如果可以，请提取我：滥用恶意软件检测器中的PDF解析器

         Curtis Carmony                          Mu张勋                      超Hu         Abhishek Vasisht Bhaskar            Heng Yin

雪城大学NEC美国实验室雪城大学雪城大学curtis.carmony@gmail.com mu@nec-labs.com xhu31@syr.edu abhaskar@syr.edu heyin@syr.edu

*抽象*

由于PDF格式的普及和Adobe Reader的不断发展，恶意PDF的检测仍然是一个问题。所有现有的检测技术在一定程度上依赖于PDF解析器，而PDF格式的复杂性为解析器混淆留下了充足的空间。为了量化这些解析器与Adobe Reader之间的差异，我们通过在大多数自动二进制分析技术识别的位置直接使用Adobe Reader创建参考JavaScript提取器。通过将此参考提取器的输出与从VirusTotal获取的大型数据集上的几个开源JavaScript提取器的输出进行比较，我们能够识别数百个样本，其中现有的提取器无法从中提取JavaScript。通过分析这些样本，我们能够确定这些提取器中的每一个的几个弱点。基于这些经验教训，我们对恶意PDF样本进行了几次混淆处理，可以成功回避所有经过测试的恶意软件检测器。我们称这种逃避技术为a*PDF解析器混淆攻击*。最后，我们演示了参考JavaScript提取器提高了现有基于JavaScript的分类器的准确性，以及如何使用它来缓解实际设置中的这些分析器限制。

                                      一        导论

即使Adobe的Acrobat Reader（通常称为Adobe Reader）通过增加高级安全机制（如沙箱）而变得越来越安全[5]，但在2014年发布的44个CVE中仍然会发现新的漏洞[1]，已发布128个CVE在2015年写作时[2]。由于继续利用Adobe Reader以及无处不在的PDF格式，恶意PDF文件的检测依然令人担忧，卡巴斯基报告称Adobe Reader在2014年是第三大受攻击目标，并且吸引了整体攻击的5％[ 18。

商业防病毒产品中的恶意PDF检测在很大程度上依赖于签名检测，并且不足以检测包含零日漏洞利用或高级持续威胁的PDF。为了解决这个限制，已经提出了两类系统来具体检测恶意PDF文件：1）基于结构和元数据的检测器[29]，[32]，[38]和2）基于JavaScript的分类器[23]，[25 ]，[37]，[26]。

通过确定哪些结构特征和元数据与每个类最相关，基于结构和元数据的检测方法区分良性和恶意PDF。然而，PDF漏洞的基本恶意并不是来源于文件结构，而是源于具有恶意意图的嵌入式有效载荷（例如JavaScript代码）。因此，通过*模仿攻击* [39]，[38]和*反向模仿攻击* [28] ，这些检测器可以很容易地被躲避，这些*攻击*隐藏PDF文件中的有害代码，这些代码显示与良性文件相关的结构特征和元数据。

要从根本上解决PDF漏洞攻击，有必要分析文档内容并搜索恶意负载。之前的工作[23]揭示了JavaScript是PDF漏洞中最常见的恶意内容，主要有两个原因：1）Adobe JavaScript API的实现暴露了漏洞; 2）JavaScript代码用于实现高级开发技术，如堆喷洒。从VirusTotal收集的样本集中的几乎所有恶意PDF文档都包含JavaScript，表明嵌入式JavaScript的提取和分析对于恶意PDF检测至关重要。

为此，之前的基于JavaScript的分类器[23,25,37]试图解析PDF文档，从中提取JavaScript，然后分析此JavaScript以将其分类为良性或恶意。这些作品都依赖于他们从PDF中准确提取JavaScript的能力。除MPScan [26]（使用与本文中介绍的类似的Adobe Reader的修改版本）外，这些工作中的每一个都依赖于开源解析器或其自己开发的解析器。由于所有这些解析器都是不完整的，并且对JavaScript可以嵌入的位置有过于简单的假设，所以这些检测方法不准确或不健全。

在本文中，我们旨在对称为*PDF解析器混淆攻击*的新规避技术进行系统研究，该技术旨在混淆恶意软件检测器中的PDF解析器以逃避检测。本质上来说，这是一种逃避

攻击例证了恶意软件探测器故意滥用文件处理的*变色龙*和*狼人*攻击[22]。然而，与之前在此工作中研究过的其他文件类型（例如ZIP，ELF和PE）相比，PDF格式的复杂性与Adobe Reader宽大的解析这些文件的潜力提供了更大的攻击空间。不幸的是，这个攻击空间在安全界尚未得到充分研究。

为了进行系统的研究，我们开发了一个*参考JavaScript提取器*，直接使用Adobe Reader，这可以说是最流行和最有针对性的PDF查看器[19]。为了开发这个参考提取器，我们提出了一种主要是自动的动态二进制分析技术，可以快速识别少量的候选分接点，可以通过简单的手动分析进一步细化。然后，我们使用从VirusTotal收集的超过160,000份PDF文件对该参考提取器和几个流行的提取器进行差异分析。对于每个提取器，我们都会识别数百个无法正确处理的样本，但根据参考提取器包含JavaScript。

通过研究参考提取器与现有提取器之间的这些差异，我们发现了几个新的混淆，并进一步量化了它们在JavaScript提取器和恶意软件检测器的解析器混淆攻击中的影响。通过结合这些混淆中的几个，**我们证明了恶意PDF可以成功规避评估的*所有*恶意软件检测器，包括基于签名的，基于结构/元数据的和基于JavaScript的检测器**。

这些发现表明，有效对策的关键是高度逼真的解析器，它非常模仿Adobe Reader的解析逻辑。一种可能的解决方案是直接部署基于JavaScript的检测器的参考JavaScript抽取器。我们的实验表明，这种部署方案不仅会导致可接受的运行时间开销，而且还会产生更高的检测精度。我们的实验表明，在用我们的参考提取器替换原始解析器之后，基于相当初级的分类器，PJScan [23]的检测率从特定版本的Adobe Reader的68％增加到96％。

纸贡献。总之，本文做出如下贡献：

•          我们提出了一种大多数自动化的平台无关点识别技术，用于正确识别与Adobe Reader中用于开发参考JavaScript提取器的JavaScript解析和执行相关的抽头点。

•          使用我们的参考提取器，我们系统地评估现有JavaScript抽取工具的缺点。我们已经识别出数百个PDF样本（无论是良性的还是恶意的），而现有的提取器无法从中提取JavaScript。我们手动调查其中的很多，并确定其根源。

•          我们通过结合我们分析中确定的几个混淆因素构建了几个PDF解析器混淆攻击。这些逃避已被证明有效地成功避开了我们测试过的所有恶意软件检测器。

•          我们讨论几种缓解技术。特别是，我们证明，使用我们的参考JavaScript提取器，现有分类器的检测率在我们的样本集中从68％显着增加到96％，并为参考提取器提供了可能的部署方案。

我们计划发布完整的数据集，并为参考JavaScript提取器启动公共服务，以帮助安全研究人员进一步研究此问题。MD5散列值列表可用于部分数据集，可在https：// goo找到*。*GL / qtbuOC。

                                       II。        B ACKGROUND

*A.元数据和基于结构特征的检测*

由于基于签名的恶意PDF检测器[31]容易受到各种传统恶意软件多态性技术[16]，[17] [34]的影响，因此已经努力寻找更强大的恶意PDF检测方法。根据观察，恶意文件除了有效载荷以外通常只有很少或没有内容，并且良性文件通常具有大量不同的内容，因此已经提出了若干系统来量化这些结构差异以促进恶意PDF检测和分类。

PDF恶意软件杀手[29]使用流行的PDFiD工具[35]的样本中确定的PDF关键词作为特征集，用于训练随机森林PDF分类器。该系统的主要限制在于其使用PFDiD工具，该工具仅执行简单的字符串匹配以识别PDF关键字子集的存在，因此不能识别由过滤器编码的字符串或区分文档结构中的字符串与那些在其内容中。

PDFrate [32]同样使用随机森林分类器，但使用更多描述性特征集。它分析PDF文件以检索样本的202个不同的结构方面，例如文件中对象的数量和类型，它们的大小，内容方面，文档中的页面以及文件的大小。同样，这项工作在很大程度上受其解析器的限制，这是由作者开发的一个程序，它利用正则表达式来提取无法解码文件中的编码流或解析其内容的这些特征。

采取类似的方法，Srndic和Laskov [38]开发了一个系统，该系统提取PDF中对象的树状结构作为分类的特征集。尽管其在离线实验中的准确性，但在操作测试中使用此系统表明，该系统无法始终正确识别新威胁。

尽管基于元数据和结构的恶意PDF检测系统已被证明既高效又有效，但它们从根本上容易被规避。先前的研究[38,39]已经证明，无论是有趣的还是对PDFrate [9]的系统研究，这些分类都受到所谓的模仿攻击。在这些攻击中，恶意样本的结构特征被修改为类似于已被分类为良性的PDF文档的结构特征。由于PDF恶意软件执行的恶意行为不一定取决于特定的结构特征，因此该技术可以避开这些分类器，同时保留原始攻击的功效。

第二种类型的攻击，即*反向模仿*攻击也已经提出[28]。模仿攻击为恶意样本增加了良性特征，而反向模仿攻击则将样本分类为良性并使其变得恶意。这种攻击的发起甚至更容易，因为它没有

