# 实用的数据唯攻击生成

布赖恩·约翰内斯迈尔　　亚洲·斯洛文斯卡　赫伯特·博斯　　克里斯蒂亚诺·朱弗里达

阿姆斯特丹自由大学

## 摘要

由于越来越复杂的CFI解决方案，控制流劫持变得越来越困难，最近的研究转而专注于自动构建仅使用数据的攻击，通常使用符号执行、简化假设（这些假设并不总是与攻击者的目标相匹配）、手动小工具链或以上所有方法。 因此，这种方法的实际采用程度很低。在这项工作中，我们将不必要的复杂性抽象出来，而是采用一种轻量级的方法，针对那些对于分析来说最易于处理且对于攻击者来说最有前景的漏洞。

具体而言，我们提出了EINSTEIN，这是一个仅使用数据的攻击利用流水线，它使用动态污点分析策略来：(i)扫描易受攻击的系统调用链（例如，执行代码或破坏文件系统），以及(ii)为这些系统调用生成利用程序，这些利用程序使用未经修改的攻击者数据作为输入。 EINSTEIN在常见服务器应用程序中发现了数千个易受攻击的系统调用，远远超出了现有方法的范围。
此外，以 nginx为案例研究，我们使用 EINSTEIN生成了944个攻击，并讨论了其中两个绕过最先进的缓解措施的攻击。

## 1 引言

> “一切都应该尽可能简单，但不要过于简单。”
> - 阿尔伯特·爱因斯坦

自从三十多年前控制流劫持攻击出现以来[81]，攻击和防御一直在争夺程序的控制流的“控制权”。 此类攻击遵循图1a中列出的一般步骤：它们覆盖程序的某些控制*data*of；强制其执行攻击者指定的代码片段，即 *gadgets*；甚至可以将足够多的这些片段链接在一起以实现任意计算，即图灵完备性[19, 20, 24, 49, 71, 77]。 这演变成了一场攻击利用程序的某些控制流和防御限制该控制流之间的军备竞赛，直到控制流完整性（CFI）[11, 86, 87, 89, 96]的广泛部署，使得攻击程序的控制流变得越来越困难。

数据唯攻击的承诺。作为回应，攻击者转向了数据唯攻击，承诺利用程序的（庞大）数据流（即任何操作中使用的程序数据）。 然而，尽管这些攻击技术上利用了非控制数据，但其中许多仍然依赖于利用控制相邻数据（例如分支条件）来执行“合法”的控制流劫持。 实际上，它们仍然利用了程序控制流的（微小）剩余攻击面。 因此，为了从如此小的攻击面中识别出数据唯攻击，这些方法采用了重量级分析（例如符号执行）来推理大量复杂的约束条件。

然而，在现实中，对于一个足够复杂的程序，在有限的时间内解决所有约束条件是极其困难，甚至是不可能的。 问题空间——即覆盖任何可能的数据到任何可能的值，将其用于任何潜在的工具，从每个可能的程序点开始，以实现任意计算——太大了。 因此，为了扩展规模，这些方法被迫做出几个简化假设，例如：（1）攻击只能覆盖特定部分的内存，或者（2）攻击者必须使用另一个漏洞来达到工具的入口点，或者（3）攻击者必须手动将工具链在一起构建攻击。 虽然这些假设使得攻击生成成为一个可处理的问题，但不幸的是，它们要么极大地限制了生成的攻击的范围，要么将其部分留给未来或手动处理。

简而言之，这些方法的普遍复杂性阻碍了它们实现图1b中列出的所有目标，从而限制了它们的实际应用。

一种轻量级的方法。在本文中，我们研究了一种新颖的轻量级技术来构建数据唯攻击。 我们的方法依赖于四个关键见解，极大地减少了问题空间。 首先，类似于牛顿在控制流劫持相关领域中的方法[88]，我们观察到动态污点分析简化了数据唯攻击的许多复杂约束。 换句话说，我们不需要推理大量未知的事实，而只需推理具体数据在具体情况下的使用情况。

步骤**1**：利用内存写入漏洞进行利用。
步骤**2**：将执行引导到小工具的入口点。
步骤**3**：通过小工具链执行一些有效载荷。

(a) 任何非控制（或控制）数据攻击的步骤。

目标**1**：自动建模任意内存写入漏洞。
目标**2**：自动识别任意*g***a**dget入口点。
目标**3**：自动生成完整的*g***a**dget链。

(b) 自动化数据唯攻击的目标。

图1：数据唯攻击：步骤和目标。

程序执行。 其次，我们观察到表达能力（尤其是图灵完备性）并非必需。 相反，攻击者只是想要实现特定的目标，例如通过 execve运行代码，或通过 write写入文件。 第三，我们观察到操纵程序的预期路径并非必需。 实际上，程序将自行调用 execve、 write或其他有趣的系统调用，而无需攻击者的任何额外干扰。 最后，我们观察到在初始化之后，程序将许多类型的数据视为不可变的，因此该数据几乎没有（如果有的话）任何约束。

通过针对这些直接的"身份"数据流，我们可以在一个程序点上损坏数据（例如文件名），并期望它在传递到另一个程序点时保持不变（例如传递给 execve的文件名）- 而同时不执行任何繁重的约束求解。

利用这些见解，我们使问题变得可处理，即使对于轻量级分析也是如此。

我们提出了EINSTEIN，一个仅使用数据的攻击利用流水线。 EINSTEIN使用自定义污点策略（而不是约束求解）来识别攻击者可控制的系统调用（而不是图灵完备性），沿着程序的（已经有效的）运行路径生成针对系统调用参数的利用，这些参数具有身份数据流（而不是复杂的数据流）来自攻击者数据。 给定一个易受攻击的目标程序作为输入，它会对代码进行仪器化，以模拟和跟踪攻击的每一步，从而使其能够在运行时识别出小工具链。

具体来说，我们首先用一种独特的颜色对可能被攻击者覆盖的任何数据进行污染。 然后，我们跟踪污染物进入安全敏感的系统调用流程[29, 32, 40, 64, 87]。 此外，我们还跟踪污染物进入某些在系统调用之间共享的库状态。 如果这个状态可以被一个系统调用控制，并被另一个系统调用使用，那么我们有证据表明攻击者可以利用它来将两个本来安全的系统调用链接在一起，进行数据唯攻击。 最后，我们生成利用程序，用攻击者的数据破坏已识别的系统调用参数，并通过在目标应用程序上运行它们来确认利用程序的有效性。

爱因斯坦的低复杂性方法使其能够识别出数千个易受攻击的工具。 以流行的Web服务器nginx为案例研究，我们使用爱因斯坦生成了944个利用程序，包括1个代码执行原语，17个写入-何处原语和41个发送-何处原语。 此外，许多这些利用程序绕过了最先进的缓解措施，例如系统调用过滤[34, 35, 44, 67]和选择性DFI [44, 79]。

作为案例研究，我们讨论了CODE-EXECUTION漏洞，以及WRITE-WHAT-WHERE漏洞之一。 前者展示了仅数据攻击表面的丰富性提供了许多易于攻击者利用的低悬挂果实。 它利用了一个从全局变量直接获取其路径名和参数的execve系统调用，使攻击者可以轻松执行任意代码。 后者展示了系统调用链的威力，进一步扩大了可用的攻击表面。 它利用了open系统调用的路径名参数和write系统调用的fd和buf参数，使攻击者能够破坏服务器的文件系统。

贡献。我们做出以下贡献：

- 我们提出了一种在复杂程序中构建仅数据攻击的实用方法。
- 我们开发了EINSTEIN，一个开源的仅数据利用流水线。
- 我们评估了流行的服务器应用程序上的EINSTEIN，以识别数千个以前未知的易受攻击的系统调用，并以nginx为案例研究生成数百个可行的攻击。

## 2 背景和相关工作

### 2.1 数据唯攻击和防御

Young和McHugh在1987年提出了第一个数据唯攻击的例子[95]，随后的研究表明它们在实践中是可行的[30, 84, 91]。然后，Chen等人在2005年首次对现实世界的程序进行了广泛的研究[26]。

运行示例。图2概述了文献中描述的经典数据唯攻击之一[26]。 在这个例子中，服务器使用sort-scr**ipt**程序来对客户端指定的数字进行排序。 在良性情况下，客户端首先向服务器发送一个POST /sort-scr**ipt**请求，请求体中包含未排序的数字，服务器将其传递给CGI-BIN目录中的/sort-scr**ipt**。 最后，程序将排序后的数字返回给服务器，服务器将其传递给客户端。 然而，N**ull** HTTPD we b服务器中的内存写入漏洞允许攻击者覆盖CGI-BIN路径，例如将其设置为/b**in**。 当攻击者现在向服务器发送一个POST /sh请求时，它将执行请求体中包含的sh**ell**命令。

实用与全面的防御。对于这些攻击的缓解通常试图阻止两种操作之一：（1）恶意数据的定义，即内存中的恶意数据的定义（即**def**）

[1] E爱因斯坦可在https://github.com/vusec/einstein获得。

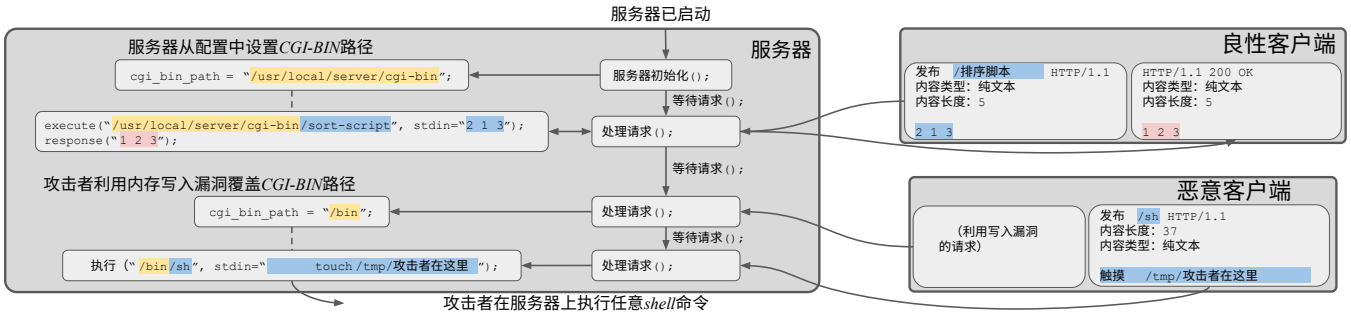



图2：Chen等人提出的仅数据攻击[26]（为了清晰起见进行了轻微修改）。 尽管EINSTEIN的发现几乎有两个十年之久，但它仍然能够识别出利用流行Web服务器的CGI-BIN路径的类似工具。

写漏洞；或（2） 使用恶意数据，例如，通过传递恶意数据给 execve。 不幸的是，全面的防止恶意数据的全面解决方案（例如，内存安全[13, 14, 22, 45, 56, 57, 94]，数据空间随机化[16, 17, 21, 68]）或防止其使用（例如，数据流完整性[23, 51, 80]）要么性能差，要么需要软件或硬件上繁琐的更改，从而限制了它们在实践中的应用。 另一方面，针对恶意数据的实用措施（例如，内存错误扫描[12, 36, 75, 76, 83]，选择性DSR [62, 63]）或针对其使用的措施（例如，选择性DFI [44, 79]，系统调用过滤[34, 35, 44, 67]）都不是全面的，因为它们使攻击表面的重要部分容易受到攻击。[2]

