arXiv:2301.10618v1 [cs.CR] 25 Jan 2023

Clueless：一种用于刻画值作为地址泄漏的工具

Xiaoyue Chen 乌普
萨拉大学 瑞典乌普萨拉
xiaoyue.chen
@it.uu.se

Pavlos Aimoniotis 乌普
萨拉大学 瑞典乌普萨拉
pavlos.ai
moniotis@it.uu.se

Stefanos Kaxiras 乌普
萨拉大学 瑞典乌普萨拉
stefanos.kaxiras
@it.uu.se

摘要

Clueless 是一种二进制插桩工具，用于刻画程序中显式的缓存侧信道漏洞。它通过跟踪动态指令依赖，检测数据值向地址的转换。如果发现内存中的数据值被用于地址计算，以进一步访问其他数据，Clueless 就会对这些值进行标记。

Clueless 可以报告执行过程中各个时刻被用作地址的数据量。还可以专门指示它跟踪内存中的特定数据（例如密码），以查看这些数据是否会在执行过程中的某个时刻转化为地址。如果发生这种情况，它会返回一份跟踪记录，说明被跟踪的数据如何转化为地址。

我们在 SPEC 2006 上展示了 Clueless，并首次分析了这些程序中转化为地址的数据值的数量。我们还在一项微基准测试和一个案例研究中展示了 Clueless。该案例研究考察了 OpenSSL 中 AES 的不同实现：T表、向量置换高级加密标准（VPAES）和英特尔高级加密标准新指令（AES-NI）。Clueless 展示了加密密钥如何在 T表实现中转化为地址，而在其他实现中则未检测到明显的缓存侧信道漏洞。

CCS概念

• **安全与隐私** → 侧信道分析与对策；信息流控制。

1 引言

缓存侧信道攻击通过一种微架构隐蔽信道——缓存——泄漏信息。通过观察共享缓存状态的变化，间谍进程可以绕过进程隔离，从受害进程中读取秘密数据。缓存侧信道攻击已在不同架构的处理器上，以及针对不同算法（例如 RSA [1], 高级加密标准 [2–5], 和 ElGamal [6]）得到演示。推测执行侧信道攻击，例如 Spectre [7],Meltdown [8] 及其变体 [9–14]，极大地改变了架构领域对安全性的看法。这些攻击利用本应被取消的推测执行指令（例如分支预测错误中的指令）访问秘密数据，并通过共享缓存传输出去。非推测执行的缓存侧信道攻击通常可以通过改进易受攻击算法的实现来缓解（例如，避免使用秘密数据查找大型查找表）；而这类攻击的推测执行变体则难以通过更改软件实现来防范，因为信息泄漏发生在推测执行过程中。

图1展示了 Spectre变体1，其中攻击者可以利用分支预测错误访问任意程序数据并通过共享缓存传输机密信息 [7]。受害程序已正确实现适当的边界检查，但由于推测执行，仍然容易受到攻击。

推测侧信道攻击已被证实构成巨大的安全威胁。人们提出了多种硬件方案来防范这类攻击。例如，InvisiSpec [15] 和 Ghost-Min-ion [16]通过使用额外的推测缓冲区，使推测执行在数据缓存层次结构中不可见，从而防止机密信息通过缓存通道泄露。未命中时延迟（DoM） [17]会延迟所有在数据缓存中未命中的推测加载，从而避免产生可观察到的时序差异；而推测污点跟踪（STT） [18] 则专注于阻止传输指令。STT 使用动态信息流跟踪（DIFT）对秘密数据进行污点标记。如果推测执行指令的结果不会通过任何潜在的隐蔽通道泄露机密信息，STT 就允许转发这些结果。

本工作并未提出针对推测侧信道攻击的新缓解方法。相反，我们旨在从一个全新的视角了解这类漏洞在程序中的普遍程度。侧信道攻击依赖一项基本的编程特性来泄露机密信息的值——将数据值转换为内存地址。除了图 1 中的受害程序之外，还可以考虑排序、哈希或许多其他根据数据值生成地址的算法。虽然我们了解了以地址形式泄露数据的机制，但尚不清楚这一问题在我们的工作负载中有多严重：在给定的应用程序中，有多少值会以地址的形式“泄露”?

本研究旨在揭示我们面临这种潜在漏洞的程度。*Clueless* 是一款基于二进制重写的工具，可追踪动态指令依赖；当它发现内存中的数据值被用于地址计算，以进一步访问其他数据时，便会为这些值添加标记。

Clueless 可通过两种模式使用：聚合模式，用于报告执行过程中各个时刻被用作地址的数据量；以及跟踪模式，在此模式下，工具会根据要求跟踪内存中的特定数据（例如密码），以查看这些数据在执行过程中是否会在任何时刻被转换为地址。若跟踪的数据确实被转换为地址，跟踪模式会返回一份记录，说明这些数据是如何被转换为地址的。

```
1 uint8 A[10];
2 uint8   B[256*64];
3 void 受害者（大小_t 地址）{
4   如果(地址 < 10) {     // 预测错误的分支
5     uint8 val   = A[addr]; // 访问机密信息
6     ...   = B[64*val]; // 传递机密信息
7   }
8 }
```

图 1：Spectre变体1。

```
机密信息 =*addr;
if (机密信息 % 2)
  a = A[0];
else
  a = A[128];
```

