计算机科学
COMPUTER SCIENCE

基于符号执行优化的PDF恶意指标提取技术

宋恩舟, 胡涛, 伊鹏, 王文博

引用本文

宋恩舟, 胡涛, 伊鹏, 王文博. 基于符号执行优化的PDF恶意指标提取技术[J]. 计算机科学, 2024, 51(7): 389-396.

SONG Enzhou, HU Tao, YI Peng, WANG Wenbo. PDF Malicious Indicators Extraction Technique Based on Improved Symbolic Execution [J]. Computer Science, 2024, 51(7): 389-396.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.230300117

基于符号执行优化的 PDF 恶意指标提取技术

宋恩舟　胡　涛　伊　鹏　王文博
国家数字交换系统工程技术研究中心　郑州 450001
(391032473@qq.com)

摘　要　恶意 PDF 文档是 APT 组织常用的攻击方法,提取分析其内嵌 JavaScript 代码指标是判定文档恶意性的重要手段,然而攻击者可以采取高度混淆、虚拟机与沙箱检测等逃逸方法。因此,文中创新性地将符号执行方法用于 PDF 指标提取,提出了一种基于符号执行优化的 PDF 恶意指标提取技术,并实现了由代码解析、符号执行和指标提取 3 个模块组成的指标提取系统 SYMBPDF。在代码解析模块中实现内嵌 JavaScript 代码提取与重组。在符号执行模块中设计代码改写方法,通过强制分支转移提高符号执行的代码覆盖率;设计并发策略和两种约束求解优化方法,以提高系统执行效率。在指标提取模块中实现恶意指标整合与记录。对 1271 个恶意样本进行了指标提取与评估,指标提取成功率为 92.2%,有效性为 91.7%,代码覆盖率较优化前提升 8.5%,系统性能较优化前提升 32.3%。

关键词:恶意文档;JavaScript 代码;指标提取;符号执行;代码改写;约束求解优化
中图分类号　TP311

PDF Malicious Indicators Extraction Technique Based on Improved Symbolic Execution

SONG Enzhou, HU Tao, YI Peng and WANG Wenbo
National Digital Switching System Engineering Technological R&D Center, Zhengzhou 450001, China

Abstract　The malicious PDF document is a common attack method used by APT organizations. Analyzing extracted indicators of embedded JavaScript code is an important means to determine the maliciousness of the documents. However, attackers can adopt high obfuscation, sandbox detection and other escape methods to interfere with analysis. Therefore, this paper innovatively applies symbolic execution method to PDF indicator extraction. We propose a PDF malicious indicator extraction technique based on improved symbolic execution and implement SYMBPDF, an indicator extraction system consisting of three modules: code parsing, symbolic execution and indicator extraction. In the code parsing module, we implement extraction and reorganization of inline Javascript code. In the symbolic execution module, we design the code rewriting method to force branch shifting, resulting in improving the code coverage of symbolic execution. We also design a concurrency strategy and two constraint solving optimization methods to improve the efficiency. In the indicator extraction module, we realize integration and recording of malicious indicators. In this paper, 1271 malicious samples are extracted and evaluated. The success rate of indicator extraction is 92.2%, the indicator effectiveness is 91.7%, the code coverage is 8.5% higher and the system performance is 32.3% higher than that of before optimization.

Keywords　Malicious documents, JavaScript code, Indicator extraction, Symbolic execution, Code rewriting, Constraint solving optimization

1　引言

近年来,通过恶意文档实现的网络攻击数量急剧上升,作为 APT 组织常用的攻击向量,其通常会带来网络钓鱼、漏洞利用、拒绝服务攻击等多种严重危害。在多种恶意文档中,恶意 PDF 文档借助各类阅读器产品的丰富功能以及软件本身存在的安全缺陷,通过嵌入脚本、远程链接、构建特定漏洞格式等多种方法实施攻击(表 1 列出了近年来威胁甚广的多个 PDF 漏洞利用实例),检测难度大、隐匿性强,通常会产生身份提权、任意代码执行等严重危害。根据 F-Secure 安全公司的统计数据,2020 年,利用恶意 PDF 文档展开的攻击次数在文档攻击总数中的占比超过 80%,给网络空间安全态势带来了严重威胁[1-4]。