## 2.2 自动化数据唯攻击

尽管在近20年前发现了数据唯攻击（例如图2中的攻击），并且缺乏实用的、全面的缓解措施，但是构建攻击的自动化解决方案仍然存在一些限制：它们的范围有限，需要大量手动工作，或者基于不切实际的假设。 这些限制通常与潜在的复杂性和不良可扩展性有关，这是大多数解决方案的基础，但有时它们的目标也不同。 例如，一些方法追求图灵完备性，这在学术视角下很有趣，但对攻击者来说并不那么相关。 在本节的其余部分中，我们将详细介绍以前的方法存在的各种不足之处（也可参见表1）。

基于漏洞的分析。 虽然防止恶意数据的使用需要进行全面的攻击面分析：识别（和消除）任何危险数据流，无论其来源如何，但以前的方法[39, 41, 66]的范围更窄，仅限于特定的内存写入漏洞，而不是通用的原语。. 换句话说，它们生成的攻击仅适用于用户提供的特定漏洞，包括可能写入的特定地址和值。 部分例外是Limbo [73]，它模拟了任意的

[2]我们将感兴趣的读者引用到以前的数据攻击调查[27, 28]和内存破坏防御[85]。

任意堆栈溢出，尽管这还有很长的路要走一般的内存破坏。 相比之下，EINSTEIN从通用攻击者能力（例如，任意读/写原语）开始进行分析，而不考虑具体的漏洞。

**预定的小工具入口点。接下来，以前的应用程序-**方法可能需要用户预先定义小工具入口点[43, 66]。 换句话说，为了构建攻击，他们需要用户已经知道如何以将其执行引导到第一个小工具的方式来利用程序。 这是任何“奇怪的编译器”的典型模型，其目标是将图灵完备的小工具链连接在一起（例如，ROP编译器[72]，它们假设用户已经可以利用

一个返回指令），它也是仅数据小工具编译器的模型[43,66]。然而，识别入口点可能是一个非平凡的任务，因为它通常需要用户开发一个单独的漏洞利用程序，只是强制程序的执行达到第一个小工具（例如，通过“合法”地改变控制流）。以前的方法可能部分地模拟入口点，要么是不完整的（例如，由于对共同执行的覆盖率不足[39, 73]），要么是不可靠的（例如，假设在内存错误之后执行的任何循环都可能是小工具调度程序[41]）。

手动小工具链接。最后，以前的方法可能要求用户手动链接小工具[41]。 然而，链接通常是一个非平凡的任务，因为CFI限制了可以链接在一起的小工具，并且小工具可能是“易变的”（即，在链中选择一个小工具可能排除了该链中其他小工具的选择[43]）。 几种以前的方法根本不处理链接，而其他所有方法只处理部分链接（例如，由于共同执行的覆盖率不足[39, 73]，或者问题是NP-**hard**[43]）"。

## 3威胁模型

我们假设与之前的工作[39, 41, 43, 66, 73]大部分相同的威胁模型。 具体而言，我们考虑一个目标程序：(1)存在内存损坏漏洞[1, 3–6]；(2)受到DEP [15]、ASLR [65]、最先进的控制流劫持防御（例如CFI [11, 86, 87, 89, 96]）和强大的堆栈保护（例如完美堆栈canaries、影子堆栈[31]）的保护。 这与之前略有不同

表1：构建仅数据攻击的方法概述以及它们如何模拟图1a中的步骤。 具体而言，它们是否能模拟攻击步骤（ ），能部分模拟攻击步骤（ ），还是需要用户定义攻击步骤（ ）。 ◐ ○

| 方法 | | 设计 | | 这个攻击步骤的模型是什么? | | | 是否实用? | |
|---|---|---|---|---|---|---|---|---|
| 名称 | 年份 | 数据流分析类型 | 目标 | 内存写入漏洞 | Gadget入口 | Gadget链接 | 模型所有基元 | 完全模拟基元 |
| FlowStitch [39] | 2015 | 符号执行 | 以攻击者为中心 | ○ | ◐ | ◐ | ✗ | ✗ |
| MinDOP [41] | 2016 | 静态污点分析 | 图灵完备 | ○ | ◐ | ○ | ✗ | ✗ |
| BOPC [43] | 2018 | 符号执行 | 图灵完备 | ● | ○ | ◐ | ✗ | ✓ |
| Steroids [66] | 2019 | 静态约束求解图灵完备 | | ○ | ○ | ● | ✗ | ✓ |
| Limbo [73] | 2020 | 符号执行 | 以攻击者为中心 | ◐ | ◐ | ◐ | ✓ | ✗ |
| EINSTEIN | - | 动态污点分析以攻击者为中心 | | ● | ● | ● | ✓ | ✓ |

以前的工作[39, 41, 43, 66, 73]不假设强栈保护。 为简单起见，我们专注于类似于该领域中的许多先前工作的流行服务器程序[18, 52, 59, 60, 74, 87–89]。

此外，我们假设攻击者：（1）可以访问与其潜在受害者部署的二进制等效物；（2）可以合法地与目标程序交互（例如，通过向目标Web服务器发送请求）；（3）可以通过信息泄漏[33, 37, 42, 70, 78]绕过ASLR；（4）可以利用内存损坏漏洞从静止程序状态下进行任意写入基元，如以前的工作[60, 88]所示；以及（5）旨在强制目标程序使用攻击者控制的参数调用系统调用[29, 32, 40, 64, 87]。 这与以前的工作不同，以前的工作假设：（1）攻击者可以从任意程序状态[43]执行任意写入基元，而不是从静止状态；（2）攻击者旨在实现图灵完备性[41, 43, 66]，而不是旨在实现安全敏感可控制的系统调用（我们稍后介绍，见表2）。

## 4 EINSTEIN在行动中的概述

在我们详细讨论设计之前，我们通过运行示例来了解EIN-STEIN的操作。 我们考虑一个目标程序，例如网络服务器，具有三个应用程序相关的执行阶段：（i）初始化阶段，程序启动并初始化长期存在的数据，例如配置数据；（ii）静止阶段，程序等待用户交互，例如等待新的客户端请求，此时攻击者可以利用内存写入漏洞；（iii）处理阶段，程序处理用户交互，例如调用execve。

为了指导我们对EINSTEIN操作的解释，我们参考图3，图中顶部显示主要组件，底部显示主要分析步骤。 接下来，我们解释EINSTEIN如何识别（步骤1-4）并构建（步骤5-6）图2的攻击。 我们将EINSTEIN的更复杂方面，例如精确的污点策略和链接，推迟到稍后。

步骤 **1.**我们使用一个程序驱动器（例如，一个测试套件）来运行带有EINSTEIN插装的受害者程序。 与以前的工作[60, 88]一样，EINSTEIN从程序处于静止状态开始分析，此时所有初始化（例如cgi_bin_path）已完成。

步骤 **2.**然后，EINSTEIN通过标记所有可能受到影响的数据（包括cgi_bin_path字符串）来模拟任意内存写入漏洞，从而实现图**1**b的目标**1**。 这是我们的第一个污点策略。
此外，它还将被污染的数据记录在内存快照中。

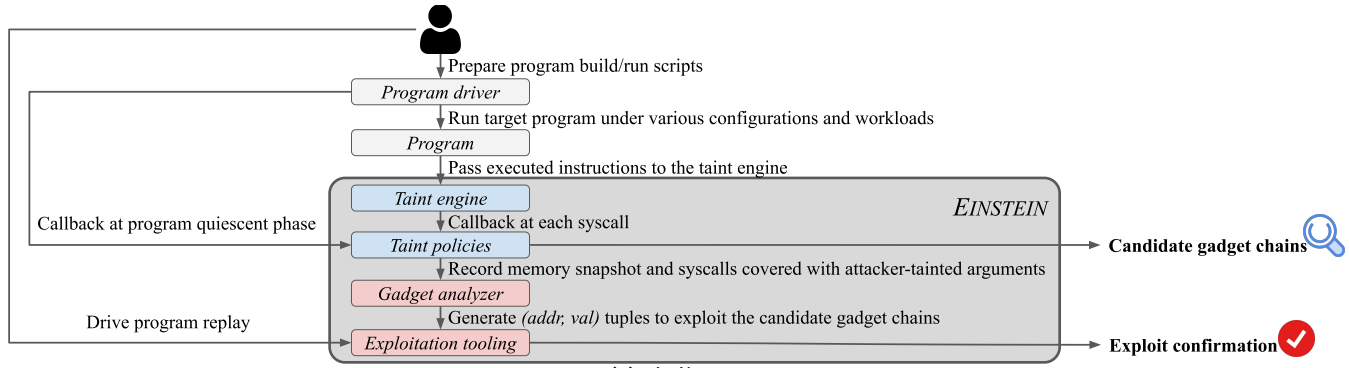
步骤 **3.**为了发现gadg**et**，驱动程序向服务器发送标准请求（例如，来自示例的POST /sort-script请求），同时污点引擎通过目标服务器的执行传播攻击者数据的流程。

第四步。当服务器处理示例 POST请求时，它使用带有攻击者污染参数的 execve调用。 识别系统调用为敏感，EINSTEIN的第二个污点策略将其识别为一个候选的工具，实现了目标**2**。 此外，它还记录系统调用的信息，例如其参数及其污染程度。

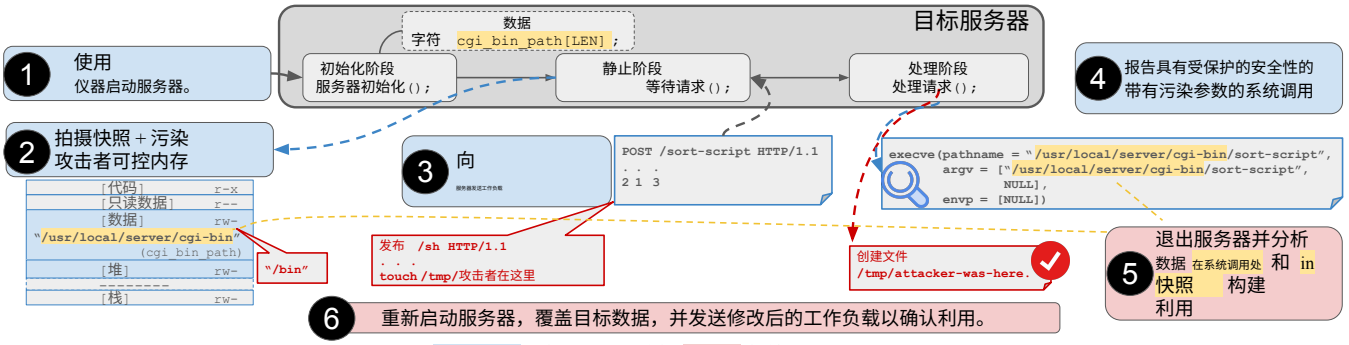
第五步。已经识别出候选工具，现在尝试构建一个利用程序。 首先，EINSTEIN的工具分析器确定了 execve的路径名的一部分 和 argv 参数都带有与cgi_bin_path相对应的颜色的污点。
经过进一步检查，发现它们实际上与cgi_bin_path完全相同。 我们将这种直接的数据流称为身份数据流。 EINSTEIN通过生成 (地址,值)对来构建一个候选的利用程序，以覆盖目标数据，从"/usr/local/server/cgi-bin"到 "/bin"（目标**3**）。

