Introduction：

几十年来，研究人员一直在研究主机安全性。攻击的演变使得现有的方法无法满足现代复杂的场景。高级持续性威胁(APT)攻击由于其持久性、隐蔽性和明确的目标，被许多有经验的攻击者用于破坏受害者的主机。APT 攻击通常是由具有国家或组织背景的黑客组织发起的。因此，这些团体组织良好，目标明确，技术高超，具有很强的侵入性。火眼公司 2019 年年度报告显示，超过 20 个活跃的 APT 组织对数十个领域的目标发起了攻击，包括政府、金融公司甚至冬季奥运会.

传统的入侵检测方法可以分为离线和在线两类。最著名的脱机检测方法之一是沙盒方法，将目标程序部署到隔离的环境中进行单独的分析，此外，还构建了一些日志记录和来源跟踪系统来监视系统的活动，然后构建来源图来检测或分析攻击，虽然这些方法可以清楚地看到攻击，但考虑到离线检测的迟滞性，人们已经开始使用在线检测方法来实时检测攻击。这些方法包括基于网络流量的分析、软件静态特征检测和钩子技术等。然而，现有的研究主要集中在 APTs 的一个特定阶段，对其内在机制和攻击载体仍知之甚少。

近年来，基于内容的检测已被证明是有效的。基于上下文方法的实时检测系统在近些年来已经被提出。StreamSpot 分析流信息流图，通过提取局部图特征并将其向量化进行分类来检测异常活动。基于学习的检测方法只能提供恶意评分或分类结果，而不能解释这些结果。此外，这些检测系统无法在没有错误警报的情况下检测攻击。因此，在实践中，基于学习的方法并不适用于企业场景。随后，sleuths 提出了一种基于来源图的基于标记的检测方法，但该方法主要针对对机密文件的可疑访问，并通过添加域白名单来减少误报。为了更好地了解 APT 攻击，分析人员将 APT 生命周期解耦为多个阶段，然后使用每个阶段的相应特征来匹配可疑行为。APT 攻击分为 7 个或 11 个阶段。多相杀伤链模型被许多研究者采用。通过建立一个基于简单规则检测每个阶段并计算可疑分数的模型，Holmes 在基于阶段的检测方面取得了实质性进展。然而，这些阶段在 APT 攻击中并不都是必要的(例如，凭据访问)，其中一些阶段(例如，检测远程代码执行漏洞)通常是通过先验知识检测到的，并且往往会随着时间的推移而改变。

此外，实时上下文工作通常在来源图中保存上下文信息。然而，随着时间的推移，图会持续增长，APT 可以持续几个月甚至几年，这使得这些方法在系统长时间运行时不可避免地会遇到效率和内存问题，特别是实时检测。因此，大多数检测方法依赖于短时间窗口。

为了解决这些挑战，特别是准确性和效率，本文提出了一个准确检测 apt 的模型。此外，它提出了一种新的基于状态的检测框架，其中每个进程和文件都表示为一个精心设计的数据结构，用于实时、长期检测。

为了高精度地检测未知 APT，我们利用控制流(即为什么执行进程或代码)和数据流(即数据如何在对象之间传递)来解释上下文行为，而不是集中在 APT 攻击中不必要的、不可检测的和易于更改的阶段。我们确定了以下三个基本攻击阶段：1)部署并执行攻击者的代码，2)收集敏感信息或造成破坏，3)与 C&C 服务器通信或窃取敏感数据。我们主要专注于准确地检测这些阶段，并将它们组合起来，以区分恶意行为和良性行为。与更复杂的基于相位的模块相比，该方法有利于准确检测未知 apt。

为了实时高效地进行这种上下文检测，我们提出了一种新的基于状态的跟踪和检测框架，以及基于法医分析思想的相应数据结构。在这种设计中，所有的语义都被存储为状态，框架只保留所有进程和文件的当前状态以供检测。状态由事件和其他实体的相关状态更新，类似于有限状态自动机(FSA)，我们称之为类 FSA 结构。因此，框架不需要存储历史数据，并且内存使用保持一致。状态随时间变化。一旦进程进入恶意状态，无论持续多长时间，攻击都会被检测到。使用该框架，我们可以长时间监控主机，以高精度和低开销自动检测 apt。

此外，我们基于该框架的检测方法可以检测攻击并提供解释。具体来说，检测结果是用重建的攻击图生成的，它说明了这些攻击是如何发生的，并有利于后续的分析。

最后，我们实现了我们的设计，叫做 Conan。Conan 从 Windows 收集数据，提取语义，然后使用智能策略进行状态传输和实时检测 apt。此外，一旦检测到攻击，它可以自动重建部分攻击链。我们在三个场景下评估 Conan：DARPA 的参与、我们的实验室和三家现实世界的公司。结果表明，Conan 可以快速检测出我们实验中所有潜在的 APT 攻击，错误检测率接近于零，并准确地重建攻击图。随着时间的推移，Conan 的内存使用保持不变(1-10 MB)，这与以前基于不断增长的来源图的设计不同。我们将我们的贡献总结如下：

我们提出了一种新的 APT 检测模型，它集中在 APT 的三个步骤上，我们提出了一套精确跟踪和检测这些步骤的设计，包括检测基于内存的攻击和可疑进程行为。

我们提出了一种新的高效的基于状态的检测框架，其中每个进程和文件都表示为一个类似于 FSA 的结构。该框架有助于以恒定和有限内存使用(1-10 MB)和高效率(比数据生成快数百倍)来进行 APT 的检测。

我们设计了 APT 攻击检测系统，能够实时、高精度地检测未知高级攻击，并快速恢复攻击链。我们在真实世界的数据集上测试了我们的系统，并确定它比以前的方法表现得更好，特别是在效率和准确性方面。

本文所使用的原子可疑指标(ASIs)、转移规则和恶意状态与评估中使用的相同。虽然 Conan 可以检测有 DARPA 参与中出现的所有攻击，但我们并不打算证明这些定义足以适用于所有攻击；相反，我们想要展示我们的设计，以准确有效地检测和重建 apt。此外，可以通过添加更多数据源轻松扩展 Conan，因为 ASIs、规则和恶意状态可以在配置文件中自定义。

A living example and threat model：

在本节中，我们给出一个简单的 APT 攻击示例，以说明现有检测方式的缺点。这种攻击始于网络钓鱼电子邮件，因为根据 PhishMe 的研究，91%的情况下，网络钓鱼电子邮件是成功网络攻击的幕后黑手。

想象一下，你正在电脑上工作，收到一封来自同事的电子邮件，要求你填写附件中的表格。你下载了附件，没有收到防病毒软件的警告，打开它，填写，然后回复发件人。你可能会在几天后忘记这封邮件，但几个月后，你发现一些敏感数据被竞争对手窃取了。

此外，这种袭击可能更加复杂。例如，攻击者首先准备一个新的远程访问木马(RAT)，它永远不会被防病毒软件所发现。然后，他收集关于你公司的信息，这样他就可以模仿员工给你发送电子邮件的方式。附件包含在打开文件时执行的模糊宏脚本。它执行 PowerShell 命令下载并在内存中运行准备好的 RAT，从而在不将恶意文件留在硬盘上的情况下访问您的计算机。攻击者可以潜伏很长一段时间，直到他得到他想要的东西，例如敏感文件或您帐户的密码。

在这里，基于网络流量的分析可能由于伪装和加密而无效。基于沙盒的检测可以通过反沙盒技术而无效。静态恶意软件分析无法工作，因为磁盘上没有恶意软件，甚至内存中的恶意软件也是全新的，没有记录任何静态特征。