针对恶意 PDF 文档带来的安全威胁,近年来诸多研究者提出了相关的检测方法和分析技术。其中比较常见的是通过分析文档中内嵌的恶意 JavaScript 代码来判断文档的恶意

到稿日期:2023-03-14　返修日期:2023-07-03
基金项目:国家自然科学基金面上项目(62176264)
This work was supported by the National Natural Science Foundation of China(62176264).
通信作者:胡涛(hutaondsc@163.com)

性[5]，此方法在 Web 安全检测中也有相应的应用，其中的某些手段与 PDF 恶意文档检测具有相似性。其技术难点在于如何尽可能多地提取代码中隐匿的恶意指标。目前的提取方法主要分为两类：静态提取技术与动态提取技术。静态提取技术通常是在代码提取后，不采取加载执行代码方式而直接采取去模糊与反混淆手段。其结合文档结构解析代码变量及字符串等相关信息作为提取指标[6-7]。动态提取技术通常是对已提取的 JavaScript 代码进行模拟执行，通过内存监视发现可疑的 Shellcode 代码或通过监测系统行为及序列等作为提取指标[8-12]。静态提取技术安全性较高，能够较好地恢复代码的原语义，具备检测效率高、速度快、开销小的优点，但是其对高度混淆代码的还原效果有限，并且对基于内存攻击的 JavaScript 代码的指标提取效果较差，部分攻击者可以构造不符合格式规范的 PDF 文档或采取代码重混淆技术干扰解析与提取[13]；动态提取技术弥补了静态方法的不足，能够充分提取代码执行过程中的信息，但对攻击者设置特定触发条件、检测执行环境、延迟执行等逃逸手段仍旧束手无策[14-15]。

表 1 PDF 漏洞利用实例

Table1 Example of PDF vulnerability exploitation

编号	阅读器产品	漏洞成因	危害影响
CVE-2022-28672	Foxit Reader	权限控制不当	任意代码执行
CVE-2022-27787	Adobe Reader	缓冲区溢出	任意代码执行
CVE-2022-24934	WPS Office	权限控制不当	任意代码执行
CVE-2021-44709	Adobe Reader	缓冲区溢出	任意代码执行
CVE-2021-21045	Adobe Reader	权限控制不当	任意代码执行
CVE-2020-14425	Foxit Reader	权限控制不当	任意代码执行

基于上述分析，本文提出了一种基于符号执行优化的 PDF 恶意指标提取技术：SYMBPDF。该系统的核心组件是针对 PDF 文档内嵌 JavaScript 代码的符号执行引擎。通过优化的符号执行与约束求解手段，推断出使代码反混淆的关键值。通过遍历代码执行路径，可以成功捕获 JavaScript 代码的所有可能行为并记录相关变量与字符串。例如，攻击者采用时间戳判断方法，只在系统时间符合要求的情况下连接恶意服务器执行下载任务。在此情况下，本系统可以成功探索该执行路径，提取服务器 URL 域名等恶意指标。

SYMBPDF 利用了 2 个不同的数据集进行评估。第 1 个数据集包含公开收集的 579 个恶意 PDF 文档样本，第 2 个数据集包含私人收集的 692 个恶意 PDF 文档样本。数据集涵盖了近 10 年来 3 种主流阅读器的多个典型漏洞样本。在共计 1 271 个恶意样本中，成功地对 1 172 个样本实现了恶意指标提取。此外，两个数据集中包括 414 个采用高度混淆和具有特定触发条件的恶意样本，本系统成功地对 392 个此类样本实现了指标提取。

综上，本文的主要贡献如下：

1）创新性地将符号执行方法用于 PDF 恶意指标提取，设计与实现了相应原型系统 SYMBPDF，可以有效应用于恶意 PDF 检测与研究。

2）对符号执行引擎进行优化，通过代码改写方法解决路径丢失问题，最大程度提高代码执行覆盖率；通过并发策略和 2 种约束求解优化方法提高符号执行效率，有效地提升了系统性能。

3）对近 10 年来 3 种主流阅读器典型漏洞的共计 1 271 个恶意样本实现了指标提取和评估，设计了提取成功率、代码覆盖率、指标有效性及提取效率相关实验。

2 背景与相关工作

本章首先对 PDF 文档解析与 JavaScript 代码执行的工作原理进行描述，其次对 PDF 以及 JavaScript 代码的恶意指标提取的相关工作进行总结，然后介绍 JavaScript 代码混淆及逃逸手段，最后介绍符号执行的相关技术。