第**6**步。为了确认其有效性，EINSTEIN的利用工具重新启动服务器，覆盖由 $(addr, val)$对定义的目标数据，并发送工作负载以利用该小工具-在这种情况下，使用PO**ST** /sh和一个sh*ell*命令在/t**m**p目录中创建一个文件。 如果文件被创建，EINSTEIN确认利用成功。

对于运行的示例，对单个系统调用的参数进行一次性覆盖就足以进行利用。




(a) 架构。




(b) 操作：E爱因斯坦 在步骤1-4中，识别并构建图2中的示例攻击。

图3：EINSTEIN的概述。

虽然简单，但这样的情况仍然很常见（第7节）。另一方面，当我们讨论完整的集合时，我们将展示EINSTEIN也支持系统调用链。
E爱因斯坦的污染策略。我们在第二个案例研究中展示了链接的有用性（第8节）。

## 5 设计

在本节中，我们通过详细介绍图3a中的每个组件来讨论E爱因斯坦的设计。

### 5.1 污染引擎

为了在运行时跟踪攻击者可控数据，动态污染分析（DTA）引擎模拟了每个执行指令的输入到输出的数据流。为了实现这一点，DTA引擎使用了以下构造（根据引擎的不同可能有不同的名称）：（1）颜色是一种程序元数据，在我们的情况下是攻击者可以覆盖的地址；（2）标签（也称为标签集）是与某些程序数据（例如内存位置）对应的颜色集，表示某些数据是攻击者可控的；（3）标签映射包含了所有程序数据的标签。这是一个相当DTA的典型设计[25, 46, 58, 69, 82, 88]，实际上我们的设计受到了之前工作中使用的DTA引擎[88]的启发，用于识别控制数据攻击[3]。

然而，我们的用例，即非控制数据攻击，带来了两个现有的DTA引擎无法解决的独特挑战
解决。首先，因为所有非控制数据都是潜在的攻击向量，我们需要一种能够唯一跟踪无限数量的污点颜色的方法，覆盖无限量的数据。例如，在我们的示例攻击中，我们需要知道，我们的攻击可能覆盖的所有可能数据中，它应该特别覆盖服务器的CGI-BIN路径。其次，因为程序通常在复杂的点上调用系统调用（例如，时间敏感的网络写入或大文件读取），我们需要一种能够扩展到覆盖复杂特性的复杂工作负载的方法。例如，在我们的示例攻击中，我们确实需要能够覆盖处理攻击的POST请求的代码路径。相比之下，我们确认之前类似的DTA引擎无法轻松扩展到单个默认请求到服务器程序之外，因为过多的减速会导致复杂请求超时[88]。

因此，我们设计了三个基本的数据无限可扩展性和支持颜色的无限数据。

[3] 他们的方法是之前的工作Galileo [77]（“空间”）的继任者，被命名为Newton [88]（“绝对空间和时间”）。
由于我们的方法将Newton的方法推广到非控制数据攻击，我们将其命名为E爱因斯坦（“时空”）。

我们的DTA引擎的几个方面，特别是：（1）它如何访问标签映射，（2）它如何初始化标签映射，以及（3）它如何组合标签。

访问标签映射。每当程序访问数据时（例如通过加载或存储），DTA引擎会访问其标签。DTA引擎通常通过获取程序数据的地址并执行一些映射来定位相应的标签。一些DTA引擎可能会慢但内存效率高通过在每次访问时通过多层动态管理的数据结构（例如页表）进行导航[9, 10, 47, 88]。 其他一些可能快但内存效率低通过在每次访问时索引到静态管理的数据结构（例如影子内存）。我们的方法结合了两者的优点，通过将快速影子内存组织与内存高效的设计相结合。 具体来说，我们指示链接器将用户地址空间限制在顶部4 GB，

并保留底部部分用于影子内存。 这种组织方式产生了（**i**）基于快速影子内存的地址和相应标签之间的映射（仅需2个算术和1个掩码操作）和（**ii**）64位用户指针可以轻松压缩为32位，因为前32位始终相同。由于我们使用指针作为污点颜色，这种指针压缩策略[50]也适用于我们的（32位）颜色，从而减小了标签映射的总体大小50%。

初始化标签映射。 在访问标签映射中的受污染条目之前，DTA引擎必须首先初始化标签映射。. 一些DTA引擎可能会快速但受污染数据受限通过使用未受污染（即零初始化）的标签映射[8–10, 75, 83]。 这种方法利用了快速的硬件MMU，因为对未初始化的标签页的任何访问都会生成页面错误，然后自动将一个零初始化的页面映射到标签映射中。 其他DTA引擎可能会慢但污染无限的数据通过使用大部分受污染的标签映射[82, 88]。 这种方法需要一个慢速的基于软件的MMU来检查每次访问，以便当识别到未初始化的标签页时，它们可以将一个自定义初始化的页面映射到标签映射中。

我们的方法结合了两者的优点，既可以污染无限的数据，又可以通过硬件MMU进行快速访问检查。具体而言，我们使用Linux的userfaultfd功能来：（1）通过硬件页面错误快速识别未初始化的标签页，以及（2）在软件中处理页面错误，以便我们的DTA引擎可以污染表示攻击者可控数据的标签映射条目。

组合标签。每当程序组合数据（例如，通过算术运算），DTA引擎会组合它们的标签。 DTA引擎通过对一个标签中的颜色与另一个标签中的颜色执行集合并运算来实现这一点。一些DTA引擎可能旨在快速但仅支持少量颜色，通过将标签表示为大小受限的数组，并为每种可能的颜色分配一个条目，例如，一个包含1种颜色[10, 47, 83]，8种颜色[8, 47]，256种颜色[7, 9]等的数组。其他DTA引擎可能旨在慢速但支持无限颜色，通过将标签表示为无限集合，并为每个在运行时添加的颜色添加一个条目[82, 88]。

我们的方法通过使用支持无限颜色的标签集合，同时限制其运行时容量以保持快速，兼顾两者的优点。 特别是，我们基于数组的标签集合包含每个在运行时添加的颜色的条目，最多限制为某个限制 $N$。 如果一个标签集合在运行时包含超过 $N$个颜色，那么我们将该标签集合标记为过满，并且不再将任何颜色组合到其中。我们依赖于这样的洞察力，即如果一个标签包含大量颜色，那么它很可能代表着复杂的数据，具有多个来源（例如，加密哈希的输出），因此，相对于只有少量颜色的标签，攻击者很难利用该数据。 对于我们的实验，我们选择 $N = 16$，因为它产生了很少的标签集合过满，同时保持了可接受的内存开销。

## 5.2 污染策略

我们设计污染策略来模拟数据攻击在图1a中列出的步骤。 在高层次上，我们最初将攻击者可控制的区域标记为污染，而EINSTEIN的DTA引擎将污染信息传播到系统调用调用点，并检查系统调用参数的标签以识别可能被利用的参数。

### 5.2.1 建模任意内存写入我们通过污染攻击

者可以覆盖的所有内存来建模任意内存写入。 我们讨论哪些数据是攻击者可控制的，并且我们用什么方式进行污染。

因为我们考虑的是任意内存写入原语，我们会污染任何可写内存段中的数据。 然而，由于我们专注于长期存在的（并排除短期存在的）数据损坏，我们会排除堆栈段。

（这不是一个基本限制，我们的方法可以轻松适应对堆栈上的长期存在的数据进行污染。）此外，为了模拟长期存在的数据损坏的效果，我们会在程序处于静止状态时对内存进行污染，这是之前的工作[88]中所做的。

对于爱因斯坦的分析来说，能够准确地识别每个受污染字节的来源至关重要。 为了以字节粒度跟踪内存依赖关系，我们使用唯一的标签为每个可控攻击者字节配置我们的DTA引擎：具体来说，是其内存地址的压缩（32位）版本。 为了跟踪它所来源的具体值，我们还记录了一个污染了所有内存的内存快照。 因此，对于任何受污染的标签，我们都有它的来源：它的（地址，值）对。 污染的源头后来成为攻击者可能破坏、控制系统调用参数并发动攻击的候选位置。

### 5.2.2 跟踪依赖关系

传统的污染实现[47]模拟了跟踪直接受攻击者控制的内存依赖关系的污染策略（即，系统调用参数$X$在受污染地址$Y$处被读取），并忽略了间接的依赖关系（即，系统调用参数$X$在地址$Y'$处使用受污染指针被读取）。 为了支持后者，我们

此外，我们还实现了负载指针传播（即通过污染指针读取的每个值都被污染），使我们能够模拟攻击者对数据指针、数组索引等进行破坏，从而间接控制系统调用参数。

#### 5.2.3 模拟系统调用利用

我们使用污点汇来模拟攻击的下一部分，即执行一个小工具，并检查系统调用参数是否可被攻击者控制。 具体来说，我们在表2中列出的安全敏感系统调用调用之前插入钩子。
如果我们发现任何被攻击者污染的参数，我们将生成一个系统调用报告，其中包括：（1）关于系统调用的信息，如其名称、参数、参数的污点和回溯；（2）用于区分它与其他报告的信息，如目标进程的PID和TID以及递增的报告编号；（3）用于帮助重现攻击的信息。

#### 5.2.4 建模系统调用链接

我们使用污点源和污点接收器的组合来建模小工具链接。 然而，我们注意到，对于许多端到端攻击，甚至可能不需要链接。 换句话说，因为我们的小工具是系统调用，而不是执行算术或条件分支的小片段。 [19, 20, 24, 43, 49, 66, 71, 77] - 一个小工具可能足以完成整个攻击。例如，图2中的示例攻击仅利用了一个系统调用，即execve。 然而，如果攻击者无法使用此类系统调用（例如，由于缓解措施），系统调用链接将大大扩展可用的小工具集。

通过文件描述符链接系统调用。就像任何其他小工具编译器[43,66,72]一样，我们通过将一个小工具的数据输出链接到另一个小工具的输入来链接系统调用。 系统调用接口可以链接许多不同类型的数据：进程ID（从**getpid**到**kill**），信号量ID（从**sem**get到**sem**op）等等。EINSTEIN链接文件描述符（*F*D），因为它们被我们目标的一些安全敏感系统调用使用。 *F*D唯一地标识一个打开的I/*O*流，例如文件或网络套接字。 如果攻击控制一个I/*O*流，那么它就控制了目标程序发送和接收数据的位置。 我们专注于输出流，因为它们允许攻击者进行更改，但我们的方法同样适用于输入流。

表2列出了EINSTEIN目标并且与链接相关的系统调用。 特别地，一些安全敏感的系统调用只对某些类型的FD（例如， sendto只接受套接字FD）起作用，因此它们只能与特定的创建FD的系统调用（例如， socket）链接。 另一方面， write可以对多种类型的FD进行操作，因此可以与多种类型的创建FD的系统调用链接。

I/O流可以直接或间接地受到攻击者的控制。 首先，如果攻击者控制了I/O流的创建方式，即FD创建系统调用的参数，例如 open的 filename参数，则该I/O流是直接可控的。
其次，如果攻击者控制了安全敏感系统调用的FD参数，并且可以将其重定向到一个直接可控的I/O流。 例如，通过控制 fd参数来 write，攻击者可以将其重定向到前述的直接可控文件。
因为间接可控流取决于直接可控流的存在，我们首先解释如何跟踪直接可控流，然后解释如何识别直接和间接可控访问。