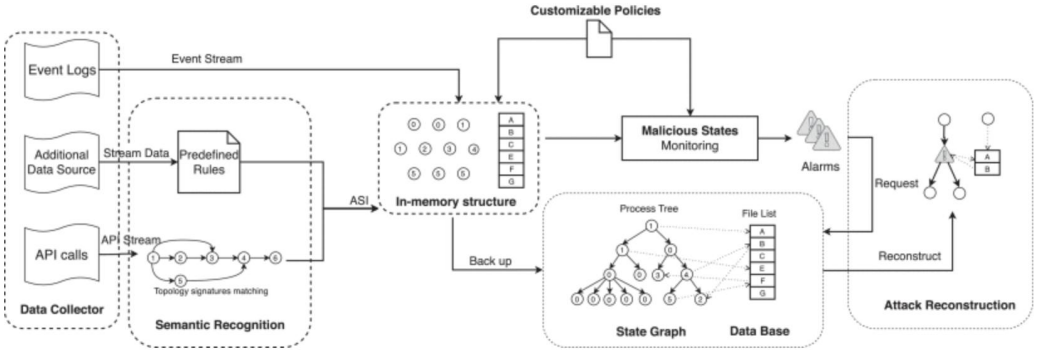
在我们的威胁模型中，攻击者非常了解他的目标，并准备新的攻击，包括零时间暴露、自行开发的恶意软件和新的 DNS。他首先让他的恶意代码在受害者的机器上运行，然后自

动或通过接受来自网络的命令执行一些恶意操作来收集信息或造成损害。最后，如果他想获得敏感信息，这些被盗的数据应该通过网络发送。虽然我们部署该系统是为了检测 APT 攻击，但它也可以检测到具有类似目标的一般攻击。但是，我们无法检测到侧信道攻击和内部攻击，在这些攻击中，攻击者有合法的方式访问机器。Conan 无法检测到的另一种攻击是面向返回的编程(ROP)攻击。使用这种技术，攻击者获得了对调用堆栈的控制，从而劫持程序控制流，然后仔细地执行已经存在于机器内存中的选定的机器指令序列。因此，在受害者的机器上没有部署恶意代码。然而，这种攻击已经可以通过现有的方法避免或检测到。

在本文中，我们假设事件日志和数字签名是可信的。评估中使用的所有攻击都无法被传统的反病毒系统检测到。相反，我们不需要假设整个攻击发生在系统安装之后。因此，可以检测到预安装的恶意软件和恶意代码。

Detection model：

在本节中，我们首先提供如图 1 所示的设计概述。我们在主机上开发了一个快速稳定的多级数据收集器，用于收集审计跟踪、调用堆栈和附加数据。然后，这些数据被发送到检测服务器，提取为高级语义，并作为进程和文件状态存储在内存结构中。同时，根据预定义的规则对所有事件日志进行处理，以改变进程和文件的状态。这些事件和状态存储在数据库中，以备以后的攻击重构。每当进程进入恶意状态时，就会触发警报以及重建的攻击图。



数据收集器部署在客户端，它试图以较低的开销从多层收集跟踪。这些跟踪在客户端进行预处理，以减少数据大小，然后发送到服务器端的检测器。检测器根据预定义的规则构造一个主存结构来维护进程和文件的状态。状态和必要的事件同步到数据库。一旦触发警报，就可以从数据库中重建攻击链。

本文提出的检测模型使系统能够准确检测 apt，而基于状态的框架使高效检测 apt 成为可能。我们将分别在第 3 节和第 4 节介绍这两个部分。

（1）Motivation：

APT 中的字母 A 代表这些攻击中使用的先进技术。传统的恶意软件检测、漏洞检测、威胁情报等检测系统只关注 APT 攻击链的单个阶段，且这些阶段所使用的攻击技术容易改变;因此，APT 攻击者很容易逃避这些传统的检测方法。MITRE ATT&CK 介绍了一个 11 阶段 APT 攻击模型来描述 APT 攻击中使用的战术、技术和程序。

然而，过于复杂的多相模型只能用于更好地理解 apt，而不能用于检测它们。例如，作者开发了一个检测每个战术(阶段)中的技术的系统，并将这些阶段通过信息流(包括数据流和控制流)连接起来作为攻击链。在每个攻击链中，检测到的阶段越多，攻击的可能性就越大。

然而，有三个主要问题。首先，每个阶段都使用了数百种技术。该系统需要检测数百种技术，实现难度大，检测开销大。其次，仍然考虑了传统方法的检测点，如漏洞。要检测这些阶段，需要有先验知识。此外，攻击所使用的技术很容易改变；因此，很难检测到未知的攻击。最后，作者认为，虽然他们不能准确地检测到所有阶段，但作为所有阶段的子集，检测到的阶段足以区分攻击和良性活动。换句话说，没有必要检测所有阶段。此外，引入一些常见和不必要的阶段会增加误报率。

更多的阶段不能表明攻击，更少的阶段不能证明其合法性。例如，新安装的浏览器可以触发 MITRE ATT&CK 11 阶段模型中的 6 个阶段(初始访问、执行、持久性、凭据访问、发现和泄露)，并通过以下方式触发错误警报。同时，高级攻击者通过零时间暴露访问机器并下载恶意软件。然后，它使用一种未知的方法实现持久性(或者在某些场景中它不需要持久性，例如在永不关闭的服务器上)。它记录键盘的输入，并通过命令和控制通道泄露数据，这可能很难被发现。最后，在这种攻击中唯一可以检测到的阶段是执行和收集，因此不能将它们视为攻击。

(2) Three-phase Detection Model:

为了检测未知 APT 攻击，我们首先找到它们的不变部分。换句话说，我们试图回答以下问题：是什么使这些活动具有“攻击性质”。在研究了数百种 APT 攻击后，首先可以观察到攻击者必须首先将其代码部署到受害主机上。不同之处在于，恶意软件可能只在内存中定制或执行，以逃避传统的基于静态文件的检测系统。第二个观察结果是，多年来，攻击者的最终目标保持不变。自 2006 年以来，APT 发起攻击，从至少 141 个组织窃取了数百 tb 的数据。如今，APT 也专注于类似的任务。这些行为类似于 Android 中的权限，可能会导致隐私泄露或损害。最后观察到的是，攻击者将与 C&C 服务器通信并窃取机密数据，恶意程序应该始终具有访问网络的能力。

为此，我们提出了一个检测 APT 的三相模型：

部署并执行攻击者的代码。任何流程行为都是代码执行的结果。攻击者必须首先向受害者部署代码以实现其目标。为了检测这个阶段，我们监控来自外部的数据流，包括网络和便携式设备，我们称之为不可信数据流。无论攻击者使用什么漏洞或技术来部署他的代码，拥有不受信任的数据流都是发起攻击的必要条件。这种设计可能会导致更多的误报，但是后面介绍的技术将有助于解决这个问题。在另一种场景中，攻击者可能使用合法进程来达到其目的。例如，攻击者可以使用预先安装的 Windows SnappingTool 捕获屏幕。在这里，不可信控制流可以提供帮助。如果进程或线程由可疑线程启动，则该进程或线程也被标记为可疑。代码部署总是必要的，除非攻击者可以通过其他方式获得对受害者的授权访问（通过密码远程登录)。这种类型的攻击可以通过 IP 白名单来阻止，通过异常检测来检测，这不在我们的研究范围之内。

