j 代表加密轮数, $K_{j_0} \cdots K_{j_3}$ 为 128 bits 轮密钥 K_j 按 32bits 字长的划分,同理可得 $D_4 \cdots D_7, D_8 \cdots D_{11}, D_{12} \cdots D_{15}$ 。

然后轮加密就变成了查表操作,轮加密要用到 T0—T3 表,而最后一轮加密要用到 S 盒,所以 S 盒单独保留一个表,前 9 轮查找表操作如图 1 所示,字节替换层和行移位层是可以交换顺序的。

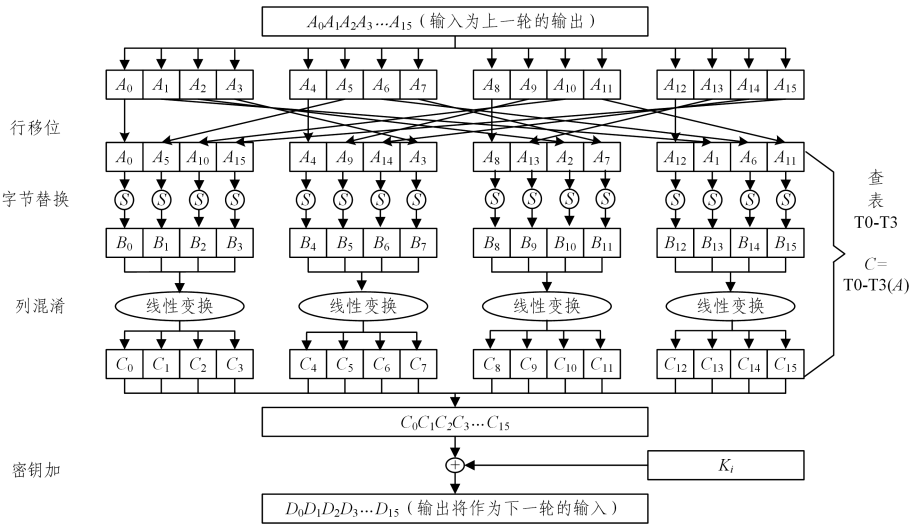


图 1 查表法轮加密操作 E(0—8)
Fig. 1 Lookup table operation for E(0—8)

在最后一轮进行行移位、S 盒字节替换和轮密钥加操作， 如图 2 所示。

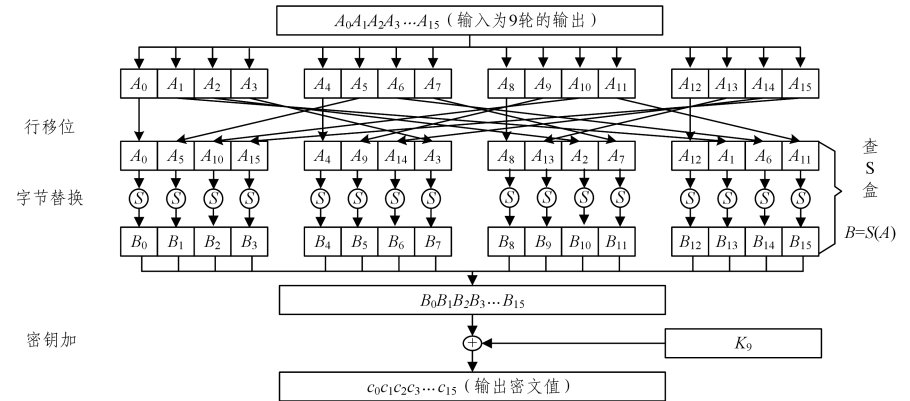


图 2 最后一轮加密操作 E(9)
Fig. 2 Last encryption for E(9)

可以得到:

$$C_0 \cdots C_{15} = (B_0 B_1 \cdots B_{15}) \oplus K_9$$
$$C_0 \cdots C_{15} = S(A_0 A_5 \cdots A_{11}) \oplus K_9 \tag{2}$$

2 flush + flush 攻击方法

2014 年,Yarom 等首次提出了 flush+reload 方法^[5],对比早期的缓存计时攻击,如 evict+time 或 prime+probe 方法^[6],新的攻击方法的明显优势是其目标是最后一个共享缓存,可跨核心完成攻击,能够适用于当代多核心的 CPU 和大