(a) 控制流

```
机密信息 =*addr;
i += 64 * 机密信息;
a = A[i];
```

（b）数据流

图 2：泄漏机密信息的代码。

```
r1 = addrX
r2 = addrY
load rX <- (r1)
load rY <- (r2)
r3 = rX * 64
r4 = r3 + rY
加载 r5 <- (r4)
```

图 3： addrX 和 addrY 为泄漏点的指令。

我们在 SPEC 2006 上以聚合模式演示了 Clueless，并首次分析了这些程序中被转换为地址的数据值的数量。我们还通过一项微基准测试和一项案例研究演示了 Clueless 的跟踪模式。该案例研究考察了 OpenSSL 中 AES 的不同实现：T表、向量置换高级加密标准（VPAES）以及英特尔高级加密标准新指令（AES-NI）。T表 AES 实现很容易受到缓存侧信道攻击（例如 Prime+Probe），而 VPAES 和 AES-NI 则不受缓存时序攻击影响。Clueless 能够清晰地展示加密密钥在 T表实现中如何被转换为地址，以及其他两种实现中缺少相应转换的情况。

2 方法

Clueless 是一种动态插桩工具，可在运行时分析指令，以跟踪作为内存地址泄漏的值。值是指不应直接或间接用作内存地址的数据，例如密码哈希、私有加密密钥。当值流向内存地址时，该值就可能作为内存地址泄漏。该工具的检测范围仅限于数据流：它会跟踪数据依赖，但忽略控制依赖。换言之，Clueless 能够检测显式通道 [18],，即在加载指令中用作地址的值，但无法检测隐式通道 [18],，即值通过控制流交互而泄漏的情况。假设机密信息是一个值，工具无法检测到图 2a 中代码的泄漏，因为&A[0] 和&A[128] 仅与机密信息存在控制依赖。另一方面，工具能够检测到图 2b 中的泄漏，因为机密信息参与了&A[i]的计算。此外，addr 将被标记为泄漏点。泄漏点是存放泄漏值的内存位置。

Clueless 使用一种基于动态信息流跟踪（DIFT）的算法 [19–21]。DIFT 已成功应用于防止针对软件的攻击 [20–25]，也见于针对推测执行攻击的硬件保护方案 [18]。DIFT 通过将污点与数据关联，并根据数据流传播污点来跟踪信息流。此外，Clueless 的算法还需要自动为数据分配污点并维护这些污点。

2.1 污点分配

每当从某个内存位置加载值（即不应被用于寻址内存的数据）时，都会为该内存位置分配一个新的污点。每个污点都与一个值的地址相关联。在图3的示例中，假设值位于内存位置addrX 和addrY，那么当第3行的加载指令执行时，会为 t_x 分配一个新的污点addrX 当第3行的加载指令执行时，随后会分配另一个污点t_y；当第4行执行时，则会分配 addrY 。

Clueless 需要知道加载的数据是否为值。大多数当代指令集架构（ISA）不会区分值 加载与地址 加载，也不会区分值 寄存器与地址 寄存器。作为一种二进制插桩工具，Clueless 无法判断哪些加载操作实际加载的是值。Clueless 为这个问题提供了两种解决方案。

一切皆为值。 一种解决方案是将内存中的所有数据最初都视为值，也就是说，Clueless 假设内存中没有任何内容应被用作内存地址。对于每条加载指令，都会为该加载操作的内存地址分配一个新的污点。因此，所有包含内存地址的内存位置都会被视为泄漏点。Clueless 实际上提供了一种方法，可根据过去的执行情况将内存中的任意数据归类为内存地址或非地址。这为分析程序提供了一个新的视角：进程内存中有多少内容可能通过缓存侧信道被另一个进程观察到？我们将这一模型称为聚合模式。聚合模式表示程序内存的可见程度——第 3 节将介绍其结果。

用户设置监视点。另一种方案是让用户标记出包含值的内存区域。只有当加载操作的地址位于已标记的内存区域内时，才会为该地址分配新的污点。Clueless 通过提供应用程序接口，支持动态注册和注销要监视的内存区域。为此，用户需要修改经插桩程序的源代码，插入 Clueless 监视点 API 调用。我们将此模型称为跟踪模式。第 4 节将介绍该模式的结果。

2.2 污点传播

Clueless 使用位数组存储污点集合，其中每个位代表一种不同的污点。采用这种表示方式时，集合并运算等同于按位或运算，执行效率很高。每个位数组都与一个寄存器或内存位置相关联。位数组中的位数有限，可在编译工具时进行配置。因此，污点的最大数量等于位数组中的位数。

在指令级，数据流可分为两类：寄存器-寄存器流和寄存器-内存流。在依赖跟踪的语境下，这两类流的主要区别之一是，寄存器-寄存器流跟踪所需的空间上限由体系结构寄存器的数量决定，而寄存器-内存流跟踪所需的空间上限则由虚拟内存位置的数量决定。例如，一对加载和存储指令（两者都会产生寄存器-内存流）可以在整个虚拟内存中复制数据，导致每个内存

表1：污点传播示例

指令	rX	rY	r3	r4	备注
r1 = addrX	{}	{}	{}	{}	
r2 = addrY	{}	{}	{}	{}	
加载 rX <- (r1)	$\{t_x\}$	{}	{}	{}	t_x 与 addrX 相关联
加载 rY <- (r2)	$\{t_x\}$	$\{t_y\}$	{}	{}	t_y 与 addrY 相关联
r3 = rX * 64	$\{t_x\}$	$\{t_y\}$	$\{t_x\}$	{}	
r4 = r3 + rY	$\{t_x\}$	$\{t_y\}$	$\{t_x\}$	$\{t_x,\ t_y\}$	
加载 r5 <- (r4)	{}	{}	{}	{}	addrX、addrY 已标记