跟踪文件描述符。为了识别直接可控输出流，我们在检查其参数的FD创建系统调用中添加污点接收器。 首先，我们的接收器确定创建的流是否确实可以是一个输出流。 对于某些系统调用，不需要进行检查，但对于其他系统调用，这由参数指定（例如， open的 flag参数确定文件是否可写）。 在这种情况下，我们得出结论，如果：（1）相关参数指定创建一个输出流，或者（2）相关参数受到攻击者污染，因此，攻击者可以强制其创建一个输出流。其次，我们的接收器确定攻击者是否可以直接控制输出流的“位置”，通过检查相关参数是否受到攻击者污染，例如， open的 pathname或 connect的 addr参数。

接下来，我们需要一种追踪这些直接可控流的方法。 也就是说，内核维护着一个FD表，它将FD映射到相应的I/O流。 如果攻击者控制了一个创建的流，那么他们就控制了它在FD表中的条目。 因此，对攻击者可控的流的任何未来引用都应该被视为有污染的，无论FD是否有污染。 为了模拟这个过程，我们维护了我们自己的可控FD表，它与内核的FD表相似，但有一些差异。 具体来说，我们的表格：（1）仅为可控输出流创建条目，（2）将FD映射到创建它的系统调用的报告编号，（3）在可能引用它的任何后续FD系统调用参数中进行污染（无论是直接还是间接引用）。

识别对可控文件描述符的访问。在我们的设计中，我们最终遇到的问题是：我们如何识别对攻击者可控的I/O流的访问？ 朴素的方法是使用在第5.2.3节中描述的相同的污染源来简单地检查安全敏感系统调用的FD参数是否有污染。 然而，这是不精确的，因为它会：（1）无法识别许多可控访问，例如，如果FD参数是未污染的，但其I/O流是有污染的；（2）错误地将许多不可控访问识别为可控访问，例如，如果FD参数是有污染的，但所有打开的I/O流都是未污染的。 因此，我们修改了对FD参数的污染源，改为检查我们的可控FD表。 特别是，我们确定系统调用的FD参数是：（1）直接可控的，如果它存在于我们的可控FD表中；或者（2）间接可控的，如果它有污染并且我们的可控FD表包含相同类型的FD（例如，如果系统调用需要一个套接字FD，则为套接字FD）。

表2：EINSTEIN攻击的系统调用。 我们针对与之前的工作[29, 32, 40, 64, 87]相似的一组安全敏感的系统调用，以及一组演示其可行性的FD配置系统调用。 由于这不是攻击者可以利用的系统调用的详尽列表，EINSTEIN可以轻松扩展以处理更多的系统调用，并针对不同类别的系统调用[48, 61, 90]。

| 系统调用类型 | 链接类型 | 系统调用 |
|---|---|---|
| 安全敏感的系统调用 | 不进行链接 | execve，execveat，mprotect，mremap，remap_file_pages |
| | 输入：文件FD | mmap |
| | 输入：文件或套接字FD | pwrite64，pwritev，pwritev2，sendfile ，write，writev |
| | 输入：套接字FD | sendmmsg，sendmsg，sendto |
| FD-配置系统调用 | 输出：文件FD | creat，open，openat，openat2 |
| | 输出：文件或套接字FD | dup，dup2，dup3 |
| | 输出：套接字FD | bind，connect，setsockopt，socket，socketpair |

## 5.3 Gadget分析器

为了构建对已识别的gadget链的利用，可以使用一种重量级分析方法，通过符号执行来确定所有攻击者数据流的确切约束。
然而，由于EINSTEIN旨在识别低成本攻击，我们的gadget分析器采用了一种轻量级方法，针对身份数据流，即在汇聚点处的值与源头处的值相同的数据流。 正如我们在第7.1节中所展示的，这已经产生了丰富的攻击面。

在我们的情况下，汇聚点值是一个受污染的系统调用参数，源头值是它在内存快照中的来源数据。 我们可以通过检查汇聚点值的污染颜色（包含快照中的源地址）来将汇聚点值映射到其对应的源头值。 换句话说，我们使用每个报告的系统调用参数的污染颜色来识别它在内存快照中的来源值。 如果这些值是等价的，则该参数具有身份流。

然后，为了利用身份流，我们生成 (地址,值)对来覆盖一些源值。 我们可能生成的具体攻击对取决于特定攻击者的目标和特定目标程序。 因此，为了简单起见，我们生成了通用的每系统调用攻击，例如 execve创建一个特定的文件，或 write写入一个特定的字符串。

如果用户愿意，可以从我们的
攻击工具中修改攻击。

## 5.4 攻击工具

由于我们的小工具分析器放弃了复杂的约束求解，我们认为它生成的 (地址,值)对集合仅构成一个候选攻击。为了确认它是一个有效的攻击，我们必须确认我们的攻击确实源自不违反任何程序不变式的身份数据流。 因此，我们不是例如符号验证我们的攻击可能影响的所有可能约束，而是将问题推给程序本身-即，一个真实的、具体的执行。 特别是，我们的攻击工具只是重新运行目标程序，并确认攻击是否确实成功。 它通过以下方式实现：(1) 从任意内存写入原语中重写由 (地址,值)对指定的数据；然后 (2) 向目标程序发送一些工作负载触发小工具链；最后（3）检查是否成功实现了所期望的攻击。 如果攻击成功，我们确认它确实是一个有效的攻击。

我们观察到一些候选攻击比其他候选攻击更有可能成功确认。 幸运的是，因为我们的方法涵盖了如此广泛的范围——例如，通过跟踪任何可能的攻击者数据到任何组合的安全敏感系统调用，我们有很多候选攻击可供选择。我们优先考虑那些我们认为更强大和更稳定的攻击原语，例如利用更高价值的系统调用（例如 execve），利用多个系统调用参数，覆盖确定性地址（例如全局数据），以及利用具有更大身份流量的系统调用参数（对于我们的实验，我们针对至少跨越4个字节的身份流量进行目标定位）。

# 6 实施

我们基于libdft [47]实现了我们的DTA引擎。
与使用libdft的其他分析一样，EINSTEIN的运行时组件（指定污点策略）与我们的libdft变体静态链接。 我们启用我们的工具并禁用ASLR来调用目标程序。 尽管我们的威胁模型已经假设了一个绕过ASLR的攻击者，但为了分析，禁用ASLR确实提供了一些实现层面的好处：（1）它允许我们强制使用32位寻址，这样我们可以将标签从64位压缩为32位；（2）它导致确定性地址在多次运行中保持不变，这对于拥有信息泄漏的攻击者来说并不重要，但它简化了针对此类地址的候选漏洞利用的确认。 我们在附录A中详细介绍了我们实现的其他部分。

# 7 评估

我们在一台装有128GB RAM的AMD Ryzen 9 3950X CPU上运行Ubuntu 22.04.3 LTS（内核版本v6.2）来评估EINSTEIN的攻击面和性能。为了运行Newton [88]，我们使用一台装有32GB RAM的Intel Xeon Silver 4108 CPU运行Ubuntu 16.04.7 LTS（内核版本v4.3）。

表3：每个系统调用和参数类型找到的gadget数量。

| 系统调用* | 总覆盖 | 总覆盖，其中该参数具有来自攻击者数据的数据流的百分比以及其中是身份数据流的百分比。 | | | | | |
|---|---|---|---|---|---|---|---|
| | | 参数1 | 参数2 | 参数3 | 参数4 | 参数5 | 参数6 |
| execve | 11 | 7 (86%) | 7 (86%) | 7 (86%) | | | |
| mmap | 787 | 55 (5%) | 441 (8%) | 55 | 16 (100%) | 17 | 73 |
| mprotect | 144 | 97 (68%) | 98 (27%) | 1 | | | |
| mremap | 11 | 11 | 7 | 11 | - | - | |
| pwrite64 | 1270 | 1217 (3%) | 741 (47%) | 399 (19%) | 506 (36%) | | |
| pwritev | 10 | 10 (100%) | 10 (10%) | - | 10 (20%) | | |
| sendfile | 1 | - | - | 1 | 1 | | |
| sendmmsg | 2 | 2 | 2 | - | - | | |
| sendmsg | 12 | 5 (100%) | 3 (100%) | - | | | |
| sendto | 431 | 389 (7%) | 410 (38%) | 408 (6%) | - | - | - |
| write | 3083 | 768 (74%) | 2877 (91%) | 1265 (10%) | | | |
| writev | 754 | 244 (18%) | 742 (76%) | - | | | |

* 未涵盖 execveat、 pwritev2或remap_file_pages。

表4：找到的gadget数量和每个目标程序的性能。

| 目标程序 | 非控制数据 gadgets（%身份数据流） | 控制数据 gadgets*（%身份数据流） | | 运行时间 (h:mm:ss) | | 峰值PSS（MB） | | | 代码覆盖率 |
|---|---|---|---|---|---|---|---|---|---|
| | | 带有寄存器操作数 | 带有内存操作数 | 基准 | EINSTEIN | 基准 | EINSTEIN（每个标签集1种颜色） | EINSTEIN（每个标签集16种颜色） | |
| httpd | 1834 (97%) | 2082 (42%) | 15,179 (94%) | 0:04:00 | 0:35:08 | 26 | 1560 | 3129 | 27.3% |
| lighttpd | 92 (98%) | 50 (34%) | 155 (94%) | 0:00:05 | 0:01:51 | 3 | 176 | 257 | 27.8% |
| nginx | 1623 (82%) | 503 (60%) | 564 (99%) | 0:08:41 | 3:27:05 | 24 | 479 | 848 | 49.1% |
| postgres | 2105 (27%) | 1382 (11%) | 4690 (13%) | 0:00:56 | 1:47:27 | 175 | 1568 | 11,621 | 46.5% |
| redis | 218 (84%) | 160 (22%) | 53 (100%) | 0:04:01 | 1:01:13 | 191 | 3257 | 32,188 | 33.6% |

* 通过代码重用防御进行缓解。

程序和驱动程序。我们的目标是Web服务器 httpd、lighttpd和 nginx；以及数据库服务器 postgres和redis—所有这些都被证明存在内存写入漏洞的风险[1–6]。 尽管我们遵循以前的工作通过针对服务器应用程序[18, 52, 59, 60, 74, 87–89]，我们注意到，即使是具有更有限用户输入交互和很少明显危险系统调用（如 execve）的程序，也可能存在风险。 因为EINSTEIN的方法只是建立在标准的写入漏洞上，所以它确实可以推广到这样的应用程序。

我们使用每个服务器的测试套件来驱动分析，因为测试套件旨在测试各种功能，因此涵盖了各种代码路径。如果爱因斯坦能够根据简单的测试套件创建利用程序，那么这证实了我们的方法极大地简化了仅数据攻击。

此外，由于nginx是一个常见的攻击目标案例研究[55, 88, 92]，我们使用我们的攻击工具来确认爱因斯坦为nginx生成的候选利用程序。

程序阶段确定。如第4节所述，程序执行的阶段（即初始化、静止和处理阶段）是应用程序相关的。 虽然更复杂的方法可以确定程序的不同执行阶段（例如系统调用监视[35]），但我们所有的目标服务器在几秒钟内初始化。 因此，在建立初始后静止状态之前，我们只需保守地等待10秒钟。