2.1 PDF 解析 JavaScript 代码工作原理

JavaScript 代码的执行引擎是 PDF 文档阅读产品中的重要组成部分。根据最新 ISO 标准[16]，PDF 文档物理结构由文件头（包含文本、图像、字体等信息的文档主体）、交叉引用表、trailer 和文件尾组成。其中，最关键的文档主体由多个以层次结构链接到一起的文档对象组成。阅读器在实际解析过程中主要分为对象检索和代码重组两个阶段。阅读器首先读取 trailer 中的 Root 字段和文件尾中的 startxref 字段，找到文档中存储文档对象结构的根节点，然后按顺序检索每个文档对象，根据其中的 FlateDecode，Type 等编码和类型字段，对流内容进行解码。如果解码后的文档对象的字典为 Action 类型且包含/S，/JavaScript，/JS 等关键词，则记录该对象序号。之后将记录的对象按顺序进行代码重组。最终阅读器执行重组完毕的 JavaScript 代码。

2.2 PDF 及 JavaScript 代码恶意指标提取相关工作

为了后续 PDF 恶意性检测，许多研究者提出了分析并提取 PDF 内嵌 JavaScript 代码中恶意指标的方法，Web 安全中的部分 JavaScript 代码分析研究方法也可以用于指标提取工作。以上工作可大致分为两类。

一类是在不模拟执行 JavaScript 代码的情况下，通过反混淆等手段最大程度还原代码，将其中的内容、关键词等信息以及文档结构信息作为文档指标。Maiorca 等[17]对文档中的对象及 JavaScript 触发等关键词进行提取，将其频率作为最终指标。Lin 等[18]提取了文档内容中的功能关键字、习惯性用词和一些常数段作为指标。Sun[19]对文档中的 JavaScript 代码进行反混淆处理，然后提取求值函数、堆喷射特征、长字符串、API 函数等关键信息作为指标。Wang 等[20]不关注代码本身的特征而是提取其视图结构作为指标。Ndichu 等[21]收集了代码中关键字频数、行数、字符等共计 45 个特征作为指标。Fraiwan 等[22]在代码中提取了 URL 属性、代码结果、代码活动、代码内容作为指标。此类静态提取技术无法对高度混淆的 JavaScript 代码进行指标提取，容易受到代码重混淆等技术的影响。部分提取技术得到的指标本身并不具有明显的恶意性，而是作为特征向量代入后续的分类模型，无法对利用新特征进行攻击的恶意样本实现检测，且容易受到对抗样本的攻击，并且提取的指标无法作为分析人员研究恶意 PDF 文档攻击方式的显性依据。

另一类主要通过模拟执行代码提取其中的变量信息、中间语言或 Shellcode 代码作为指标。Laskov 等[23]利用 SpiderMonkey 引擎模拟执行 JavaScript 代码，将中间语言作为提取指标，但是中间语言指标只能作为分类依据而很难直接

表征恶意性。Li 等[24]结合了静态提取技术，将代码反混淆后提取的关键词和模拟执行获得的 Shellcode 共同作为指标。Zhuge 等[25]提出的 MPScan 检测模型对 Adobe Reader 进行了 Hook，然后提取疑似 Shellcode 代码作为指标。Cova 等[26]对 JavaScript 进行模拟执行并提取其预先定义的相关特征作为指标。Ma 等[27]动态执行 JavaScript 代码，由变量名读取器对变量初值和终值进行读取并实现分类检测。此类动态提取方法无法对抗只在特殊环境和要求下触发恶意行为的文档，对抗逃逸手段效果较差。Hu 等[28]引入了较为成熟的符号执行思路 JSForce，设计了强制执行引擎，可以沿着不同路径执行代码片段。但是此方法对使用特定阅读器 API 功能代码的指标提取效果较差，有时会产生过载与路径爆炸的问题，并且，如何重组 PDF 文档中的 JavaScript 代码也是后续模拟执行的关键前提。