缓存结构。Ristenpart 等^[7]展示了攻击 CPU 和缓存活动图;2014 年 Apecechea 等^[8]在使用 Xen 和 VMware VMMs 跨 vm 设置的虚拟化环境中首次实现了 Bernstein 的攻击;而后 Irazoqui 等^[9]利用 AES 中的缓存冲突作为时间泄漏源,使用 flush+reload 方法恢复 AES 的密钥;2016 年,Gruss 等在 flush+reload 的基础上,提出了 flush+flush 方法,通过实验证明了 FF 拥有更高的灵敏度和更好的隐蔽性^[10]。

flush+flush 实为 flush+reload 方法的变种,flush+flush 的步骤(见图 3)如下:

步骤 1 间谍进程与加密进程共享的数据块被加载到 cache 中。

步骤 2 使用间谍进程用 flush 命令将该数据块从 cache 中驱逐。

步骤 3 执行加密进程。

步骤 4 间谍进程再次驱逐该数据块,如果该数据块被加密进程使用,它就会被加载至 cache 中,则对应较短的 flush 计时值;否则对应较长的计时值。

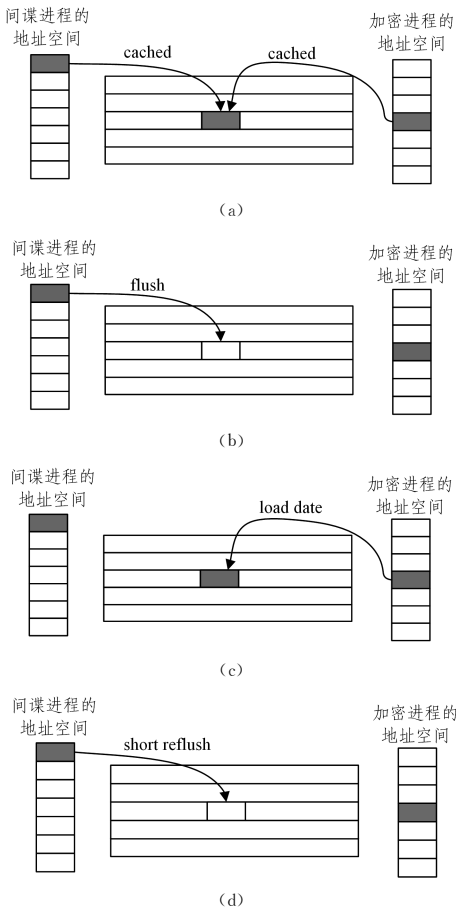


图 3 flush+flush 的步骤

Fig. 3 Process of flush+flush

3 执行攻击

本次攻击是使用间谍进程完成对加密进程的监测及数据采集,也可以将数据分析和逆运算写入间谍进程,一次完整的加密中 S 盒只使用了一次(最后一轮加密),这样使用 flush+flush 攻击方法就能够监测加密时 S 盒内表项的使用情况,此次实验监测的目标是 S 盒偏移位中的表项,攻击步骤(间谍进程的架构)如下:

- (1) 测量缓存驱逐(flush)能力,并找到阈值的最优解;
- (2) 通过 flush+flush 攻击方法获得 T 表、S 盒的主存地址;
- (3) 获得 S 盒偏移地址;
- (4) 对 S 盒偏移地址监控,采集相关密文及对应的监测计时;
- (5) 使用 S 盒偏移地址内的表项值和发生命中的密文恢复最后一轮密钥。

3.1 求解最优阈值

测量缓存的驱逐性能,通过反复载入单个缓存行数据,驱逐 cacheline 数据,测量 Cached 和 Uncached 两种情况的驱逐时间,再根据时间差找到能提高最大分辨率的阈值解。算法

1 展示了求解阈值($F_Threshold$)的步骤,而 $F_Threshold$ 是在对 Cache 性能测试后,根据命中、未命中的数量和时间信息算出。 $F_Threshold$ 选择驱逐缓存的数据和驱逐主存的数据计时时间之和的 $1/2$ 作为阈值,恰好能够以最大分辨率区分命中和未命中。