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 表I：现有的PDF分类器   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 技术 | 探测器 | 发现 | 分析器 | 闪避 | |  |  | 能力 | 需求 | 技术 | | 基于签名 | AV扫描仪Shafiq et al。[31] | 不定 | 低 - 中 | 恶意软件多态性[16]，[17]，[34] | |  | PDF恶意软件杀手[29] |  |  | 模仿攻击[39]，[38] | | 元数据和结构为基础 | PDFrate [32] Srndi        c和Laskov [38]' | 中 | 中 | 反向模仿攻击[28] | |  | 刘等人。[25] |  |  |  | | 基于JavaScript | MDScan [37] PJScan [23] | 不定 | 高 |  | |

需要知道目标分类系统，这种模仿攻击很大程度上取决于。

*B.基于JavaScript的检测*

这些系统的逃避表明，仅依靠结构和元数据相似性的恶意PDF检测技术是不够的。鉴于大多数恶意PDF使用JavaScript来触发或设置漏洞，以前的分类器将重点放在提取和分析嵌入式JavaScript上。由于这些检测方法依赖JavaScript分类进行分类，所以。

刘等人。[25]尝试识别和测试文档中自动执行的JavaScript，以便将Adobe Reader在运行时表现出来的可疑行为归因于正在执行的JavaScript。随后将运行时观察数据与其他启发式数据一起用于计算分类得分。该系统受限于其过于简单化的JavaScript抽取，仅将JavaScript与两个关键字相关联，并假设它们必须始终以明文形式出现。事实上，JavaScript可以嵌入到多个图层中，使用格式的扩展（例如XFA），并且可以通过使用各种PDF功能（如对象流和过滤器）进行编码。

意识到恶意JavaScript通常利用Adobe JavaScript API读取PDF中对象的内容，MDScan [37]不仅解析PDF以提取嵌入式JavaScript，而且还加载文档的内部结构。提取的JavaScript然后在修改后的JavaScript引擎中运行，并被增强以支持作者反向设计的Adobe JavaScript API的某些元素，以便对这些受支持的API函数的调用模仿Adobe Reader的行为。虽然这个系统为动态分析这个JavaScript提供了一个更完整的平台，但由于部分的API支持，它本质上是不完整的，这是不平凡的，容易出错的，并且相当耗时地进行改进。这项工作意识到JavaScript可以以各种方式进行编码，但它只将JavaScript与一个关键字相关联，

PJScan [23]提取JavaScript并使用标记化的JavaScript作为用于训练单类支持向量机的特征集。这个系统的精度不够准确，主要是由于它的PDF解析器libpdfjs，它建立在第三方解析器Poppler上[10]。虽然Poppler声称要实施整个PDF ISO 32000-1规范[24]，但它并没有声称解决规范和Adobe Reader的封闭源实现之间的差异，规范中的所有附录或所有的规范扩展，如XFA。

在不考虑每个检测器特有的缺点的情况下，这些基于JavaScript的PDF分类器都受到JavaScript提取功能的限制。不仅必须正确解析PDF，而且这些检测器必须静态标识文档中嵌入的所有JavaScript组件。由于JavaScript可以以多种不同方式嵌入，甚至可以使用这些检测器未实现的规范的扩展，所以它们不可能始终生成文档中的所有JavaScript，特别是那些已经被混淆的JavaScript。

为了解决这些分析问题，Lu等人引入**了MPScan，**它钩住Adobe Reader的JavaScript引擎，在打开文档时生成由Adobe Reader执行的JavaScript，然后使用shellcode和堆喷检测技术将文档分类为恶意或良性[26]。虽然作者能够缓解所有这些解析问题，但他们只能钩住一个版本的Adobe Reader，并且没有提供任何技术来识别钩住二进制文件的点或描述它们是如何做的。在没有这些信息的情况下，人们必须假设他们是通过手动分析来完成这项工作的，这对于像Adobe Reader这样庞大而复杂的程序来说是一项艰巨的任务，并且每一个新版本的Adobe Reader都必须重复这些任务。

*C.总结和假设*

如表I所示，以前的PDF分类器一直在发展以掌握恶意有效载荷的语义，以击败造成多态恶意软件或模仿良性文件结构的基本攻击。

假设1：一个关键的观察是以前的所有检测方法都依赖于解析和解释PDF文件到一定程度。因此，他们的检测准确性主要取决于其PDF解析和JavaScript提取的质量。因此，我们假设可以发起一个微妙的攻击来规避所有这些分类器，前提是它可以成功地混淆用于检测的PDF分析器。

假设2：我们也意识到，为了执行准确和强大的PDF分类，实际检查PDF文件中的嵌入式JavaScript有效负载至关重要。因此，我们假设JavaScript的改进

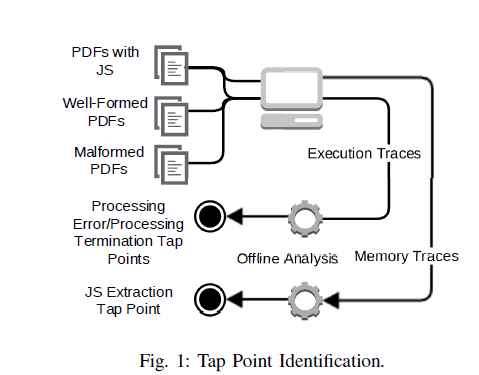


图1：Tap Point Identification。

提取可以方便检测现有的基于JavaScript的PDF分类器未检测到的恶意文档。

                III。              R EFERENCE J AVA S CRIPT E XTRACTOR

为了验证我们的假设，我们需要开发一个参考JavaScript抽取器，以便定量测量部署在检测器和Adobe Reader中的现有PDF解析器之间的差异。

*A.对新技术的需求*

MPScan表明可以在Adobe Reader中手动识别JavaScript抽取点。由于Adobe Reader-IDA Pro仅在主AcroRd32.dll组件中标识了91,753和133,835个函数，因此我们未描述所需 der的9.5.0和11.0.08版本，我们的工作量与，分别 - 它不可能是一个简单的任务。

由于参考提取器只能精确地模仿单个版本的Adobe Reader的行为，因此我们必须为要保护的每个Adobe Reader版本确定三个分支点，使得该技术的手动应用不可行，作为通用解决方案恶意PDF检测的问题。通过开发一种可重复和自动化的技术，我们可以以最小的努力为许多版本的Adobe Reader可靠地生成参考提取器。这种技术也可以想象应用于从其他文件格式中提取可执行代码，例如Microsoft Office文档中的嵌入式VBA宏。虽然我们无法完全自动化这个过程，但我们已经能够自动化大部分分析。总体，

*B.概述*

图1描述了构建参考提取器的工作流程。我们首先在执行监视器中用Adobe Reader打开三类带标签的PDF样本（即带有JavaScript的格式良好的PDF，没有J​​avaScript和格式错误的PDF格式良好的PDF），以收集内存访问和执行痕迹。最后，我们对轨迹执行离线分析，以识别与JavaScript提取，PDF处理终止和处理错误相关的三个*分*接点。然后，我们可以通过在这三个分接点处修改Adobe Reader来创建参考JavaScript提取器。

特别是，我们通过比较多个类别的独特痕迹来识别这些点。通过使用自动执行的JavaScript处理格式完整的PDF的存储器访问痕迹，我们确定了*JavaScript提取点*，其中提取并执行嵌入式JavaScript代码。通过检查格式良好和格式错误的文件的不同执行痕迹，我们发现*处理终止点*和*处理错误分接点*，它们分别代表这两个类中文件处理的结束。

值得注意的是，我们根据Adobe Reader在打开文件时的行为来定义PDF文件的格式。我们不依赖于PDF规范来进行这样的确定，因为1）某些规范项目含糊不清，并且不容易解释，2）Adobe Reader的实现实际上偏离了规范以增加兼容性。因此，我们认为Adobe Reader成功打开的PDF格式良好。相反，如果Adobe Reader打开警报，指出无法打开示例，我们认为PDF格式不正确。用于执行我们的分析的样本集是通过使用Adobe Reader打开样本手动构建的，以根据我们的定义确定它们是正确的还是畸形的，

虽然我们使用现有的工作作为JavaScript提取的基础[15]，但由于三个原因，此技术不足以用于参考提取器的开发。首先，因为它依靠一个执行监视器进行提取，所以处理任何大量的样本都太慢了。其次，该技术侧重于在监控系统运行时提取所有目标信息，因此不提供用于确定何时提取所有数据的机制，这是我们为了便利地处理样本而需要的。最后，该技术不处理没有目标信息存在或无法提取的情况，即PDF不包含JavaScript，或者Adobe Reader由于格式不正确而无法处理PDF。

为了解决这些限制，我们的技术与以前的三个方面不同。首先，与之前的着重于访问数据的指令并因此仅对存储器轨迹执行分析的工作相比，我们考虑了数据访问和控制传输。增加控制流分析是必要的，因为并非我们希望从Adobe Reader中提取的所有信息都可以通过只监视内存访问来确定。例如，确定Adobe Reader是否遇到错误或是否已完成打开PDF可能只能通过检查程序的状态来确定。其次，这项先前的工作监视“操作”级别的内存访问，并且只对固定数量的内存访问中的连续内存访问进行分组。然而，此分组不适合分段存储器访问模式，这些模式很可能出现在JavaScript处理中。我们会根据“操作组”的粒度跟踪内存访问情况，以捕获遍布许多访问的连续操作。最后，原始技术选择指令作为分支点，这只能将目标数据交给完整的系统仿真器。为了提高性能，我们调整技术来定位函数，以便可以使用现有的函数挂钩技术。它只能产生完整的系统仿真器的目标数据。

下面的小节将给出详细的解释。总而言之，我们已经量身定制了现有技术来识别JavaScript抽取点并将其扩展以识别参考抽取器所需的新处理错误和处理终止点。

*C. JavaScript提取Tap Points*

定义。我们认为*JavaScript抽取点*是一个函数，Adobe Reader从PDF文档中提取和执行JavaScript代码。形式上，这样一个抽头点被定义为一个三元组：