2.3 JavaScript 代码混淆及逃逸手段

针对上述静态与动态提取方法，攻击者可以对 JavaScript 代码进行高度混淆并设计相关逃逸手段来干扰分析。图 1 展示了本文构造的一段内嵌在恶意 PDF 文档中的 JavaScript 代码，该代码利用 Foxit Reader 阅读器对某 API 函数执行策略管理不当的缺陷(CVE-2020-14425[1])来执行恶意脚本。我们对该代码进行编码混淆，并设置了只在特定时间触发的逃逸手段，可以成功绕过恶意文件分析网站 Virustotal[2]的静态及动态检测。

```
9 0 obj
<<
/S/JavaScript
/JS(var last=new Date()["\\u0067\\u0065\\u0074\\u0054\\u0069\\
u006d\\u0065"]();if(0x1865a700c10<last){app["\\u006f\\u0070\\
u0065\\u006e\\u0063\\u0050\\u0044\\u0046\\u0057\\u0065\\u0062\\
u0050\\u0061\\u0067\\u0065"]("exe.clac\\\\\\\\\23metsyS\\\\\\\\
swodniW\\\\\\\\:C".split("").reverse().join(""));} )
>>
endobj
```

图 1 CVE-2020-14425 恶意样本

Fig. 1 Malicious sample of CVE-2020-14425

除此之外，JavaScript 代码还具有变量替换、控制流平坦化、字符串阵列化等多种混淆手段，对静态提取技术造成了极大干扰。攻击者还可以采取检测文件、检测进程、检测用户名、检测用户交互以及延迟执行等逃逸手段来对抗沙箱和虚拟机环境。传统的动态技术无法针对此类攻击实现指标提取。

2.4 符号执行

符号执行是一种常见的程序分析技术，其主要原理是在符号变量的抽象域中执行程序。该技术在代码中每个条件指令后执行，并且对执行过程中引入的所有的约束条件进行跟踪。例如，当 JavaScript 代码读取一个环境变量相关的整数值 X 时，该变量初始不受任何约束。但当其执行带有条件 $X\geqslant 0$ 的条件指令时，程序执行流程将分为 2 个新的分支并将变量的值约束为正($X\geqslant 0$)或负($X<0$)。该分析技术可以通过约束求解来确定程序中执行特定分支的输入，进而提高代码整体的执行覆盖率。但是在执行过程中，用实际值替换相应符号值后会存在路径丢失问题，造成代码执行的不完全性，本系统针对此问题进行了优化。

3 SYMBPDF 系统设计与实现

3.1 设计架构

本文提出的 SYMBPDF 方法包括代码解析、符号执行、指标提取 3 个模块，其基本框架如图 2 所示。代码解析模块主要负责 PDF 文档对象解析、JavaScript 代码提取与重组；符号执行模块为系统的核心部分，首先拓展 API 功能，使用相关的符号变量来构建执行环境，并进行符号探索执行，最后结合约束求解器求解变量的可满足性，将表达式从符号域转为具体域；指标提取模块主要对各执行路径中的返回值、变量值等进行解析与记录，最终生成恶意指标结果。

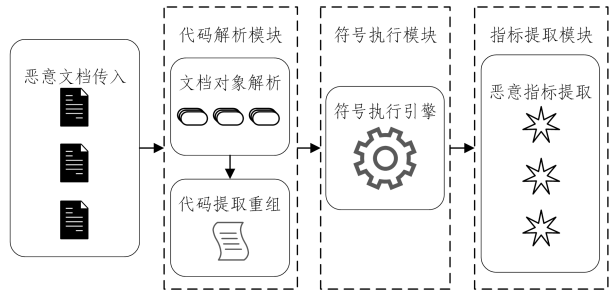

图 2 SYMBPDF 流程图

Fig. 2 Flowchart of SYMBPDF

3.2 代码解析模块

代码解析模块主要实现 PDF 文档结构解析和 JavaScript 代码提取与重组。结合 2.1 节中的工作原理，本模块首先根据文档结构遍历文档对象，依次进行解码操作；之后根据对象属性关键词提取包含的 JavaScript 代码；最后根据对象顺序进行代码重组，整体流程如图 3 所示。