算法 1 $F_Threshold$ 阈值计算函数

```
input: int Plaintext[16]
output: FF_count[], MF_count[], F_Threshold

1. void detect_F_Threshold()
2. for (i=0; i < repet_Num; i++)
3.     maccess(* Plaintext);
4.     start=rdtsc();
5.     flush(* Plaintext);
6.     MF_time=rdtsc()-start;
7.     MF_count[MF_time]++;
8. end
9. return MF_count[MF_time];
10. flush(* Plaintext);
11. for (j=0; j < repet_Num; j++)
12.     start=rdtsc();
13.     eflush(* Plaintext);
14.     FF_time=rdtsc()-start;
15.     FF_count[FF_time]++;
16. end
17. return FF_count[FF_time];
18. FF_time += MF_time * MF_count[MF_time];
19. MF_time += FF_time * MF_count[MF_time];
20. F_Threshold=(FF_time+MF_time)/(2 * repet_Num)
21. return F_Threshold
```

3.2 反复攻击获得 T 表、S 盒的主存地址

在连续加密执行的同时,使用 flush+flush 方法对共享空间中每隔 cacheline(64bytes) 大小的每个物理地址重复遍历,通过对比阈值判断在加密中被频繁使用的内存地址集(内存块)。使用算法 2 能确定 S 盒的地址集,因为没有互相依赖关系,其中第 1—11 行代码与第 12—15 代码是并列执行的。需要注意的是:密钥生成操作只执行一次后就会存储在缓存或寄存器中直接使用,不需要多次执行密钥生成运算,因此在连续的 Cache 微结构攻击中,不需要考虑使用 S 盒、T0—T3 表生成密钥操作。因为 S 盒的大小是 256 字节,而 T0—T3 表的大小均为 1024 字节,所以很容易确定地址集是否为 S 盒的地址集。

算法 2 flush+flush 求解 T 表、S 盒地址

```
Input: Address_space, long unsigned Plaintext
Output: Cached_Address_space

1. for * addr in Address_space do
2.     flush(* addr)
3.     wait
4.     st=rdtsc()
5.     flush(* addr)
6.     if (rdtsc()-st) < F_Threshold do
7.         * addr → Cached_address_space
8.     addr += 64
9. end
10. P[16]=Plaintext.split(16)
11. for i=0 to 50000 do
12.     Encryption(P[16])
```

```
13. end
14. return Cached_ Address_space
```

3.3 偏移地址求解算法

内存与缓存的基本单位是 cacheline (64 字节), S 盒由 256 个 8bits 的项构成, 需要占用 4 个 cacheline, 但是在实际情况中, S 盒在 Cache 结构中是不对齐的, 要占用多于 4 个 cacheline 的缓存空间, 所以必须得到偏移地址才能够确定 S 盒的起始位置和终止位置, 如图 4 所示, 只有①点、②点、④点出现不对齐分布特征才可实施此攻击。

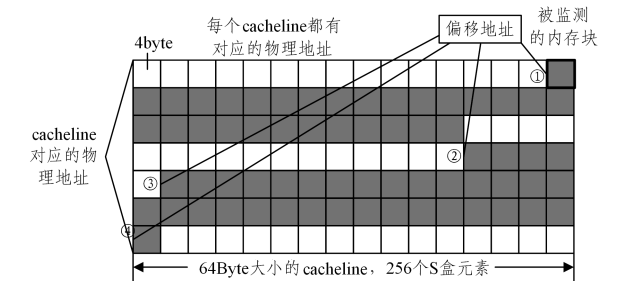


图 4 S 盒占用 cache 情况
Fig. 4 S box stored in cache

首先执行 flush(addr+i), 然后执行加密进程 flush(addr+(i++)) 个偏移位 ($i \in [0, 64]$), 通过将计时值与阈值对比推断是否命中, 从而确定偏移地址。这是利用 S 盒在 LLC 中的不对齐分布和 ASLR 特性。

算法 3 计算偏移地址算法

```
Input: byte * addr
Output: i
1. count[64]={0}
2. for i=1 to 63 do
3.   for j=0 to Repet_number do
4.     Encryption(P [16])
5.     st=rdtsc()
6.     flush (addr+i)
7.     ed=rdtsc()
8.     t[j]=ed-st
9.     flush(addr+i)
10.  end
11. ifeval(t [repet_number])< F_Threshold
12. return i
13. end
```

