Load-to-store: store buffer 暂态窗口时间泄露的利用

唐明，胡一凡
（武汉大学国家网络安全学院，湖北 武汉 430072）

摘 要：为了研究现代处理器微架构中的漏洞并制定对应防护，针对负责管理访存指令执行顺序的内存顺序缓冲（MOB）进行分析，发现前向加载会把存在依赖的 store 指令的数据直接旁路到 load 指令，推测加载会提前执行不存在依赖的 load 指令，在带来效率优化的同时，也可能导致执行出错与相应的阻塞。针对 Intel Coffee Lake 微架构上现有 MOB 优化机制，分析如何利用内存顺序缓冲的 4 种执行模式与对应执行时间，构造包括暂态攻击、隐蔽信道与还原密码算法私钥的多种攻击。利用 MOB 引发的时间差还原内存指令地址，该地址可泄露 AES T 表实现的索引值。在 Intel i5-9400 处理器上对 OpenSSL 3.0.0 的 AES-128 进行了密钥还原实验，实验结果显示，30 000 组样本能以 63.6%概率还原出一个密钥字节，且由于内存顺序缓冲的特性，该利用隐蔽性优于传统 cache 时间泄露。

关键词：内存顺序缓冲；微架构侧信道漏洞；OpenSSL AES；时间侧信道
中图分类号：TP309.2
文献标志码：A
DOI: 10.11959/j.issn.1000−436x.2023051

Load-to-store: exploit the time leakage of store buffer transient window

TANG Ming, HU Yifan
School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China

Abstract: To research the vulnerability of modern microarchitecture and consider the mitigation, memory order buffer which was responsible for managing the execution order of memory access instructions was analyzed and found that load forward would directly bypass the data of dependent store instructions to load instructions, and speculative load would execute independent load instructions in advance. While bring efficiency optimizations, it might also lead to errors and corresponding blocking. The existing optimization mechanisms on the Intel Coffee Lake microarchitecture, and the leak attack scheme by using them were analyzed. Using the four execution modes of MOB and the corresponding duration, a variety of attacks were constructed including transient attack, covert channel, and recovery of the private key of the cryptographic algorithm. The time difference caused by MOB was used to leak the address of memory instructions, and the implementation of AES T table was attacked. Key recovery experiments were conducted on AES-128 with OpenSSL 3.0.0 on an Intel i5-9400 processor. The experimental results show that 30 000 sets of samples can recover a key byte with a probability of 63.6%. Due to the characteristics of memory order buffer, the concealment of the exploit is better than traditional cache time leaks.

Keywords: memory order buffer, microarchitectural side-channel vulnerability, OpenSSL AES, timing side-channel

收稿日期：2022−07−31；**修回日期：**2022−10−08
基金项目：国家自然科学基金资助项目（No.61972295, No.62072247）；武汉市科技项目应用基础前沿专项基金资助项目（No.2019010701011407）
Foundation Items: The National Natural Science Foundation of China (No.61972295, No.62072247), Wuhan Science and Technology Project Application Foundation Frontier Special Project (No.2019010701011407)

0　引言

为了优化计算机系统的性能，研究者对体系结构进行了多种改进，包括超标量流水线、分支预测、多级缓存（cache）、超线程等[1]。有的优化会导致系统进入不确定状态，并对访问权限外的数据进行访问。处理器会在指令集提交的阶段等待并检查，以确保写入指令集体系结构（ISA, instruction set architecture）的状态是正确的。然而，现有研究工作[2-14]显示，在 ISA 外，现代处理器执行过程中会在部件上留下痕迹。暂态攻击（TA, transient attack）利用这些痕迹，让处理器在错误执行状态下将信息泄露到系统安全边界以外。例如，Meltdown 攻击[9]利用处理器指令发生越权访问异常，旁路该内存数据到其他低权限指令。由于现代处理器包含种类丰富的核内/核间共享资源，攻击者可利用共享资源进行秘密信息传递，即隐蔽信道攻击。此外，利用隐蔽信道进行侧信道攻击可以还原加密算法私钥，如文献[7]利用 cache 还原 RSA（Rivest, Shamir, Adleman）密钥。现有微架构隐蔽信道的防护方法只能针对特定共享资源[15-16]，无法覆盖新出现的隐蔽信道。

本文选择内存顺序缓冲（MOB, memory order buffer）机制进行泄露分析研究。已有研究 MOB 漏洞的工作如下。Spoiler 攻击[6]逆向恢复 MOB 的地址匹配机制，确定 MOB 只对部分物理地址进行匹配。文献[4]利用 MOB 在超线程之间的共享，构建隐蔽信道攻击，要求发送方与接收方分处同一个物理核的 2 个逻辑核，利用前向加载（LF, load forward）失败与正常工作的时间差编解码数据。文献[17]介绍内存访问模型。本文逆向分析 MOB 所有优化机制与存在的漏洞，并基于 MOB 工作机制提出了一种新型侧信道攻击，可实现 OpenSSL 高级加密标准（AES）的密钥还原。

已有利用微架构侧信道攻击密码算法的工作大部分基于 cache 的工作原理，包括 Flush+Reload[7]、Prime+Probe[14]、Flush+Flush[11]。Flush+Reload 利用不同核共享 L3-cache，首先冲刷 cache 行，在 RSA 算法运行后重加载 cache 行并测量时间，如果时间较长，则代表 RSA 访问了对应 cache 行的内存地址。文献[18-19]进一步利用 Flush+Reload 的 cache 泄露攻击更多密码算法。文献[16]把此类 cache 侧信道攻击应用于网络传输的场景。文献[20-23]对 cache 替换策略与多级缓存的漏洞进行进一步利用。MemJam[5]利用时间泄露恢复密码算法访问内存模式，通过时间差确定访问内存是否发生 4k alias 异常，进而还原 AES S 盒实现私钥，但未对产生时间泄露的微架构进行分析。文献[15]研究 MOB 部分机制，并构造了泄露系统地址随机化的攻击。

本文研究微架构中 MOB 的优化机制与其所有可能的安全问题，并提出一种新型攻击方法 $A_{\mathrm{lts}}^{\mathrm{MOB}}$，用于内存访问指令地址还原，进一步可还原密码算法的私钥。本文主要贡献如下。

1) 逆向分析 MOB 的所有工作状态，并分析 MOB 工作状态中可利用的侧信道漏洞。

2) 设计了一种利用 MOB 机制的新型微架构侧信道攻击，成功还原了最新版本 OpenSSL AES 的私钥。

1　背景

1.1　处理器内存访问过程

Intel 处理器会对微指令进行乱序发射。微指令将先后进入指令重排序缓冲（ROB, re-order buffer），等待操作数准备好乱序进入端口执行，微指令完成后按序退出 ROB。指令运行期间，如果发生异常或中断，需要清除它与之后所有微指令的计算结果。

处理器通过总线与内存控制器连接。由于主存储器访问时间很长，在 DRAM（dynamic random access memory）与物理核之间部署 cache 能极大减少内存读写的时间开销。但执行 store 指令的时间还是远长于 load 指令与其他指令的时间。为了在执行 store 指令期间可以提前执行不依赖于其数据的 load 指令或其他指令，CPU 在 L1-cache 和访存执行单元之间引入了 MOB，如图 1 所示，主要包括加载缓冲（LB, load buffer）和存储缓冲（SB, store buffer），与 L1-cache、执行单元相连。load 指令、store 指令执行过程分为三步，即内存地址生成、内存地址翻译、内存数据访问。

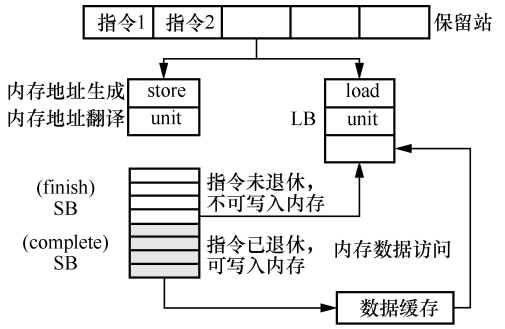


图 1　CPU 访存单元结构

内存地址翻译指如果开启地址虚拟化，需要通过操作系统的页表完成虚拟地址到物理地址的映射。内存数据访问会因指令类别的不同而不同。load 指令只需要地址寄存器准备好，就可以从数据缓存或 SB 中读取数据。而 store 指令需要数据寄存器和地址寄存器同时准备好才可以执行，执行过程需先把目标地址与数据写入 MOB 的 SB，值得注意的是，这个目标地址可能未完全翻译成物理地址[24-25]。当 store 指令进入 finish 状态时，还不可写入内存；当 store 指令之前的指令完成提交，进入 complete 状态时，才可写入内存。MOB 负责控制 store 指令何时写入数据到内存以及 load 指令何时执行。特别是当 load 指令与 store 指令存在数据依赖时，需要更精密的控制。