（*调用者，函数入口点，参数号*）*，*其中调用者指示函数的调用上下文，参数号是函数参数的索引，该参数保存对包含JavaScript的以空值终止的字符串的引用。F:\PDFpy\God_with_me\2018Q2\paper\1_T_ExtractMeIfYouCanAbusingPDFParsersInMalwareDetectors_NDSS2016\ExtractMeIfYouCan.files\image006.gif

我们维护分接点功能的调用上下文以提高识别的准确性。一些常用函数（如*memcpy*）很可能会被程序中的多个调用者调用。但是，这些调用中只有一部分与JavaScript操作相关联，因此引入上下文感知可显着帮助消除错误识别。

我们在功能级别而不是指令级别定义分接点，因为功能入口点对条件执行更具弹性，并为挂钩提供更清晰的接口。例如，根据字符串的长度，*memcpy*中的不同指令用于复制字符串。

尽管功能级抽头点有优势，但实际上，我们必须首先确定指令级抽头点，我们称之为*原始抽头点*。每个原始点都被正式定义为一对：

（*调用者，程序计数器*[[1]](file:///F:\\PDFpy\\God_with_me\\2018Q2\\paper\\1_T_ExtractMeIfYouCanAbusingPDFParsersInMalwareDetectors_NDSS2016\\ExtractMeIfYouCan.htm" \l "_ftn1" \o ")）*，*其中调用者也是主机函数的调用者，并且程序计数器唯一地表示指令的地址。

一旦发现原始分接点，需要进行数据流分析以将识别的指令与主机功能的某个参数相关联。因此，我们最终可以检索功能级别的分接点。

内存访问跟踪。通过分析Adobe Reader在打开包含自动执行JavaScript的PDF时进行的内存访问来识别原始JavaScript抽取点。所有这些*内存访问*都记录在内存跟踪中，其中每个访问*m*被格式化为一个元组：*m* =（*调用者，程序计数器，类型，数据，地址*）

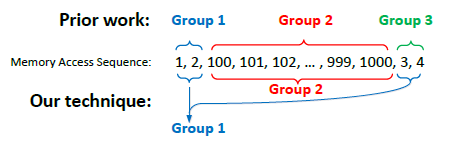


图2：识别先前和这项工作之间的连续内存操作的比较

也就是说，调用上下文，产生访问的指令的地址，访问的类型（读或写），写或读的数据以及正在访问的存储器的地址。在一层调用上下文不足的情况下，我们可以通过添加另一层调用者信息来增加上下文敏感度。我们将为这些样本收集的记忆痕迹集合称为M *JS*= [ *M*0*，M*1*，...，M n* ] ，其中每个*M i*表示单个样本的迹线。

原始分接点的识别。一旦收集到M *JS中*的内存痕迹，我们将分两步进行离线分析，以确定原始分接点。首先，我们将每个跟踪中的内存访问分组为*连续的内存操作，*然后我们检查这些内存操作以搜索JavaScript字符串。

我们跟踪“操作组”级别的内存访问，而不是单个操作级别，以便容忍JavaScript处理中经常发生的间歇性内存访问，其中字符串可能会被分段分段执行。先前的工作只监视一次有限的窗口（即五次）的*操作*，并且超出此限制的任何操作（即使与前一次操作相邻）都不能与以前的操作相关联。图2展示了我们的方法与之前的工作相比的优势[15]。在这个例子中，只有三个组（即{ 1,2 } ，{ 100 ... 1000 } 和{ 3,4 }）由先前的工作来标识，尽管访问存储器区域3和4继续在1和2上的操作。为了解决这个限制，我们以更高的粒度跟踪存储器操作并且同时直接监视几个访问组。因此，对从100到1000的存储区域的顺序访问变成一个单独的组; 可以在一个窗口内观察{ 1,2 } 和{ 3,4 }的访问，因此可以进一步分组为一个。虽然以前的工作可能会增加窗口大小以正确分组分段内存访问，但这大大增加了计算开销。

我们将*连续内存操作*定义为具有相同调用上下文的指令列表，以相同的方式访问内存中的连续位置。形式上，一组连续的内存操作*g*被定义为一个三元组：

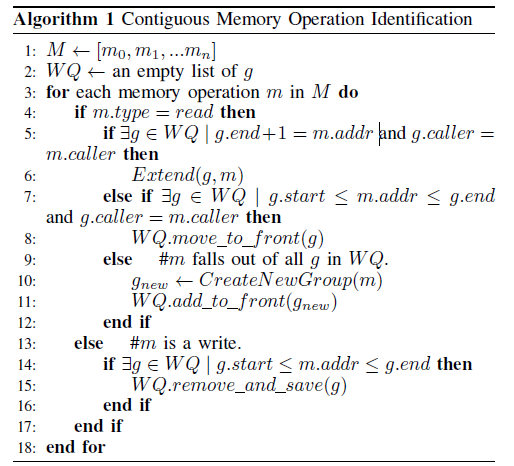
*g* =（*开始，结束，m列表*）*，*

开始和结束分别是连续访问的开始和结束地址，*m列表*是单个内存访问[ *m start，...，m end* ]的列表。为了对这些内存操作进行分组，我们给出算法1。

该算法将内存访问跟踪*M*作为输入并输出连续内存访问组列表。我们还引入了一个工作列表*WQ*，它用作LRU缓存来存储固定数量的访问组。实际上，缓存大小为10就足够了，尽管潜在的分接点可能由于被提前推出缓存而丢失。尽管可以增加此大小，但它会增加执行分析所需的内存量。

为了识别这些组，我们遍历轨迹*M*中的每个存储器操作*m*。如果读取操作*m*与现有的组相匹配，则根据调用上下文对缓存*WQ*中的*g*和*m*读取的地址恰好成功地访问最后一个由*g*访问的地址，我们执行*Extend*（），它在*g*的末尾插入*m*。如果读操作*m*落在现有组的中间，我们将该组移到缓存的前面，以避免丢弃最近访问过的组。否则，*米*落在指示存在新的连续存储器访问组的每个现有组的边界之外。我们使用*m*创建这个新组，并将其添加到缓存的前面，如有必要，删除并保存最近最少使用的组。在*m*实际写入缓存中的现有组*g*的情况下，我们通过将旧组从缓存中删除并将其保存为输出来使旧组无效。

一旦为样本收集了连续的内存操作，就会对它们进行检查以找到自动执行的JavaScript代码。由于JavaScript的内存中编码未知，所以必须搜索UTF-8和UTF-16表示。如果发现JavaScript字符串，则这些内存操作将被视为原始分接点。



点按点优化。一旦发现了原始的分支点，我们进行一次使用定义链分析，以查看所标识的内存操作是否可以追溯到任何函数参数。如果是这样，找到JavaScript抽取点的候选人。由于静态分析可能会导致不准确，我们然后执行运行时测试来验证这些候选人。

对于每条内存跟踪都会重复搜索候选项，并将潜在点击点集缩减为在所有处理的内存轨迹中生成嵌入式JavaScript的点。然后使用少量手动分析来确定要使用哪些候选项以及如何从中提取JavaScript。在实践中，这相当于在反汇编器中检查点击点以识别容易产生JavaScript的点。如果找不到合适的分接点，则可重复执行该离线分析，包括额外级别的呼叫上下文。

*D.处理终止和处理错误*

定义。除了提取自动执行的JavaScript之外，还需要确定Adobe Reader是否可以成功打开文件，或者是否由于某种错误而无法执行此操作。我们将这些状态定义为

*处理终止：* Adobe Reader已成功完成与打开PDF相关的所有初始处理的时间点。任何自动执行的JavaScript必须在此之前执行，并且必须完全呈现页面上的所有元素。

*处理错误：*在完成PDF的初始处理之前，Adobe Reader遇到错误导致其中止处理。在这种情况下，Adobe Reader将打开一条警报，向用户描述遇到的错误。

Adobe Reader的行为将永远属于这两种情况之一。除了生成有关样本的有用信息之外，Adobe Reader的这些案例中的其中一个表明何时不再有自动执行的JavaScript提取。通过终止流程，只要有一个流程到达，就可以方便地处理样品。

我们将处理终止和处理错误分接点定义为指令的程序计数器，它们在执行时指示Adobe Reader已达到这些状态中的每一个状态。实际上，我们发现单独的程序计数器足以识别这些点，但如果需要，可以添加调用上下文。

执行跟踪。通过仅分析由Adobe Reader执行的指令来执行识别处理终止和处理错误分接点。由于只使用这些抽头点的执行来揭示有关程序状态的信息，因此可以通过对所执行的基本块执行该分析来简化该分析，而不失一般性。所产生的轨迹是包含每个执行的基本程序段的第一条指令地址的列表。

尤其是，这些跟踪集合ET *JS*，ET *WF*和ET *MF*被收集用于具有JavaScript的格式良好的PDF，没有J​​avaScript的格式良好的PDF以及格式错误的PDF（即导致Adobe Reader产生错误的PDF），分别。更具体地说，ET *JS*包含从用于生成内存跟踪的样本中收集的跟踪以及手动标识为包含JavaScript的其他恶意PDF。ET *WF*中的痕迹是从良性样本中获得的，这些样本经过手动验证不含JavaScript。最后，ET *MF*包含使用和不使用JavaScript的格式错误的样本以及完全不同格式的文件（如PE和DOCX）的跟踪。

收集ET *JS*和ET *WF中的*每条痕迹，直到Adobe Reader出现完成其样本的初始处理，而收集ET *MF*中的痕迹直至Adobe Reader提出警告，表明它无法打开文件。