位置都被数据的污点所污染，因此需要大量空间来存储污点集合。若跟踪的数据只有少量几项，这或许不是问题，因为这些数据不太可能流经内存的很大一部分。然而，当跟踪点数量很大时，空间开销会使完整跟踪寄存器-内存流变得不切实际。Clueless 在聚合模式下正是如此——它将内存中的所有内容都视为值，并跟踪整个内存。另一方面，为每个寄存器存储一个污点集所需的空间要少得多，因为体系结构寄存器的数量很少。

通过寄存器跟踪依赖关系。 Clueless 通过检查指令来跟踪寄存器-寄存器流，识别源寄存器、目标寄存器和内存寻址寄存器，并遵循传播规则。表1演示了执行图3中的指令时污点如何通过寄存器传播。传播规则如下：

(1) 对于每条加载指令，其目标寄存器的污点集会变为单元素集合或空集。如果加载的是一个值，污点集就是一个单元素集合，其中的元素是与该值的地址相关联的新污点。如果加载的不是值，污点集就是空集。(2) 对于将目标寄存器设为常量的指令（例如，两个相同源寄存器之间的异或，将常量移动到寄存器），其目标寄存器的污点集会变为空集。(3) 对于源操作数和目标操作数均为寄存器的指令（规则 2 中的指令除外），其目标寄存器的污点集会变为其源寄存器污点集的并集。(4) 对于加载和存储指令，内存寻址寄存器的污点集会被清空。所有与被清除的污点相关联的内存地址都会被标记为泄漏点。(5) 对于存储指令，如果所有内存寻址寄存器的污点集均为空集，则该地址不再是泄漏点，并会取消标记。

将依赖跟踪扩展到内存。 仅跟踪寄存器-寄存器流时，数据的污点集合可能会丢失，因为程序通常会将某些数据存入内存，用包含这些数据的寄存器执行其他操作，之后再从内存中重新加载这些数据。此类情况需要跟踪寄存器-内存流，以存储和重新加载污点集合。为将依赖跟踪扩展到内存，我们引入了另外两条传播规则：

(6) 对于每条存储指令，内存地址的污点集变为包含所存储数据的源寄存器的污点集。(7) 对于每条加载指令，除规则 1 外，目标寄存器的污点集变为规则 1 得出的污点集与内存地址的污点集的并集。

由于空间需求，使用完整方法追踪所有寄存器-内存流并不切实际，但追踪这些流仍然很重要，因为将数据临时存储到内存中非常常见。因此，采用组相联缓存作为尽力而为的方法，用于存储与内存地址关联的污点集合。缓存采用先进先出替换策略。编译该工具时，可以配置缓存的组数以及相联度。

2.3 污点维护

由于使用了静态大小的位数组作为污点集，Clueless 中的污点数量有限。因此，必须对污点进行维护和复用。只有当某个污点不属于任何污点集时，才能复用该污点。传播规则1、2和4可以清空污点集，从而使污点可供复用。根据传播规则4，与已清空污点关联的地址已被标记为泄漏点，因此这些已清空的污点不再包含有用信息，可以从所有污点集中移除，从而立即变为可复用状态。

尽管采用了回收机制，污点仍可能耗尽。例如，程序可以通过循环从内存中加载大量值并将它们相加。在这种情况下，Clueless 会从所有污点集中移除最早一次加载所分配的污点，使其可供使用。

2.4 局限性

不跟踪推测执行。Clueless 是一种二进制插桩工具。它不是硬件模拟器，也无法获取诸如推测执行的指令等微体系结构信息。因此，Clueless 无法跟踪推测执行。

跟踪不完整。 Clueless 是一种特征分析工具，而非验证工具。Clueless 的跟踪并不完整。Clueless 能够在有限窗口内跟踪数据依赖。这种不完整性是由我们的实现所导致的，该实现使用数量有限的污点和容量有限的缓存。编译该工具时，用户可以调整这些参数，以找到所需的跟踪窗口大小。

依赖项。 Clueless 依赖于 Intel Pin [26]。Clueless 被编译为共享库，需要由 Intel Pin 加载。Clueless 的传播算法以平台无关的方式实现，但 Intel Pin 仅支持对 IA-32、x86-64 和 MIC 指令集架构进行插桩。因此，Clueless 目前仅支持这些指令集架构。

2.5 源代码

Clueless 的源代码依据 GNU 通用公共许可证第 3 版发布。其 Git 仓库可通过 https: //github.com/xiaoyuechen/dift-addr.git 访问。

3 聚合模式

在聚合模式下，Clueless 将经过插桩的程序内存中的所有内容都视为值。在此模式下，Clueless 会将数据转换为地址的所有内存位置标记为泄漏点。此外，Clueless 会收集程序使用的所有内存地址集合，即在任何内存访问指令中使用的地址。借助泄漏点集合和所有内存地址集合，我们可以引入一个度量指标，用于描述程序在一次给定执行中被用作地址的数据所占的比例。