## 7.1 攻击面分析

我们根据我们的方法揭示的攻击面进行了调查：攻击者能够有效地控制安全敏感的系统调用，以及攻击者可以通过这些系统调用做什么。 在附录B中，我们还评估了这种攻击面与控制流劫持攻击的攻击面的比较。

与之前的工作[54]一样，我们根据唯一的回溯来呈现系统调用的数量，因为其他可能的度量标准存在问题：呈现总执行的系统调用会人为地增加这些数字（因为程序越长

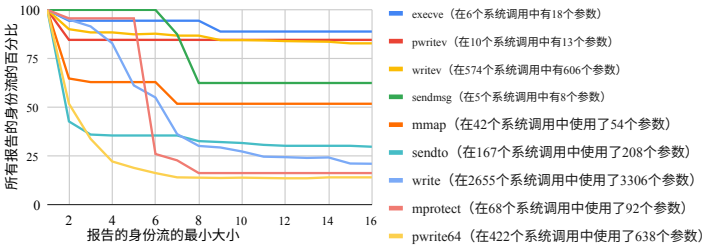



图4：报告的所有身份流的百分比与报告的身份流的最小大小之间的关系。

运行次数越多，执行的系统调用越多）；而展示唯一的调用点将人为地降低这些数字（因为一个系统调用点可能从程序的多个部分调用）。可控性。为了评估攻击者

可以控制系统调用参数的程度，我们首先参考表3，该表列出了有污染的系统调用参数的数量，特别是具有身份流的参数的百分比。我们首先观察到的是，大多数参数都有一个身份流。毫不奇怪，许多身份流是针对通常被视为“不可变”的数据类型的，例如在execve中用于pathname和argv参数的字符串（86%）以及用于write和writev的buf参数（91%）

(76%)。同样，FD参数往往具有高比例的标识流（例如，对于 pwritev为100%，对于 sendmsg为100%，对于 write为74%）；这也不足为奇，因为这些FD通常由同时接受基于字符串的参数的系统调用配置，例如 open和 creat的 pathname，以及connect的 addr（根据其 sa_family可能是字符串）。

我们基于图4做出的第二个观察是，标识流的长度按系统调用进行了细分。我们观察到许多标识流至少跨越16个字节，因此，攻击者可以破坏的数据在脆弱的系统调用之前保持不变。鉴于攻击者数据在如此多的系统调用参数中保持不变的普遍性，我们得出结论，我们的仅数据攻击是低成本的。

基于系统调用的利用为了评估攻击者可以对这些可控制的系统调用做什么，我们再次参考表3，并观察到每个参数都可能受到攻击者的污染。即使对于我们不希望受到攻击者污染的系统调用参数（例如，*m*prot的prot参数和send*m*sg的flags参数，通常编译为常量），我们仍然遇到了受污染参数的情况。例如，我们发现pthread_create调用*m*protect时，prot参数从堆上的可控数据中获得了身份流。

接下来，我们参考表5，展示了我们对nginx的确认利用，并观察到它们为攻击者提供了许多基本操作。例如，一个易受攻击的execve给我们提供了一个代码执行的基本操作；易受攻击的文件配置系统调用（例如openat）结合易受攻击的文件写入系统调用（例如write）给我们提供了17个写入基本操作。

表5：对于 nginx的已确认漏洞利用。

| 攻击原语 | 计数 |
|---|---|
| CODE-EXECUTION | 1 |
| WRITE-WHAT-WHERE | 17 |
| WRITE-WHAT | 375 |
| WRITE-WHERE | 79 |
| SEND-WHAT-WHERE | 41 |
| SEND-WHAT | 372 |
| SEND-WHERE | 59 |
| 总计 | 944 |

WHAT-WHERE原语；易受攻击的套接字配置系统调用（例如 connect）与易受攻击的套接字写入系统调用（例如 sendmsg）给我们提供了41个SEND-WHAT-WHERE原语；易受攻击的系统调用与非易受攻击的系统调用给我们提供了更弱但更多的原语（即，总共885个WRITE-WHAT、WRITE-WHERE、SEND-WHAT和SEND-WHERE原语）。相比之下，FlowStitch [39]——之前的一种方法也（i）生成基于系统调用的漏洞利用，并且（ii）在评估nginx[2]时假设了任意内存写入原语——受到了符号执行覆盖率差的限制，因此只为 nginx生成了2个漏洞利用。由于EINSTEIN识别出如此广泛的受污染参数，并生成各种攻击原语，我们得出结论，我们的仅数据攻击是多样化的。

## 7.2 性能分析

我们研究了 EINSTEIN在两个维度上的性能。首先，我们研究了整体性能
EINSTEIN每个目标程序的性能，如表4所示。我们比较的基准是原始目标程序，即没有任何插装。

其次，我们研究了 EINSTEIN的 DTA 引擎的性能。为了做到这一点，我们将 EIN-STEIN的开销与 Newton [88]的开销进行比较，与 EINSTEIN不同的是，Newton 没有在第5.1节中描述的 DTA 引擎优化。具体来说，Newton 的 DTA 引擎：(1) 通过页表遍历访问其标签映射，(2) 通过基于软件的 *MMU* 检查初始化其标签映射，(3) 将标签表示为无界集合。图5展示了运行 nginx在最小配置下并处理多个 HTTP 请求时，Newton、EINSTEIN及其基准的开销。我们注意到两个问题导致了这个评估设置：(1) 任何中等复杂的工作负载（例如，使用加载多个模块的配置，或处理 HTTPS 请求）都会导致 Newton 立即崩溃，因此我们只运行最小配置并处理简单请求；(2) 由于各种技术原因（例如，Newton 需要较旧的内核版本，完全支持32位等），我们无法在上述配置上评估EINSTEIN和 Newton。

在同一台机器上，我们展示了两台机器上的基准开销；T1是EINSTEIN的机器，T2是牛顿的机器。

运行时开销。如表4所示，EINSTEIN的运行时开销范围为8倍至26倍。 我们广泛的实验，包括完整的测试套件，证实了DTA引擎可以轻松扩展到完整的测试活动。 此外，从图5**a**中可以看出，EINSTEIN处理GET请求的速度至少比牛顿快两个数量级。我们将EIN-STEIN的加速主要归因于其访问标签映射的方式：在每次访问时，它执行快速的影子内存查找，而牛顿则导航其页表结构。 由于我们的DTA引擎优化使得分析平台实用，我们认为运行时开销是可以接受的。

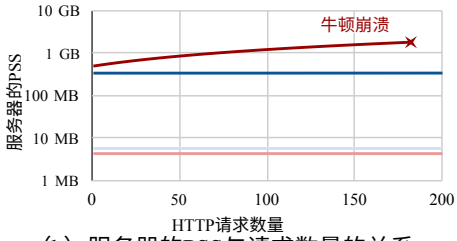
内存开销。接下来，我们评估了EINSTEIN在使用每个标签集1个颜色和使用每个标签集16个颜色（即默认设置）时的内存开销。 使用1个颜色/标签集时，标签很容易过填充，因为将超过1个颜色添加到标签集中将导致它不再跟踪各个颜色。 因此，这种配置无法为许多系统调用参数的标签构建候选利用链，因为许多标签已经过填充。 尽管如此，它仍然识别出16个颜色/标签集配置所识别的所有候选利用链。

我们在表4中发现，EINSTEIN的内存开销与1个颜色/标签集相对于未经过仪器化的基线为9-60倍。 这个开销包括内部数据结构的开销
EINSTEIN，libdft64-ng等等，其中最重要的是标签映射，它包含了程序内存每个字节的标签集。 如果我们将颜色/标签集的数量从1增加到16，我们预计最坏情况下的内存开销将增加16倍。 然而，我们发现我们的目标数据库有7.4-9.9倍的开销，而我们的目标Web服务器只有1.6-2.0倍的开销。 我们将更糟糕的开销归因于目标应用程序具有更细粒度的空间访问模式，因为它们需要每个程序内存页加载更多的标签映射。

接下来，在图5b中我们可以看到，爱因斯坦的内存开销是恒定的，而牛顿的内存开销会随着每个额外的 GET 请求而增长，直到最终因为达到32位进程的2 GB限制而崩溃。 我们将这种差异归因于DTA引擎表示标签的方式：爱因斯坦使用有界的16个标签/集合，而牛顿使用无界的集合。 因此，牛顿会将任何新的颜色与标签结合，直到内存用尽为止。

原则上，这会导致最坏情况下的$O(n^2)$内存开销（其中 $n$ 是总程序内存），即每个程序数据字节都与其他程序数据字节结合。
因此，像 nginx这样的应用程序在正常运行时需要大约16 MB的内存，但在最坏情况下，需要256 TB的内存才能运行牛顿。然而，在实际情况下，2 GB的限制掩盖了这种开销，并且在达到此限制之前就会崩溃。 我们得出结论，由于我们的DTA引擎的存在-

（a）请求运行时间与请求数量的关系。



（b）服务器的PSS与请求数量的关系。

图5：将nginx与EINSTEIN和Newton [88]一起运行与仅运行nginx（即基准）进行比较。

由于其可配置的每个标签集的颜色数量（即在可用内存总量和生成的利用程序数量之间进行权衡），我们的内存开销是可以接受的。

覆盖率。与任何动态分析一样，其有效性取决于程序的覆盖率：覆盖率不足通常会导致结果不佳，反之亦然。 然而，尽管测试套件仅覆盖了程序代码的27-49%，EINSTEIN仍然发现了许多可利用的工具。 未来的工作可以通过利用sysc**a**ll-**guard**变量[93]来增加代码覆盖率，这无疑会产生更多的可利用工具。

## 8个案例研究

我们已经证明EINSTEIN可以为流行的服务器构建多样化、低成本的攻击。 为了证明这种攻击构成了严重威胁，我们提供了两个针对nginx的案例研究。 对于每个案例研究，我们将：（1）解释EINSTEIN如何识别gadget链，构建候选利用程序，并确认利用程序的有效性；（2）讨论利用程序如何绕过最先进的防御措施。

针对的防御措施。在考虑我们讨论的防御措施时，我们回顾了第2节中关于针对易受攻击的定义和用法的实用和全面的防御措施的解释。 因为我们假设存在内存写入漏洞，任何以 *def*为中心的防御措施（例如内存安全、**D**SR和内存错误扫描）都不在讨论范围内。 此外，由于全面的防御措施（例如内存安全、完整的**D**SR和完整的**D**FI）并未广泛部署，我们也将它们排除在讨论范围之外。 因此，我们将讨论集中在实用的以使用为中心的防御措施上，即系统调用过滤[34, 35, 44, 67]和选择性

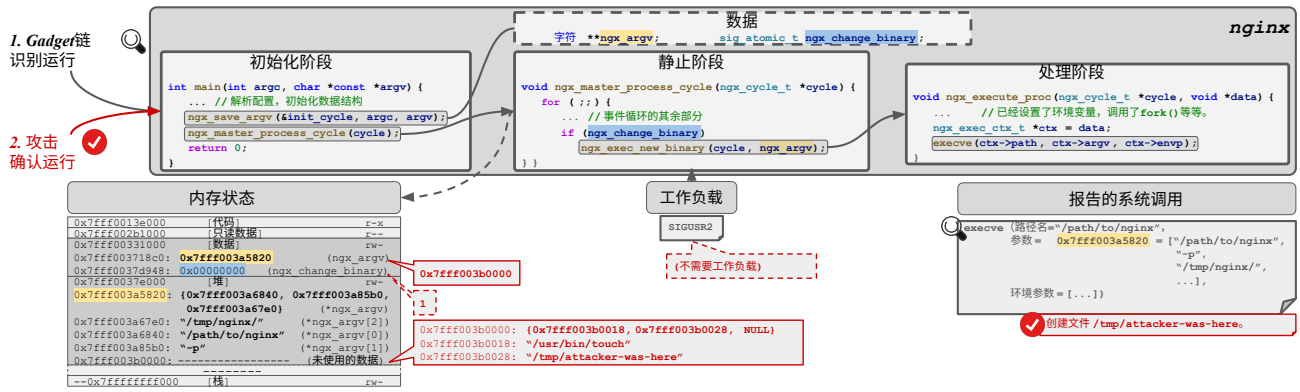



图6：EINSTEIN在nginx中识别并构建攻击的基于execve的gadget。

DFI [44, 79]. 正如我们所证明的那样 实际的 防御是根本不安全的，我们得出结论需要部署更多 全面的防御措施。[4]

## 8.1 绕过系统调用过滤

这个案例研究利用了一个 execve 在 nginx的实时二进制升级功能中的小工具，其中 nginx在保持所有连接的同时替换自己的运行二进制文件。图6
展示了我们如何识别和利用这个小工具 EINSTEIN.