处理终止。为处理终止点选择的基本块必须满足三个要求：1）基本块总是在处理格式良好的PDF时执行一次且仅执行一次; 2）在处理格式不正确的PDF时，Adobe Reader永远不会执行它; 3）如果PDF包含JavaScript，则仅在JavaScript抽取点后执行。

根据要求1，我们计算一组*BB WFUnique*，它包含ET *WF*中所有迹线共有的基本模块，每个迹线只出现一次。根据要求2，我们需要从*BB WFUnique中*排除ET *MF*中任何跟踪中执行的所有基本块。我们将这些基本块收集在一个*BB MF中*，然后排除它：（*BB WFUnique BB MF*）。由于要求3，我们必须首先找到设置*BB JSTrunc*，它保存在ET *JS的*任何轨迹中执行的所有基本块*之后*到达JavaScript抽取点，然后将交点计算为潜在处理终点抽头点集合：可以选择该集合中的基本块作为处理（*BB WFUnique BB MF*）\ *BB JSTrunc*。任何终端点。

处理错误。与处理错误抽头点关联的基本块必须满足两个要求：1）处理格式不正确的PDF时，基本块总是执行一次且仅执行一次; 2）在处理格式良好的PDF时，Adobe Reader永远不会执行它。为了满足这两个要求，我们计算了三组中间块。具体而言，我们首先计算一组*BB MFUnique*，其中包含所有基本块，对于ET *MF中*所有迹线都是*共有的*，这些块在每条迹线中只出现一次。然后我们排除所有为构造良好的PDF执行的基本块，无论是否带有JavaScript。为此，我们收集了两套*BB JS*和*BB WF*，它们分别表示集合ET *JS*和ET *MF*中的所有基本块。因此，这组潜在的处理误差抽头点被计算为

*BB MFUnique*（*BB JS* [ *BB WF*））。同样，这个集合中的任何基本块都可以用作处理错误抽头点。

*E.点击行动*

一旦确定了分支点后，需要修改Adobe Reader以记录在JavaScript抽取点生成的JavaScript，并在达到处理终止和处理错误分接点时终止。如果抽头点定义包含调用上下文，则这些修改还需要在运行时确定其当前执行是否与该上下文相匹配，否则不应更改该程序的行为。通过检查日志文件的存在并检查过程的退出代码，这是特定于每个点击点的，可以通过编程方式轻松确定已到达哪些点击点以及是否提取了任何JavaScript。

通常情况下，在初始化文件的过程中会自动执行多个JavaScript语句。通常我们希望抓住所有这些，因此Adobe Reader可以运行，直到它到达处理错误或处理终止点。但是，为了处理意外情况，我们在每次达到JavaScript抽取点时设置超时。此可配置超时限制了Adobe Reader在处理特定JavaScript语句时通过在达到时间限制时终止进程的时间。选择超时一秒用于评估，以便有足够的时间执行，同时允许我们处理大量样本。

由于性能问题，我们选择执行热补丁而不是动态仪器（例如Pin [27]）。Microsoft Detours库[21]被选中修改二进制文件，主要是因为它的简单性和易用性，但需要少量的手动工作来将该库与分接点连接起来。在JavaScript抽取点的情况下，这相当于识别目标函数的调用约定和参数。对于处理错误和处理终止分接点，我们不需要关注此分析，因为它们会导致进程立即终止。请注意，必须先禁用Adobe Reader的沙箱机制，以防止处理PDF的进程执行某些操作（如创建文件）。

                            IV。            D 有效的分析

*A.实验设置*

为了评估参考JavaScript提取器的有效性并确定现有提取器的局限性，我们基于Adobe Reader 9.5.0和11.0.08版本生成了两个不同的提取器，然后将它们相互比较，以及其他几个开源工具提供类似的功能。libpdfjs [7]工具是利用Poppler PDF渲染库并为PJScan恶意PDF检测器供电的JavaScript提取器。Origami [8]是一个PDF解析，分析和创建的框架，与使用此框架的JavaScript抽取工具打包在一起。JavaScript解包工具jsunpack-n [6]试图从多种格式中提取和分析JavaScript，但包含一个特别从PDF中提取JavaScript的模块。最后，PDFiD工具不是JavaScript提取器，而只是扫描文档以查看特定关键字的外观。除此之外，VirusTotal使用此工具将样本标记为包含JavaScript或不包含JavaScript [36]，此工具包含的结果是VirusTotal报告的结果。

所有这些工具都是针对2015年2月份从VirusTotal采购的一组163,306 PDF文件运行的。评估的每个工具都有20秒的时间来处理每个样本，然后该工具被视为超时并终止。Origami和jsunpack-n工具在运行Ubuntu 6.04的6GB RAM和2.93GHz CPU的机器上运行。

libpdfjs工具支持的最新操作系统是Ubuntu 11.04，因此它运行在运行此版本的虚拟机中。Adobe Reader工具还必须在Windows虚拟机内运行，该虚拟机特别运行XP Service Pack 3。这两个虚拟机都是

|  |
| --- |
| 表II：JavaScript抽取 |

在与其他提取器相同的裸机上运行，​​并分别获得4GB的RAM。尽管这比其他工具稍微少一些，但实际上，内存使用似乎不是任何提取工具的限制。

由于Adobe Reader工具实际上执行从样本中提取的JavaScript，因此我们必须小心谨慎，以防止Adobe Reader的任何潜在漏洞对未来样本的处理产生影响。为了防止出现这种情况，在处理每个新样本之前，先处理干净状态下的虚拟机快照，然后进行恢复。可以在https：// goo找到与结果的某些部分相对应的哈希列表*。*gl / qtbuOC和每个描述。

*B.总结结果*

表II中列出了每个工具提取至少一个JavaScript项目的样本总数以及它们与每个参考提取器的结果比较。请注意，对于PDFiD，列出了它标识为包含JavaScript的样本数。综合起来，我们评估的开源提取器为5250个独特样本生成了JavaScript。我们的9.5.0版参考提取器从4397个样本中生成JavaScript，其中2956个为良性，1441个为恶意，我们的版本11.0.08参考提取器从4704个样本生成JavaScript，其中3261个是良性的，1443个是恶意的， VirusTotal至少有15次检测为恶意。总共，所有提取器都为5267个独特样本生成了JavaScript。

表III显示了两种不同的参考提取器之间的比较。每个提取器可以生成JavaScript的文件在很大程度上都是相似的，并且所有生成的提取都与下面描述的相似性度量相匹配。特定参考提取器独有的提取中，除两个之外的所有提取都是良性的，仅由版本11.0.08提取器生成。这两个样本的格式不正确，允许多种解释，并且似乎不同版本的Adobe Reader选择不同的解释。版本11.0.08提取器独有的大量提取主要是由使用旧版本Adobe Reader不支持的功能的样本引起的，并且9.5.0提取器特有的提取很大程度上是由使用现在不推荐使用的功能的样本造成的。

由于缺乏基本事实，验证这些提取的正确性非常困难。因此，我们试图确定哪些提取会产生与参考提取程序发现的JavaScript至少部分匹配的结果。尽管我们假设我们的参考提取器生成的JavaScript是正确的，但它仅提取打开PDF时自动执行的JavaScript代码，该代码可能只是文档中JavaScript的一个子集。

由于其他提取器试图生成嵌入在文档中的所有JavaScript，因此参考提取器生成的JavaScript应至少匹配其他提取器生成的某些JavaScript。如果找到PDF样本的这种部分匹配，我们称之为“匹配”。如果抽取器对于我们的参考抽取器产生抽取的样本不产生任何东西，我们将其称为“零”。如果提取器产生的提取与我们工具的提取完全不匹配，并且手动验证为无效，则这被认为是“无效提取”。由于PDFiD实际上并未提取JavaScript，因此无效提取的概念不适用于此工具。

为了安全起见，我们没有推断抽取参考样本没有产生样本的抽取的有效性。结果，提出的无效提取量仅代表下限。我们称这些情况“不确定”。

为了识别匹配的提取，我们首先查找子串匹配。由于不同的提取器会产生空白区略有差异的提取，例如不同的结束序列，并且现有的提取工具在处理非ASCII可编码字符时经常遇到问题，因此我们在此搜索中不考虑它们。如果找不到子字符串匹配，我们将使用Python difflib [4]库有效地计算提取之间的相似性启发式。如果抽取与我们的工具产生的抽取看起来至少有50％相同，则认为抽取匹配。

虽然在大多数样本中，现有提取器生成的JavaScript与我们的工具相匹配，但结果显示先前的提取器错过了大量PDF文档的JavaScript提取。平均而言，这些提取器会在10.1％的文件中丢失JavaScript代码，与11.0.08版本的参考提取器相比，PDFiD无法检测到17.01％的样本中存在JavaScript。对于恶意PDF，情况更为严重。平均而言，每个现有的提取器无法从22.47％的由参考提取器标识为包含JavaScript的恶意样本中提取JavaScript。

请注意，每个提取器的“无效”和“零”类别中的样本被分解为良性和恶意类别，正如VirusTotal的检测器所识别的。从这些类别中分析出来的许多恶意样本都存在混淆现象，似乎是解析器混淆攻击，表明攻击者已经意识到了一些解析器的弱点，并且正在积极利用它们来企图逃避检测。jsunpack-n提取器在处理这些样本中效率最高，并且似乎已经被编码以专门解决许多规避问题。

表III：JavaScript抽取（版本9.5.0和11.0.08）

                       9.5.0仅          11.0.08只有          普通          总数

F:\PDFpy\God_with_me\2018Q2\paper\1_T_ExtractMeIfYouCanAbusingPDFParsersInMalwareDetectors_NDSS2016\ExtractMeIfYouCan.files\image016.gif                      21                         328                           4376                  4725