3.1 Λ度量指标

令 L_i 为泄漏点集合， A_i 为程序执行第 i 条指令后所有地址的集合（显然，$L_i \subseteq A_i$）。令 n 为整个程序执行过程中的指令总数，度量指标 Λ定义为

$$\Lambda = \frac{\sum_{i=1}^{n} |L_i|}{\sum_{i=1}^{n} |A_i|}$$

表示整个程序执行期间，转换为地址的数据平均占比。形象地说， Λ是图 5 中 $|L_i|$曲线下方的面积除以 $|A_i|$曲线下方的面积。

3.2 Λ个 SPEC 基准测试

我们在 SPEC 2006 上使用 Clueless 的聚合模式，通过分析 $|A_i|$ 和 $|L_i|$的变化，并比较不同基准测试的 Λ值，来表征数据到内存地址的转换。由于 Clueless 使用不完整的方法跟踪 L_i，而对 A_i 的跟踪是完整的，因此报告的 Λ 值是实际 Λ的下界。

数据到地址的转换普遍程度由 Λ表示，是程序的固有属性。图 4 展示了不同基准测试程序的 Λ 值。astar 程序和 soplex 程序将超过三分之一的内存用于存储地址。另一方面，在 bwaves 程序和 sjeng 程序中，这类转换则很少出现。

3.3 深入观察

为了进一步了解数据向地址的转换情况，我们进一步研究了一些基准程序在执行的每个时刻有多少数据被转换为地址。

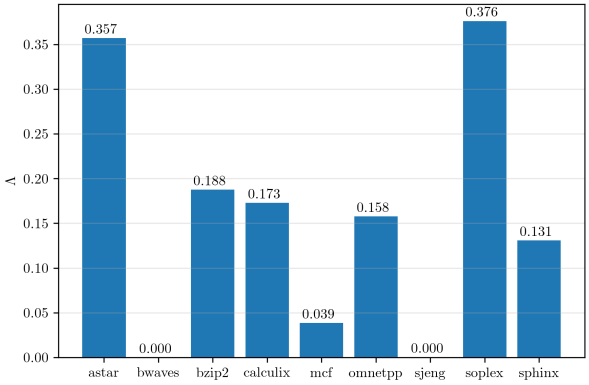


图4： Λ个 SPEC 基准测试。

Astar。 图5a 展示了在 astar 程序的一次执行过程中，$|A_i|$和 $|L_i|$如何变化。请注意， $|A_i|$单调递增，而 $|L_i|$则不然。当第 $i+1$条指令写入地址 $l \in L_i$时， $L_{i+1} = L_i \setminus \{l\}$。当 l是栈地址时，这种情况很常见，因为栈内存往往会被频繁重写。在 astar 程序中，当 $i \in [2\times10^{11}, 3\times10^{11}]$时， $|L_i|$ 会波动。这种波动的原因是，包含地址的同一批内存块被反复加载和写入。

Bzip2。 图5b显示，bzip2程序会周期性地将新地址存储到相同的内存块中。在

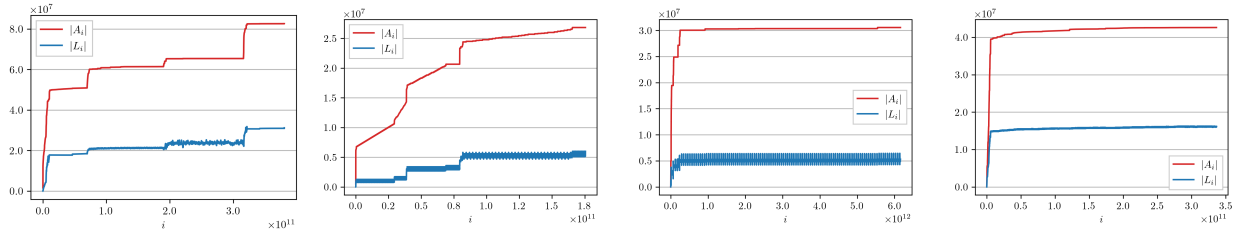
$i \in [0, 3.9\times10^{10}]$ $i \in [3.9\times10^{10}, 7.9\times10^{10}]$ $i \in [7.9\times10^{10}, 1.75\times10^{11}]$ 和 $|L_i|$ 时，可以观察到一种常见的数据地址转换模式—— $|L_i|$ 迅速增加，随后周期性波动；接着 $|L_i|$ 再次迅速但幅度较小地增加，并再次周期性波动。这种重复模式可能是因为 bzip2程序重新分配内存来存储内存地址，但在重新分配的内存上使用了相同的算法。

Calculix。 图5c显示，calculix程序具有明显的周期性内存访问模式。在$|A_i|$ 和 $|L_i|$初始增加后， $|A_i|$ 趋于稳定，而 $|L_i|$ 呈现周期性变化。 $|L_i|$ 的振幅相对较大，约为 2.1×10^6，这表明包含 2.1×10^6 地址的内存块会被反复写入新地址。

Soplex。 图5d展示了 soplex 程序的 $|A_i|$ 和 $|L_i|$ 如何变化。经过最初的增长后， $|A_i|$ 和 $|L_i|$ 都趋于稳定。这并不意味着该程序中的数据只会被转换为内存地址一次。数据被标记后，仍可能以不同方式多次转换为内存地址，但 L_i 会保持不变。 $|L_i|$ 保持稳定仅表明没有新的数据被标记，也没有已标记内存位置被写入。

4 跟踪模式

Clueless 在跟踪模式下允许用户动态注册和注销监视点，即包含值的内存块。如果任何监视点中的数据被转换为内存地址，Clueless 将针对每次泄漏提供详细诊断信息。诊断信息包括泄漏点、该泄漏点中的值转换成的内存地址、