算法 3 中 i 代表偏移位, 1 个偏移位对应 1 个字节大小, 使用 S 盒地址+偏移位在偏移地址中完成 S 盒表项的访问, 并对偏移地址进行监控。

3.4 利用 flush+flush 方法求解密文值对应的监测计时

在实际攻击中可以不指定监测的 S 盒内表项数, 只要在 4~16 个 byte 间都可以 (即监测 4~16 个表项), 这样攻击具有更好的适用性, 测量的时间戳差值存储在密文字节计时向量 $\langle C[], t \rangle$ 中, 并使用键值对 $\text{dict}('C[]':t)$ 保存这组值, 算法 4 描述了求解该键值对的过程。其中, 间谍进程可以使用相同明文多次加密取平均值以消除噪声, 而在实际加密环境下, 只能监测一次计时。

算法 4 flush+flush 方法

```
Input: byte * S(A[0]), longunsignedint Plaintext[]
Output: X=dict ('C'[16]':t)
```

```
1. P[16]=Plaintext. split(16)
2. for j=0 to Repet_number do
3.   flush (* S(A[0]))
4.   C'[16]=Encryption(P[16])
5.   st=rdtsc()
6.   flush (* S(A[0]))
7.   ed=rdtsc()
8.   t[j]=ed-st
9. end
10. return X=dict ('C'[16]':eval(t))
```

3.5 密钥恢复算法

将 $C_0 \cdots C_{15} = S(A_0 A_5 \cdots A_{11}) \oplus K_9$ 按照 byte 大小进行拆分, 可得 $C_0 = S(A_0) \oplus k_0$, 同理可得 $C_1, C_2, \cdots, C_{15}$, 即 $C_i = S(A_j) \oplus k_i$ (其中 $i \in [0, 15]$), 则 $k_i = S(A_j) \oplus C_i$, i 代表一组命中密文值中的第 i 个 (byte) 值, j 代表行移位引起的变换。对于有小于 flush 阈值的每组 $C[]$ 值的每个 byte 值 C_i , 有 n 个可能的候选值, n 代表被监测的内存行所包含的 S 盒中表项 (bytes) 的数量, 正确的 key 是 C_i 的所有 n 个有效值中唯一具有相同解的值。

算法 5 给出了密文恢复简码, 其中用 $C'[i]$ 表示 C_i , 密钥字节 k 是从被测量的 M 组密文数据中筛选 m 组低计时值的密文中恢复。 $X = \text{dict} \{ 'C^0[]':t^0, 'C^1[]':t^1, \cdots, 'C^M[]':t^M, \cdots, 'C^M[]':t^M \}$ 保存了每组密文对应的计时值, $\text{Count_key}[]$ 是每个密钥字节候选 k 的向量, 它是密钥恢复步骤中相同密钥计算值出现的次数。

算法 5 恢复密钥算法

```
Input: long unsigned int X[], long unsigned int Threshold
Output: byte K[]
1. int Count_key[]
2. byte Y[][16]
3. C'[16]=Ciphertext. split(16)
4. for t=0 to M do
5.   if X('C'[16]')<F_Threshold
6.     Y[m][16]← C'[16]
7. end
8. for i=0 to 15 do
9.   int Count_key[16]={0}
10.  for t'=0 to m do
11.    for j=0 to n-1 do
12.      Count_key[ Y[t'][i] ⊕ S(A_j) ]++
13.    end
14.  end
15. k[i]=Count_key. index(max(Count_key[]))
16. end
17. return K9[]={k[0],k[1],...,k[15]}
```

如图 5 所示, 本次实验中取 $n=4$, 即被监测的内存行保存了 S 盒的前 4 个表项值, 需要恢复密钥 k_0 。在 M 组密文值中有 m 组密文被检测为低 flush+flush 计时值 (每组密文值 16 bytes)。以第一个 byte 密钥 k_0 为例对其进行计算, 将监控内存行的每个表项与 C_0 字节进行异或, k_0 有 4 种可能的解。若监控的内存行中的不同 S 盒中的表项使用同一个密钥 byte 进行加密, 则运算中得到了相等的 k 值, 进而对 m 组相关密文值的第一个 byte 位置做 $4m$ 次计算, 通过计数 $\max(\text{Count_key}[k_0])$ 就能得到正确的 k_0 , 由于 $k_i = C_i \oplus S[A_j]$, 以滑动