尽管这些样本已被VirusTotal上的许多检测器分类为恶意，但它们利用来逃避提取的弱点对于这些检测器仍然有效，如第五部分所示，表明这种分类可能基于JavaScript分析之外的因素。JavaScript无法从许多良性样本中提取的事实要么表明这些提取器不完整，要么解析PDF与Adobe Reader不同。

*C.点击验证*

我们的参考提取器不生成来自546个样本的JavaScript（其中只有10个满足我们的恶意定义），其中至少有一个其他提取器能够提供。为了演示参考提取器只能从不包含自动执行JavaScript或者格式不正确的PDF中提取JavaScript，而不是因为技术失败或者由于选择错误的抽头点，我们会进一步执行验证。通过在执行监视器中使用原始Adobe Reader二进制文件打开这些示例，我们可以观察程序的行为以确定它是否处理其他提取器生成的任何JavaScript。

这些样本中的每一个都由Adobe Reader打开，然后给出三十秒来加载EScript.api模块，并知道它包含JavaScript引擎。如果在这段时间内加载模块，则允许Adobe Reader运行另外四分钟来收集内存跟踪。

在546个样本中，只有274个实际加载了EScript.api模块。对于其余的274个样本，我们将内存操作从它们的轨迹分组到相邻的操作中，然后将这些操作访问的数据与其他工具产生的抽取进行比较。在这些样本中产生的所有连续操作中，仅识别出1006个唯一匹配字符串。

绝大多数这些字符串都是来自Adobe JavaScript API或JavaScript关键字的模块或函数名称。一些小的JavaScript片段被识别出来。手动分析表明这些片段只是在模块内生成的JavaScript语句之间的子字符串匹配，可能作为JavaScript引擎初始化的一部分执行。在这些连续的操作中没有发现任何提取的完整JavaScript语句，这表明参考提取器已经在格式完整的PDF中正确捕获了所有自动执行的JavaScript。

*D.教训*

在确定了包含JavaScript但其中一个提取器未正确处理的样本之后，我们开始着手确定这些失败的原因。在很多情况下，这些失败的原因很容易被识别为一个不完整的解析器实现，因为许多工具都知道它们的一些局限性，并会输出消息

预告片<< / Root 1 0 R / Size 8 >>

（a）  原始预告片

预告片<< / Root％！@＃！@＃1 0 R / Size 8 >>

（b）  带注释评论的预告片

/ XFA'[（config）42 0 R（template）％195 0 R 111 0 R（datasets）44 0 R（localeSet）45 0 R]'

（c）   具有注入评论的缩写XFA条目

图3：注释注入

这表明他们遇到了他们无法处理的规范方面。在其他情况下，提取器的源代码和样本本身被手动分析以确定失败的原因，这通常是设计错误或实施错误的结果

通过检查单个提取器唯一产生正确输出的样本，我们能够确定提取器为什么成功，而其他则不是。因此，虽然只对这些样本的相对较小的子集进行分析，但我们能够确定这些提取器中的几个弱点。表四概述了这些限制及其对JavaScript提取器的影响，通常可将这些限制分为四类。

实施错误。通常，提取器会正确解释规范，但在实现时会出现编程错误。虽然这些错误和错误中的很多很容易解决，但枚举却很困难。

PDF规范规定，以“％”开始并以换行符序列结尾的注释应被忽略并视为单个空白字符。尽管规范的这一方面很简单，但jsunpack-n或Origami工具都不会正确解析注释。如图3a和3b所示，将注释注入PDF的预告片中会导致折纸工具过早终止并阻止提取JavaScript。同样，注入字典的评论也阻碍了jsunpack-n工具。如图3c所示，字典中的缩写XML Forms Architecture（XFA）条目演示了此错误。在这种情况下，jsunpack-n没有意识到应该忽略“％195 0 R”字符串。不是查找包含恶意负载的ID为“111 0”的对象，

流数据将由行尾标记结尾，后跟“endstream”关键字。但是，正则表达式jsunpack-n用于在“endstream”之前识别与零个或多个换行符匹配的流数据。这意味着在碰巧具有与换行符相关联的值的流中的尾随字节将被错误地认为是空白而不是流数据。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 表四：失败和限制   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  |  | 受影响的提取器 | | | | libpdfjs | jsunpack-N | 折纸 | | 实施错误 | 在预告片中评论 | 7 | 7 | 3 | | 在字典中评论 | 7 | 3 | 3 | | 尾随流数据中的空白 | 7 | 3 | 7 | | 安全处理程序版本5十六进制编码的加密数据解析 | 7 | 3 | 7 | | 安全处理程序修订版3,4的加密密钥计算 | 7 | 3 | 7 | | 编码对象中的十六进制字符串文字 | 7 | 3 | 7 | | 设计错误 | 使用孤立的加密对象 | 7 | 3 | 3 | | 没有加密元数据的安全处理程序修订版5加密密钥计算 | 7 | 3 | 7 | | 遗漏 | 无XFA支持 | 3 | 7 | 7 | | 没有安全处理程序版本5支持 | 3 | 7 | 7 | | 没有安全处理程序版本6支持 | 3 | 3 | 7 | | 歧义 | 没有交叉引用表和无效的对象关键字 | 7 | 7 | 3 | |

该规范允许多种不同的加密方案和算法。在这些算法中生成和应用加密密钥相当复杂，并且可能取决于文档的其他几个特征。规范如何执行的规范和算法被称为“安全处理程序”，随着规范的发展，正在开发几个修订版本。根据PDF规范，可以使用空白的“默认”密码来应用加密算法，这意味着即使文件的某些内容存储在密文中，任何正确实现该算法的解析器都可以解密并检查它们。Jsunpack-n似乎已经正确解释了修订3和4安全处理程序的加密密钥生成算法; 然而，

十六进制字符串文字有时不能被jsunpack-n正确处理。当这些字符串文字置于编码对象的内部时，它们在对象解码后没有被正确解析。此外，jsunpack-n无法解析用于使用修订版5安全处理程序存储加密数据的十六进制字符串。

设计错误。这些是提取器似乎错误地解释了规格或者故意采用快捷方式来简化开发的情况。这些错误在规范的更复杂方面很常见，例如文档加密，这些文档加密很难正确解释，或者在破坏开发人员假设的特殊情况下解决。

例如，jsunpack-n和Origami会为定义加密参数的对象扫描文档，如果找到，则使用它们来解密规范说明应加密的所有内容。但是，PDF规范中的增量更新机制允许创建不再加密的更新文件。这样的更新不会删除旧内容，而会生成一个新的文档结构，以停止引用旧对象。因此，定义加密参数的对象的存在不一定意味着文档的当前版本被加密。然后这些提取器将错误地解密已经以纯文本形式的数据，产生“垃圾”数据。

根据文档的元数据是否被加密，修订版5安全处理程序有两个稍微不同的密钥生成算法。这些算法没有被jsunpack-n正确解释。因此，当元数据被加密时，它只能产生正确的密钥。

遗漏。没有一家提取器评估声称完全实施了所有的PDF规范及其扩展。通过使用规范的这些未实现的方面，攻击者从提取器隐藏恶意内容是微不足道的。

libpdfjs提取器的遗漏最大，这主要是由于它依赖于老版本的Poppler解析器，该解析器没有实现更新的规范添加（例如修订版本4和5安全处理程序）。Poppler解析器也不支持PDF规范的XFA扩展，该规范通常用于将JavaScript嵌入到PDF中。尽管Origami或jsunpack-n提取器都不支持XFA，但它们支持足够的规范来识别和提取以这种方式嵌入的JavaScript。

只有Origami工具支持版本6安全处理程序，这是处于开发中的PDF 2.0规范的一部分[14]。尽管该算法不是PDF规范的正式部分，但它仍被Adobe产品使用，因此任何有效的恶意PDF检测器也必须这样做。

歧义。PDF规范在某些情况下含糊不清，留下了多种解释空间。同样的，Adobe Reader试图“正常工作”，通常会处理偏离规范的PDF。由于规格不包括这些情况，因此不清楚应如何处理。寻找这些含糊不清的东西非常困难，并且决定如何处理。

例如，PDF规范指出，所有PDF文档都应包含一个“交叉引用”表或流，其中包含有关文件中所有对象及其位置的信息。如果一个文档没有这个表格，Adobe Reader和我们评估的所有提取器都会尝试通过扫描文档中的对象来重建此表格，这通常在文档中的对象格式正确时成功。

在恶意PDF中经常出现的另一个歧义是使用格式错误的“objend”关键字来终止对象。规范规定对象应该用“endobj”关键字来终止，但Adobe Reader和所有现有的提取器都会通过接受两者来偏离规范。当这两个歧义组合在包含以不正确的“objend”关键字终止的对象并且没有交叉引用表的文档中时，Origami不能识别对象并且不能解析文档。

                        V.            P ARSER C FORFUSION A TTACKS

*A.攻击定义*

通过系统地研究提取器的这些弱点并确定它们的根本原因，我们可以对其他PDF文件进行修改，我们称之为混淆，这些文件利用这些弱点并阻止JavaScript提取。我们对导致这些弱点的解析器限制的理解也使我们能够开发新的混淆，这也阻止了抽取。由于Adobe Reader对文件的处理及其对嵌入式JavaScript的执行不受影响，因此这些混淆的应用程序会阻止任何基于JavaScript的恶意PDF检测，同时不会影响漏洞利用的功效。我们称这些混淆的应用程序具有允许恶意PDF逃避检测的特定目的，*PDF解析器混淆攻击*。