识别和利用。

首先，当启动时，nginx将其 argv保存到全局变量 ngx_a rgv中，然后被 EINSTEIN污染。 接下来，当向 nginx发送工作负载时，EINSTEIN向 nginx主进程发送 SIGUSR2 信号，从而启动实时二进制升级。. 这引发了ngx_chang e_binary条件，导致 nginx调用ngx_exec_new_binary并使用**ngx_a**rgv 进行二进制文件更改。 作为一个参数。 最终，这导致nginx调用execve，将初始保存的相同argv传递给它。 EINSTEIN识别execve的受攻击参数并报告它。

EINSTEIN通过首先识别execve的argv参数到ngx_argv之间的身份数据流来构建此漏洞利用。 全局变量。然后，EINSTEIN通过生成覆盖ngx_argv指向包含两个字符串的数组的(*addr, val*)对，EINSTEIN构建了一个“任意代码执行”候选漏洞利用。字符串为："/usr/bin/touch","/tmp/attacker-was-here"，以及一个NULL终止符。 重新运行nginx以确认此漏洞利用后，它确实创建了文件/tmp/attacker-was-here，表明我们的漏洞利用成功。

因为这个小工具是在服务器端触发的（通过 SIGUSR2 ）而不是客户端（通过网络请求），乍一看似乎远程攻击者无法触发它-相反，攻击者必须等待 nginx进程自行升级。 然而，我们的利用工具（ 解析小工具的回溯）立即清楚地表明
我们确实可以远程触发这个小工具。在
特别是，通过简单地覆盖被污染的全局变量
ngx_change_binary，我们可以强制 nginx在ngx_exec_ne w_binary之前进入条件分支，从而调用
execve。 因此，通过仅仅进行一点手动检查，攻击者可以将这样的不可触发小工具升级为可触发小工具。 我们注意到，这种“合法”的控制流劫持方法与以前的仅数据方法有几个不同之处。 首先，它是可选的，因为这个小工具也可以被触发：（1）通过 nginx，在它自身升级时，和（2）通过攻击者，通过劫持一个被污染的 killsys call。
其次，它的工作量相对较低，因为我们只需翻转
一个已经识别出的gadget之前的一个分支，
而其他方法则需要分析所有可能的分支
以寻找新的gadget [41, 43, 73]。

讨论。基于系统调用过滤器的防御的目标[34,35, 67]是禁用高价值的系统调用，例如 execve。 然而，系统调用过滤器在根本上是不准确的，因为它们无法禁用合法的系统调用。 换句话说，如果 nginx可以合法地调用 ex ecve，那么防御就无法禁用 execve——否则它将破坏 n ginx。 EINSTEIN利用了这种不准确性，因为它不会改变控制流，因此它只涵盖合法的系统调用，从而避免了基于系统调用过滤器的防御。

原则上，如果一个注重安全的系统管理员为 nginx设计了一个完全禁用所有合法使用 execve的配置（例如通过禁用CGI、实时二进制升级等），那么系统调用过滤器确实可以禁用 execve，从而减轻这个gadget的影响。 然而，在实践中，这几乎是不可能的，因为：（1）无法从配置中禁用某些功能（例如实时二进制升级）；（2）很难确定没有可能的程序点会调用 execve（例如由于动态加载的库和模块的存在）。 对 nginx上最先进的系统调用过滤器的评估证实了这一观察结果：它们都无法完全禁用 execve[34, 35, 67]。

[4]的确，开发人员可以部署全面的（但代价高昂的）防御措施来阻止EINSTEIN的攻击。 此外，他们可以使用EINSTEIN的报告来加固他们的程序（例如，通过清理特定的系统调用参数）。

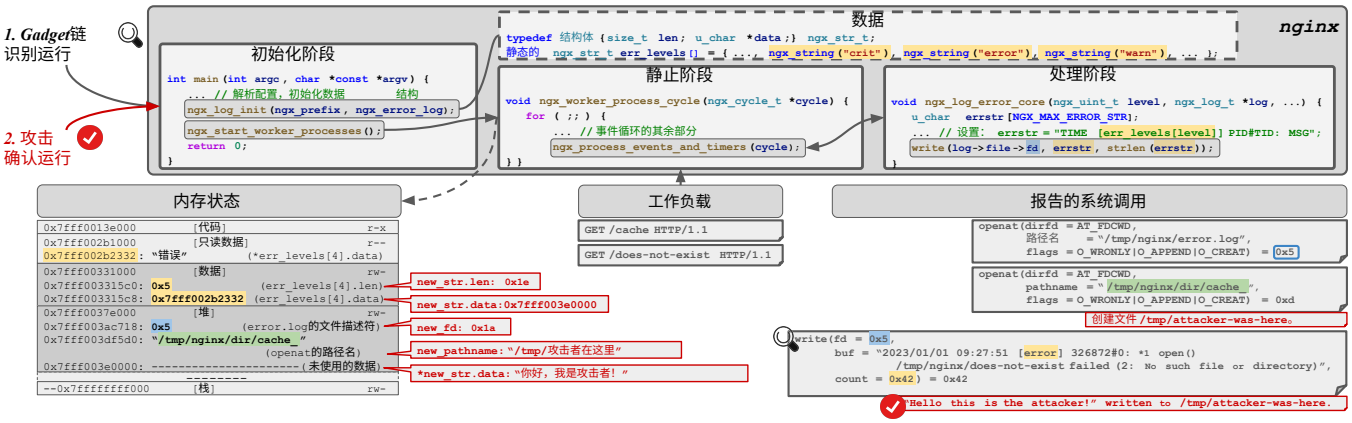



Figure 7: An openat-to-write-based gadget chain in nginx that EINSTEIN identified and built an attack for.

## 8.2 绕过选择性DFI

这个案例研究利用了一个 openat来进行写入gadget链中nginx的错误日志。图7展示了我们如何识别和利用带有EINSTEIN的gadget。

**识别和利用。在nginx进入其静止阶段后，EINSTEIN污染了三个** nginx对象在部分中特别是：（1）err_levels[]全局数组，其中包含指向日志字符串（例如，"crit"，"error"，"warn"）及其长度的指针；（2）配置定义的缓存文件的路径，nginx在对/cache的请求上打开；以及（3）nginx错误日志的FD。然后，EINSTEIN向nginx发送请求，其中两个触发了我们gadget链的系统调用。 首先，它发送一个GET /cache请求，导致nginx打开缓存文件。 它通过一个带有污染的路径名的 openat来打开它。 然后，EINSTEIN将创建的文件添加到其可直接控制的FD表中。 其次，EINSTEIN发送一个GET /does-not-exist请求，导致nginx将一个 "error"消息写入错误日志。它通过一个带有污染的 fd、 buf和 count的 write来实现这一点。 因为FD参数是污染的，并且直接可控的FD表中有一个文件，

E爱因斯坦确定写入可以被重定向到另一个文件，从而识别从openat到write的链。

E爱因斯坦通过首先识别openat的路径和堆上的缓存之间的身份流来构建此漏洞利用。 和缓存堆上的路径名，(2) write的fd 和错误日志的FD 堆上的buf和指向"error"字符串的指针。 然后，E爱因斯坦通过生成$(addr, val)$对来覆盖：(1) 缓存文件的路径名为"/tmp/attacker-was-here"，(2) 错误日志的FD为缓存文件的FD，以及(3) 指向"error"字符串的指针为指向"Hello this is the attacker"字符串的指针，从而构建了一个"写入文件系统的write-what-where"候选漏洞利用。 接下来，我们重新运行E爱因斯坦来确认此漏洞利用，但不幸的是，它只向我们的文件写入"Hello"。 仔细检查后，我们很快注意到从write的count参数的标记集中，它取决于"error"字符串的长度，即0x5。

从而解释了为什么只有我们的tar文件的前五个字符被写入了文件。 我们的分析器最初没有针对字符串长度进行目标定位，因为它没有一个身份数据流写入的计数值不同，因此我们修改我们的攻击以覆盖err_levels[]字符串长度为0x1e，即我们字符串的长度。结果，我们能够成功地写入我们整个攻击者指定的字符串到我们攻击者指定的文件。

讨论。选择性**DFI**的目标是仅对选择的*def*-**use**链路强制执行**DFI**，例如对系统调用参数使用[44]。 然而，选择性**DFI**在根本上是不精确的，因为：(1)它使许多数据流不安全（即它是"选择性的"），(2)对于任何复杂的数据流，它必须静态地过度近似给定使用的合法*def*集，否则就会破坏目标程序。 EINSTEIN利用了这种不精确性，因为它使用了轻量级的动态分析，对于复杂的数据流没有任何问题。

具体来说，如果选择性DFI的指向分析无法确定所有可能的某些加载数据的def，例如由于众所周知的困难的过程间分析、指针别名问题等[38]，那么它必须假设任何def都是合法的。例如，在我们的gadget中，从"error"字符串的def到write的use of buf的数据流很难解决：err_levels[]有四个地址被占用的位置，可能在过程间传递多达五个调用深度（例如，传递给可变参数sprintf的自定义实现）。 因此，保护write的buf的每一个可能的def是非常困难的，而且过度近似可能会导致显著的性能开销。 毫不奇怪，将选择性DFI应用于系统调用参数的最近工作并不尝试保护write[44]。 最后，我们注意到这个gadget也对系统调用过滤具有抵抗力，因为许多合法的程序功能依赖于openat和write。

## 9个限制

覆盖率。由于几个原因，我们的分析范围有限。 首先，像任何动态分析一样，我们的代码覆盖率不完整。尽管如此，我们仍然发现了许多漏洞，

正如第7.2节所解释的那样，进一步改进代码覆盖率也将识别出更多的攻击。 其次，像其他DTA引擎一样，我们不追踪隐式流。 然而，我们认为这是可以接受的，因为隐式流对攻击者来说不太有用，因为它们通常只传播一位数据。 第三，EINSTEIN不构建每一种数据攻击，例如那些旨在实现图灵完备性的攻击。相反，