收集敏感信息或造成损害。攻击者通常试图窃取机密数据或破坏受害者的数据或机器，这就是攻击者实施攻击的原因，也是受害者希望避免此类结果的原因。我们不认为能够接触到受害者但不会导致任何有害行为的入侵是真正的攻击。从文件中窃取机密数据可以通过监视来自机密目标的数据流作为机密数据流来检测。其他可疑行为由我们的预定义签名检测，如第 4.1 节所述。

与 C&C 服务器通信或窃取敏感数据。这两种操作在 APT 中都是必要的，没有这两种操作，攻击就无法完成。虽然有很多方法可以实现这一点(例如，可移动磁盘)，但典型的

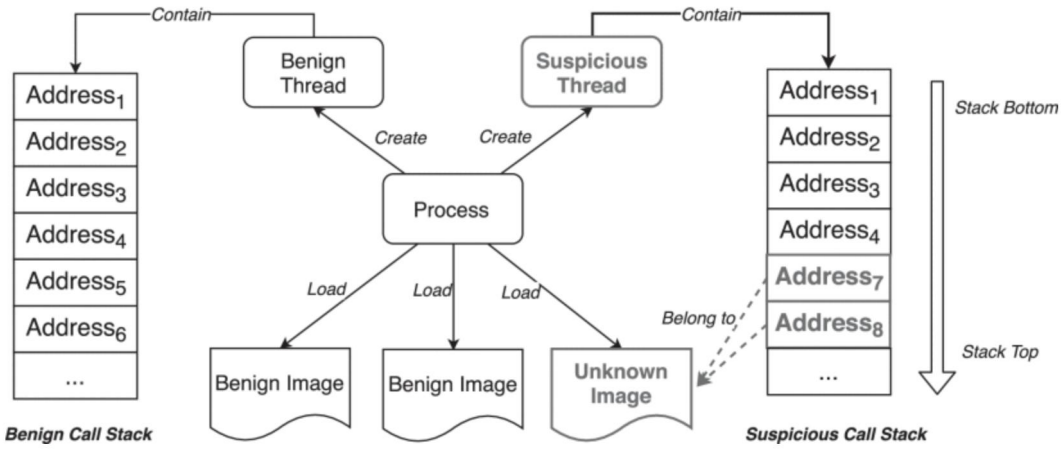
现实攻击方法是通过网络连接。

这三个阶段对于大多数 APT 攻击来说都是简单且必要的。因此，我们尝试检测执行可疑代码以执行恶意行为的进程。此外，我们还提供了额外的特征来说明不同的攻击场景。与以前的工作不同，我们没有分配分数或简单的标签；相反，我们使用更详细的描述来描述攻击的组件，以便在不增加额外开销的情况下更好地理解和进一步分析。

（3）Tracking Suspicious Code Execution

基于内存的攻击，包括注入和无文件攻击，可以帮助攻击者在良性应用程序的内存中执行他们的代码，这种攻击手段被越来越多地使用，因为传统的检测系统对它们是无效的。虽然现有的研究将流程视为存储上下文信息的实体，但它们很容易受到基于内存的攻击。在基于内存的攻击中使用的检测技术可能会有所帮助，但实现这些攻击的方法有多种，这使得检测变得困难。因此，在本节中，我们提出一种通过跟踪可疑数据流和检查执行调用堆栈来检测可疑代码执行的方法，忽略攻击使用的技术。

如果攻击者想要执行恶意行为，他必须 1)直接执行恶意代码或 2)借助良性进程来实现。检测的主要挑战是确定 1)哪些代码被执行，2)它来自哪里，3)它是如何执行的。尽管在跟踪细粒度数据流时，污点跟踪有很大帮助，但由于其高开销，这种方法不能实时使用。为了解决第一个挑战，我们采用了调用堆栈。调用堆栈是一种堆栈数据结构，用于在生成事件时存储有关活动子例程的信息。调用堆栈中的地址是属于不同代码块的返回地址(例如，图像)。如果一个调用堆栈中的所有地址都来自可信代码块，则调用此线程执行可信代码。由于良性进程可以很容易地强制执行外部代码，因此我们根据代码执行将一个进程分成多个子单元(线程)。如图 2 所示，这些执行未知或可疑代码的线程与那些完全良性的线程是分开的。我们考虑以下场景：



执行无符号代码的可疑线程(红色)与完全正常的线程(红色)是分开的。该特性对于检测可疑的代码执行非常重要，包括无符号图像和内存攻击。

图像加载和内存执行。映像加载是进程将可执行文件加载到内存中的基本操作。攻击者可以用恶意图像文件替换良性图像文件，或者迫使良性应用程序加载恶意图像，然后以良性进程为幌子进行攻击。此外，攻击者可以将恶意代码直接写入另一个进程的内存空间，或者从内存而不是从磁盘加载映像，而不会触发系统事件。前者被称为进程注入，通常用于攻击，后者是一种称为反射加载的技术，在最近的高级攻击中使用。我们的系统监视加载图像的动态事件，并为每个进程存储基址和已分配内存的大小。当无符号图像的内存地址或已分配内

存出现在调用堆栈中，这意味着在该线程中执行了一些未知代码;因此，这个线程应该与其他线程分开。为了减少解析完整调用堆栈时的开销，我们对每个线程进行低频率的抽样检查。

脚本执行。基于脚本的攻击近年来变得很常见，因为主机进程是完全良性的，进程读取执行代码，比如 PowerShell 和 VBscript，而不是加载它。探测此类攻击具有挑战性。我们的系统列举了大多数常见的脚本主机，并将它们与正常进程分开处理。

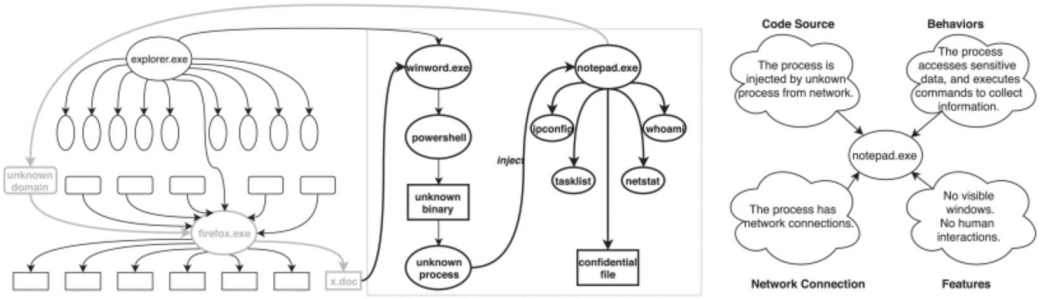
为了解决第二个挑战，我们通过基于粗粒度数据流的推断来跟踪可疑数据。例如，如果一个进程有一个网络连接，那么这个进程写入的任何文件都可能包含来自网络的数据。这些推理规则将在第 4.3 节中介绍。粗粒度数据流可能会导致较高的假阳性率，但真阳性率保持不变。第 5 节中的结果表明，即使是粗粒度的数据流，我们的系统也可以以较低的假阳性率检测 APT。

为了解决第三个挑战，我们跟踪控制流(特别是由可疑线程创建的进程)。虽然它们可能是良性进程，但它们可以用来实现攻击者的目标，例如抓取屏幕和窃取敏感数据。

State-Based Framework：

APT 中的字母 P 代表 Persistence，意思是攻击者可以潜伏很长时间，直到得到他想要的东西。这与攻击链模型中的 Persistence 不同，后者表示在操作系统重新启动后使恶意软件自动启动的技术。检测持久性技术是不实际的。一方面，已知的持久性技术有 59 种，检测它们的成本很高。另一方面，即使检测到某个进程是持久性的，也不能将其视为恶意软件。攻击者可以潜伏很长一段时间而没有任何可疑的行为。此外，恶意文件可能会在下载后几天被打开。因此，基于上下文信息很难检测攻击。