(a) Astar (b) Bzip2 (c) Calculix (d) Soplex

图 5： $|A_i|$ **和** $|L_i|$ **基准程序。**

```
// 该常量 共享 库:
uint8 T[256 * 64];
int foo (常量 uint8 *s, size_t n) {
  for (size_t i = 0; i < n; ++i) {
    uint8 r = T[s[i] * 64];
    ...
  }
}

// 该 受害者 程序:
uint8 s[] = "cLUe";
foo (s, sizeof (s) - 1);
```

图6：微基准测试程序。

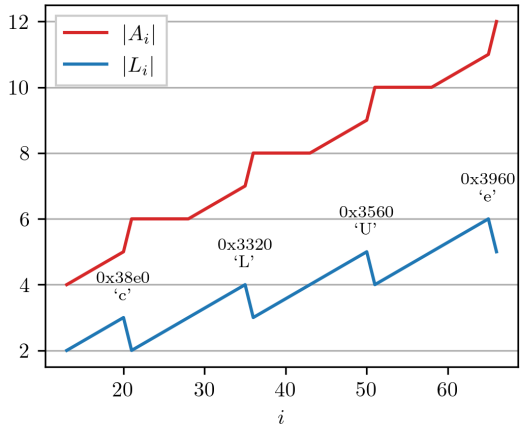


图7：微基准程序的 $|A_i|$ **和** $|L_i|$ **。**

显示值到地址转换过程的指令跟踪记录，以及发生泄漏的相关例程和映像名称。如果程序存在此类漏洞，此模式可用于测试程序的侧信道漏洞，并帮助了解机密信息在何处以及如何泄漏。

4.1 微基准测试

我们在图 6 中演示了 Clueless 在微基准程序上的跟踪模式。数组 `T[]` 和函数 `foo` 定义在受害程序链接的共享库中。受害程序将机密信息存储在 `s[]` 数组中。受害程序将机密信息作为参数传递给 `foo` 。函数 `foo` 加载其参数数组中的每个字节，将字节值乘以 64，并将结果用作常量数组 `T[]` 的索引进行查表。

该程序容易受到 Flush + Reload [1] 等侧信道攻击。攻击程序可以 `mmap` 共享库并刷新包含 `T[]`, 的缓存行，等待受害程序调用 `foo` 函数，然后测量重新加载这些缓存行所需的时间，以确定 `foo`访问了哪些缓存行。假设受害程序所在机器的缓存行大小为 64 字节，攻击者便可完全恢复机密信息——对 `T[s[i]*64]` 的每次访问都会落在不同的缓存行上，因此可以通过 `(l-T)/64` 计算出机密信息的每个字节，其中 `l` 是被访问缓存行的地址。由于 `T` 只是共享库中的一个符号，因此也很容易找到它的偏移量。在我们的示例中，`T` 的偏移量为 `0x2020`。

4.2 定位泄漏

Clueless 的聚合模式可用于分析微基准程序。图 7 展示了其 $|A_i|$和 $|L_i|$的变化情况。在 Clueless 处于跟踪模式时，我们可以精确定位哪一次增加的

```
0x7fff41801683 { 0 } ->%rax                    movzbl (%rax) ,%eax
%rax { 0 } ->%rax                              movsbl %al ,%eax
%rax { 0 } ->%rax                              shl $0x6 ,%eax
%rax { 0 } ->%rax                              cltq
%rdx %rax { 0 } ->%rdx                         add %rax ,%rdx
%rdx { 0 } ->%rcx                              mov %rdx ,%rcx
[%rcx { 0 } ] = 0x55baae46d8e0 ->              mov (%rcx) ,%rax
```

（a）传播 **(b) 指令**

图 8：追踪微基准测试。

$|L_i|$ 导致被跟踪的秘密泄漏。图 7 中的注释揭示了泄漏发生时泄漏的机密信息内容。例如，当 $i = 20$（即 `&T[s[0]*64]`）的计算结果为 `0x38e0` （减去映像加载偏移量后），并在一次加载中被用作内存地址时，`0x38e0` 就会泄漏。要还原 `s[0]`, ，我们计算 `(0x38e0-0x2020)/64` ，得到 `0x63`，它是字母 'c' 的 ASCII 码。

4.3 追踪变换

为了进一步了解这种泄漏，Clueless 会给出导致泄漏的传播轨迹。图 8a 展示了导致 `s[0]` 泄漏的传播轨迹部分，图 8b 则展示了相应的指令。

传播轨迹的语法如下：

- `{ NUM, .. }` 表示一个污点集，例如，`%rax { 0, 3 }` 表示寄存器 `rax` 的污点集为 $\{t_0, t_3\}$。
- `->` 表示数据流和污点传播，例如，`%rdx { 0} -> %rcx` 表示数据从寄存器 `rdx` 流向寄存器 `rcx`，且污点 t_0 传播到寄存器 `rcx`。
- `[ 寄存器 { NUM, .. } ]` 表示将寄存器用作内存地址，后接等号和有效地址，例如：`[ %rcx { 0 } ] = 0x55baae46d8e0 ->` 表示寄存器 `rcx` 的污点集为 t_0，并在加载时被用作内存地址。

通过分析微基准测试的跟踪记录，我们发现第 1 行的指令将 `&s[0]`（即污点集为 $\{t_0\}$ 的 `0x7fff41801683`）加载到寄存器 `rax`。接下来的 3 条指令将 $\{t_0\}$ 从寄存器 `rax` 传播到其自身。然后，第 5 行的指令通过将寄存器 `rax` 加到寄存器 `rdx` 上，将 $\{t_0\}$ 传播到该寄存器。随后，$\{t_0\}$ 进一步传播到寄存器 `rcx`，后者最终被用作内存地址 `0x55baae46d8e0`。