虽然我们只专门分析了我们评估的开源提取器的弱点，但我们能够确定这些提取器中可能存在于其他PDF解析器中的几个常见弱点。为了确定这些弱点的普遍性以及解析器混淆攻击的强度，我们将模糊处理应用于包含工作漏洞的PDF，然后评估不同的JavaScript提取器和恶意PDF检测器是否仍然能够正确提取JavaScript或对其进行分类抽样为恶意。通过识别几个强烈的混淆并将它们组合在一起，我们能够阻止所有评估的JavaScript提取器，VirusTotal上的所有商业AV产品以及基于元数据的PDFrate检测器。

*B.攻击构造*

为了演示这些攻击在逃避检测中的有效性，使用Metasploit模块[3]创建了一个恶意PDF，该模块利用Adobe Reader [12]版本9.0.0至11.0.3中的“免费使用”漏洞作为有效载荷。我们嵌入到这个恶意示例中的有效载荷在网络上的另一台机器上打开了一个反向shell，并且对于应用于原始示例的每个修改，验证了该漏洞的功能。

最初的Metasploit模块生成了一些混淆，这些混淆在我们的任何之前首先被删除

|  |  |  |
| --- | --- | --- |
| 3 0 obj  << / JS 6 0 R / S / JavaScript /类型/  行动>> endobj  ...  6 0 obj  << /长度3907 >>  流  函数heapSpray（str，str\_addr，r\_addr） { ... } endstream endobj  （a）恶意JavaScript和引用未被混淆。 | 3 0 obj  << / JS 6 0 R / S / JavaScript /类型/  行动>> endobj  ...  6 0 obj  << /长度1552 /过滤器/ FlateDecode  >>流  <编码的JavaScript> endstream endobj  （b）恶意JavaScript被编码，但引用没有被混淆。  图4：流混淆 | 2 0 obj  << / Type / ObjStm /长度1696 /过滤器  / FlateDecode / N 4 / First 20 >>流  <编码对象>  endstream endobj  （c）放入流中的对象，然后进行编码。恶意的JavaScript和引用被混淆。 |

这样会产生一个明确标识为恶意的样本，并且可以被所有提取器正确处理。除了在使用十六进制编辑器轻松应用的文档预告片中嵌入注释之外，所有这些修改均使用开源qpdf工具[13]或PyPDF2 Python库[11]进行。

一般来说，这些攻击背后的目的是使用检测器分析程序不支持或处理不正确的规范方面来掩盖恶意样本的有效载荷。图4概括了一个简化的例子，演示了这种攻击如何从图4a中未混淆的恶意内容开始。

通过对文档中的对象流应用过滤器或编码，如图4b所示，所有不支持所使用编码的检测器都会遮住恶意JavaScript。但是，文档中包含JavaScript的事实并未隐藏，可能会与其他启发式技术一起使用，将样本分类为恶意。如果在对流数据进行编码之前首先将对象放置在流中，如图4c所示，则对于未正确处理规范的这些方面的恶意软件检测器，不存在对嵌入式恶意JavaScript的指示。

为了评估这些针对VirusTotal上的商业恶意软件检测器的攻击的有效性，我们从基本未混淆的恶意文件开始，然后应用不同的混淆，包括Flate压缩，R5和R6安全处理程序，十六进制编码等。通过确定哪些混淆是成功的以及他们回避的探测器，我们能够结合几个迷惑，以最大限度地提高逃避探测器的数量。虽然许多这些混淆的应用很容易，但它们在实践中是强大的。

由于知道这些解析器混淆攻击可能不足以单独遏制基于元数据的检测系统，因为样本的核心内容没有改变，所以我们将解析器混淆攻击与反向模仿攻击相结合[28]。为了应对这种攻击，我们从恶意样本中删除了有效载荷，并将其应用于良性根文件，这是一份来自IRS网站的税务表格[[2]](file:///F:\\PDFpy\\God_with_me\\2018Q2\\paper\\1_T_ExtractMeIfYouCanAbusingPDFParsersInMalwareDetectors_NDSS2016\\ExtractMeIfYouCan.htm" \l "_ftn2" \o ")。我们

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 表五：解析器对商业探测器和JS提取器的混淆攻击   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 困惑 | MD5哈希 | 检测率 | O 1 | l 2 | P 3 | j 4 | | 没有 | ae91ec6a96dc4d477beba9be6b907568 | 30/55 | 3 | 3 | 3 | 3 | | Flate Compression，物流 | eb64df4dbd733b5aa72fb0c41995f247 | 24/56 | 3 | 3 | 7 | 3 | | Flate Compression，R5安全处理程序 | 2b1071b27f96d9cdcfc59e35040d28b7 | 56分之19 | 3 | 7 | 3 | 7 | | Flate压缩，R5安全处理程序，对象流 | 8887439e33d15bcc8716634cbcbb392e | 14/54 | 3 | 7 | 7 | 7 | | Flate Compression，R6安全处理程序 | 4e05ad44febe26f25629f27c155a7a0e | 57分之4 | 3 | 7 | 3 | 7 | | Flate Compression，R6安全处理程序，对象流 | c82643a1388a2645409395ef3420d817 | 0/56 | 3 | 7 | 7 | 7 | | Flate Compression，R6安全处理程序，对象流，预告片中的评论 | 6b6abbce700027f7935e3eeacd43618d | 0/57 | 7 | 7 | 7 | 7 | | JS以十六进制字符串编码为UTF-16BE | ab09a01fe61a1066f814e3ffc2548f0a | 55分之23 | 3 | 3 | 3 | 3 | | JS以十六进制字符串编码为UTF-16BE。Flate压缩，对象流 | b21e264efbb14b928f0121b22030c3a7 | 10/55 | 3 | 3 | 7 | 7 | | JS编码为UTF-16BE的十六进制字符串，Flate Compression，R5安全处理程序，对象流，注释在预告片中 | 5039c273435300a46cd42ad0de0bb4ff | 1/57 | 7 | 7 | 7 | 7 |   F:\PDFpy\God_with_me\2018Q2\paper\1_T_ExtractMeIfYouCanAbusingPDFParsersInMalwareDetectors_NDSS2016\ExtractMeIfYouCan.files\image017.gif1折纸 2 libpdfjs 3 PDFiD 4 jsunpack-n  表六：PDFrate逃避   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 样品 | MD5哈希 | Contagio  恶意软件转储 | 乔治梅森  大学 | PDFrate社区 | | 未混淆的恶意文件 | ae91ec6a96dc4d477beba9be6b907568 | 86.4％ | 89.6％ | 91％ | | 恶意软件仅解析器混淆攻击 | 6b6abbce700027f7935e3eeacd43618d | 70％ | 65.8％ | 82.2％ | | 良性的根文件 | 303b209708842adf30b81f437c5ec0ed | 0.7％ | 13.9％ | 13.5％ | | 根文件w /解析器混淆+反向模仿攻击 | d48a343058503f931eadec99f3a89e70 | 7.8％ | 2.3％ | 11.0％ | |

然后应用模糊处理来成功地阻止商业恶意软件检测器对该样本进行评估，以评估其对检测器的影响。这种攻击是根据PDFrate分类器[32]进行评估的，该分类器已作为公开的在线服务发布[9]。

*C.攻击有效性*

商业探测器和JS提取器。表V列出了一系列不同的解析器混淆攻击，这些攻击使用此示例挂载，并对VirusTotal上的两个检测器以及我们在差异分析中使用的JavaScript提取器的影响进行了检测。此表列出了AV扫描仪的“检测比率”，即扫描仪的数量，该扫描仪将文件标识为可能及时返回结果的扫描仪数量为恶意文件。此外，还提供了每个样本的MD5校验和，可用于从VirusTotal获取有关该文件和扫描的附加信息。有足够权限访问VirusTotal API的研究人员也可以获取这些文件的副本，但必须谨慎行事，因为它们确实包含有效的漏洞利用。

从大部分探测器识别为恶意的完全未混淆的恶意文件开始，我们开始应用不同的混淆。每个解析器都有自己的弱点，通过将不同组合的混淆结合起来，我们可以利用几个解析器中的弱点来降低检测率。我们还可以通过分析混淆来降低检测率，分析表明解析器不太可能预期或正确处理。通过将这些技术与对象流的使用相结合来增加被编码的文档中的信息量，我们能够确保检测需要成功解析，同时降低检测器能够这样做的机会。

尽管常见的编码（如Flate压缩）只会稍微降低检测率，但更复杂的修订版本5（R5）安全处理程序和隐蔽修订版本6（R6）处理程序的应用程序在避开检测中非常有效。尽管折纸工具正确地实现了修订版本6的安全处理程序，但通过在文档预告片中注入评论，生成完全避开检测的样本，很容易受到挫败。最后，使用Flate压缩，修订版本6安全处理程序，对象流和注释中注释的注释相结合的示例在VirusTotal上产生零检测结果，并且可以击败所有JavaScript提取器。

同样，只应用相对简单和深入理解的UTF-16BE和十六进制编码，检测率只会稍微降低。从我们的分析中知道，许多解析器处理相对简单的编码，但没有正确处理复杂的组合，我们将这些与Flate压缩和加密结合起来，几乎完全避开了检测。事实上，这可以在没有使用未经批准的版本6安全处理程序的情况下完成，这表明即使在多年存在的规范中仍然存在许多需要解决的缺陷。

基于元数据的检测器。表六显示了解析器混淆攻击对PDFrate的有效性。我们评估了此分类器的版本，该分类器作为在线服务公开可用。分类器通过使用针对三种不同数据集，Contagio恶意软件转储，乔治梅森大学收集的样本以及PDFrate社区提交的样本训练的模型，产生样本具有恶意的可能性[9]。PDFrate没有提供用于确定样本是否有恶意的阈值，但攻击的影响是明确的。