为了检测这类攻击，系统事件日志需要长期存储，每天需要占用几 gb 的硬盘空间。一些研究旨在减少日志大小，但这样的措施只能缓解这个问题，因为攻击可以持续数年，数据应该存储很长时间。此外，通过查看这些日志来识别相关事件也需要时间。因此，来源图(也称为依赖关系图或信息图)通常用于更快地遍历日志。基于来源图的实时检测系统通常将其存储在内存中，以便在计算和图匹配方面获得更好的性能，但图会随着时间的推移而不断增长。由于 apt 可以在没有可疑行为的情况下长时间处于休眠状态，因此这些方法存在与存储不断增长的图相关的内存问题，以及与跟踪长期攻击相关的效率问题。如图 3a 所示，对于长期攻击，几乎不可能实时执行这种类型的检测。



(a) A provenance graph of an attack.

(b) Aggregate context information to one process in our approach.

在本节中，我们将介绍一个基于状态的框架。在这个框架中，流程和文件的每个实例都类似于一组自动机，它允许我们通过聚集图 3b 所示的每个流程中用于帮助检测的所有上下文信息，以低开销实时检测不同的攻击场景。首先阐述了语义识别、数据结构、状态转换条件和恶意状态的概念。接下来，我们将解释基于框架重构攻击的方法。

（1）Semantic Recognition：

我们的语义状态定义是受到法医学分析的启发；我们自动识别数据流、控制流和流程行为的高级语义。这些语义表示在基于上下文的检测中使用的基本证据。我们称这种语义为原子可疑指标(ASIs)。

ASI 包括以下类型的语义之一，并通过相应的跟踪来检测或推断：

攻击者为达到目标而执行的行为。通常由 api 检测或由机密数据流推断。

可疑代码的来源，换句话说，进程可以执行此类行为的原因；由不可信的数据流推断。

进行外部通信的能力，由网络活动推断。

由不受信任的控制流推断执行流程的原因。

每 ASI 都可以被描述为一个三元组：〈No,Ty,De〉。每个 ASI 都被分配了一个唯一的数字 No，它表示它在位图中的位置，以记录状态。Ty 表示类别，其中包括以下内容:1)可疑代码源，以跟踪潜在的不可信代码执行;2)可疑行为;3)网络连接;4)特征，是用来说明不同攻击场景的附加特征。De 表示用人类可读的语义解释检测结果的描述。

这些 ASIs 是通过直接从源数据提取或通过规则推断(第 4.3 节)来识别的。如果 1)新的 ASI 与现有的 ASI 具有不同的语义，或者 2)有多种方法可以以不同的精度识别相同的语义，则应该声明一个可以帮助检测 apt 的新 ASI。表 1 列出了一组选择性的 ASIs。不同 ASIs 的组合最终可以描述不同的攻击场景。下面的检测步骤基于这些 ASIs、数据流和控制流。

Number	Type	Description
P1	Net.	Network traffic.
P2	Beh.	Access sensitive files.
P3	Beh.	Audio recorder.
P4	Beh.	Keylogger.
P5	Beh.	Execute sensitive commands.
P6	Beh.	Grab screens.
P7	Beh.	Steal Windows credentials from memory.
P8	CS	Injected by untrusted code.
P9	CS	Load unsigned images.
P10	CS	Load unsigned images from the network.
P11	CS	Execute scripts from the network.
P12	CS	Find unknown code in call stacks.
P13	CS	Executed by suspicious threads.
P14	Fea.	No human interactions.
P15	Fea.	Ancestor process has network connections.
P16	Fea.	The process has visible windows.
P17	Fea.	Access data from the network.
F1	-	The file contains data from the network.
F2	-	The binary file does not have a certificate.
F3	-	Contain data from malicious behaviors.
F4	-	The file is sensitive.

一些 ASIs 可以很容易地检测到，或者通过基于系统事件日志的推断来生成。虽然攻击者通常执行的行为是最重要的 ASIs 之一，但目前还没有成熟的方法来检测它们。

为了应对这一挑战，我们对具有此类 ASIs 的现实世界恶意软件进行了最大规模的研究，称为远程访问木马(RAT)，涉及 500 多份白皮书和 50 多个活跃在过去十年的 RAT 家族。结果表明，rat 通常存在了几十个 ASIs，这些 ASIs 的实现基本相同。我们开发了在不同层的数据源上检测它们的方法。为了平衡准确性和效率，我们手动生成了一组拓扑 API 签名来识别这些基于代码实现的行为(例如，如图 4 所示;箭头表示 api 之间的数据流，因此，行为必须

按照这样的顺序实现)。API 签名的覆盖率和真阳性率很高(90%)，但它们可能不明确。例如，用于截屏的 api 也用于在窗口中绘制图片;甚至序列也保持相同(对于这两种用途，这些 api 用于将设备上下文的内容从一个复制到另一个)。为了解决这一问题，我们采用了一些外部信息来提高准确性(例如，我们发现抓取屏幕的 API 序列也可以用于绘制窗口，我们的解决方案是:当一个进程正在抓取屏幕，而当时没有可见窗口时，我们可以将其视为截图操作而不是绘图操作)。为了实时高效地匹配这些拓扑 API 签名，我们将这些拓扑转换为 fsa，将签名与流数据进行匹配，这些 API 需要在一个窗口中进行匹配，以减少误报。在实践中，我们采用 6 秒的时间窗口。我们不使用系统调用，因为它们的级别太低，无法反映语义。

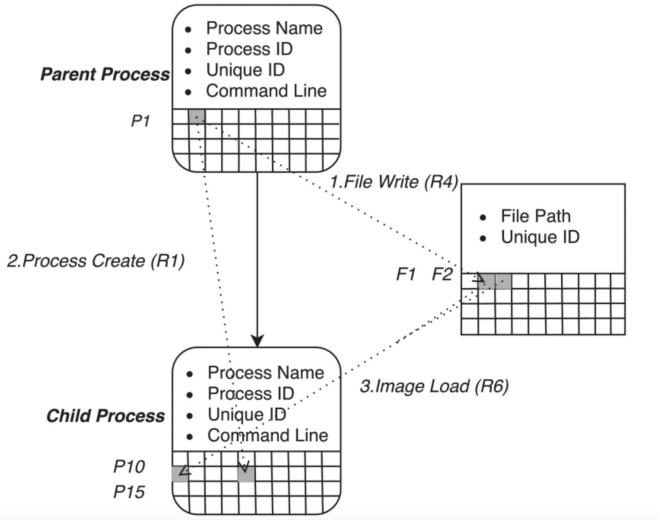
一个简单的截屏签名示例。同一框架中的 api 是实现类似功能的替代方案。签名是基于其代码实现生成的，序列是基于 api 之间的数据流。

API 调用通常通过沙盒 API 挂钩来记录，由于性能和稳定性较差，无法用于实时检测系统。我们使用 ETW 内核调用堆栈跟踪来恢复 API 调用。ETW 可以捕获内核事件(包括 SysCallEnter 事件，它表示对系统调用的调用)及其调用堆栈。我们可以有效地从调用栈中恢复 api。然而，这个过程不是我们的主要贡献，我们在这里不详细描述它。