1.2　MOB 与数据依赖引起的时间差

内存数据依赖由两条访问同一个地址的 store 指令或 load 指令产生。在超标量乱序处理器中，为了发射尽可能多的指令，内存指令的执行顺序会与程序中原始代码顺序不一致，引发 RaW（read after write）、WaW（write after write）、WaR（write after read）数据依赖。

Intel 内存顺序模型[17]的基本原则如下：load 指令之间按序执行，store 指令之间按序执行，load 指令可以越过之前的 store 指令提前执行。只有 load 指令会发生有限的乱序，引发 RaW 数据依赖。

RaW 数据依赖指程序中访问同一个内存地址的 load 指令发生在 store 指令后。如果处理器在乱序执行中提前执行 load 指令违反了 RaW，就会使 load 指令读取到错误的内存值。MOB 负责管理内存指令的执行顺序。MOB 结构[24-25]如图 2 所示，由 SB 与 LB 组成，每个条目包含对应指令要访问的数据、目标地址与执行状态，其中，目标地址包括物理地址（PA）和虚拟地址（VA）。SB 会把一条 store 指令分为 2 个操作码（OP）来存储，一个在存储地址缓冲（SAB, store address buffer）中存储地址，另一个在存储数据缓冲（SDB, store data buffer）中存储数据。store 指令会在提交后进入 SB，这样处理器不会因为 store 指令的低效而阻塞，其他指令（包括 load 指令）可以乱序发射。

Intel 利用推测加载（SL, speculative load）与 LF 机制在加速的同时保证 RaW 数据依赖。SL 与 LF 示意如图 3 所示。当处理器判断 load 指令地址与之前 store 指令地址无关时，SL 允许 load 指令越过之前的 store 指令优先执行；当判断出现 RaW 数据依赖时，LF 使 load 指令可以从 SB 中直接得到数据，而不用等依赖的 store 指令完成内存写入，再从 cache 获取数据。这 2 个优化机制所需的地址检查通过对 LB 中 load 指令地址与 SAB 中 store 指令地址的匹配来完成。然而，MOB 不是总能正确完成地址检查的，在地址检查中，MOB 仅对部分位进行匹配来判断 load 指令与之前的 store 指令是否存在依赖关系。显然，这种数据依赖判断机制存在出错的可能。

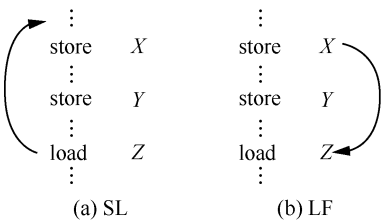


图 3　SL 与 LF 示意

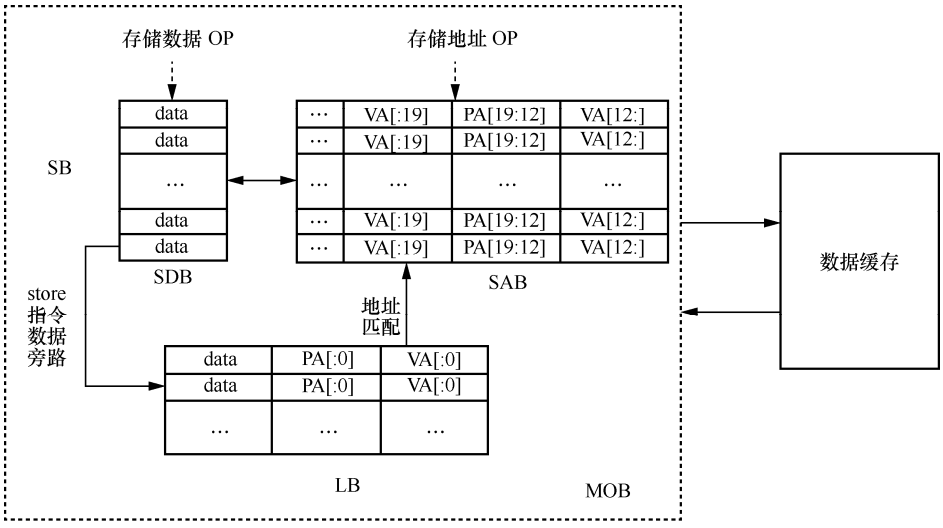


图 2　MOB 结构

2　MOB 攻击原理

本节介绍如何利用 MOB 在 RaW 数据依赖下执行错误构造攻击。根据攻击者能力目标与场景，可以把攻击分为以下三类。1) 攻击者想直接获取 load 读取的脏数据，需要在一个有限的窗口内把数据传递出去，属于 TA。2) store 指令和 load 指令都属于攻击代码，分别属于云服务器上的 2 个程序，它们的目标是通过 MOB 传递一个数据而不被管理者发现，属于隐蔽信道攻击。3) store 指令和 load 指令中的一个属于攻击代码，攻击目标是泄露受害者程序中的秘密数据，用于还原敏感数据，如密码算法的私钥。

Intel 关于 MOB 的地址匹配方式未完全公开，但官方文档[17]说明会出现错误的内存依赖关系识别的情况。MOB 可能会错误地旁路 store 指令的数据到 load 指令，或提前执行不该提前的 load 指令。内存指令会在退出 ROB 之前检查出错且不会把错误的数据写进 ISA，但对错误数据的清除回滚将带来执行时间上的差异。RaW 指令序列与 MOB 在不同地址关系下的不同执行情况如图 4 所示，具体如下。

情况 1　SL 正确触发。addr3 与 addr1、addr2 的低十二位都不一致，MOB 正确触发 SL，store1、store2 写入内存之前就进行 load3 的读取内存操作。

情况 2　LF 正确触发。addr3 与 addr2 完全一致，MOB 正确触发 LF，直接把 store2 在 SDB 里存放的数据发送给 load3，省去 load3 从 cache 中读取数据的步骤。

情况 3　LF 出错（即 4k alias）。addr3 与 addr2 的低十二位一致，但其他位并不相等。MOB 如果只对低十二位地址进行匹配，会误以为可以进行 LF。于是 load3 会得到 store2 中 reg2 的内容，这是一个错误的数据。当 load3 进入退休阶段，它将完全检查 SAB 中地址，会发现之前的匹配出错，这时候需要清除 load 指令之后所有的计算结果，并重新从 cache 读取数据。这会带来明显的时延。

情况 4　SL 出错。addr3 与 addr2、addr1 完全一致。store2 因为 reg2 依赖于之前的指令结果，仍在保留站中还未执行进入 SB。此时 MOB 错误地认为 load3 不存在依赖，使其错误地越过 store2 而读取到 store1 存放的数据。类似情况 3，在检测出错后需要回滚，从而带来明显的时延。