窗口的方式扩展到全部字节位,进而恢复最后一轮所有的密钥值 $k_0 k_1 k_2 k_3 \cdots k_{15}$ 。图 6 为本次实验 k_0 对应的有效计算值, $k_0^{(i')}$ 中的 i' 代表 m 组中密文组中的第 i' 个 k_0 ,每计算一个 k_i 时,其他 15 bytes 位置就会产生噪音值,因此实际产生的噪音值是正确密钥值的 $16 * n - 1$ 倍,图中异或值 a_3 出现的次数最多,因此可将该 byte 的密钥值推测为 a_3 。

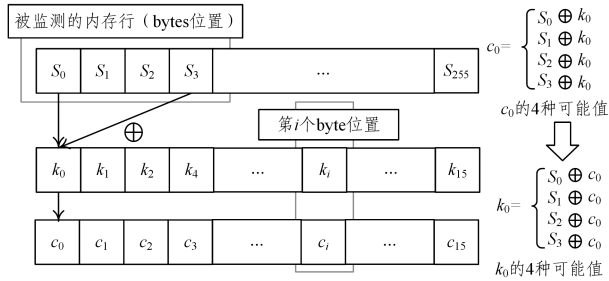


图 5 本次实验中 $n=4$ 时计算 k_0 步骤

Fig. 5 Calculate $k_0 (n=4)$

$$k_0^{(0)} = \begin{cases} 56 \\ a_3 \\ b_5 \\ 67 \\ \dots \end{cases} \quad k_0^{(82)} = \begin{cases} 18 \\ c_8 \\ 78 \\ 67 \\ \dots \end{cases} \quad k_0^{(129)} = \begin{cases} b_0 \\ 90 \\ a_3 \\ 50 \\ \dots \end{cases}$$

图 6 a_3 是计数最多的值,则该 byte 对应的 $k_0=a_3$

Fig. 6 a_3 is the value with the largest count, then $k_0=a_3$

4 验设置及结果分析

4.1 实验设置

本实验配备在 Intel i5-4590 四核心、频率为 3.3 GHz 的机器上进行;该处理器具有三级缓存架构,有 4 个 core,每个 core 有 1 个 32 kB 的 L1 数据缓存、1 个 32 kB 的 L1 指令缓存、1 个 256 kB 的 L2 缓存,并且 4 个 core 共用一个 6 MB 的 L3 共享一致性缓存,缓存行 (cacheline) 大小是 64 B,攻击目标为 Linux 4.15.0 操作系统中 Bouncy Castle JDK1.0 库的 AES 快速加密法——AESFastEngine.java^[4]。间谍进程与加密进程在同一 CPU 中运行。使用 Read Time-Stamp Counters (rdtsc) 测量时序,攻击被设置为跨核心工作,加密进程与间谍进程运行在不同的 core 中。

4.2 实验结果分析

图 7 中展示了 flush+flush 对 cache 命中和未命中的 50000 次攻击的计时分布,通过实验看到命中计时平均值为 162 个硬件周期,未命中计时值在 175 个硬件周期,差距为 13 个硬件周期,取阈值为 168,可最大分辨率区分 93% 以上的计时量。

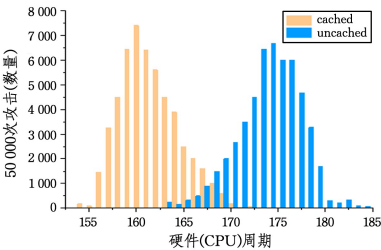


图 7 对 i5-4590 进行 50000 次 flush+flush 计时攻击

(区分命中、未命中)

Fig. 7 Perform 50000 flush+flush timing attacks on i5-4590

(distinguish hit and miss)

图 8 绘制了恢复正确的密钥与同步加密数之间的关系,采集 120×10^3 组密文数据 (即少于 40 s 的交互) 就能检测到泄露,采集 250×10^3 组密文值就能恢复最后一轮密钥值。

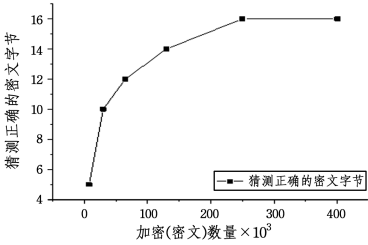


图 8 猜测 AES-128 位正确的 key bytes 与其加密请求数的关系

Fig. 8 Relationship between guessed correct key and number of encryption in AES=128