(2) Data Structure:

为了支持实时分析和长期监控，我们提出了一个类似于主存 fsa 的结构来记录可能涉及攻击的每个进程和文件的状态。请注意，我们不需要存储任何历史事件来执行检测，但是有了重建攻击的额外目标，我们只保留了一小组使状态更改到数据库的事件。

如图 5 所示，我们将每个进程和文件处于特定状态时的基本信息和状态保存在内存中。每个流程实例都可以描述为⟨Na,Pi,Cl,Ui,St⟩，其中包含流程的基本信息。Na 为进程名，Pi 为进程号，Cl 为命令行。由于 Na 和 Pi 可以重复，我们为每个实例分配一个唯一的 ID (Ui)。St 表示该进程的状态。



数据结构和操作。标签 P1, P10, P15, F1，F2 为表 1 所示状态。R1、R2 和 R15 是表 2 中描述的规则。这个例子记录了一个下载驱动攻击的过程。

每个文件实例都是一个三元:⟨Na,Ui,St⟩。Na 代表文件路径，Ui 是唯一 ID, St 是状态。状态是预定义的，如第 4.1 节所述。一旦实例包含特定的状态，相应的位就被设置为 true。

所有不活跃的进程和文件都从内存中删除到数据库中，以确保内存是恒定的。只有当文件被其他进程操作时，文件才会被恢复。

(3) State Transition:

我们的方法基于这样一种见解，即相同类型的事件具有不同的高级语义，这取决于所涉及的主题和对象之间的差异。例如，读取下载的文件与读取个人目录中现有的文件是不同的。前者涉及访问未知数据源，可能导致不可信的代码执行，而后者涉及访问个人数据，最终可能导致用户数据泄露。此部分的目的是跟踪机密数据流、不受信任的数据流和不受信任的控制流。这种分析通常由法医分析人员进行;但是，我们的系统会自动执行这个分析。

为了自动区分这些事件并记录语义，我们创建了一组预定义的规则，为事件分配更详细的语义。表 2 中有一组选择性规则。

No.	Subject ASI	Event	Object ASI	Direction	Description
1	P1/P15	start	P15	forward	This process is executed by a process with network connections.
2	[Beh.]	write	F3	forward	A process writes data acquired through malicious behaviors to a file.
3	P17	read	F1	backward	A process accesses data from the network.
4	P1/P17	write	F1	forward	A process writes data from the network to a file.
5	P2	read	F3/F4	backward	A process accesses a sensitive file.
6	P10	load	F1&F2	backward	A process loads a downloaded image with no valid certificate.
7	[CS.]	start	P13	forward	One process is executed by a suspicious process.

每条规则都是六元组:⟨No,Ss,Ev,So,Di,De⟩。No 是规则的序列号，在边中使用，表示如何生成此操作。Ss 代表一个主题的特定状态。在我们的设计中，主体始终是一个过程。表示对象的特定状态。对象可以是进程、文件或 IP。Ev 是主体对客体执行的事件。Di，向前或向后，表示一个实体影响另一个实体的方向。当主体处于某种状态并对一个对象执行事件时，对象的状态在我们所说的前进方向上发生变化，其中主体和对象分别是源和目的。反之，如果主体受客体的影响，我们称它为落后；在这种情况下，主体是目的地。注意，当 Ss 和 So 是源时，它们都可以是一个或多个状态。De 是对规则目的的描述，用来解释重构后的攻击链。

我们系统中的每个实体(即主体或对象)就像一个 FSA，可以被描述为五元组:⟨S，Σ，δ，S0,F⟩。

S:状态集。St 中的位的组合，表示进程和文件的当前状态。

Σ:输入字母。由系统事件 Ev 组成。

δ:状态转换函数。

$$\delta : S_f \times \Sigma \rightarrow S_l. \tag{1}$$

与传统的原始 FSA 不同，状态 Sf 已经存在于一个实体中，状态 Sl 是在这个 Ev 中涉及的另一个实体中新生成的。

S0:初始状态。一旦一个新的进程或文件出现在我们的系统中，我们就在内存中创建一个相应的实例。St 中的所有位都设置为 false。只有可能包含机密数据的文件才使用状态 F5 初始化。

F:最终状态的集合。一旦进程进入其中一种状态，它将触发警报。这些恶意状态将在第 4.4 节中讨论。

如图 5 所示，由于具有网络连接的进程向该文件写入数据，因此推断该文件是从网络下载的。然后，一个新进程加载这个下载的文件，结果，该进程变为表示此语义的状态。换句

话说，状态包括前一个对象状态和此事件的语义。因此，在我们的系统中，我们不需要存储任何历史事件，并且可以有效地检测攻击。

但是，为了更好地理解和进一步分析，我们将导致状态变化的事件存储到数据库中。事件存储有四个属性:规则号、时间戳、源和目标。规则号表示改变状态的原因。事件就像起源图中的一条边，但是操作的源和目标是位图中用来记录状态的位，而不是进程或文件。我们将在第 4.5 节讨论这种设计。

第 5.8 节的结果表明，现有的规则已经覆盖了 95%以上的原始事件，只有不到 1%的规则存储在数据库中用于攻击重构。

请注意，虽然我们可以获得读取和写入文件的偏移量和长度，但我们将文件视为一个整体。这种简化可以显著减少开销。

（4）Malicious States：

恶意状态是各个状态的各种组合，指示检测所需的上下文信息。如 3.2 节和 4.1 节所述，ASIs 可分为 4 种不同类型:可疑代码源、网络连接、可疑行为和特征。如果一个进程至少包含前三类中的每一类中的一个 ASI(不包括特征)，我们就说它进入了恶意状态。每种恶意状态说明了不同的攻击场景。例如，如果一个进程加载了从网络下载的未签名映像，执行了恶意行为并连接到网络，我们将其识别为“下载&执行”攻击，并根据其执行的恶意行为知道攻击者的目标。如果主机中已经存在无符号映像，则可以将其识别为“现有恶意软件”。因此,我们不需要假设攻击的所有阶段都发生在 Conan 开始监控系统之后,我们认为这是 Conan 对现有作品的一个改进。

一旦进程进入恶意状态，系统就会发出警报。换句话说，所有的检测过程都可以通过简单地检查一个进程的状态来执行。例如，这样的检查可以揭示进程是否正在执行来自网络的无符号代码。我们不需要知道代码的确切来源，因为这些信息对检测提供的帮助很小。此外，由于我们的系统只检查进程状态，因此它的开销比基于图的检测机制要低得多，但达到的效果几乎相同。

我们还利用多种通用特征来识别不同的攻击场景。例如，“人工交互”特性反映了进程是否自动运行，而“无可见窗口”特性表明用户是否可以明显地识别该进程的存在。特征越多，恶意行为的置信度越高，这代表了不同的攻击场景。使用更多的特性将有助于系统管理员分析攻击并减少误报。请注意，除了代码认证，我们不使用任何文件、过程或域的白名单。在系统部署之前安装的恶意软件也可以作为一种特殊的攻击场景被检测出来。

（5）Attack Reconstruction：

来源分析极大地帮助理解和检测攻击。因此，我们不仅检测这些恶意攻击，而且尝试用语义重构这些攻击，类似于来源分析函数的方式。这样的重构极大地有助于攻击分析，进一步减少误报，并帮助保护主机免受未来的攻击。