情况 3、情况 4 中 load3 读取了错误的数据，可能直接泄露脏数据。上述 4 种情况还存在明显的

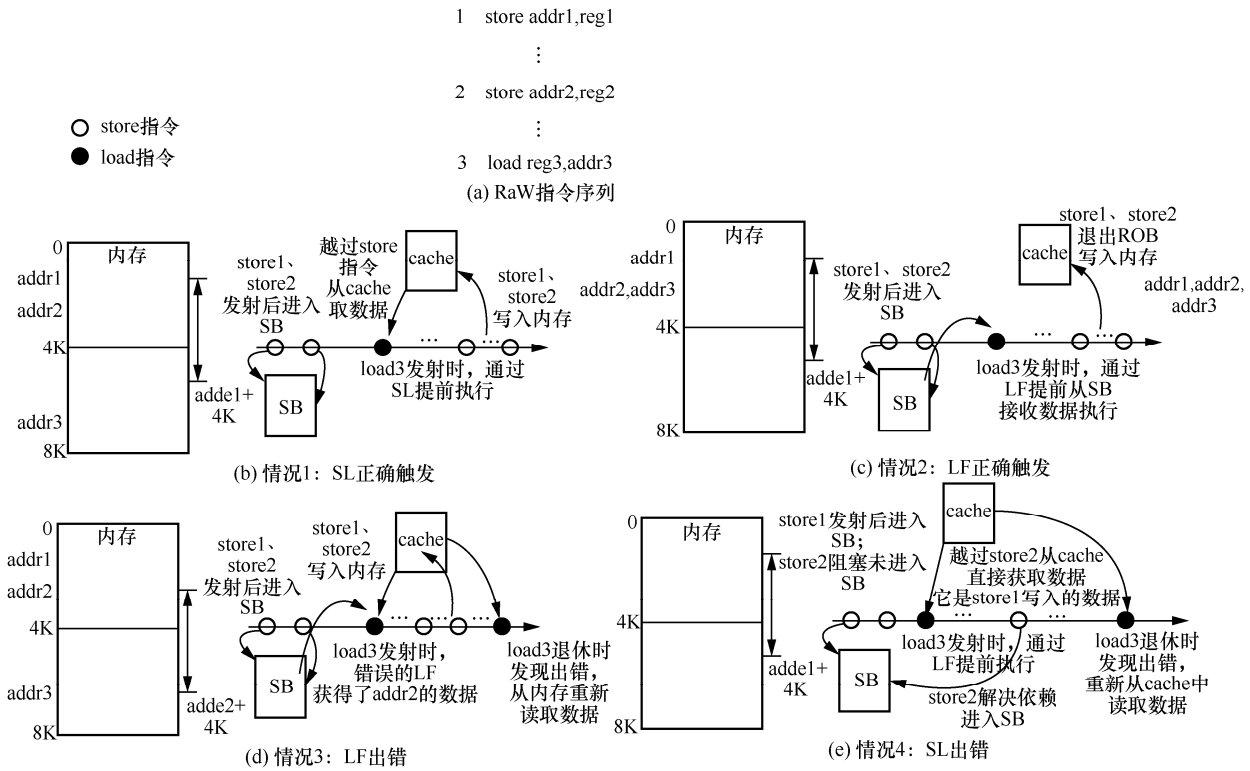


图 4　RaW 指令序列与 MOB 在不同地址关系下的不同执行情况

时间差，可以通过时间差推测出 3 条指令访问地址的关系，从而产生间接的数据泄露。MOB 的 4 种执行情况的时间差异如表 1 所示，其中，时间为归一化值。

表 1　MOB 的 4 种执行情况的时间差异

情况	时间
SL 正确触发	1.00
LF 正确触发	0.98
LF 出错	1.05
SL 出错	1.02

下面讨论利用不同执行情况构造的攻击场景与代码示例。

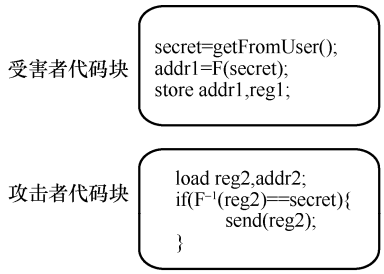
2.1　利用 MOB 触发的暂态攻击

TA 是利用处理器错误预测执行的攻击。攻击触发处理器出错，使之进入一个暂态窗口。在窗口内，处理器会访问到正常情况下无法访问的数据。本文介绍利用 MOB 的两类错误执行情况触发的暂态攻击。

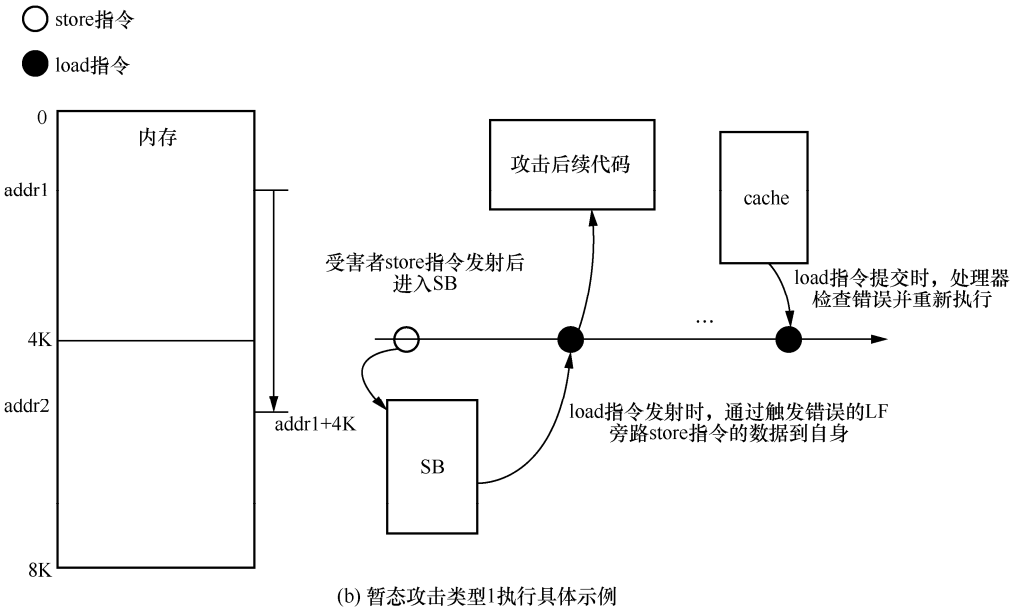
类型 1　受害者代码与攻击者代码在一个物理核上的 2 个不同逻辑核上运行（需要处理器开启超线程），且它们共享一段内存。攻击者目标是通过情况 3 利用错误的 LF 来获取受害者的数据。如图 5 所示，受害者存储秘密数据到内存，攻击者尝试读取不同内存地址，通过低十二位（页偏移）与秘密数据内存相等触发错误的LF来获取受害者的数据。需要注意的是，虽然攻击者和受害者共享一段内存，但不包括受害者 store 指令的内存。

因为攻击者与受害者难以同步，需要假设攻击者已知受害者可能进行内存访问的时间段，并在对应时间段不断执行 load 指令。若受害者在该时间段的某个时刻进行敏感数据访问，攻击者的 load 指令会触发情况 3，通过 LF 直接从 SDB 中获取受害者刚存储的秘密数据。该攻击类别属于微架构数据采样攻击[2]。

类型 2　受害者代码与攻击者代码在一个核上交替运行。攻击者可以利用情况 4，通过 SL 让受害



(a) 受害者代码块与攻击者代码块



(b) 暂态攻击类型1执行具体示例

图 5　MOB 暂态攻击类型 1

者代码的 load 指令越过最近的 store 指令，来读取之前 store 指令在对应内存位置的敏感数据。具体攻击已在 Spectre v4[3]中完成。

2.2　利用 MOB 的隐蔽信道

发生 SL 的执行时间，与发生 LF 出错的执行时间存在明显的差异，显然可以选择其中一个编码为 1，另一个编码为 0 构建隐蔽信道。MOB 隐蔽通道的数据发送与接收如图 6 所示，store 指令在发送方，load 指令在接收方。发送方通过选择 store 指令地址来决定传输不同的比特。当传输 1 时，存储与 load 指令低十二位一致的地址；当传输 0 时，存储任意地址。接收方每隔一定时间，加载固定地址，根据加载时间的长短解码 0 和 1。

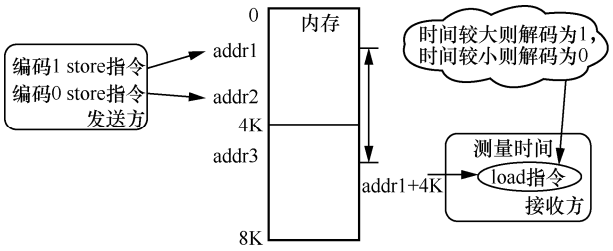


图 6　MOB 隐蔽通道的数据发送与接收

文献[4]针对多种云服务器进行了 MOB 隐蔽信道攻击，包括协议双方如何进行同步、如何保证数据的准确率。

2.3　利用 MOB 泄露受害者访存信息 $A_{\mathrm{lts}}^{\mathrm{MOB}}$

本节提出利用 Intel 处理器中 MOB 泄露敏感数据的通用攻击方法。本文提出的 $A_{\mathrm{lts}}^{\mathrm{MOB}}$ 攻击中受害者与攻击者在一个逻辑核上交替运行。攻击目标为受害者代码中某条 load 指令的地址（该地址包含敏感信息），攻击者通过设置 store 指令的地址，测量受害者代码整体运行时间来还原敏感信息，即通过 MOB 机制实现 load 指令地址向 store 指令地址的泄露。$A_{\mathrm{lts}}^{\mathrm{MOB}}$ 利用 MOB 机制出错造成的时间差，实现对受害者当前 load 指令地址的恢复。

$A_{\mathrm{lts}}^{\mathrm{MOB}}$ 利用 MOB 中 SL 正确触发与 LF 出错回滚的时间差还原地址信息。代码块（gadget）结构如图 7 所示，包含一段与敏感数据有关的 load 指令。攻击者与受害者需要共享一块内存区域，这个区域不需要包含敏感数据，最多只需要一个页面的大小。对于受害者进程，如加密算法，假设攻击者可以任意次监听它的输入，并测量它的整体运行时间。