5 案例研究：高级加密标准

我们已经看到，第 4 节中的微基准测试可能会因值到地址的转换而泄露机密信息。出于相同的原因，高级加密标准 (AES) 的某些实现 [27] 容易受到缓存侧信道攻击。这些实现通常依赖大型查找表来加快加密过程 [28]。如果加密密钥被转换为大型查找表的索引以进行查表，攻击者就可能通过观察相应的缓存状态变化，部分或完全恢复密钥。过去已有大量针对 AES、利用此类漏洞的攻击被证明 [2–5]。此外，人们还提出了 AES 的不同实现方案，以防范此类攻击，同时保持或提升加密速度 [3, 28, 29]。

在本案例研究中，我们使用 Clueless 的跟踪模式，分析 OpenSSL 3.0.3 中的三种不同 AES 实现：T表、向量置换高级加密标准 (VPAES) 以及英特尔高级加密标准新指令 (AES-NI)。我们将扩展加密密钥设为监视点，以观察它是否会被转换为内存地址。

5.1 T表

OpenSSL 中的高级加密标准 T表实现使用了 9 个 T表，也就是预先计算好的查找表，其中 8 个各为 8kiB，另一个为 2kiB。加密密钥首先会扩展为轮密钥。第一轮密钥与 16 字节明文进行 异或 运算，生成初始状态向量。随后，状态向量中的元素被用作 T表索引，以查出相应的值；这些值再与下一轮密钥结合，生成下一个状态向量。这种实现很容易受到 Prime+Probe 攻击，仅需 300 次加密即可完全恢复 128 位加密密钥 [3]。

Clueless 检测到潜在泄漏，并将密钥的所有字节标记为泄漏点。此外，Clueless 会提供一条传播轨迹，展示密钥如何转换为内存地址。图 9a 显示了加密第一轮的传播轨迹（所有轮次都类似），图 9b 则显示了对应的指令。跟踪记录显示，第一轮轮密钥的泄漏过程如下：

```
0x7fff609ec660 { 1 } ->%xmm5
%xmm2 %xmm5 { 1 } ->%xmm2
%xmm0 %xmm2 { 1 } ->%xmm0
%xmm1 %xmm0 { 1 } ->%xmm1
%xmm1 { 1 } ->%xmm1
%xmm0 { 1 } ->%xmm0
%xmm5 %xmm0 { 1 } ->%xmm5
%xmm0 %xmm1 { 1 } ->%xmm0
%xmm3 %xmm1 { 1 } ->%xmm3
%xmm3 %xmm5 { 1 } ->%xmm3
%xmm4 %xmm0 { 1 } ->%xmm4
%xmm4 %xmm5 { 1 } ->%xmm4
%xmm2 %xmm3 { 1 } ->%xmm2
%xmm2 %xmm0 { 1 } ->%xmm2
%xmm3 %xmm4 { 1 } ->%xmm3
. . .
%xmm3 %xmm1 { 1 2 ... 10 } ->%xmm3
%xmm4 %xmm2 { 1 2 ... 10 } ->%xmm4
%xmm4 %xmm5 { 1 2 ... 11 } ->%xmm4
%xmm0 %xmm3 { 1 2 ... 10 } ->%xmm0
%xmm0 %xmm4 { 1 2 ... 11 } ->%xmm0
%xmm0 { 1 2 ... 11 } ->%xmm0
```

图 10：跟踪 VPAES 的实现。

（1）第 1 行至第 4 行的指令加载第一轮轮密钥，并将其与寄存器 `rax`、`rbx`、`rcx` 和 `rax` 中的明文进行异或，从而构成初始状态向量。监视的内存位置会被赋予不同的污点，并传播到目标寄存器。（2）第 5 行至第 8 行的指令从初始状态向量中提取一些字节，并将它们复制到另一组寄存器（`r10`、`r11`、`r12` 和`r8`）。污点从源寄存器传播到目标寄存器。

(3) 第 13 至 16 行的指令使用提取出的字节在其中一个 T表中进行查表。受污点影响的寄存器用于构成有效地址。此时，Clueless 将第一轮密钥所在的内存位置标记为泄漏点。

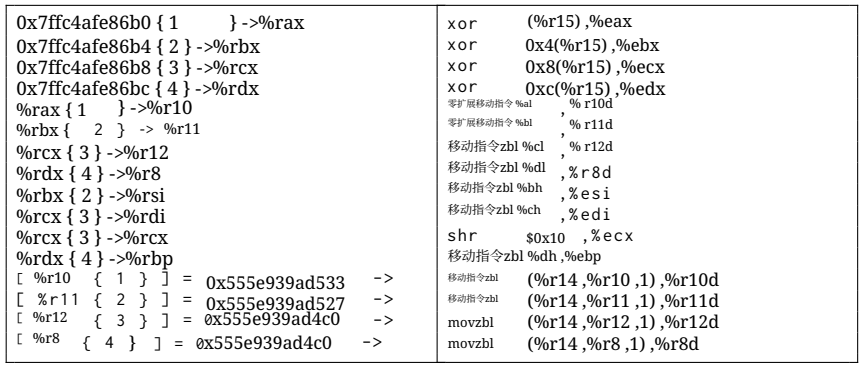
5.2 VPAES

VPAES 是一种利用向量置换指令加速高级加密标准的技术。它避免了依赖密钥的内存引用，因此能够免受已知的缓存时序攻击 [28]。硬件必须支持向量置换指令才能使用 VPAES。