由于数据结构的特殊性，可以有效地执行这些任务。由于我们将来自目标进程的所有证据汇总为状态，因此重构攻击的基本思想是解释为什么该进程被归类为恶意的，具体来说，就是通过回溯该进程中 ASIs 的来源。由于我们的系统将每个 ASI 的来源保存在数据库中，作为状态之间的边(见第 4.2 节)，通过沿边的反向跟踪，可以在线性时间内找到攻击的来源。对于正向分析，由于我们的系统可以立即检测到恶意进程，因此可疑进程几乎不会造成额外的影响;因此，不存在前向跟踪的依赖项爆炸。

在实践中，具有依赖关系爆炸的重构图对进一步分析的贡献较小。为了获得图，我们只重建了足够的证据来证明这是一次攻击，而不是试图重建整个攻击，因为跟踪粗粒度数据流的攻击是一个未解决的研究问题，已经研究了多年。

Evaluation：

我们使用三种不同的长时间运行环境来评估 Windows 上的 Conan。结果表明，Conan 能够以较高的精度和较低的开销检测多种类型的攻击。

（1）Implementation：

我们采用事件跟踪 Windows (ETW)作为主要的数据提供者。ETW 是 Windows 自带的审计系统，可以提供 1000 多种审计日志，包括系统调用、调用栈、应用级日志等。我们通过添加数据源来扩展 ETW，例如二进制文件的证书、人工交互和剪贴板。数据收集器是用 c++实现的，由大约 9.2 万行代码(KLoC)组成。其余组件是用 Java 实现的，大约由 6.2 KLoC 组成。所有状态、规则和恶意状态都可以在大约 112 行的配置文件中定制和指定。

（2）Datasets：

我们在三个环境中检查我们的系统:我们的研究实验室，DARPA Engagement 和几个现实世界的企业。表 3 总结了前两种环境中使用的数据集，我们将单独讨论现实世界中的大规模评估。

Dataset	Duration (hh-mm-ss)	File Read (Original)	File Read (Optimization)	File Write	File Create /Delete/Rename	Process /Thread	Loadlib	Network	Other	Total # of Events	Call Stack
D-1	8:50:25	45.63%	0.63%	29.60%	15.10%	0.91%	0.64%	7.99%	0.14%	10.66M	107M
D-2	8:48:10	43.33%	0.60%	27.43%	17.00%	0.93%	0.63%	10.65%	0.02%	9.96M	136M
D-3	6:53:40	46.92%	0.38%	25.18%	18.20%	0.87%	0.54%	8.27%	0.02%	6.13M	175M
L-1	3:27:48	20.00%	0.56%	21.62%	24.27%	0.61%	2.65%	30.6%	0.17%	102.5M	201M
L-2	284:36:05	34.40%	0.18%	2.41%	1.73%	0.51%	2.15%	58.75%	0.04%	100M	10B

每个活动的数据集，包括不同事件的持续时间和分布，以及事件和调用堆栈的总数。

前 3 行对应于 DARPA 透明计算(TC)计划的一部分，由红队进行的攻击活动。这个集合的时间跨度为 23 个小时，包括大约 30 M 个事件和 400 M 个调用堆栈。表的后两行分别对应于我们研究实验室收集的攻击和良性数据。

这些数据是由我们在 Windows 上收集的。表 3 中的“duration”列指的是收集器在目标计算机上运行的时间长度。注意，持续时间既包括主机上的良性活动，也包括与攻击相关的活动。接下来的几篇专栏文章将事件分解为不同类型的操作。文件读写不仅包括文件读写，还包括一些附加的数据流，如剪贴板。在每个数据集中，原始文件读事件的数量接近 40%，但通过以下优化，这个数字减少了大约 100 倍:当一个进程读取一个文件时，收集器忽略两个相邻写事件之间的重复读事件，因为这个文件的状态在写入之前不能改变，因此，进程不会受到这些读事件的影响。进程/线程列包括进程和线程开始/结束事件。网络只包含 TCP 和 UDP 包，因为 ETW 不能提供底层网络事件。“Other”列包括我们为更好地理解攻击而收集的功能，包括图像证书、可视窗口和人机交互。L-2 中的网络事件数量非常重要，因为用户在测试期间观看了许多电影。

（3）Environments and Experimental Setups：

我们的实验包含三个场景。

实验室的设置。我们在实验室的两台主机上部署了 Conan，并在一台主机(L-1)上实现了图 3a 所描述的攻击，此外还下载并安装了一组具有类似行为的良性应用程序，包括即时通讯工具(例如 Skype)、远程访问工具(例如 TeamViewer)、图像编辑器(例如 Photoshop)和其他(例如 PuTTY)。同时，我们在另一台主机上运行我们的系统数周，以确保其长期稳定性和低假阳性率(L-2)。

DARPA 参与下的设置。我们评估中的攻击场景配置如下。有三台主机安装了 Windows 10。由于设置涉及对抗性交战，我们对红队准备的攻击没有事先的了解:我们不知道攻击何时发生，也不知道攻击目标是什么。值得注意的是，红队在攻击目标主机的同时，也在对主机进行良性的后台活动。这些活动包括浏览网站、下载二进制文件并执行它们、读写电子邮件和文档。总体而言，近 99.9%的事件与良性活动有关。因此，任务是在大量良性事件中实时自动识别攻击。

在这次交战中，我们检测到所有攻击，没有误报。因为在这些场景中，测试团队试图模拟一个严格的业务或政府环境，所以没有实现那些可能产生误报的行为，例如未签名的应用程序安装。重构结果不能覆盖攻击产生的所有活动。在下一节中，我们将把我们的部分结果与红队发布的真实数据进行比较。

真实的企业设置。我们被允许将我们的系统部署在三家现实世界的公司中进行长期运行，以验证 Conan 的稳定性和效率。为了安全起见，所有部署的机器都在他们的办公网络中，而不是在他们的业务网络中。所有的机器都可以上网，办公时间由员工操作。网络和机器还受到其他安全产品的保护，如反病毒软件和防火墙。准确性、稳定性和效率都可以在这样的场景中检验。

（4）Attack Scenarios and Reconstruction：

在本节中，我们将详细介绍一个活动的检测结果，以说明 Conan 如何检测这些攻击。其他结果将在附录中讨论。表 4 列出了所有特定状态下的检测结果。

DataSet	Process	Code Source	Behavior	Network	Feature
D-1	cloud	10,13	16	1	-
	firefox	12	2,5	1	14,16
D-2	dll_loader_x64	10,12,13	7	1	15
D-3	telnet	13	2	1	-
L-1	notepad	8	2,5	1	-
E-1	firefox	9	6	1	14,16

如图 7a 所示，攻击者将网站重定向到另一个 IP 地址。

当用户试图浏览此网站时，firefox.exe 进程将导航到假 IP 地址 138.113.2.43。此时，firefox 被标记为 P1，这意味着该进程有外部网络流量。

然后，攻击者可以利用 Firefox 的漏洞远程成功执行代码。当它执行未知代码时，Conan 使用动态调用堆栈检测它，将该线程与可信线程分离，并将其标记为 P12。

firefox 被攻破后，攻击者首先执行主机名和任务列表，然后读取 Deafult。通过 RDP 协议收集主机信息。Conan 记录分离实体执行敏感命令和访问敏感文件的情况。