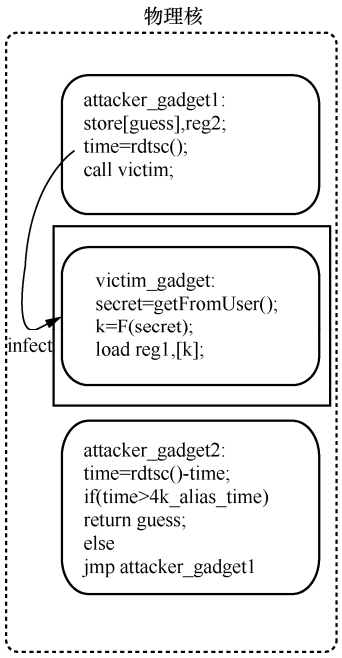


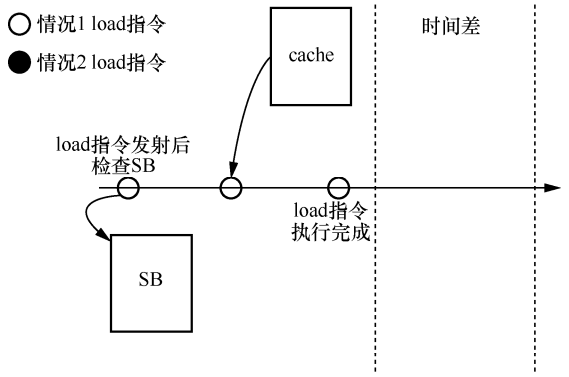
图 7　gadget 结构

设置步骤　假设攻击者已知受害者代码，根据受害者 load 指令的地址范围，选择一组与它低十二位可能一致的冲突地址集合。每次受害者运行前，攻击者选择一个可能的冲突地址执行若干次 store 指令。

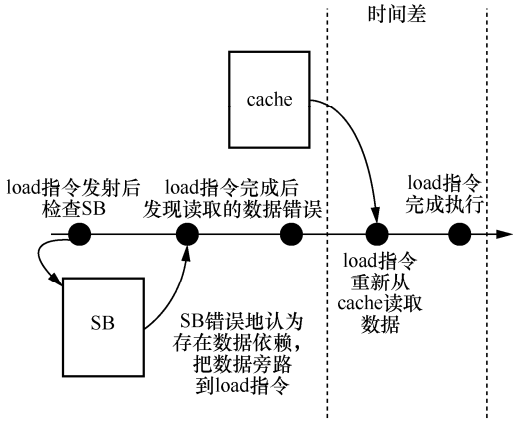
运行测量步骤　攻击者在完成 gadget1 中 store 指令后，设置第一个时间测量点。这时受害者代码运行，观测到完成后攻击者设置第二个时间测量点（gadegt2 中），计算两个点的差值来统计受害者代码运行的时间。基于 MOB 的工作机制，受害者代码运行时间存在以下 2 种情况。

情况 1　受害者 load 指令的地址与攻击者在设置步骤中选取的 store 指令的地址无关。load 指令执行过程中，在解码发射后检查 SAB，发现没有相匹配的地址，通过 SL 越过 store 指令完成对内存的访问。此时运行时间较短。

情况 2　受害者 load 指令的地址与设置步骤中选择的 store 指令的地址的低十二位相等。这时 load 指令检查 SAB，发现部分地址（低十二位）匹配（但实际上完整地址不匹配），于是错误地通过 LF 让 load 指令从 SDB 中直接获取攻击者 store 指令的数据。直到 load 指令进入退休阶段，对地址进行完整匹配，才发现之前 LF 出错。这时候需要清除之前的计算结果，并重新读取数据，从而受害者代码运行时间会因 load 指令的异常而较长。



(a) 情况1：正常运行，时间较短



(b) 情况2：发生4k alias，时间较长

图 8　$A_{\text{lts}}^{\text{MOB}}$ 实施时 load 指令在 CPU 中运行状态

地址还原步骤　攻击者穷举冲突地址集合内所有地址作为 store 指令的目标地址，通过上述设置步骤与运行测量步骤，最终得到一个时间明显较长的冲突地址，其低十二位作为受害者代码块 load 指令地址的低十二位。

当受害者程序运行时，敏感数据与 load 指令的地址相关，可通过 $A_{\text{lts}}^{\text{MOB}}$ 攻击构造还原敏感数据的方案。

3　$A_{\text{lts}}^{\text{MOB}}$ 在 AES 中的攻击

OpenSSL 中 AES-128 轮函数采用 T 表实现，运算过程的中间值 x 会作为 T 表索引，计算对应地址后用 load 指令读取内存。如果攻击者可以利用 store 指令，与 T 表 x 值所处地址偏移 4 096N（N 为任意自然数）处发生冲突，那么 AES 在查 T 表时，会受到干扰引发 LF 出错，导致时延；否则 SL 正确触发，加密时间会很短。该时间差用于推断 AES 中间结果，进一步还原轮密钥。

假设实际攻击场景如下。

1) 受害者代码运行在主机 A 上，处理器支持 MOB，主机 A 上部署 AES 加密算法，向多个用户提供加密服务，在特定内存区域存储管理不同用户私钥$\{K_i\}$，用户 C_i 无权限访问用户 C_j 的私钥 K_j。

2) 攻击者与受害者均为 A 的注册用户，攻击者 Ca 的攻击目标是受害者 Cv 的私钥 Kv，攻击者需要在主机 A 中部署一个可控代码 Code_Ca，并对 Code_Ca 中 store 指令地址进行控制。

3) 攻击者 Ca 首先以合法用户身份调用 AES 加密算法，得到 T 表内存地址（该地址对应内存是所有用户共享的）；Ca 在 Cv 调用 AES 加密时同步执行 Code_Ca，根据 AES 运行时间，还原 AES 加密中间结果和 Kv。假设受害者加密明文的分布均匀，AES 攻击步骤如下。

步骤 1　获取 AES 的 T 表地址。

步骤 2　选择轮密钥猜测值，根据猜测值，Ca 对窃听到的明文进行分类。

步骤 3　Ca 运行 Code_Ca 的同时，Cv 运行加密算法，Ca 采集 Cv 的加密运行时间（步骤 1 和步骤 2 同步进行）。

步骤 4　重复步骤 2 和步骤 3，对轮密钥某一字节进行穷举，得到评估值排前 4 的密钥猜测结果。

步骤 5　对轮密钥 16 个字节分别重复步骤 2～步骤 4，完成对轮密钥的完整还原。

3.1　攻击的准备–获取 T 表地址与相关性攻击方法

假设 AES 的 4 个 T 表在内存中连续存放，只要获得一个 T 表（Te0）的初始地址，就可以得出其余 3 个 T 表的初始地址。下面介绍利用 $A_{\text{lts}}^{\text{MOB}}$ 的 Te0 初始地址获取过程（对应 AES 攻击步骤 1）。

1) setup 阶段。攻击者代码申请一个 4 KB 的 char 数组，并对数组中某一元素进行 store 操作。

2) run and measure 阶段。作为注册用户，攻击者已知自己的私钥 Ka，选取一个等于 Ka 的明文。调用 AES，运行中会对 addr_{Te0} 地址的内存进行 load 操作，同时记录 AES 运行时间。

3) recover address 阶段。重复执行 setup 阶段和 run and measure 阶段 4 096 次，实现 setup 阶段对数组元素遍历，根据最长 AES 执行时间还原 Te0 初始地址低十二位。

上述步骤得到的 char 数组元素地址 addr_{c} 是与 Te0 初始地址（addr_{Te0}）发生低十二位冲突的地址，可以推出 $\text{addr}_{\text{c}}+4n$ 会与 Te0[n]发生冲突。依此类推，分别得到 Te0、Te1、Te2、Te3 所有表项的冲突地址（如图 9 所示），用于之后的攻击步骤，获

取 T 表地址的过程如算法 1 所示。T 表元素地址（addr）都会与 char 数组元素地址（addr'）一一对应，满足

$$\text{addr}' - \text{addr}_{\text{c}} = \text{addr} - \text{addr}_{\text{Te0}} \tag{1}$$

算法 1　addr_generate 冲突地址生成（AES 攻击步骤 1）

输入　addr_array[]（构造冲突地址的集合）

输出　addr_char, addr_array[]

```
1) for   addr in addr_array
2)       store(addr, random_data);
3)       time ← rdtsc();
4)       AES_encypt(key=Ka, msg=Ka);
5)       time ← rdtsc() – time;
6)       append(time_list,time);
7) end for
8) addr_char, addr_temp ← addr_array;
9) for   time in time_list
10)      if   max_time < time   then
11)           addr_char ← addr_temp;
12)           max_time ← time;
13)      end if
14)      addr_temp ++;
15) end for
16) return addr_char, addr_array[]
```

使用 addr' 与 addr_{c} 的差值，可以获得 AES 访问 T 表地址 addr 与 addr_{Te0} 的差值，进一步计算可以得到在 T 表的索引 n。对于第一轮查 Te0 表第一个字节，该索引就是明文第一个字节 m_0 与第一轮子密钥 sk_0 第一个字节异或的结果，即

$$m_0 \oplus \text{sk}_0 = \frac{\text{addr} - \text{addr}_{\text{Te0}}}{4} \tag{2}$$