EINSTEIN旨在实现更简单但更强大的原语：可控制的系统调用。 识别和消除这些将显著提高攻击者的难度。

约束条件。与使用DTA的其他方法一样，我们的分析受到限制，因为它不以两种方式跟踪所有可能的数据流约束。 首先，它不跟踪任意程序操作（例如算术）对攻击者数据的影响。 然而，对我们来说，这不是一个问题，因为我们的身份数据流分析会过滤掉这些更复杂的小工具，而我们仍然有很多候选小工具。 其次，它不跟踪攻击者数据如何影响程序的路径。 然而，对我们来说，这也不是一个问题，因为我们的攻击确认会过滤掉这些更复杂的候选攻击，而我们仍然有很多可行的攻击。

## 10 结论

我们提出了EINSTEIN，一种轻量级的方法来构建实用攻击，这些攻击是以前的解决方案所无法达到的。 我们使用EINSTEIN构建了数百个仅数据攻击，并展示了两个低成本的案例研究，绕过了最先进的防御措施。 我们得出结论，仅数据攻击对攻击者来说是完全可行的，因此，供应商应考虑采用更强大的防御措施，如全面的DFI和内存安全性。

## 致谢

我们要感谢匿名审稿人的反馈意见。 我们还要感谢Alyssa Milburn对初始影子内存实现的帮助，以及Mans van Someren修复了一些libdft的错误。 这项工作得到了荷兰研究委员会（NWO）通过“INTERSECT”和“Theseus”项目的支持。

## 参考文献

[1] “CVE-2007-4727”，https://nvd.nist.gov/vuln/detail/CVE-2007-4727.

[2] “CVE-2013-2028,” https://nvd.nist.gov/vuln/detail/CVE-2013-2028.

[3] “CVE-2021-32027,” https://nvd.nist.gov/vuln/detail/CVE-2021-32027.

[4] “CVE-2022-23943,” https://nvd.nist.gov/vuln/detail/CVE-2022-23943.

[5] “CVE-2022-41741,” https://nvd.nist.gov/vuln/detail/CVE-2022-41741.

[6] “CVE-2023-36824,” https://nvd.nist.gov/vuln/detail/CVE-2023-36824.

[7] “DataFlowSanitizer (Clang 12 documentation),” https://releases.llvm.org/12.0.0/tools/clang/docs/DataFlowSanitizerDesign.html.

[8] “数据流灌输器（Clang 13 文档）”，https://releases.llvm.org/13.0.0/tools/clang/docs/DataFlowSanitizerDesign.html.

[9] “KernelDataFlowSanitizer”，https://github.com/vusec/kdfsan-linux/.

[10] “KernelMemorySanitizer”，https://github.com/google/kmsan.

[11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti,“控制流完整性原理、实现和应用”， *TISSEC*，第13卷，第1期，2009年.

[12] S. Ahmed, H. Liljestrand, H. Jamjoom, M. Hicks, N. Asokan, and D. D. Yao，“并非所有数据都是平等的：数据和指针优先级排序以实现可扩展的数据导向攻击防护”，在*USENIX Security*，2023年.

[13] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, “使用WIT防止内存错误利用”，在 *S&P*, 2008年。

[14] P. Akritidis, M. Costa, M. Castro, and S. Hand,“Baggy Bounds Checking: 一种高效且向后兼容的防御措施，用于防止越界错误”，在*USENIX Security*, 2009年。

[15] S. Andersen and V. Abella, “Microsoft Windows XP Service Pack 2中对功能性的更改，第3部分：内存保护技术，数据执行预防”，http://technet.microsoft.com/en-us/library/bb457155.aspx, 2004年。

[16] B. Belleville, H. Moon, J. Shin, D. Hwang, J. M. Nash, S. Jung, Y. Na, S. Volckaert, P. Larsen, Y. Paek 等, “硬件辅助的数据随机化”，在 *RAID*, 2018年。

[17] S. Bhatkar和R. Sekar，“数据空间随机化”，in *DIMVA*，2008年。

[18] A. Bittau，A. Belay，A. Mashtizadeh，D. Mazières和D. Boneh，“盲目黑客”，在 *S&P*，2014年。

[19] T. Bletsch，X. Jiang，V. W. Freeh和Z. Liang，“跳转导向编程：一种新的代码重用攻击”，在 *AsiaCCS*，2011年。

[20] E. Bosman和H. Bos，“Framing Signals——A Return to Portable Shellcode”，在 *S&P*，2014年。

[21] C. Cadar，P. Akritidis，M. Costa，J.-P. Martin和M. Castro，“数据随机化”，技术报告TR-2008-120，微软研究。, Tech. Rep.，2008年。

[22] S. A. Carr 和 M. Payer，“DataShield: 可配置的数据机密性和完整性，在 *AsiaCCS*，2017年。”

[23] M. Castro，M. Costa 和 T. Harris，“通过强制数据流完整性来保护软件，在 *OSDI*，2006年。”

[24] S. Checkoway，L. Davi，A. Dmitrienko，A.-R. Sadeghi，H. Shacham 和 M. Winandy，“无返回的返回导向编程，在 *CCS*，2010年。”

[25] P. Chen 和 H. Chen，“Angora: 基于原则的搜索的高效模糊测试，在 *S&P*，2018年。”

[26] S. Chen，J. Xu，E. C. Sezer，P. Gauriar 和 R. K. Iyer，“非控制数据攻击是现实威胁，在 *USENIX Security*，2005年。”

[27] L. Cheng, S. Ahmed, H. Liljestrand, T. Nyman, H. Cai, T. Jaeger, N. Asokan, and D. Yao, “利用现有和潜在的防御方法进行面向数据的攻击的利用技术”， *TOPS*, vol. 24, no. 4, 2021.

[28] L. Cheng, H. Liljestrand, M. S. Ahmed, T. Nyman,T. Jaeger, N. Asokan, and D. Yao, “面向数据的攻击的利用技术和防御”，在*SecDev*, 2019.

[29] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng,“ ROPecker: 一种通用且实用的防御ROP攻击方法”，在 *NDSS*, 2014.

[30] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, “Point-GuardTM: 保护指针免受缓冲区溢出漏洞的影响,” in*USENIX Security*, 2003.

[31] T. H. Dang, P. Maniatis, and D. Wagner, “影子栈和栈金丝雀的性能代价,” in*AsiaCCS*, 2015.

[32] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, “路径敏感控制安全的高效保护,” in *USENIX Security*, 2017.

[33] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh,“ 跳过ASLR: 攻击分支预测器绕过ASLR,” in *MICRO*, 2016.

[34] S. Ghavamnia, T. Palit, S. Mishra, 和 M. Polychronakis, “Temporal System Call Specialization for Attack Surface Reduction,” 在*USENIX Security*, 2020.

[35] S. Ghavamnia, T. Palit, 和 M. Polychronakis, “C2C: Fine-grained Configuration-driven System Call Filtering,” 在 *CCS*, 2022.

[36] F. Gorter, E. Barberis, R. Isemann, E. van der Kouwe, C. Giuffrida, 和 H. Bos, “FloatZone: Accelerating Memory Error Detection using the Floating Point Unit,” in *USENIX Security*, 2023.

[37] B. Gras, K. Razavi, E. Bosman, H. Bos, 和 C. Giuffrida, “ASLR on the Line: Practical Cache Attacks on the MMU,” 在 *NDSS*, 2017.

[38] M. Hind，“指针分析：我们还没有解决这个问题吗？”在 *PASTE*，2001年。

[39] H. Hu，Z. L. Chua，S. Adrian，P. Saxena和Z. Liang，“自动生成数据导向的攻击”，在*USENIX Security*，2015年。

[40] H. Hu，C. Qian，C. Yagemann，S. P. H. Chung，W. R.Harris，T. Kim和W. Lee，“强制执行控制流完整性的唯一代码目标属性”，在 *CCS*，2018年。

[41] H. Hu，S. Shinde，S. Adrian，Z. L. Chua，P. Saxena和Z. Liang，“数据导向编程：非控制数据攻击的表达能力”，在 *S&P*，2016年。

[42] R. Hund, C. Willems, 和 T. Holz, “针对内核空间ASLR的实用时序侧信道攻击,” 在 *S&P*, 2013.

[43] K. K. Ispoglou, B. AlBassam, T. Jaeger, 和 M. Payer, “块导向编程: 自动化仅数据攻击,” 在 *CCS*, 2018.

[44] C. Jelesnianski, M. Ismail, Y. Jang, D. Williams, 和 C. Min, “保护系统调用，用BASTION保护（大部分）世界,” 在 *ASPLOS*, 2023.

[45] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, 和 Y. Wang, “Cyclone: 一种安全的C方言,” 在 *ATC*, 2002.

[46] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, andC. Giuffrida, “Kasper: 在Linux内核中扫描广义瞬时执行工具,” in *NDSS*,2022.

[47] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D.Keromytis, “libdft: 用于商品系统的实用动态数据流跟踪,” in *VEE*, 2012.

[48] K. Koning, H. Bos, and C. Giuffrida, “使用硬件辅助的过程虚拟化进行安全高效的多变体执行,” in *DSN*, 2016.

[49] B. Lan, Y. Li, H. Sun, C. Su, Y. Liu, and Q. Zeng, “面向循环的编程: 一种绕过现代防御的新代码重用攻击,” in *TrustCom*, 2015.

[50] C. Lattner 和 V. S. Adve，“透明指针压缩用于链式数据结构”，在 *MSP*, 2005年。

[51] T. Liu, G. Shi, L. Chen, F. Zhang, Y. Yang 和 J. Zhang，“TMDFI: 基于标记内存辅助的细粒度数据流完整性保护，针对嵌入式系统的软件攻击”，在 *TrustCom*, 2018年。

[52] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim 和 W. Lee，“ASLR-Guard: 阻止代码重用攻击中的地址空间泄漏”，在 *CCS*, 2015年。

[53] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser,G. Lowney, S. Wallace, V. J. Reddi 和 K. Hazelwood，“Pin: 使用动态插装构建定制程序分析工具”，在 *PLDI*, 2005年。

[54] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “模糊测试的艺术、科学和工程：一项调查”， *TSE*, vol. 47, no. 11,2019.

[55] M. Morton, J. Werner, P. Kintis, K. Snow, M. An-tonakakis, M. Polychronakis, and F. Monrose, “异步Web服务器的安全风险：当性能优化放大数据导向攻击的影响时”，在 *EuroS&P*, 2018.

[56] S. Nagarakatte, J. Zhao, M. M. Martin, andS. Zdancewic, “SoftBound：C语言的高度兼容和完整的空间内存安全性”，在 *PLDI*,2009.