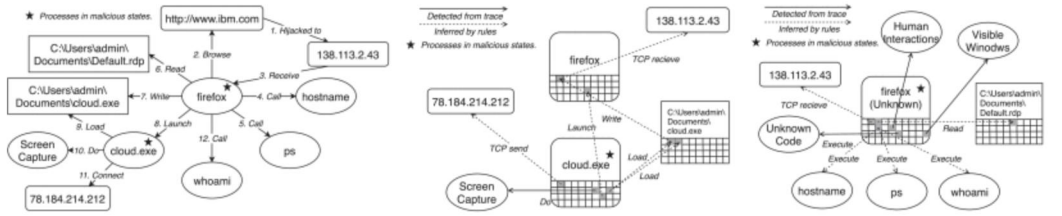
此时，当该实体进入恶意状态时，将触发警报。该实体包括表 4 (D-1 firefox)中列出的状态。具有可见窗口和与用户交互的功能意味着它应该被用户感知。由于它既具有网络连接又执行未知代码，因此可以推断执行的未知代码可能是从网络下载的。因此，这种攻击被认为是“对良性应用程序的利用”。

然后，Firefox 下载二进制文件 cloud.exe。这个文件被标记为 F1，这意味着它是从网络下载的。

Firefox 创建了一个进程 cloud，并加载无签名映像 cloud.exe。它使该进程由可疑进程执行，并执行来自网络的不受信任的代码。

最后，可疑进程截取屏幕截图，执行 whoami，并将屏幕截图发送到一个新的 IP 地址 78.184.214.212。这种攻击被公认为“下载并执行恶意软件”，这是现实情况中最常见的攻击场景之一。

重构后的攻击链如图 7b 和 7c 所示。Conan 通过跟踪每个 ASI 的来源自动解释了为什么这些进程是恶意的。重建后的攻击链与原始攻击链相似。



(a) Provenance graph of Attack D-1　(b) Reconstruction of the attack from cloud.exe　(c) Reconstruction of the attack from firefox.

(5) Large-Scale Real-World Deployment:

为了研究 Conan 的效率和稳定性，我们将其部署在三家现实世界的公司中:一家金融公司、一家通信公司和一家能源公司。我们在 Windows 7/8/10 上开发了收集器。它们都是 64 位的。一共在 226 台办公室机器上部署了收集器。这些收集器已经运行了三个多月，结果表明，Conan 可以像我们预期的那样在现实场景中长时间高效稳定地运行。

同时，我们检验了方法的准确性。3600 万个进程中有 422 个告警，包括 14524 个程序，如表 5 所示。正如我们在第 4.4 节中所述，我们不假设计算机在安装 Conan 之前是干净的，并且我们假设计算机上现有的程序也可能是恶意的。结果如表 5 所示，其中 C 表示攻击链完整的攻击，I 表示攻击链不完整的攻击，没有检测到初始访问阶段。不幸的是，我们通过使用威胁情报和沙盒检查所有这些警报。到目前为止，这些攻击还没有被归类为恶意攻击。我们将在下一节中讨论假警报。

Company	Total Machine	Total Process	Total Program	Alarm Type	Alarms	FPR(Alarm/Process) (10^{-5})	Program	FPR(Program) (10^{-2})
Financial	61	9,014,881	4,664	C	15	0.17	9	0.19
				I	98	1.09	68	1.46
Communication	85	16,512,358	7,152	C	17	0.10	8	0.11
				I	124	0.75	73	1.02
Energy	80	11,251,325	5,215	C	22	0.20	11	0.21
				I	142	1.26	91	1.75
Total	226	36,778,564	14,524	C	54	0.15	22	0.16
				I	364	0.99	172	1.28

真实世界评估中的假警报。

(6) False Alarms:

为了研究 Conan 在良性环境中的有效性，我们在实验室的主机和三家真实的公司中部署了它。

在实验室:因为我们的检测系统主要关注远程访问和可疑行为，所以我们下载、安装并运行了一组具有类似能力和行为的良性应用程序。我们还选择了一组常见的应用程序，包括系统应用程序和其他流行的应用程序。我们从网络下载每个应用程序，用默认配置安装它，然后手动执行它以触发额外的程序行为(L-1)。另外，我们邀请了一位不了解我们系统的志愿者使用另一个宿主(L-2)进行长期测试。结果表明，Conan 可以准确地识别良性应用程序，没有假阳性发生。

在现实场景中:我们将 Conan 部署在三家不同领域的公司，它已经运行了三个多月。近80%的不可信数据流来自内部网络。触发告警的程序中，大部分(70%以上)是本领域的定制程序，剩下的是 firefox(浏览器)、PuTTY(SSH 客户端)等普通程序。最糟糕的是，有一些被破解的程序和重新打包的程序。很难判断它们是恶意的还是灰色的。我们选择了一个假阳性进行详细分析(如表 4,E-1, firefox 所示)。

这个应用程序在部署 Conan 时已经存在，但它仍然可能是一个恶意软件实例。因为 Firefox 是一个开源应用程序，我们发现它包含 11 个实现屏幕捕捉的代码块;因此，它可能出于合理的原因截取屏幕截图。出于认证目的，在安装柯南之前下载了 firefox.exe 的二进制文件;因此，我们无法追踪 Firefox 的下载源。事实上，Conan 无法将这种情况与攻击者创建具有正常功能但包含恶意代码的假应用程序的情况区分开来。例如，攻击者可以在开源项目中插入恶意代码，重新编译，然后上传恶意软件，诱使人们下载它。

表 6 显示了在每个数据集中生成的一些重要的 ASIs(不包括真阳性攻击和假阳性攻击)。他们每个人都很可疑，可能是袭击的一部分；例如 P2 表示访问某些敏感文件的进程数。然而，使用这些上下文方法和提出的检测模型，不会再有假阳性了。

Data set	P1	P2	P3	P4	P5	P6	P10	P11	P15	P16	F1	F2	F3	F4
D-1	2588	26	16	119	10	164	85	5	207	42	8161	17	2	15
D-2	3874	17	7	78	7	99	122	9	171	17	7411	12	7	9
D-3	22	8	47	9	6	25	40	18	26	6	3697	10	2	6
L-1	177	10	7	14	2	20	5	48	463	43	36723	114	19749	11
L-2	2438	12	10	15	5	126350	573	1	5681	126	145161	53	106350	5

每个数据集中生成的 ASIs 的选择性列表(不包括攻击)。

(7) Runtime Overhead and Memory Usage:

我们的探测系统可以分为两个主要部分:收集器和探测器。在客户端，收集器的开销可以忽略不计:它可以运行在具有 Intel i5-7500 CPU(4 核和 3.40 GHz)的主机上, 内存小于 10 MB, CPU 使用率为 5%。具体来说，表 8 显示了收集器中事件解析和签名匹配的平均效率。由于 100 ns 是 Windows 上的最小时间粒度，结果表明收集器可以有效地处理事件和 api。根据主机工作负载的不同，带宽利用率约为 1 ~ 10kb /s。

我们测量了服务器上的检测开销，结果如表 7 所示。我们在一台使用 Intel Xeon Silver 4116 CPU(12 核，每个核 2.1 GHz)和 256 GB 内存的服务器上进行实验，运行在 Ubuntu 18.04 上。内存:每个检测器有固定的内存，平均大约 2.1 MB 的内存。与不断增长的起源图(开始时为 100 MB, 5 天数据集[22]结束时为 600 MB)相比，Conan 是唯一可以长时间监视主机的实用系统。效率:检测器对一个数据流使用单核。平均而言，Conan 分析数据的速度比生成数据的速度快 200 到 1500 倍，如“加速”一栏所示。换句话说，如果 CPU 是唯一的限制，一个核的 Conan 可以实时处理多个数据流。图 6c 为 Conan 监控不同数量主机时 CPU 和内存的使用情况。我们通过实时模拟不同数量的数据流(L-1)同时传输给柯南来验证这一结果。结果表明，Conan 可以监视数百个具有线性 CPU 和内存使用情况的主机。