实际运行中受到噪声的影响，在不能同时控制密钥和明文的情况下，不能明确地获得 addr' 的值。中间值与 T 表地址关系如图 10 所示。

攻击者尝试还原其他用户轮密钥的策略需要对某个轮次子密钥的字节逐一还原，但攻击者只能获取受害者 AES 加密的完整执行时间，一次 AES 完整加密包括 144 次 T 表查表。AES 共有 10 轮运算，前 9 轮运算查 T 表，最后一轮运算查 S 盒。以第一轮子密钥第一个字节还原为例，共有 36 次查 Te0 表，第一轮对第一个字节以外的 35 次查 Te0 表都会对 $A_{\text{lts}}^{\text{MOB}}$ 还原索引造成干扰。具体步骤如下。

步骤 1　计算一个内存访问变带宽矩阵 $\boldsymbol{P}$。如图 11 所示，$\boldsymbol{P}$ 的每一行代表一个随机的明文（M 个

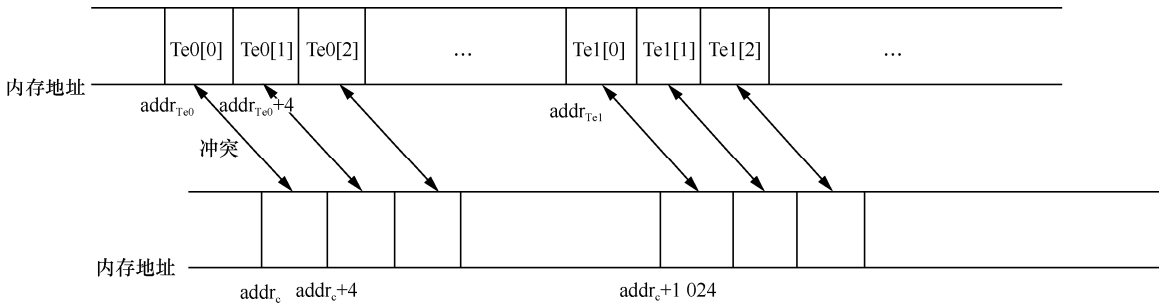


图 9　T 表地址的获取与冲突地址的构造

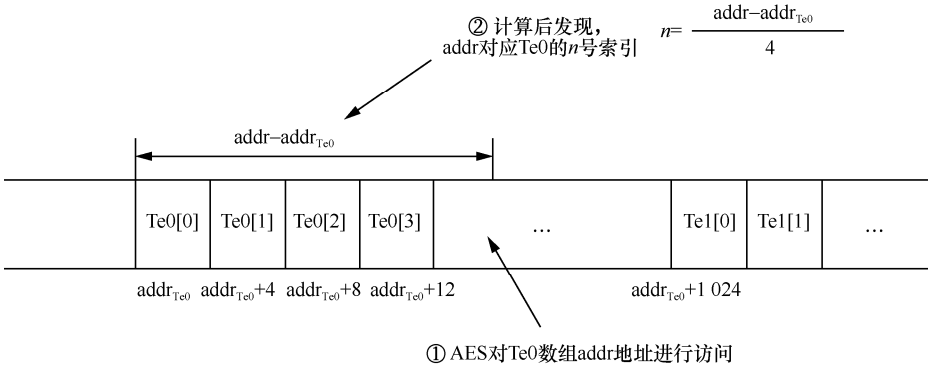


图 10　中间值与 T 表地址关系

明文），每一列代表该明文在一个轮密钥字节猜测值下是否访问 Te0[0]（元素为 0 表示不访问，为 1 表示访问）。

步骤 2　采集 M 组 AES 加密对应明文的运行时间，得到一个长度为 M 的时间向量 $\boldsymbol{T}$。

步骤 3　利用 $\boldsymbol{T}$ 与矩阵 $\boldsymbol{P}$ 每一列计算相关性系数，得到长度为 256 的相关性向量，取向量中排前 4 的猜测值作为候选密钥字节。

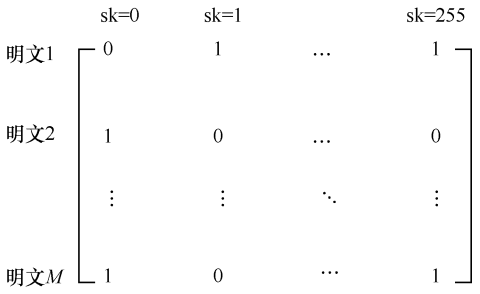


图 11　$\boldsymbol{P}$ 含义

3.2　已知明文还原一个子密钥字节

利用已知明文攻击改进 $A_{\text{lts}}^{\text{MOB}}$，还原轮密钥的过程如下。

设 AES 第一轮密钥第一个字节为攻击目标。攻击者对窃听到明文按第一个字节取值排序，并选取猜测密钥第一字节 $\text{sk}_{\text{guess}_i}$ 与对应明文第一字节相等。对 $\text{sk}_{\text{guess}_i}$ 每个取值（0～255）收集明文数量为 M，共有 256 组明文，合并存放在一个明文二维数组中，如图 12 所示。若每组明文的第一个字节与一个轮密钥字节猜测值相等，意味着当猜测值正确时，对该组明文调用 AES 时会发生 Te0[0] 查表操作。每组明文首字节以外的值因为均匀采样而近似为随机值（random）。

实际攻击过程中，Ca 窃听 Cv 的加密明文与对应的加密时间。由于受害者明文分布的均匀性，只要窃听时间够长，Ca 就可以获得 256 组所需明文。Ca 把窃听的 256M 次加密时间按明文首字节分类后存放在对应数组 time_array_i 中，数组编号 i 对应明文首字节取值，每个数组 time_array_i 有 M 个单元，存放 M 次加密时间。

对一个猜测密钥 $\text{sk}_{\text{guess}_i}$，攻击者 Ca 要在 Cv 调用 AES 执行前对 Te0 数组初始地址对应冲突地址 addr_{c} 进行 store 操作。在 Intel 处理器中可以使用 rdtsc 指令获取精确的时间点。每次加密采集的时间存放在数组 time_array_i（长度为 M）中。具体过程如算法 2 所示。

算法 2　sample 对明文加密时间的采集（AES 攻击步骤 2 和步骤 3）

输入　addr_char（冲突地址），M（选择明文组数），msg_array[]（明文向量）

输出　time_array[]

1) for　i from 0 to M
2)　　store(addr_char, random_data);
3)　　time ← rdtsc();
4)　　AES_encrypt(key=Kv, msg=msg_array[i]);
5)　　time ← time – rdtsc();
6)　　time_array[i]　　← time;
7) end for
8) return time_array[]

为了得到正确密钥，需要对密钥猜测值进行排序。对于任意 2 个密钥猜测值 $\text{sk}_{\text{guess}_i}$ 与 $\text{sk}_{\text{guess}_j}$，比较对应时间数组 time_array_i 与 time_array_j，会出现以下 2 种情况。

情况 1　$\text{sk}_{\text{guess}_i}$ 为正确密钥，time_array_i 中大

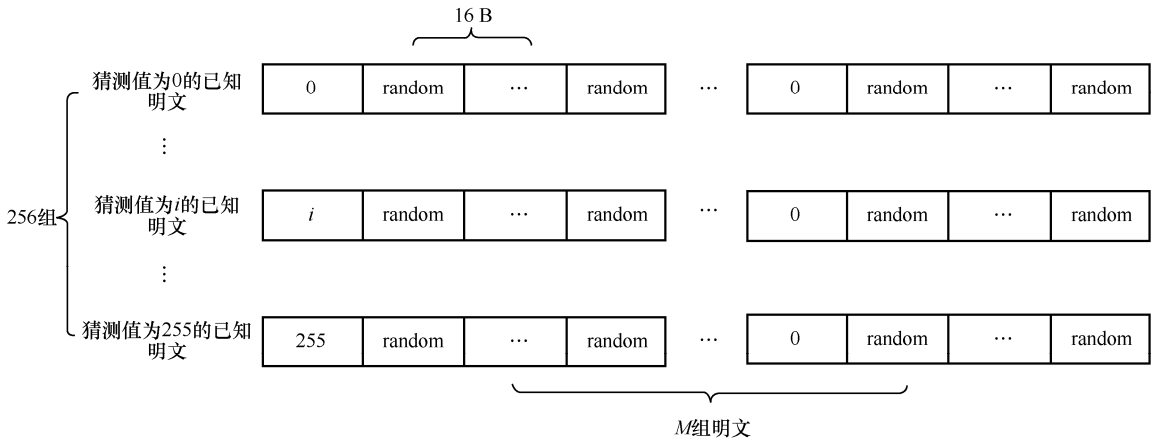


图 12　256 组已知明文

多数元素明显大于 $time\_array_j$ 中元素。因为 $time\_array_i$ 对应明文加密过程会访问 Te0[0]，Ca 的 store 指令将触发 LF 出错产生较大的时延。而 $time\_array_j$ 对应明文在查 Te0 表时将触发 speculative load，运行时间较短。