分类器可以正确识别Metasploit生成的未混淆的恶意文件为恶意文件，尽管我们可以使用完全阻止VirusTotal上的检测器的相同混淆（即R6安全处理程序，对象流，预告片评论）来降低其恶意评级，但我们无法这样做显着。这是由于PDFrate没有根据内容检测恶意软件

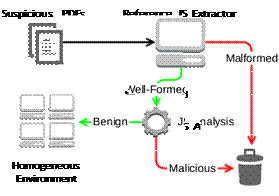


图5：可能的部署方案

嵌入式JavaScript而不是文件的元数据，而且这些混淆不会显着改变用于分类的许多特征（例如文件大小）。相反，反向模仿攻击的应用产生了被分类为不那么恶意的样本。但是，取决于用于分类的阈值，样本可能仍然显示为可疑。

除了对该文件进行反向模仿攻击之外，通过安装解析器混淆攻击，分类器将该样本识别为恶意的能力显着降低，直至其被分类为比原始样本更好的两个点分类。由于PDFrate的基于正则表达式的解析器无法执行解密文档所需的计算，因此大部分文档对分类器来说都是大型随机数据流。因此，将用于将样本分类为恶意的样本的许多特征被解析器混淆攻击掩盖。

                                       VI。        M ITIGATIONS

为了缓解解析器混淆攻击，可以使用三类缓解方法。

*A.基于运行时的利用检测*

第一种可能的解决方案是在运行时完全捕获JavaScript执行。由于解析器混淆攻击只能解决静态解析和进一步的分类工作，因此它不会阻止JavaScript执行，因此，当Adobe Reader打开托管PDF文件时，观察到这种情况。

已经提出了先前的恶意软件检测系统，虽然该恶意软件检测系统没有专门用于检测恶意PDF，但可以通过分析目标应用程序的运行时行为来检测恶意软件的存在。例如Nozzle [30]和ShellOS [33]都被证明能够有效检测针对Adobe Reader的攻击。

尽管这些工具能够完全规避与解析相关的问题，但它们有很大的开销。这些系统不仅取决于恶意JavaScript的执行情况，还取决于实际执行攻击的情况，以便将PDF分类为恶意含义，即恶意的JavaScript有选择地执行或无法正确执行的恶意JavaScript无法正确分类。

表VII：PJScan性能

|  |  |  |
| --- | --- | --- |
| 工具 | 真正的积极 | 假阳性 |
| 原始PJScan | 68.34％（1453） | 0.18％（3814） |
| PJScan＆Adobe Reader 9.5.0 | 96.04％（1441） | 0.32％（3521） |
| PJScan和Adobe Reader 11.0.08 | 94.02％（1021） | 0.20％（3677） |

表VIII：平均运行时间

|  |  |
| --- | --- |
| 工具 | 平均。运行时（s） |
| libpdfjs | 0.05 |
| jsunpack-N | 0.78 |
| 折纸 | 1.86 |
| 参考JS提取器 | 3.93 |

*B.解析器的改进*

第二个可能的解决方案是改进用于恶意PDF分类器的现有解析器。由于提出的攻击关注第三方PDF解析器与Adobe Reader之间的差异，如果我们可以提高解析器的质量并解决这些差异，则攻击本质上可以被击败。

但是，一个程序一般很难模拟另一个程序的行为。尽管我们的参考提取器可以帮助识别其他PDF解析器的弱点，但绝不能保证改进的解析器忠实地遵循Adobe Reader的解析逻辑。即使可以开发完美的解析器，但由于它们具有独特的解析行为，因此必须对每个Adobe Reader版本重复执行此项工作。

*C.参考提取器的部署*

鉴于现有解析器的改进取决于参考提取器的存在，因此仅使用参考提取器就可以更简单。图5演示了减轻攻击的第三种可能性，它在现实世界中部署了参考JavaScript提取器和基于JavaScript的分类器。由于参考提取器只能精确地模仿它所基于的Adobe Reader版本的行为，因此它最适合于已知使用的Adobe Reader版本的受控，相对均匀的环境（例如企业）。

为了演示使用参考提取器可以提高现有基于JavaScript的分类器的准确性，我们比较了使用libpdfjs作为其提取器的原始PJScan检测器与使用参考JavaScript提取器的修改版本的性能。

由于PJScan工具只能对包含JavaScript的样本进行分类，因此我们仅针对任何提取器能够生成JavaScript的样本进行评估。虽然我们无法确定每个样本实际上是否包含JavaScript，但仍可证明检测的改进。由于参考提取器也能够识别哪些样本是畸形的，我们在评估中排除了这些样本，认为格式错误的文件可以被阻止而不会对最终用户产生不利影响。

由于PJScan采用一类支持向量机，在对任何样本进行分类之前需要针对恶意集合进行训练，进行双重交叉验证。表VII显示了该评估的结果，每个集合中的样本数量显示在括号中（对于参考提取器，这些是他们认为良好形成的样本的数量）。可以看出，我们能够极大地提高PJScan通过使用参考提取器来检测恶意PDF的能力。

这种改进的明显原因是，参考JavaScript提取器能够从比libpdfjs更多的恶意样本中提取JavaScript。在报告的1453个恶意样本中，包含JavaScript的任何工具报告为libpdfjs仅能够产生1021个提取，而版本9.5.0和11.0.08参考提取程序会从他们识别的1441和1021恶意文件中为1429和1013产生提取分别形成。

此外，由于只有每个版本的Adobe Reader可以实际处理的样本用于训练PJScan，因此训练数据中的噪音较少，并且可以生成更好的模型。例如，我们使用的Adobe Reader版本不会打开包含许多已修补的旧版漏洞的样本。通过使用参考提取器丢弃这些较旧的畸形样本，可以训练分类器并仅针对较新的漏洞进行评估，从而提高其准确性。这似乎也是为什么版本9.5.0提取器的性能稍微好一点 - 因为我们能够过滤出不能被此版本打开的较新的PDF，所以剩余的恶意样本之间存在更多相似性。

然后，我们使用我们混淆的样本来测试PJScan和参考JavaScript提取器的有效性。结果表明，当PJScan与版本11.0.08参考提取器配对使用时，如果在训练集中使用包含相同恶意负载的PDF，则现在可以检测解析器中使用的所有样本混淆攻击。与版本9.5.0参考提取器配对使用时，PJScan可以检测除使用R6安全处理程序的那些样本以外的所有样本，因为该版本不支持该样本。

表VIII显示了从VirusTotal获得的所有样本的每个评估的JavaScript提取器的平均运行时间。正如可以预料的那样，Adobe Reader工具在每次迭代之后都必须将虚拟机恢复到干净状态中付出重大代价。请注意，参考提取器的性能与MPScan相差无几，即使它们似乎不能在样本之间重置系统[26]。使用参考监视器而不是动态挂钩也需要重置系统，并且比运行虚拟机要慢很多。

在该系统的实际实施中，可以执行某些优化。例如，通过将虚拟机放在RAM磁盘上而不是硬盘驱动器上，我们可以在虚拟机快照恢复上节省大约2秒的时间。由于可以在提取之后执行快照恢复，因此可以大大减少接收样本分析的延迟时间，并且可以生成一系列分析器来缓解剩余的开销。

                                       七。         L 模仿

参考提取器的主要限制是它只能提取由Adobe Reader自动执行的代码。此外，由于我们无法无限处理任何单个样本，因此我们可能无法提取拖延执行或未在分配时间内完成的JavaScript。然而，实际上，这些问题在恶意PDF检测方面似乎并不重要。在参考提取器没有从1453中提取JavaScript的10个恶意文件中，一个是畸形的，其余9个依赖于用户交互。从根本上说，这些限制是由于使用动态分析而引起的，但正如我们的评估所显示的，静态分析也有其自身的局限性。

恶意软件通常使用“反VM”或“反沙箱”技术来避免通过选择在虚拟环境中表现出恶意行为来进行检测。虽然我们并不知道恶意PDF所使用的任何技术，并认为通过Adobe Reader JavaScript API可获得的有限系统信息会使其难以实现，但我们不能声称这种检查是不可能的。例如，高级攻击者可能会测试是否禁用沙箱，这是引用提取程序运行所必需的。但是，任何此类检查都必须取决于执行一些JavaScript，这些JavaScript将被提取出来，并可用于将文档分类为恶意或至少可疑。

                                      八。         C 结论

在本文中，我们对一种称为*PDF解析器混淆攻击*的新规避技术进行了系统研究，该技术旨在混淆恶意软件检测器中的PDF解析器以逃避检测。为了进行系统的研究，我们开发了一个*参考JavaScript抽取器*，直接使用Adobe Reader并提供了一种用于开发它的大部分自动化技术。通过研究参考提取器和现有提取器之间的这些差异，我们发现了几个新的混淆，并进一步量化了它们在JavaScript提取器和恶意软件检测器的解析器混淆攻击中的影响。通过结合其中几个混淆，我们制作了一个可以成功回避*所有问题*的恶意PDF评估恶意软件检测器，包括基于签名的，基于结构/元数据的和基于JavaScript的检测器。为解决解析器混淆攻击，我们讨论了几种缓解技术。特别是，我们证明，使用我们的参考JavaScript提取器，现有分类器的检测率已经从我们的样本集的68％显着增加到96％，并为参考提取器提供了可能的部署方案。

一个CKNOWLEDGEMENT

我们想感谢匿名审稿人和我们的牧羊人Guofei Gu博士的深刻反馈。本研究部分得到美国国家科学基金会拨款＃1054605，空军研究实验室拨款＃FA8750-132-0115和＃FA8750-15-2-0106以及DARPA拨款＃FA875014-C-0118的支持。本文中的任何观点，发现和结论都是作者的观点，结论和结论，并不一定反映出资助机构的观点。