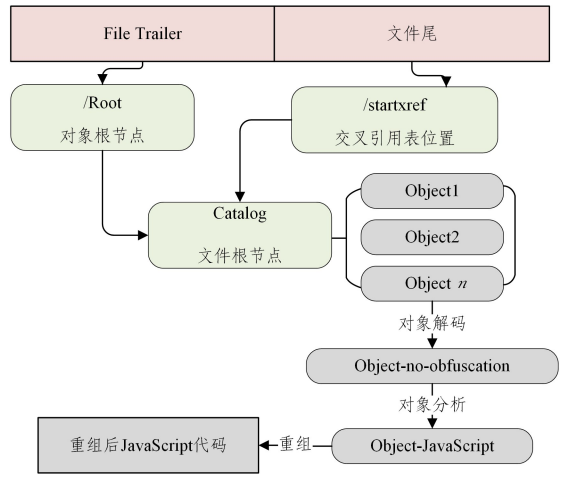

图 3 代码解析模块流程图

Fig. 3 Flowchart of code parsing module

[1] https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-14425

[2] https://www.virustotal.com/gui/home/upload.

文档结构解析主要负责在文档主体部分找到所有的对象。由于 PDF 阅读器会对不符合标准格式规范的文档进行解析，攻击者会故意删除交叉引用表中的部分对象和文件尾中的关键词来干扰静态分析。如图 4 所示，此类不符合规范的结构(不包含交叉引用表)依旧可以被阅读器正常读取，进而影响静态解析。因此本模块不依赖交叉引用表而是通过基本结构对文档对象进行识别。当解析器识别出所有对象后，将对每个对象进行反混淆处理进而提取相关信息。PDF 文档通常采用不同的编码方式将代码隐藏在流对象中，该编码方式由 FILTER 字段指定。解析器首先读取编码类型，然后采取对应解码方法对内容进行解码。解析器还会对内容进行额外的检查，将其中以不同进制来代替 ASCII 字符的编码进行还原，最终实现对文档对象的识别与解析。

```
%PDF-1.4
1 0 obj<<
/Type /Catalog
/Pages 1 0 R
/OpenAction <<
/S /JavaScript
/JS(app.alert{'It's malicious!'})
>>
>>
endobj
Trailer <<
/Root 1 0 R
>>
```

图 4 不符合格式规范的恶意文档

Fig. 4 Malicious sample with irregular formatting

代码解析模块在对 PDF 文档对象进行解析后，还要进一步对 JavaScript 代码进行提取与重组。根据相关规范，包含 JavaScript 代码的对象用关键词/JS 标识。代码可以在对象本身的位置中存储，也可以在对象链接到的其他位置存储。解析器首先确定所有包含 JavaScript 代码的对象序号，之后通过/OpenAction 以及/Names 等标志字段确认代码执行的入口点，最后根据对象顺序结合链接位置将代码进行重组。

3.3 符号执行模块

符号执行模块为本系统的核心模块，主要负责 API 功能拓展、构建符号执行引擎进行符号探索、利用约束求解器求解变量。本模块的基本架构如图 5 所示，通过本模块可以获取条件指令中变量的满足值，探索代码的所有有效分支，得到不同路径下的执行结果。

本模块首先需要构建 JavaScript 代码执行环境，执行环境基于 V8 执行引擎[1]实现。在构建执行环境前，本模块丰富并拓展了执行引擎中的相关函数。市面上主流的阅读器产品(如 Adobe Reader，Foxit Reader 等)为丰富和拓展功能提供了广泛的 API 函数。攻击者构造的 JavaScript 代码也可以通过此类 API 函数执行相应功能。因此，为了更好地对代码进行模拟，本模块参考相关产品的开发者文档[2),3)]，在 V8 引擎中对较常用的且易构成攻击的 API 调用进行了实现，本模块拓展的 API 函数如表 2 所列。

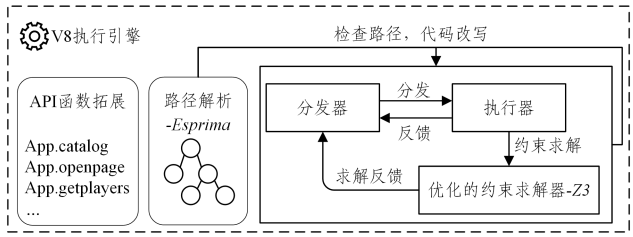

图 5 符号执行模块流程图

Fig. 5 Flowchart of symbolic execution module

表 2 符号执行引擎拓展 API 函数

Table 2 Extended API functions for symbolic execution engine

app. addMenuItem	app. alert	app. browseForDoc	app. buttonGetIcon( )
app. SubMenu	app. execDialog	app. execMenuItem	app. mouse
app. getAltTextData	app. getPlayers	app. getURLData	app. userName
app. getURLSettings	app. getWindowBorderSize	app. openDoc	app. media
app. newDirectory	app. launchURL	app. listMenuItems	app. getURL
app. newDoc	app. getPath	app. setTimeOut	app. signatureInfo
app. opencPDFWebPage	app. setAction	app. pageWindowRect	