情况 2　sk_{guess_i} 与 sk_{guess_j} 都不正确，2 个时间数组的差别不大。攻击者设置的 store 指令对两组 AES 第一轮的计算时间都没有影响，时间差异来自噪声。

为了方便不同密钥猜测值的时间数组排序，用一个大小为 256 的整数数组 score_array 保存密钥猜测值评分，取评分前 4 的猜测值作为候选密钥，具体评分规则如算法 3 所示。

算法 3　score_key 密钥猜测值的穷举与评分（AES 攻击步骤 4）

输入　addr_char（冲突地址），M（已知明文组数），msg_matrix[][]（明文矩阵）

输出　score_array[]

```
1)for   i from 0 to 256
2)     conut ← 0;
3)     for   j from 0 to 256
4)          if   j == i   then
5)              continue;
6)          end if
7)          time_array1 = sample(addr_char , M,
             msg_ matrix[i]);
8)          time_array2 = sample(addr_char , M,
             msg_ matrix[j]);
9)          if sum(time_array1) > sum(time_array2)
             then
10)             count ++;
11)         end if
12)         score_array[i] ← count;
13)    end for
14) end for
15) return score_array[]
```

已知明文分析对 A_{lts}^{MOB} 攻击 AES-T 表实现效果有明显的改进，本节分别进行 M=20 000、30 000、40 000 的对比实验来比较相关性分析与已知明文攻击的准确率，结果如图 13 所示。

利用 A_{lts}^{MOB} 对 AES 的攻击存在以下 3 种噪声。1) 系统中其他用户进程的噪声；2) 除第一轮外，其他轮次对 Te0 表的查找操作；3) 采集过程中引入的环境噪声。其中，第一种和第三种噪声由外部因素引入，已知明文分析较相关性分析的改进在于对第二种噪声干扰的削弱。

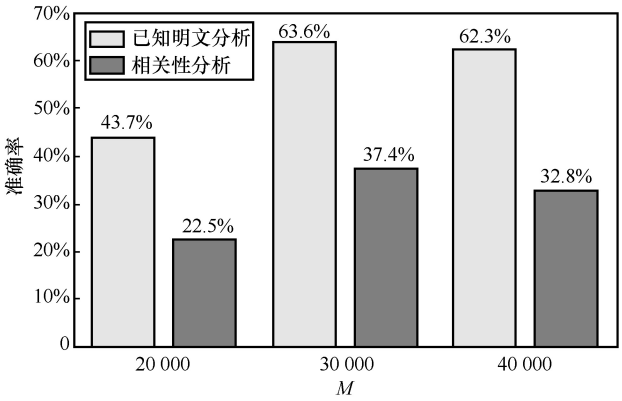


图 13　相关性分析与已知明文分析的准确率对比

完整的 AES 有 9×4=36 次对同一个对应 Te 表的查表操作。这 36 次查表的时间构成了 AES 加密时间的主要信息。当攻击目标为第一轮密钥首字节时，实测时间中只有一次查表的时间信息是有效的，其他 35 次的信息都是噪声。

基于相关性模型分析方法中，所有 36 次查表与 store 指令的地址发生冲突的概率同分布，干扰的 35 次查表会极大掩盖有效查表的时间。

已知明文分析方法中，当猜测密钥字节正确时，第一轮首字节查表时间较长且固定，由于轮密钥要求随机独立，查得 Te0[0]的概率为 $\frac{1}{256}$，剩余 35 次查表时间的累加值随机且均匀分布。当样本量足够大时，剩余 35 次查表引入的噪声将被削弱。

4　A_{lts}^{MOB} 攻击实验结果

按 3.2 节的密钥还原方法对 AES 轮密钥 16 个字节进行还原后，会在每个字节上分别得到 4 个候选密钥字节，组合后得到 $4^{16} = 2^{32}$ 个候选轮密钥，进一步穷举得出轮密钥最终猜测值。

4.1　有效性验证

本节采用开源加密算法库 OpenSSL(3.0.0)验证上述攻击的有效性。实验硬件配置如表 2 所示。

表 2　实验硬件配置

硬件	配置
CPU	Intel(R) Core(TM) i5-9400 CPU @ 2.90GHz
系统环境	Windows10
内存	8.00 GB (7.87 GB 可用)

实验 1 展示对 AES 第一轮密钥的首字节还原过程。攻击准备与轮密钥字节还原流程如 3.1 节和 3.2 节所述，设每个猜测值对应的每组已知明文数量 M= 30 000。一次还原实验后会得到 256 个密钥猜测值的评分，用该分减去算术均值并归一化，统计柱状图如图 14 所示。从图 14 中可以看到，实验中轮密钥正确值为 134。当猜测密钥正确时，分数最高，此外与正确值汉明距离较小的猜测值都有较高的评分。因为与正确密钥汉明距离小的猜测值对应的中间结果与正确中间结果取值更接近，意味着会对相邻的 T 表索引进行查询。

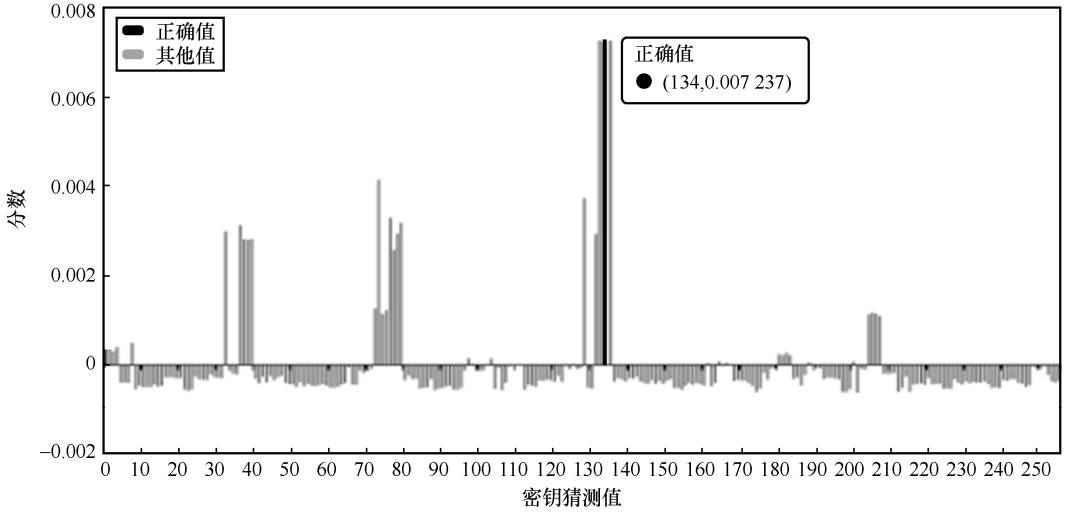


图 14　256 个密钥猜测值的评分

实验 2 验证 $A_{\mathrm{lts}}^{\mathrm{MOB}}$ 在已知明文攻击条件下还原 AES 密钥的准确率。准确率为 n 次密钥猜测实验中正确密钥字节排在前 4 的比例。对实验 1 重复进行 1 000 次。发现正确密钥分数的排序大部分在前 4，仅有 6.8%排在前 10 之后。正确密钥排序分布如图 15 所示，横坐标为 0 表示正确密钥评分最高，纵坐标表示落入该排序区间的正确密钥数量。$A_{\mathrm{lts}}^{\mathrm{MOB}}$ 已知明文攻击还原 AES 密钥的准确率能达到 63.6%。

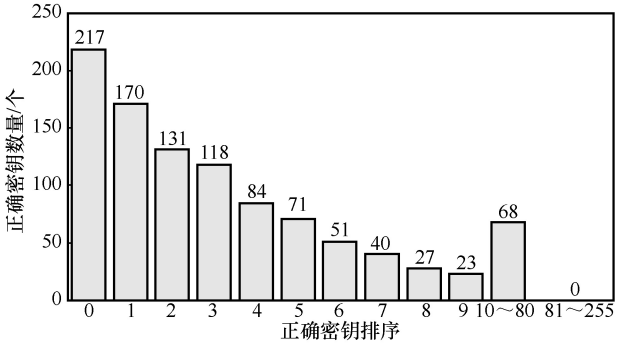


图 15　正确密钥排序分布

实验 3 统计明文数量 M 对准确率的影响。分别对 M=10 000、20 000、30 000、40 000、50 000 进行 500 次实验 1，还原密钥的准确率如图 16 所示。从图 16 可以看出，M=30 000 时能达到较高的准确率。上述 3 个实验验证了已知明文方法下 $A_{\mathrm{lts}}^{\mathrm{MOB}}$ 可以在 M=30 000 的明文组数下以较高准确率还原 T 表实现 AES-128 密钥。