此实验方法是针对 L3 缓存进行计时攻击,而 evict+time 和 prime+probe^[11] 这两种攻击的间谍进程都是针对 L1 缓存的,因此本实验攻击的一个明显优势是:可以跨内核工作,更适用于当代的多核处理器环境,当然 evict+time 和 prime+probe 方法也能应用到缓存的最后一层,但是它们在跨内核攻击中性能会显著降低,这是由于其需要进行大量数据、大缓存的清除和检测。相比使用 flush+reload 攻击方法,flush+flush 攻击在下次加密进程执行前不用 flush 上一次加密所载入的缓存行,比 flush+reload 算法少一个 flush 步骤,因此 flush+flush 算法具有更好的时效性。Gullasch 等利用 Linux 的完全公平调度程序 (CFS),在受害者 AES 加密进程运行时,控制 CPU 并挂起 AES 进程,中断 AES 轮加密操作然后进行测量^[12]。而本次实验不需要特殊手段中断加密进程,使得此攻击更加有效、更加适用于真实的加密环境。

本文方法的缺点是:利用多次刷新偏移位确定 S 盒起始地址的方法,虽然能够确定偏移地址但却依赖于内存与缓存间的映射关系;也可以通过加密前的预处理来确定偏移地址,由于进入缓存的数据只有明文及 S 盒、T 表的数据,因此可以对明文进行标记,使其区别于 S 盒、T 表的数据作为明文输入,来更加精准地定位偏移地址,从而进行更加准确的攻击,然而此实验依赖于加密算法单次访问 S 盒,如果多次访问 S 盒,就涉及概率学和去除更大体量的噪音值的方法。

结束语 本文介绍了 Rijndael 查表法的实现原理,并在配置 Intel i5-4590 四核心、3.3 GHz CPU 处理器的机器上,对 Linux 系统虚拟环境下 Bouncy Castle JDK1.0 库中的 AES 快速加密法——AESFastEngine.java 进行 Cache 微结构攻击实验,从 AES 快速加密法寻找薄弱之处,而后使用 flush+flush 方法对 AES 查表法最后一轮加密进行攻击,采集到 120×10^3 组密文数据 (即少于 40 s 的交互) 就能检测到泄露恢复部分密钥值,收集 250×10^3 组密钥值就能完全恢复最后一轮密钥值。

参 考 文 献

[1] VISCAROLA P, MASON W. 实用技术 Windows NT 和 Windows 2000 设备驱动及开发[M]. 北京:电子工业出版社,2000.

[2] ART B, JERRY L. Windows 2000 设备驱动程序设计指南[M]. 施诺,译. 北京:机械工业出版社,2001.

[3] 刘鸿雁,袁平,吴恒柏. Rijndael 算法实现方案的设计策略研究[J]. 计算机工程与设计,2008(23):38-41.

[4] The Legion of the Bouncy Castle. JDK 1.0-lcrypto-jdk10-133. zip \src\org\bouncycastle\crypto\engines \ AESFastEngine. java[OL]. http://www.bouncycastle.org/latest_releases.html.

[5] YAROM Y, FALKNER K. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack[C]// 23rd {USENIX} Security Symposium ({USENIX} Security 14). 2014:719-732.

[6] ZHANG Y, JUELS A, REITER M K, et al. Cross-VM side channels and their use to extract private keys[C]// Proceedings of the 2012 ACM Conference on Computer and Communications Security. ACM, 2012:305-316.

[7] RISTENPART T, TROMER E, SHACHAM H, et al. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds[C]// Proceedings of the 16th ACM Conference on Computer and Communications Security. ACM, 2009:199-212.

[8] APECECHEA G I, INCI M S, EISENBARTH T, et al. Fine grain Cross-VM Attacks on Xen and VMware are possible! [J]. IACR Cryptology ePrint Archive, 2014, 2014:248.

[9] IRAZOQUI G, INCI M S, EISENBARTH T, et al. Wait a minute! A fast, Cross-VM attack on AES[C]// International Workshop on Recent Advances in Intrusion Detection. Springer, Cham, 2014:299-319.