R EFERENCES

[1]        “Adobe acrobat reader：2014年发布的安全漏洞”，http：// www *。*Cvedetails *。*com / vulnerability-list / vendor id-53 / product id497 / year-2014 / Adob​​e-Acrobat-Reader *。*HTML。F:\PDFpy\God_with_me\2018Q2\paper\1_T_ExtractMeIfYouCanAbusingPDFParsersInMalwareDetectors_NDSS2016\ExtractMeIfYouCan.files\image033.gif

[2]        “Adobe acrobat reader：2015年发布的安全漏洞”，http：// www *。*Cvedetails *。*com / vulnerability-list / vendor id-53 / product id497 / year-2015 / Adob​​e-Acrobat-Reader *。*HTML。

[3]        “CVE-2013-3346    Adobe          Reader         ToolButton   使用    后  免费”，http：// www *。*rapid7 *。*com / db / modules / exploit / windows / browser / adobe toolbutton。

[4]        “difflib - 助手计算三角洲”，https：// docs *。*python *。*org / 2 / library / difflib *。*HTML。

[5]        “介绍Adobe Reader保护模式”，http：// blogs *。*土坯*。*com / security / 2010/07 / introduction-adobe-reader-protected-mode *。*HTML。

[6]        “jsunpack-n”，http：//代码*。*谷歌*。*COM / P / jsunpack-N /。

[7]        “libpdfjs”，http：// sourceforge *。*净/项目/ libpdfjs /。

[8]        “折纸”，https：//代码*。*谷歌*。*COM / P /折纸PDF /。

[9]        “PDFrate基于机器学习的分类器，基于文档元数据和结构”，http：// pdfrate *。*COM /。

[10]     “Poppler”，http：// poppler *。*freedesktop *。*组织/。

[11]     “PyPDF2”，https：// github *。*COM / mstamy2 / PyPDF2。

[12]     “漏洞详细信息：CVE-2013-3346”，http：// www *。*Cvedetails *。*com / cve / 2013-3346。

[13]     J.Berkenbilt，“QPDF：内容保留PDF转换系统”，http：// qpdf *。*sourceforge *。*净/。

[14]     G. Delugre，“adobe reader x的无证密码验证算法”，http：// esec-lab *。*sogeti *。*COM /后/的-undocumentedpassword验证算法-的-的Adobe阅读器-X。

[15]     B.Dolan-Gavitt，T.Leek，J.Hodosh和W.Lee，“Tappan zee（north）bridge：mining memory *access* for introspection”，在*Proceedings of the 2013 ACM SIGSAC conference on Computer＆communications security*。ACM，2013年，第839-850页。

[16]     P. Fogla和W. Lee，“Evading network anomaly detection systems：Formal reasoning and practical techniques”，*Proceedings of the 13th ACM Conference on Computer and Communications Security*，2006。

[17]     P. Fogla，M. Sharif，R. Perdisci，O. Kolesnikov和W. Lee，“Polymorphic blending attacks”，*在USENIX安全研讨会第15次会议论文集 - 第15卷*，2006年。

[18]     M. Garnaeva，V. Chebyshev，D. Makrushin，R. Unuchek和A. Ivanov，“卡巴斯基安全公报2014”。http：

//安全列表*。*COM /分析/ kasperskysecurity-公告/ 68010 / kasperskysecurity -公告- 2014年总体统计，对2014 /。

[19]     D. Goodin，“官方：Adobe读者是世界上利用率最高的应用程序”，http：// www *。*家伙*。*合*。*2010/03/09 / Adob​​e Reader读者攻击/，2010。

[20]     A. Henderson，A. Prakash，LK Yan，X. Hu，X. Wang，R. Zhou，and H. Yin，“让它工作，做对，快速建立一个平台中立的整体系统动态二进制分析平台“ *，在2014年国际软件测试和分析研讨会会议记录中*，ser。ISSTA 2014.纽约，纽约州，美国：ACM，2014，第248-258页。

[线上]。可用：http：// doi *。*acm *。*org / 10 *。*1145/2610384 *。*2610407

[21]     G. Hunt and D. Brubacher, “Detours: Binary interception of win32 functions,” in *Third USENIX Windows NT Symposium*. USENIX, July 1999, p. 8. [Online]. Available: http://research*.*microsoft*.*com/ apps/pubs/default*.*aspx?id=68568

[22]     S. Jana and V. Shmatikov, “Abusing file processing in malware detectors for fun and profit,” in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 80–94. [Online]. Available: http://dx*.*doi*.*org/10*.*1109/SP*.*2012*.*15

[23]     P. Laskov and N. Srndiˇ c, “Static Detection of Malicious JavaScript-´ bearing PDF Documents,” in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC ’11. New York, NY, USA: ACM, 2011, pp. 373–382. [Online]. Available: http://doi*.*acm*.*org/10*.*1145/2076732*.*2076785

[24]     M. Lee, “GNU PDF project leaves FSF High Priority Projects list; mission complete!” https://www*.*fsf*.*org/blogs/community/gnu-pdf-projectleaves-high-priority-projects-list-mission-complete.

[25]     D. Liu, H. Wang, and A. Stavrou, “Detecting Malicious Javascript in PDF Through Document Instrumentation,” in *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN ’14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 100–111. [Online]. Available: http://dx*.*doi*.*org/10*.*1109/DSN*.*2014*.*92

[26]     X. Lu, J. Zhuge, R. Wang, Y. Cao, and Y. Chen, “De-obfuscation and Detection of Malicious PDF Files with High Accuracy,” in *46th Hawaii International Conference on System Sciences, HICSS 2013, Wailea, HI, USA, January 7-10, 2013*, 2013, pp. 4890–4899. [Online]. Available: http://dx*.*doi*.*org/10*.*1109/HICSS*.*2013*.*166

[27]     C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[28]     D. Maiorca, I. Corona, and G. Giacinto, “Looking at the Bag is Not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection,” in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS ’13. New York, NY, USA: ACM, 2013, pp. 119–130. [Online]. Available: http:

//doi*.*acm*.*org/10*.*1145/2484313*.*2484327

[29]     D. Maiorca, G. Giacinto, and I. Corona, “A Pattern Recognition System for Malicious PDF Files Detection,” in *Proceedings of the 8th International Conference on Machine Learning and Data Mining in Pattern Recognition*, ser. MLDM’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 510–524. [Online]. Available: http: //dx*.*doi*.*org/10*.*1007/978-3-642-31537-4 40

[30]     P. Ratanaworabhan, B. Livshits, and B. Zorn, “Nozzle: A defense against heap-spraying code injection attacks,” in *Proceedings of the Usenix Security Symposium*. USENIX, 2009. [Online]. Available: http://research*.*microsoft*.*com/apps/pubs/default*.*aspx?id=81085

[31]     M. Z. Shafiq, S. A. Khayam, and M. Farooq, “Embedded malware detection using markov n-grams,” in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[32]     C. Smutz and A. Stavrou, “Malicious PDF Detection Using Metadata and Structural Features,” in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC ’12. New York, NY, USA: ACM, 2012, pp. 239–248. [Online]. Available:

http://doi*.*acm*.*org/10*.*1145/2420950*.*2420987

[33]     K. Snow, S. Krishnan, F. Monrose, and N. Provos, “Shellos: Enabling fast detection and forensic analysis of code injection attacks,” in *USENIX Security Symposium*, 2011. [Online]. Available: http://static*.*usenix*.*org/events/sec11/tech/full papers/Snow*.*pdf

[34]     Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo, “On the infeasibility of modeling polymorphic shellcode,” in *Proceedings of the 14th ACM Conference on Computer and Communications*

*Security*, 2007.

[35]     D. Stevens, “PDF Tools,” http://blog*.*didierstevens*.*com/programs/pdftools/.

[36]     ——, “PDFiD On VirusTotal,” http://blog*.*didierstevens*.*com/2009/04/ 21/pdfid-on-virustotal/.

[37]     Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos, “Combining static and dynamic analysis for the detection of malicious documents,” in *Proceedings of the Fourth European Workshop on System Security*, ser. EUROSEC ’11. New York, NY, USA: ACM, 2011, pp. 4:1–4:6. [Online]. Available: http:

//doi*.*acm*.*org/10*.*1145/1972551*.*1972555

[38]     N. Srndiˇ c and P. Laskov, “Detection of malicious PDF files based on´ hierarchical document structure,” in *In Proceedings of the Network and Distributed System Security Symposium, NDSS 2013*. The Internet Society, 2013.

[39]     N. Srndic and P. Laskov, “Practical evasion of a learning-basedˇ classifier: A case study,” in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP ’14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 197–211. [Online]. Available: http://dx*.*doi*.*org/10*.*1109/SP*.*2014*.*20

[[1]](file:///F:\\PDFpy\\God_with_me\\2018Q2\\paper\\1_T_ExtractMeIfYouCanAbusingPDFParsersInMalwareDetectors_NDSS2016\\ExtractMeIfYouCan.htm" \l "_ftnref1" \o ") For the brevity of presentation, we assume Address Space Layout Randomization (ASLR) is disabled. When ASLR is enabled, we in fact use module name plus offset to specify this raw tap point.

[[2]](file:///F:\\PDFpy\\God_with_me\\2018Q2\\paper\\1_T_ExtractMeIfYouCanAbusingPDFParsersInMalwareDetectors_NDSS2016\\ExtractMeIfYouCan.htm" \l "_ftnref2" \o ") http://www.irs.gov/pub/irs-pdf/fw4.pdf