符号执行引擎依赖开源工具 Jalangi2[4]实现，其本身由 JavaScript 编写且提供回调功能。如图 5 所示，本引擎主要由分发器、执行器和约束求解器构成。分发器负责管理符号探索的全局状态，汇总统计相关数据并为符号执行调度测试用例；执行器并发运行执行程序中的多个测试用例并进行相应的符号跟踪；约束求解器由开源工具 Z3[5]实现，用来求解分支变量的满足值，生成新的测试用例返回至分发器。下面描述该执行引擎的具体细节。

路径解析：为了能够掌握程序的所有分支，引擎首先对重组代码进行路径解析，生成抽象语法树。解析器基于标准 ECMAScript 解析工具 Esprima[6]实现。当掌握代码执行的全部路径后，结合后续符号执行的路径约束情况，可以进一步采取针对性的代码改写方法提高代码覆盖率。

执行生成测试：执行器对实际状态和符号状态进行区分，

1) https://github.com/nodejs/Release

2) https://opensource.adobe.com/dc-acrobat-sdk-docs/library/jsapiref/JS_API_AcroJS.html#

3) https://forums.foxitsoftware.com/forum/portable-document-format-pdf-tools/foxit-phantompdf/180511-javascript-api-documentation

4) https://github.com/Samsung/jalangi2

5) https://github.com/Z3Prover/z3

6) https://github.com/jquery/esprima

在运算前会检查当前值是否为实际状态。如果是实际状态，则继续按照源代码执行程序；如果存在符号化的值，则执行器会更新当前路径条件，对约束条件取反，通过符号化执行走向新的路径，而将原路径的相关条件返还给分发器。

测试用例并发性：在执行生成测试中，当执行器进入符号化执行时，会将原路径相关条件返还给分发器，分发器将该条件重新提交给新的执行器线程执行。因此，不同的测试用例相互独立，包含自己的内存和约束条件。内存主要包含执行语句、变量信息等；约束条件主要包含当前的符号约束和路径约束。通过测试用例并发，可以节省符号执行中深度优先搜索策略的时间。

代码改写：动态符号执行将求解器无法求解的部分用实际值进行替换，可以极大降低因外部代码交互和约束求解超时造成的不精确性，但也在一定程度上牺牲了路径探索的完全性。为解决此问题，在符号执行引擎执行完毕后，系统会记录已经执行过的代码路径，并将该路径同路径解析得到的所有分支进行对比。如果发现存在路径丢失问题，则改写该对应路径分支节点的代码，强制程序执行丢失的路径，最大程度提高代码覆盖率。为了对覆盖率进行分析与计算，我们对其形式化表示如下：

$$coverage_1=\frac{record_{\text{symbol}}}{\sum_{i=1}^{n}\prod_{j=1}^{k}b_{ij}} \tag{1}$$

其中，$record_{\text{symbol}}$ 为符号执行过程中的分支总数，n 是代码抽象语法树中的节点数目，k 是代码抽象语法树中的分支节点数目，b_{ij} 为第 i 个节点的第 j 个可能性，即该节点存在的路径数目。

约束求解优化：约束求解器的作用为将符号表达式具体化，生成符号执行引擎可以继续执行的具体方案，将其从符号状态转为实际状态。但是约束求解有时会存在符号值爆炸的情况。为了提高约束求解效率，我们采用过载优化和自适应优化两种方法。

(1)过载优化。如式(2)所示，该语句使用的是一个基于 get 返回指定名称字段值的符号表达式。该变量存在 2^{32} 个具体解，因此在执行完代码后，该具体化策略会产生 2^{32} 个执行状态，导致出现过载现象。因此我们引入了一个额外的符号作为中间变量，当执行符号比较操作、符号布尔操作或者符号索引操作时，用该符号变量来表示相应操作表达式。如式(2)所示，求解器识别出该表达式使用了一个符号比较操作，于是引入新的符号变量 $Temp1$ 表示该操作，最终将该符号表达式简化为 $Temp1+80$。该表达式此时只有 81 和 82 两个具体解，极大地优化了约束求解效率。