Clueless 在跟踪模式下显示，OpenSSL 的 VPAES 实现中没有任何密钥部分会作为内存地址泄漏。图 10 中报告的传播轨迹表明，密钥会被加载（例如，通过 `86e1` 处的指令加载），并在不同轮次中参与运算，但密钥的任何部分都不会用于引用内存。

5.3 AES-NI

英特尔于 2010年推出了 AES-NI 指令集，为 AES 提供直接的硬件支持。这些新指令的运行时间与数据无关，也不使用查找表 [29]。AES-NI 可通过单条指令完成整个轮次的加密。正确使用 AES-NI 的 AES 实现应能免受缓存侧信道攻击，因为这些指令不会涉及缓存。



(a) 传播 **(b) 指令**

图 9：跟踪 T表实现。

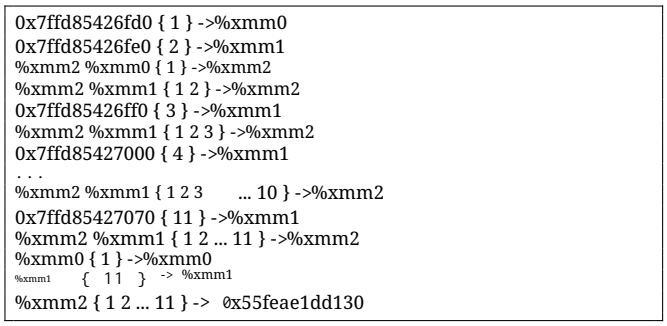


图 11：跟踪 AES-NI 实现。

尚未发现 OpenSSL 中使用 AES-NI 的 AES 实现会将密钥转换为内存地址。图 11 中的传播跟踪记录表明，密钥的任何部分都不会用于访问内存。

6 结论与未来工作

我们介绍了 Clueless：一种用于表征以地址形式泄漏的值的工具。使用 Clueless 的聚合模式，我们首次表征了 SPEC 2006 基准程序中转化为内存地址的数据值数量。一些基准程序使用超过三分之一的已访问内存来引用内存。Clueless 的跟踪模式提供了机密信息在微基准测试和 OpenSSL 中的 AES 实现中传播和泄漏的跟踪记录。AES 的 T表实现存在遭受缓存侧信道攻击的潜在漏洞，而 VPAES 和 AES-NI 实现则不受此类攻击影响。

Clueless 报告的“泄漏”仍需进一步研究。我们希望从误报中识别出可能导致敏感信息泄漏风险的数据到地址的转换（例如，机密信息转换为位于同一缓存行中的地址）。我们有兴趣在硬件模型上应用类似的动态信息流跟踪技术，以缓解 Spectre 等缓存侧信道攻击。

某些程序中数据到地址的转换频率较高，这也表明缓存系统存在优化机会：转换为内存地址的数据可能与这些转换后的地址所指向的数据相关联。这可能是我们未来工作的一个重点。

致谢

本研究得到微软研究院欧洲、中东和非洲地区博士奖学金计划（资助编号：2021-020）、瑞典研究理事会（VR）（资助编号：2018-05254）、瑞典创新局（资助编号：2021-02422）、瑞典战略研究基金会（SSF）（资助编号：FUS21-0067）的资助，以及乌普萨拉大学网络安全专项经费的支持。

参考文献

[1] Y. Yarom 和 K. Falkner，“Flush+Reload：一种高分辨率、低噪声的 L3 缓存侧信道攻击”，载于 第 *23* 届 *USENIX* 安全研讨会（*USENIX Security 14*），2014年，第 719–732 页。[2] O. Acıiçmez、W. Schindler 和 Ç. K. Koç，“针对高级加密标准的基于缓存的远程定时攻击”，载于 *RSA* 大会密码学家专题。施普林格，2007年，第 271–286 页。[3] D. A. Osvik、A. Shamir 和 E. Tromer，“缓存攻击与对策：高级加密标准案例”，载于 *RSA* 大会密码学家专题。施普林格，2006年，第 1–20 页。[4] J. Bonneau 和 I. Mironov，“针对高级加密标准的缓存冲突定时攻击”，载于 国际密码硬件与嵌入式系统研讨会。施普林格，2006年，第 201–215 页。[5] D. Gullasch、E. Bangerter 和 S. Krenn，“缓存游戏——将基于访问的缓存攻击应用于高级加密标准”，载于 *2011* 年 *IEEE* 安全与隐私研讨会。IEEE，2011年，第 490–505 页。[6] F. Liu、Y. Yarom、Q. Ge、G. Heiser 和 R. B. Lee，“末级缓存侧信道攻击具有实际可行性”，载于 *2015* 年 *IEEE* 安全与隐私研讨会，2015年，第 605–622 页。[7] P. Kocher、J. Horn、A. Fogh、D. Genkin、D. Gruss、W. Haas、M. Hamburg、M. Lipp、S. Mangard、T. Prescher、M. Schwarz 和 Y. Yarom，“Spectre 攻击：利用推测执行”，载于 *SP*，2019年5月，第 1–19 页。[8] M. Lipp、M. Schwarz、D. Gruss、T. Prescher、W. Haas、A. Fogh、J. Horn、S. Mangard、P. Kocher、D. Genkin 等，“Meltdown：从用户空间读取内核内存”，载于 第 *27* 届 *USENIX* 安全研讨会（*USENIX Security 18*），2018年，第 973–990 页。[9] A. Bhattacharyya、A. Sandulescu、M. Neugschwandtner、A. Sorniotti、B. Falsafi、M. Payer 和 A. Kurmus，“SMoTherSpectre：通过端口争用利用推测执行”，载于 *ACM SIGSAC* 计算机与通信安全会议论文集。计算机协会，2019年11月，第 785–800 页。 [在线]。获取地址：https://doi.org/10.1145/3319535. 3363194[10] V. Kiriansky 和 C. Waldspurger，“推测性缓冲区溢出：攻击与防御”，*arXiv:1807.03757 [*计算机科学 *]*, 2018年7月。 [在线]。获取地址：http: //arxiv.org/abs/ 1807 03757 。jannh@google.com，“问题 1528：推测执行，变[11] 体 4：推测性存储绕过 - project-zero”，2018年2月。 [在线]。获取地址：https://bugs.chromium. org/p/project-zero/issues/detail?id=1528