[57] ——，“CETS: 编译器强制的时间安全性 for C”，在 *ISMM*, 2010年。

[58] J. Newsome 和 D. X. Song，“动态污点分析用于自动检测、分析和生成漏洞签名的通用软件”，在 *NDSS*,2005年。

[59] B. Niu 和 G. Tan，“每个输入的控制流完整性”，在 *CCS*, 2015年。

[60] A. Oikonomopoulos, E. Athanasopoulos, H. Bos 和 C. Giuffrida，“在信息隐藏中挖洞”，在 *USENIX Security*, 2016年。

[61] OpenBSD, “pledge(2),” https://man.openbsd.org/pledge.2, 2016年。

[62] T. Palit, F. Monrose 和 M. Polychronakis，“通过保护驻留在内存中的敏感数据来减轻数据泄漏”，在 *ACSAC*, 2019年。

[63] T. Palit, J. F. Moon, F. Monrose, and M. Polychronakis, “DynPTA: 结合静态和动态分析的实用选择性数据保护”，在 *S&P*, 2021年。

[64] V. Pappas, M. Polychronakis, and A. D. Keromytis,“使用间接分支跟踪的透明ROP漏洞缓解”，在*USENIX Security*, 2013年。

[65] PaX团队，“PaX ASLR（地址空间布局随机化）”，http://pax.grsecurity.net/docs/aslr.txt, 2003年。

[66] J. Pewny, P. Koppe, and T. Holz, “DOPed应用程序的类固醇：用于自动化数据导向编程的编译器”，在 *EuroS&P*, 2019年。

[67] A. Quach, A. Prakash, and L. Yan, “通过分段编译和加载减小软件体积”，在*USENIX Security*, 2018年。

[68] P. Rajasekaran, S. Crane, D. Gens, Y. Na, S. Volckaert, 和 M. Franz，“CoDaRR: 连续数据空间随机化抵御仅数据攻击”，在 *AsiaCCS*,2020年。

[69] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, 和 H. Bos，“VUzzer: 应用感知的进化模糊测试”，在 *NDSS*, 2017年。

[70] G. F. Roglia, L. Martignoni, R. Paleari, 和 D. Bruschi, “手术式返回随机化的lib(c)”，在 *ACSAC*, 2009年。

[71] A. Sadeghi, S. Niksefat, 和 M. Rostamipour, “纯调用导向编程 (PCOP): 使用调用指令链接小工具”， *JCVHT*, 2018年。

[72] E. J. Schwartz, T. Avgerinos, and D. Brumley, “Q: Exploit Hardening Made Easy,” in *USENIX Security*, 2011.

[73] E. J. Schwartz, C. F. Cohen, J. S. Gennari, and S. M. Schwartz, “A Generic Technique for Automatically Finding Defense-Aware Code Reuse Attacks,” in *CCS*, 2020.

[74] J. Seibert, H. Okhravi, and E. Söderström, “Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code,” in *CCS*, 2014.

[75] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A Fast Address Sanity Checker,” in *ATC*, 2012.

[76] J. Seward和N. Nethercote，“使用Valgrind检测未定义值错误与位精度”，在 *ATC*，2005年。

[77] H. Shacham，“骨头上无辜肉的几何：无需函数调用的返回到libc（在x86上）”，在 *CCS*，2007年。

[78] H. Shacham，M. Page，B. Pfaff，E.-J. Goh，N. Modadugu和D. Boneh，“地址空间随机化的有效性”，在 *CCS*，2004年。

[79] C. Song，B. Lee，K. Lu，W. Harris，T. Kim和W. Lee，“通过数据流完整性强制执行内核安全不变量”，在 *NDSS*，2016年。

[80] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim,W. Lee, 和 Y. Paek, “HDFI: 硬件辅助数据流隔离,” 在 *S&P*, 2016.

[81] E. H. Spafford, “互联网蠕虫程序: 一项分析,”*SIGCOMM CCR*, vol. 19, no. 1, 1989.

[82] M. Stamatogiannakis, P. Groth, 和 H. Bos, “深入黑盒子: 使用动态检测捕获数据溯源,” 在 *IPAW*, 2015.

[83] E. Stepanov 和 K. Serebryany, “MemorySanitizer: C++中快速检测未初始化内存使用的工具,” 在 *CGO*,2015.

[84] 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,” 在 *ASPLOS*, 2004.

[85] L. Szekeres, M. Payer, T. Wei, and D. Song, “SoK: Eternal War in Memory,” in *S&P*, 2013.

[86] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, “Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM,” in *USENIX Security*, 2014.

[87] V. Van der Veen, D. Andriesse, E. Gökta¸s, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, “Practical Context-Sensitive CFI,” in *CCS*, 2015.

[88] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, “The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later,” in *CCS*, 2017.

[89] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos,and C. Giuffrida, “一个艰难的选择：在二进制级别上缓解高级代码重用攻击”，在 *S&P*，2016年。

[90] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu,P. Larsen, B. De Sutter, and M. Franz, “安全高效的应用程序监控和复制”，在 *ATC*，2016年。

[91] D. Wagner and R. Dean, “通过静态分析进行入侵检测”，在 *S&P*，2000年。

[92] X. Wang, S. Wang, P. Feng, K. Sun, S. Jajodia, S. Benchaaboun, and F. Geck, “PatchRNN：基于深度学习的安全补丁识别系统”，在 *MIL-COM*，2021年。

[93] H. Ye, S. Liu, Z. Zhang, and H. Hu, “Viper: 检测数据攻击中的系统调用保护变量,” in *USENIX* Security, 2023.

[94] S. H. Yong and S. Horwitz, “保护C程序免受通过无效指针引用的攻击,” in *ESEC*,2003.

[95] W. D. Young and J. Mchugh, “为可信的规范到实现映射进行编码,” in *S&P*,1987.

[96] M. Zhang and R. Sekar, “用于COTS二进制文件的控制流和代码完整性: 一种有效的防御手段对抗现实世界的ROP攻击,” in *ACSAC*, 2015.

## 一个实现的离题

在这个附录中，我们讨论了我们添加和合并到我们的DTA引擎中的功能，以使其具有通用性和可扩展性。 我们基于libdft [47]实现了我们的DTA引擎，libdft是建立在Intel Pin [53]之上的DTA引擎，已经发布了10多年。

合并的特性。自从lib**dft**发布以来，许多分支版本已经出现，每个版本都添加了自己的特性以适应自己的特定分析[25, 69, 82, 88]。 不幸的是，没有一个单独的lib**dft**版本支持所有这些特性，因此我们将其中许多特性合并到一个通用的DTA引擎中，即我们的lib**dft**变体。 特别是，我们合并了以下支持：(1) 64位指令；(2) 加载指针传播，即在加载指令上，不仅传播加载数据的标签到输出，还传播指针的标签；(3) 使用自己的地址标记所有数据；(4) 与进程的内存映射接口，例如检查某个内存段是否可写入；以及(5) 统一的日志记录接口，支持多线程应用程序和重定向标准日志接口的应用程序（例如n**g**inx）。

添加的功能。我们还为libdft添加了几个新功能。
我们添加的第一组功能提供了可扩展性（如第5.1节所讨论）。具体来说，我们添加了：（1）基于userfaultfd的tagmap初始化，（2）基于有界数组的tagsets，以及（3）基于影子内存的tagmap。

图8描述了我们的影子内存实现的布局。 从高层次来看，从 MAIN_START到 MAIN_END的区域是主地址空间，从SHADOW_START到 SHADOW_END的区域是相应的影子地址空间。 由于从 MAIN_START到MAIN_START+RESERVED_BYTES的区域将映射到空影子内存，我们不能将其用于程序内存。 因此，我们保留这个区域来存储自定义标签。 例如，要污染一个文件读取（如Angora所做的[25]），我们可以为文件的每个字节在这个保留区域创建唯一的标签。 因此，通过支持自定义的、唯一的标签，例如为文件的每个字节、网络读取等创建标签，我们的影子内存为其他分析提供了一个表达能力强的接口。

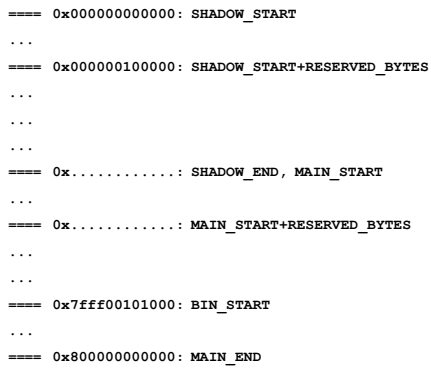



图8：影子内存布局。

影子内存布局假设目标程序是一个以位置无关方式构建的x86_64二进制文件（大多数编译器的默认设置），因此其堆栈和mmap_base位于地址空间的末尾。 接下来，布局要求我们重新链接目标程序，使其基地址位于BIN_START处。 布局中的RESERVED_BYTES和SHADOW_END的值取决于标签集中的颜色数量（即每个标签集的颜色数）以及每个颜色的大小（取决于是否启用指针标签压缩）。

我们添加的第二组功能提供了其他各种功能。 首先，我们添加了记录内存快照的支持。 我们通过在污染所有内存时调用gcore来生成核心转储来实现这一点。 由于影子内存占据了地址空间的相当大一部分，我们建议内核不要转储影子页面，以便核心转储文件不会太大。 此外，为了帮助我们的利用工具，我们在记录快照时还使用lldb来记录所有运行时符号信息。 其次，我们添加了一个接口来接收各种运行时信息。 我们使用这个接口来接收程序驱动器最近执行的行，以便我们可以将其包含在系统调用报告中，从而帮助我们的利用工具了解覆盖特定gadget链的工作负载。 第三，我们修复了许多指令级污染传播规则，例如：（1）考虑掩码操作的字节级语义（例如AND，OR）；（2）添加对各种矢量指令的支持（例如VZEROUPPER，PUNPCKLQDQ）；（3）正确处理各种x86-64的怪癖（例如零扩展32位MOV，但不是8位、16位或64位MOV）；等等。第四，我们更新了我们的libdft变体，使用最新版本的Pin并支持更新的系统调用（例如execveat，openat2）。

## B控制数据攻击表面比较

在这个附录中，我们将我们的非控制数据攻击表面与控制数据攻击表面进行比较。 为了做到这一点，我们重新实现了Newton [88]的污点策略（使用我们的污点引擎）来识别控制流劫持攻击，并重新运行我们的评估。 我们注意到我们的非控制数据小工具比控制数据小工具更强大，因为即使攻击者可以改变分支，攻击者仍然需要将其转向某个目标代码，例如，调用具有可控参数的系统调用。 相比之下，我们的非控制数据小工具已经是具有可控参数的系统调用。

表4列出了我们在每个目标程序中识别到的非控制数据小工具（即具有攻击者污染参数的系统调用）和控制数据小工具（即具有攻击者污染分支目标的间接分支），以及具有相同流的百分比。 就像以前的工作一样[88]，我们计算间接分支站点，而不是回溯；然而，计算回溯给我们带来了类似的结果。 我们根据分支目标的类型（即寄存器或内存操作数）区分控制数据小工具。

首先，我们观察到具有内存操作数的控制数据小工具比具有寄存器操作数的小工具具有更高的身份流率。 仔细观察后，我们注意到这是因为寄存器操作数往往更受限制，因为它们通常用于具有有限目标集的分支（例如，switch语句），而内存操作数通常用于具有相对无限目标集的分支（例如，间接调用）。 其次，我们观察到非控制数据小工具和具有内存操作数的控制数据小工具的身份流率大致相似。 这并不令人惊讶，因为程序将许多类型的长期存在的数据视为“不可变”，无论它是控制数据（例如，函数指针）还是非控制数据（例如，字符串）。

因此，我们得出结论，我们的轻量级基于身份流的分析适用于超出仅数据攻击的领域。