(2)自适应优化。如式(3)所示，该语句调用了一个错误的 API 函数，但约束求解引擎依旧会将其具体化。因此我们引入自适应优化，对公式的有效性进行检查，如果发现其调用了无效函数，则过滤该公式，删除该代码路径所在分支，进一步提高约束求解效率。

$$[\text{Var1}]=\text{Function}((\text{app.getFieldValue}(\text{“tes”})>100)+80) \tag{2}$$

$$[\text{Var2}]=\text{Function}((\text{app.getFildeVulae}(\text{“test”})>100)+80) \tag{3}$$

当样本触发约束求解优化时，在计算代码覆盖率时，需要排除无效的代码分支。因此定义了集合 ω 作为无效的分支节点集合，此时代码覆盖率的形式化表示如下：

$$coverage_2=\frac{record_{\text{symbol}}}{\sum_{i=1}^{n-len(\omega)}\prod_{j=1}^{k}b_{ij}}(b_{ij}\notin\omega) \tag{4}$$

3.4 指标提取模块

指标提取模块主要用于提取代码中的危险指标。根据符号执行过程中的变量值及约束求解得到的具体值生成各个变量的变化记录，将其中包含文件名、URL、Shell 命令、Shellcode 代码等特征的记录进行筛选，生成最终的指标报告。为了能够按照正确顺序记录变量值的变化情况，采用的具体流程如算法 1 所示。首先根据语法树，按顺序记录所有的变量名称及其定义区间，然后对代码逐行解析并记录每条代码的执行区间，之后按照代码执行区间顺序，建立代码同变量的映射关系，最后根据符号执行记录，按正确顺序记录变量值。

算法 1 指标提取算法

输入：重组 JavaScript 代码 JSCode，符号执行记录 Record

输出：以列表形式存储的变量值记录结果

result_list=[[identifier1，value1，value2，…]，…，]

1. for identifier in JSCode. Tokens do
2. identifier_list. append(identifier，identifier. range)/ * 按顺序提取记录变量及定义区间 * /

endfor

3. for code in JSCode. Syntax do
4. code_list. append(code，code. range)/ * 按顺序提取记录代码及定义区间 * /

endfor

5. for number in code_list do
6. if identifier_list[range] in code_list[range]：
7. code_with_id_list. append(code，identifier) / * 按执行顺序建立代码同变量的映射关系 * /

endif

endfor

8. Read_and_Insert(Record，code_with_id_list)/ * 根据映射关系及执行顺序，对变量进行记录，将结果以列表的形式呈现 * /

4 实验与测试

4.1 实验设置

为了验证 SYMBPDF 系统的性能与效果，本文分别从指标提取成功率、代码覆盖率、指标提取有效性、指标提取效率 4 个方面进行测试。指标提取成功率实验对收集样本进行提取测试，体现系统设计与实现的可靠性；代码覆盖率实验主要对符号执行中的路径覆盖率进行测试，证明代码改写方法的优化效果；指标提取有效性实验主要通过 YARA 规则的触发率证明指标提取可以有效地提高恶意 PDF 文档检测效果，并同市面上主流的 JavaScript 反混淆工具及 JavaScript 符号执行相关论文对比；指标提取效率实验主要探究本文设计的约束求解优化手段对系统效率的提升效果。

本文收集了共 1 271 个恶意样本，其中公开数据集 579 个，

私人数据集 692 个。数据集涵盖了近十年来 3 种主流阅读器的多个典型漏洞样本。收集的恶意样本中包括 414 个采用高度混淆和具有特定触发条件的复杂恶意样本。

实验在 Intel(R) Core(TM) i7-1165G7 @ 2.80 GHz, RAM 16.0 GB, Ubuntu18.04 上进行。软件方面采用 Python 3.6.9 实现, Nodejs 版本为 12.18.1, Esprima 版本为 4.0, 设置单个样本测试时间最长为 10min。