[12] E. M. Koruyeh、K. N. Khasawneh、C. Song 和 N. Abu-Ghazaleh，“Spectre 卷土重来！利用返回栈缓冲区的推测执行攻击”，载于 第 *12* 届 *USENIX* 攻击性技术研讨会（*WOOT 18*），2018年。[13] G. Maisuradze 和 C. Rossow，“ret2spec：利用返回栈缓冲区进行推测执行”，载于 *2018* 年 *ACM SIGSAC* 计算机与通信安全会议论文集，2018年，第 2109–2122 页。[14] P. Aimoniotis、C. Sakalis、M. Själander 和 S. Kaxiras，“重排序缓冲区争用：针对推测不变指令的前向推测干扰攻击”，第 20 卷，第 162–165 页，2021年7月。[15] M. Yan、J. Choi、D. Skarlatos、A. Morrison、C. Fletcher 和 J. Torrellas，“InvisiSpec：使推测执行在缓存层次结构中不可见”，载于 *IEEE/ACM*微体系结构国际研讨会论文集，2018年10月，第 428–441 页。[16] S. Ainsworth，“GhostMinion：用于幽灵漏洞缓解的严格性排序缓存系统”，载于 *IEEE/ACM*微体系结构国际研讨会论文集。计算机协会，2021年10月，第 592–606 页。 [ ] 10 1145 3466752 3480074 在线。获取地址：
[17] https://doi.org/ . / . C. Sakalis、S. Kaxiras、A. Ros、A. Jimborean 和 M. Själander，“通过选择性延迟和值预测实现高效的不可见推测执行”，载于 计算机体系结构国际研讨会论文集，2019年6月，第 723–735 页。[18] J. Yu、M. Yan、A. Khyzha、A. Morrison、J. Torrellas 和 C. W. Fletcher，“推测污点跟踪（STT）：对推测访问数据的全面保护”，载于 *IEEE/ACM*微体系结构国际研讨会论文集。计算机协会，2019年10月，第 954–968 页。 [ ] 10 1145 3352460 3358274 在线。
[19] 获取地址：https://doi.org/ . / . D. E. Denning 和 P. J. Denning，“用于安全信息流的程序认证”，《*ACM* 通讯》，第 20 卷，第 7 期，第 504–513 页，1977年7月。 [在线]。获取地址：https://doi.org/10.1145/359636.359712

[20] G. E. Suh、J. W. Lee、D. Zhang 和 S. Devadas，《通过动态信息流跟踪实现安全程序执行》，*ACM SIGPLAN*通讯，第39卷，第11期，第85–96页，2004年。[21] J. Clause、W. Li 和 A. Orso，《Dytan：一种通用动态污点分析框架》，载于 *2007*年软件测试与分析国际研讨会论文集，2007年，第196–206页。[22] L. C. Lam 和 T.-c. Chiueh，《面向安全应用的通用动态信息流跟踪框架》，载于*2006*年第二十二届年度计算机安全应用大会（*ACSAC'06*）。IEEE，2006年，第463–472页。[23] V. Haldar、D. Chandra 和 M. Franz，《Java 的动态污点传播》，载于第二十一届年度计算机安全应用大会（*ACSAC'05*）。IEEE，2005年，第9页–第页。[24] J. Kong、C. C. Zou 和 H. Zhou，《通过运行时指令级污点检查提升软件安全性》，载于提升软件可靠性的体系结构与系统支持第一届研讨会论文集，2006年，第18–24页。[25] F. Qin、C. Wang、Z. Li、H.-s. Kim、Y. Zhou 和 Y. Wu，《LIFT：一种用于检测安全攻击的低开销实用信息流跟踪系统》，载于*2006*年第三十九届 *IEEE/ACM* 微体系结构国际研讨会（*MICRO'06*）。IEEE，2006年，第135–148页。[26]《Pin——一种动态二进制插桩工具》。 [在线]。网址：https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html [27] M. Dworkin、E. Barker、J. Nechvatal、J. Foti、L. Bassham、E. Roback 和 J. Dray，《高级加密标准（AES）》，2001-11-26 2001。[28] M. Hamburg，《利用向量置换指令加速 AES》，载于国际密码硬件与嵌入式系统研讨会。施普林格，2009年，第18–32页。[29] S. Gueron，《英特尔高级加密标准（AES）新指令集》，2010年。