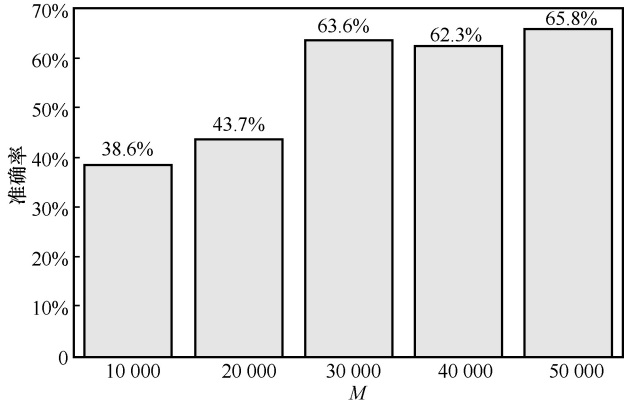


图 16　还原密钥的准确率

综上所述，$A_{\mathrm{lts}}^{\mathrm{MOB}}$ 是一种强大的敏感数据泄露原语，只要一个程序中包含敏感信息，且它利用敏感信息直接或间接地作为索引访问了内存中的一个数据结构，就可以利用 $A_{\mathrm{lts}}^{\mathrm{MOB}}$ 构造一个敏感信息还原攻击，如 AES、SMS4 加密算法的查表实现。

4.2　与已有工作对比

选择微架构侧信道领域最常用且对 AES 软件实现攻击最有效的 3 种算法 Flush+Reload[7]、Prime+ Probe[10]、Flush+Flush[11]与本文算法进行对比，对 OpenSSL AES T 表实现进行私钥还原对比实验，比较结果如表 3 所示。由于最新版本 OpenSSL AES T 表实现已经针对 cache 类的时间攻击实施了防护，对比实验在旧版本 OpenSSL-fips-2.0.8 的 AES 上进行。

首先，比较不同算法对 AES 攻击的有效性和效率，统计不同算法以超过 50%概率还原出轮密钥一个字节需要采集多少次加密。然后，统计不同算法根据采集的加密还原出密钥字节的消耗时间。$A_{\text{lts}}^{\text{MOB}}$ 所需的加密次数平均是其他算法的 8 倍，攻击时间是其他算法的 1.36 倍。$A_{\text{lts}}^{\text{MOB}}$ 的时间效率不如对比算法。

Gruss 等[11]对时间侧信道攻击提出隐蔽性的指标，从攻击过程中 CPU 性能计数器是否会发生异常来判断攻击是否隐蔽，并统计了 Flush+Reload、Prime+Probe、Flush+Flush 这 3 种算法在 cache 相关的计数器上的影响，结果显示 Flush+Reload 和 Prime+Probe 会在“cache references”与“cache misses”出现明显异常。本文提出的 $A_{\text{lts}}^{\text{MOB}}$ 不需要频繁刷新 cache，不会在 cache 访问上出现异常；构造内存模式冲突的 4k alis 仅在出现正确密钥字节时触发，不会在性能计数器上出现异常。

5　防御措施

现有关于微架构攻击的防护方案可以按照目标攻击类型分为两类，一类是针对 TA 的防护方案，另一类是针对隐蔽信道的防护方案，具体如表 4 所示。

处理器中 MOB 的优化机制导致了多种类型的微架构攻击，包括 TA、隐蔽信道和对加密算法的侧信道分析。TA 的防护分为软件级别的防护与对微架构设计的防护。

软件级别的防护与漏洞检测如通过代码级别的修改[16,26-27]或自动化漏洞定位[28-29]避免分支预测出错可能产生的泄露，但是对 $A_{\text{lts}}^{\text{MOB}}$ 而言，代码特征仅为 load 指令或 store 指令，相应防护与检测会出现很高的假阳率。

微架构级别的防护分为两类。一类是在整个系统中对攻击的特征进行检测，文献[30-31]利用动态数据流分析的方法，跟踪检测所有敏感数据相关的操作是否存在泄露，但这类方案开销过大；另一类

表 3　$A_{\text{lts}}^{\text{MOB}}$ 与其他微架构时间侧信道对比

算法	所需加密次数/次	攻击时间/s	隐蔽性	已知防御
Flush+Reload	500	215	无	有
Prime+Probe	9 600	234	无	有
Flush+Flush	700	163	有	有
$A_{\text{lts}}^{\text{MOB}}$	30 000	279	有	无

表 4　现有微架构攻击的防护方案

防护方案	防护措施	具体方法
针对 TA 的防护方案	软件级别防护 TA 触发	对 spectre v1 触发：文献[16,26]
	软件级别检测是否存在漏洞	对分支条件进行掩码防护：文献[27]
		模糊测试：文献[28]
		符号执行：文献[29]
	改进微架构设计	对条件分支相关部分的修改：文献[32-33]
		解码阶段的隔离：文献[34]
		修改数据传播的控制策略：文献[35]
		利用污点分析检测泄露：文献[30-31]
		安全内存设计：文献[36]
针对隐蔽信道的防护方案	阻止发送方发送有效信息	InvisiSpec：文献[13] Muontrap：文献[37] CleanupSpec：文献[38]
	降低信道质量	限制接收方带宽：文献[39]

是对于具体部件的安全设计，如文献[32-33]修改分支预测单元（BPU）条件分支相关的部分来防止 Spectre 这类利用分支预测错误的攻击。这类方案无论是在设计开销还是在防护效果上都是已有防护中最优的设计。但根据本文研究，尚无针对 MOB 资源的微架构级别的安全防护设计。

利用 MOB 构造的攻击基于以下 3 个条件：1) LF 与 SL 出错使攻击者可以在处理器回滚之前操作脏数据；2) LF 与 SL 触发的条件与访存的地址相关，导致它们触发正常与否的时间差会泄露对应指令访存的地址；3) SB 在进程切换间、超线程的逻辑核间共享。攻击者不仅可以利用资源共享其他用户的脏数据，也可以直接构建隐蔽信道，所以条件 3)是利用 MOB 构造攻击的核心。

因此，防护目标是切断 MOB 在不同用户之间的共享。一方面，在超线程中 2 个逻辑核上的不同线程对 SB 的访问设置隔离，只要处理器核启用超线程，SB 就均等地分成 2 个独立的部件，每个线程的内存指令只能访问对应的 SB 部分。另一方面，进程切换中需要保证处理器在发生进程切换时，一个进程的 store 指令全部完成执行并退出 ROB 后清除 SB 中内容 。这样可以保证每个线程的 load 指令无法访问另一个线程在 SB 中的数据，它的 store 指令也无法对其他线程的 load 指令执行产生影响。

6 结束语

MOB 在现代处理器中负责管理内存指令的执行顺序，为了提高执行效率的优化策略，引入多个安全漏洞。本文详细分析了内存顺序缓冲相关的优化机制，并总结出 4 种 RaW 指令序列执行模式。这 4 种执行模式因指令地址的不同而产生，对应不同的执行时间。

本文分析基于 MOB 不同执行模式构造暂态攻击和隐蔽信道，并提出一种新型攻击，即 $A_{\mathrm{lts}}^{\mathrm{MOB}}$ 攻击。攻击者通过 store 指令与其他用户 load 指令执行引发冲突，根据造成的时间差恢复 load 地址。本文基于 $A_{\mathrm{lts}}^{\mathrm{MOB}}$ 设计对 OpenSSL 3.0.0 AES T 表实现的已知明文攻击，成功还原 AES 私钥。一次 AES 加密过程中会进行 36 次 T 表的查询，其中与 $A_{\mathrm{lts}}^{\mathrm{MOB}}$ 攻击相关的时间泄露对应其中一次查表。本文利用已知明文攻击对剩余 35 次查表时间引入的噪声进行过滤，加强有效泄露。在 Intel i5-9400 CPU 上进行的实验显示，已知明文分析方法对原有 $A_{\mathrm{lts}}^{\mathrm{MOB}}$ 的改进能有效提高分析成功率，当明文样本量达到 30 000 组时，能以 63.6%概率还原出一个密钥字节。

对现有针对微架构部件的防护方案分析发现，现有防护方法无法有效抵御本文提出的 MOB 攻击，因此提出微架构级别的可能防护构想，以对抗本文提出的 MOB 攻击。

参考文献：

[1] HENNESSY J L, PATTERSON D A. Computer architecture: a quantitative approach[M]. Fifth Edition. San Francisco: Margan Kaufmann, 2006.