(8) Benefit of Semantic Recognition of Events:

由于近 99.9%的系统事件与良性活动相关，因此减少无关数据以保持 Conan 的效率和准确性是非常重要的。如前所述，我们使用预定义的规则来识别系统事件的高级语义。表 9 显示了不同流程步骤之间的事件数:原始事件(O)，由规则匹配的事件(M)和导致状态改变并存储在数据库中的事件(R)。M 列的数字与 O 列的数字相似，这意味着 Conan 跟踪了大部分原始事件。列 M 中的数字表示只有不到 1%的数据被存储用于重建。预过滤重复的 Read 事件。

Discussion：

在本节中，我们将讨论了解 Conan 设计的攻击者如何试图逃避检测机制。（对应相应攻击的解决方法）。

内存的攻击：调用堆栈中的地址是下一个命令的入口。为了避免调用堆栈中的可疑地址，恶意代码不能调用任何会导致内核事件(如读/写文件、系统调用和内存操作)的 api。但是根据我们的经验，如果没有这些 api，攻击者就无法实现他的目标。另一种可能的方法是将恶意代码挂钩或插入到良性图像中;由于内存操作，这些攻击也会在一开始就被记录下来。此外，它还描述了一个称为内存身份验证的研究问题。Conan 无法检测到的另一种攻击是 ROP 攻击，如第 2 节所述。

系统扩展：实现可疑行为的方法不止一种，这意味着我们应该为每种实现开发相应的签名。例如，现有的研究介绍了三种在 Windows 上截取屏幕的方法，还有一些其他的方法。然而，实现的总数是有限的，基于操作系统本身，它远远小于攻击的数量。因此，监视额外的行为及其实现是可行的，也是值得的。此外，我们的系统主要依赖于跟踪信息流来确定代码执行的原因以及敏感信息的去向。通过添加更多的数据源和相应的规则来覆盖更多的信息流，可以很容易地扩展我们的系统。

系统恢复：因为我们的检测方法是基于状态的，所以在系统崩溃或重新启动时恢复状态是很重要的。由于所有的状态都存储在数据库中，一旦崩溃，它们可以从数据库中恢复。当实体的状态改变时，它会同步到数据库。当一个实体被删除时，它将被标记为数据库中的 out 数据。因此，当我们的系统重新启动时，它将从数据库中恢复内存中的状态，并且系统能够继续其工作。

白色的清单。我们基于代码认证的白名单机制在 DARPA 的参与和我们的实验室工作得很好，即零虚报。然而，在现实场景中，我们会得到一些假警报。为了进一步减少这些误报，我们可以将现有的白名单机制与进程和/或 IP 白名单机制结合起来，这些机制在文献中被证明是有效的。

图像重建。由于 Conan 只保留了导致状态改变的第一个事件，它将错过对实体具有相同影响的后面的事件。因此，重构图可能是不完整的。我们认为这种方法是因为我们系统的主要目的是准确有效地检测攻击，而重构的图只用于理解为什么会产生检测信号。另一种选择是在将重复事件插入数据库时删除重复事件。在本例中，duplicate 表示事件的来源和目的地，并且匹配的规则都相同。为了确定一个事件是否重复，我们必须存储和搜索它，这需要内存存储和 CPU 计算。在我们的方法中，我们只需要检查一个实体的状态来决定是否存储这个事件。它的效率要高得多。

Related work：

在本节中，我们将调查相关研究，并将其与我们的方法进行比较，以突出我们方法的新颖性。

主机入侵检测技术可以分为三种检测器类型:误用检测器、异常检测器和混合检测器。误用检测主要依赖于已知的攻击模式;采集到的原始数据被转换成既定的格式，然后传递给检测模块，检测模块做出判断。不幸的是，误用检测技术很难检测未知攻击(即零日攻击)。我们可以看到，基于知识的方法依赖于需要定期更新的攻击特征数据库，而基于机器学习的方法通常缺乏泛化的能力。异常检测用于检测未知攻击。良性程序的行为概要文件被频繁地存储和更新。任何偏离概要文件的情况都会被标记为潜在的攻击。Forrest 等人提出了一个异常检测系统，该系统使用固定长度的系统调用序列来定义 UNIX 进程的正常行为。Shu 等人

提出了一个正式的框架,用于调查基于主机的异常检测,并详细讨论了各种动态和静态方法。基于异常的技术的优点是它们可以检测零日攻击。然而,由于误用检测和异常检测不能同时考虑假阴性和假阳性,这些方法会导致很多假阳性。针对上述两种方法的不足,提出了混合技术。除了将误用检测和异常检测技术相结合之外,混合技术还涉及特定的策略。基于策略的方法设计得很好,并以 SLEUTH 和 HOLMES 为例。SLEUTH 利用可信度标记和机密性标记来定义代码和数据。HOLMES 构建自定义策略,从 APT 攻击链的每个步骤中利用语义。上述作品很少讨论攻击生命周期的本质意图,导致一些假警报或假阴性仍然存在(我们在第 3 节中列出了这些弱点)。

Conan 与之前的作品有很大不同。我们提出了一个检测模型,总结了 APT 攻击中存在的三个基本阶段。通过该模型,我们可以揭示整个攻击链,准确检测潜在的危险。

溯源跟踪旨在发现复杂环境下完整的攻击路径。回溯是以前工作中常用的解决方案,其灵感来自于先驱工作 BackTracker。此后,PriorTracker 优化了文中提出的过程,并支持前向跟踪能力,以便及时进行攻击因果关系分析。在正向/向后搜索过程中,构建起源图来记录系统对象/主题依赖关系。研究利用系统调用数据来跟踪信息流。为了提高精度,细粒度数据由的作者收集,但是盲目地增加数据量会导致开销的增加。SLEUTH 通过使用标签进行高效的事件存储和分析进行了创新,但是所提出的策略具有固有的局限性。虽然 SLEUTH 维护了一个没有标记来源不可信的内部 IP 地址白名单(DNS 查找等),需要经常维护以减少误报,但基于标记的方法本质上是一种基于图的存储方法,难以应对长时间的 APT 攻击。由于 SLEUTH 需要很长时间来处理大量数据的依赖关系,因此很难保证实时性能。数据量与内存消耗成正比,不断增长的数据将导致内存爆炸。

与以前的工作不同,Conan 很好地利用了来源图的概念进行实时检测。Conan 采用了一种新颖的基于状态的框架,具有恒定的内存使用和低开销。我们的系统是上下文敏感的,并引入了类似于 fsm 的结构来自动传输状态。此外,无论 APT 攻击持续多长时间,我们都可以实时分析审计数据、进行状态转换并产生告警。

Conclusion:

在本文中,我们提出了 Conan,它使用类似于 fsa 的状态转换方法提供高效和准确的 APT 攻击检测。我们确定了 APT 攻击的三个最基本的组成部分,以减少误报和误报,以及一些更好地识别语义的设计。与相关研究不同的是,我们使用状态而不是来源图来记录语义,这确保了随着时间的推移恒定的内存使用。在评估过程中,通过精心设计的检测模型,Conan 快速检测出了所有攻击,只有一次误报。在我们基于状态的框架的帮助下,Conan 在一段时间内保持恒定的内存使用(1-10 MB),这与以前建立在不断增长的起源图上的方法不同。