[10] GRUSS D, MAURICE C, WAGNER K, et al. Flush+Flush: a fast and stealthy cache attack[C]// International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, Cham, 2016:279-299.

[11] OSVIK D A, SHAMIR A, TROMER E. Cache attacks and countermeasures: the case of AES[C]// Cryptographers' Track at the RSA Conference. Springer, Berlin, Heidelberg, 2006:1-20.

[12] GULLASCH D, BANGERTER E, KRENN S. Cache Games-Bringing Access-Based Cache Attacks on AES to Practice[C]// IEEE Symposium on Security and Privacy. 2011:490-505.



LU Yao, born in 1987. His main research interests include computer hardware security, and side-channel attacks.

(上接第 363 页)
而离服务器部署较近,因此其密钥分配宜选用推模式。

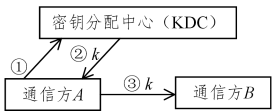


图 1 基于 KDC 的 Kerberos 协议密钥分配拉模式
Fig. 1 KDC-based Kerberos protocol key distribution pull mode

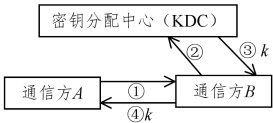


图 2 基于 KDC 的 Kerberos 协议密钥分配推模式
Fig. 2 KDC-based Kerberos protocol key distribution push mode

基于认证的密钥分配可采用密钥传送和密钥交换两种方式。密钥传送使用非对称密码算法,对本地生成的加密密钥加密发送至密钥管理中心。密钥交换使用本地和远端的密钥管理实体来共同生成成密钥。该方法无需 KDC 参与,发送方生成通信会话私钥,通过零知识证明或数字签名等方式来实现通信密钥的安全传送。工业网络中的典型应用有 Diffie-Hellman 协议,实际多采用 IETF 基于该协议开发的 Okaley 密钥交换协议。IP 安全协议集 IPSec 采用开放标准架构,其安全功能包括加密、访问控制、源地址验证、完整性校验以及防重放攻击等。IPSec 还引入了 IKE,用于共享密钥在不安全网络环境中的安全建立或更新,实现密钥管理。IKE 信息由 ISAKMP 协议头、SKEME 与 Oakley 字段构成,其特定格式取决于信息状态和模式。IKE 既能够用于 IPSec(VPN)的安全关联协商,也可以用于 RIPv2,OSPFv2,SNMPv3 等有保密要求的网络协议安全参数协商,在资源不受限的工业数据通信场景中具有广阔的应用前景。

结束语 在建设工业互联网的过程中,我们要重视平台、网络及数据的安全问题,尤其不能忽视网络传输的安全问题。

如果不解决安全问题,可能为企业带来灾难性的打击。本文就网络传输中的传输介质问题提出了规避之策,并就网络传输协议的选择、软硬件加密、加密算法的对比、密钥管理等问题提出了独到的建议,网络传输安全问题任重而道远。

参 考 文 献

[1] 卢坦,林涛,梁颂.美国工控安全保障体系研究及启示[J].保密科学技术,2018(4):24-33.

[2] 周剑,肖琳琳.工业互联网平台发展现状、趋势与对策[J].智慧制造,2017(12):56-58.

[3] 迈克尔·布雷迪,内德·考尔德,等.企业的工业互联网平台战略[J].上海质量,2017(5):15-18.

[4] 李海花.各国强化工业互联网战略,标准化成重要切入点[J].世界电信,2015(7):24-27.

[5] 陈泓汲,罗璿璐.互联网+网络安全新特性与发展建议[J].互联网天地,2015(10):1-4.

[6] 王峰.工业互联网平台分类研究[J].电信技术,2017(10):7-10.

[7] Industrial Internet Consortium (IIC). Volume G1:Reference Architecture. 8, 2015.

[8] Industrial Internet Consortium (IIC). Volume G5:Connectivity Framework, v1. 8. 8, 2016.

[9] IEC61158-4 Industrial Communication Networks-Fieldbus Specifications.

[10] IEC61784-2 Industrial Communication Networks- Profiles-Part 2: Additional Fieldbus Profiles for Real-time Networks Based on ISO/IEC 8802-3.



WU Yu-hong, born in 1981, Ph.D, associate professor. Her main research interests include internet of things, industrial internet security and big data, etc.