[2] SCHAIK V S, MILBURN A, ÖSTERLUND S, et al. RIDL: rogue In-flight data load[C]//Proceedings of 2019 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2019: 88-105.

[3] ABU-GHAZALEH N, PONOMAREV D, EVTYUSHKIN D. How the Spectre and meltdown hacks really worked[J]. IEEE Spectrum, 2019, 56(3): 42-49.

[4] SULLIVAN D, ARIAS O, MEADE T, et al. Microarchitectural minefields: 4K-aliasing covert channel and multi-tenant detection in IaaS clouds[C]//Proceedings of 2018 Network and Distributed System Security Symposium. Virginia: the Internet Society, 2018: 1-14.

[5] MOGHIMI A, WICHELMANN J, EISENBARTH T, et al. MemJam: a false dependency attack against constant-time crypto implementations[J]. International Journal of Parallel Programming, 2019, 47(4): 538-570.

[6] ISLAM S, MOGHIMI A, BRUHNS I, et al. SPOILER: speculative load hazards boost rowhammer and cache attacks[C]//Proceedings of the 28th USENIX Conference on Security Symposium. New York: ACM Press, 2019: 621-637.

[7] YAROM Y, FALKNER K. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack[C]//Proceedings of the 23rd USENIX Conference on Security Symposium. New York: ACM Press, 2014: 719-732.

[8] KOCHER P, HORN J, FOGH A, et al. Spectre attacks: exploiting speculative execution[C]//Proceedings of 2019 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2019: 1-19.

[9] LIPP M, SCHWARZ M, GRUSS D, et al. Meltdown: reading kernel memory from user space[J]. Communications of the ACM, 2020, 63(6): 46-56.

[10] OSVIK D A, SHAMIR A, TROMER E. Cache attacks and countermeasures: the case of AES[C]//Proceedings of the Cryptographers' Track at the RSA Conference on Topics in Cryptology. New York: ACM Press, 2006: 1-20.

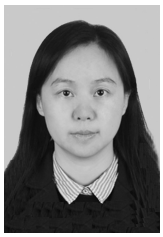
[11] GRUSS D, MAURICE C, WAGNER K, et al. Flush+Flush: a fast and stealthy cache attack[M]. Cham: Springer International Publishing, 2016.

[12] CANELLA C, GENKIN D, GINER L, et al. Fallout: leaking data on meltdown-resistant CPUs[C]//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM Press, 2019: 769-784.

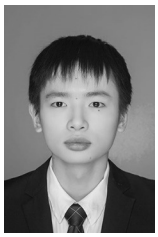
[13] YAN M J, CHOI J, SKARLATOS D, et al. InvisiSpec: making speculative execution invisible in the cache hierarchy[C]//Proceedings of 51st Annual IEEE/ACM International Symposium on Microarchitec-

ture. Piscataway: IEEE Press, 2018: 428-441.
[14] BENGER N M, POL J V D, SMART N P, et al. “Ooh aah... just a little bit”: a small amount of side channel can go a long way[C]// International Workshop on Cryptographic Hardware and Embedded Systems. Berlin: Springer, 2014: 75-92.
[15] SCHWARZ M, CANELLA C, GINER L, et al. Store-to-leak forwarding: leaking data on meltdown-resistant CPUs (updated and extended version)[J]. arXiv Preprint, arXiv: 1905.05725, 2019.
[16] KURTH M, GRAS B, ANDRIESSE D, et al. NetCAT: practical cache attacks from the network[C]//Proceedings of 2020 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2020: 20-38.
[17] RAMANATHAN R M, CURRY R, CHENNUPATY S, et al. Extending the world’s most popular processor architecture[J]. Intel Whitepaper, 2006, 1(1): 2-10.
[18] BERNSTEIN D J, BREITNER J, GENKIN D, et al. Sliding right into disaster: left-to-right sliding windows leak[C]//International Conference on Cryptographic Hardware and Embedded Systems. Berlin: Springer, 2017: 555-576.
[19] LOU X X, ZHANG T W, JIANG J, et al. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography[J]. ACM Computing Surveys, 2022, 54(6): 1-37.
[20] XIONG W J, SZEFER J. Leaking information through cache LRU states[C]//Proceedings of 2020 IEEE International Symposium on High Performance Computer Architecture. Piscataway: IEEE Press, 2020: 139-152.
[21] SCHAIK S V, MINKIN M, KWONG A, et al. CacheOut: leaking data on Intel CPUs via cache evictions[C]//Proceedings of 2021 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2021: 339-354.
[22] PURNAL A, GINER L, GRUSS D, et al. Systematic analysis of randomization-based protected cache architectures[C]//Proceedings of 2021 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2021: 987-1002.
[23] YAN M, FLETCHER C W, TORRELLAS J. Cache telepathy: leveraging shared resource attacks to learn {DNN} architectures[C]//Proceedings of 29th USENIX Security Symposium. Berkeley: USENIX Association, 2020: 2003-2020.
[24] ABRAMSON J M, AKKARY H, GLEW A F, et al. Method and apparatus for performing a store operation: US6378062[P]. [2002-04-23].
[25] ABRAMSON J M, AKKARY H, GLEW A F, et al. Method and apparatus for dispatching and executing a load operation to memory: US5717882[P]. [1998-02-10].
[26] BARBERIS E, FRIGO P, MUENCH M, et al. Branch history injection: on the effectiveness of hardware mitigations against Spectre-v2 attacks[C]//Proceedings of 31st USENIX Security Symposium. Berkeley: USENIX Association, 2022: 971-988.
[27] OLEKSENKO O, TRACH B, REIHER T, et al. You shall not bypass: employing data dependencies to prevent bounds check bypass[J]. arXiv Preprint, arXiv: 1805.08506, 2018.
[28] OLEKSENKO O, TRACH B, SILBERSTEIN M, et al. SpecFuzz: bringing Spectre-type vulnerabilities to the surface[C]//Proceedings of the 29th USENIX Conference on Security Symposium. Berkeley: USENIX Association, 2020: 1481-1498.
[29] WANG G H, CHATTOPADHYAY S, BISWAS A K, et al. KLEESpectre: detecting information leakage through speculative cache attacks via symbolic execution[J]. ACM Transactions on Software Engineering and Methodology, 2020, 29(3): 1-31.
[30] YU J Y, YAN M J, KHYZHA A, et al. Speculative taint tracking (STT): a comprehensive protection for speculatively accessed data[J]. IEEE Micro, 2020, 40(3): 81-90.
[31] FUSTOS J, FARSHCHI F, YUN H. SpectreGuard: an efficient data-centric defense mechanism against Spectre attacks[C]//Proceedings of 2019 56th ACM/IEEE Design Automation Conference. Piscataway: IEEE Press, 2019: 1-6.
[32] KELSEY K, BAI T X, DING C, et al. Fast track: a software system for speculative program optimization[C]//Proceedings of 2009 International Symposium on Code Generation and Optimization. Piscataway: IEEE Press, 2009: 157-168.
[33] MCILROY R, SEVCIK J, TEBBI T, et al. Spectre is here to stay: an analysis of side-channels and speculative execution[J]. arXiv Preprint, arXiv:1902.05178, 2019.
[34] TARAM M, VENKAT A, TULLSEN D. Context-sensitive fencing: securing speculative execution via microcode customization[C]//Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM Press, 2019: 395-410.
[35] WEISSE O, NEAL I, LOUGHLIN K, et al. NDA: preventing speculative execution attacks at their source[C]//Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. New York: ACM Press, 2019: 572-586.
[36] SUN K, BRANCO R, HU K. A new memory type against speculative side channel attacks[J]. Intel-Strategic Offensive Research & Mitigations, 2019, 1(1): 2-16.
[37] AINSWORTH S, JONES T M. MuonTrap: preventing cross-domain Spectre-like attacks by capturing speculative state[C]//Proceedings of 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture. Piscataway: IEEE Press, 2020: 132-144.
[38] SAILESHWAR G, QURESHI M K. CleanupSpec: an “undo” approach to safe speculation[C]//Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. New York: ACM Press, 2019: 73-86.
[39] SCHWARZ M, LIPP M, GRUSS D. JavaScript zero: real JavaScript and zero side-channel attacks[C]//Proceedings 2018 Network and Distributed System Security Symposium. Virginia: the Internet Society, 2018: 1-12.

[作者简介]



唐明（1976− ），女，湖北武汉人，博士，武汉大学教授、博士生导师，主要研究方向为信息安全、密码学和密码芯片等。



胡一凡（1998− ），男，浙江衢州人，武汉大学硕士生，主要研究方向为体系结构安全、密码学等。