4.2 指标提取成功率

本实验对指标提取结果及成功率进行探究。按照在过程中是否触发符号执行的原则, 将普通和复杂恶意样本分别分为实际执行和符号执行两类。如表 3 所列, 普通样本提取成功率为 96.5%, 复杂样本提取成功率为 83.3%, 样本总计提取成功率为 92.2%。但是实际执行和符号执行样本中都存在提取失败的情况: 1)因为样本中调用的 API 函数没有得到系统支持; 2)因为 JavaScript 代码重组后, 存在具有语法错误的代码片段; 3)因为部分代码的入口点被高度混淆, 没有成功定位。与普通恶意样本相比, 复杂恶意样本提取成功率明显下降, 证明攻击者采取的多种逃逸方式对指标提取工作具有一定的干扰与影响。

表 3 指标提取成功率

Table 3 Success rate of indicator extraction

类型	普通恶意样本			复杂恶意样本		
	实际执行	符号执行	总样本	实际执行	符号执行	总样本
样本数	623	234	857	119	295	414
成功数	601	226	827	98	247	345
成功率/%	96.5	96.6	96.5	82.4	83.7	83.3

4.3 代码覆盖率

本实验对符号执行的代码覆盖率进行了探究。如图 6 所示, 在引入代码改写机制前, 大部分样本都具有较高的代码覆盖率, 证明符号执行系统本身具有可靠性。引入改写机制后, 覆盖率在 0.9～1.0 之间的样本比例提升了 9.6%, 因为系统对所有没经过的路径分支进行改写, 除了语法错误导致执行失败的情况外, 覆盖率均可以达到 100%。如图 7 所示, 代码改写后平均覆盖率较改写前提高了 8.5%。可以看到, 在 6.1 s, 即第一轮符号执行完成后, 由于代码改写机制的触发, 平均覆盖率有明显提升, 证明了该机制的有效性。

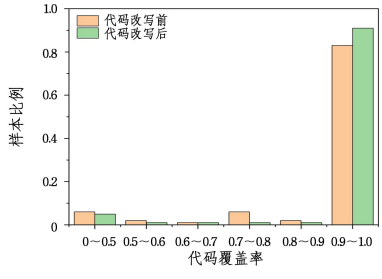

图 6 不同代码覆盖率对应样本比例

Fig. 6 Proportion of samples with different code coverage

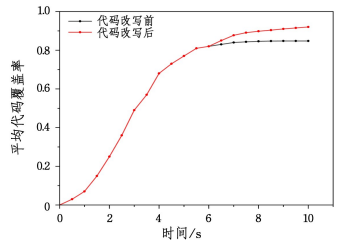

图 7 平均代码覆盖率

Fig. 7 Average code coverage

值得注意的是, 覆盖率在 0～0.5 之间的样本比例明显高于 0.6～0.9 之间的样本比例, 因为在重组代码没有语法以及格式错误的情况下, 代码改写可以保证遍历全部分支, 但是当重组代码无法正常执行时, 代码覆盖率会显著降低。

4.4 指标提取有效性

本实验主要探究并证明系统提取恶意指标的有效性, 设计相关对比实验, 对提取指标在后续 PDF 恶意性检测中的作用进行研究。我们利用自编写的 YARA 规则作为恶意性检测的依据, 该规则主要匹配敏感文件名、URL 域名、Shell 命令、执行函数(如 downLoad, EXEC, Fopen, Fwrite, http, process 等)。本实验选择市面上主流的 JSDetox[1)], JS-beautify[2)], Prepack[3)] 3 种 JavaScript 反混淆工具及 2.2 节中所述的 JavaScript 符号执行工具 JSForce 进行对比。首先利用反混淆工具将重组的 JavaScript 代码进行反混淆处理, 提取还原后代码中的变量及字符串信息作为指标。然后利用 JSForce 符号执行重组 JavaScript 代码, 并利用本系统的指标提取模块进行指标提取。最后将上述指标与 SYMBPDF 提取的指标分别传入 YARA 规则。当规则命中时, 证明基于该指标可以成功对恶意文档进行检测。本系统及其他工具检测概率及检出数量如表 4、图 8 所示。其中, 恶意脚本攻击样本总数为 872, 检出率为 90.9%; Shellcode 攻击样本总数为 399, 检出率为 93.3%, 指标提取整体有效性为 91.7%。

表 4 恶意文档检出率

Table 4 Malicious document detection rate

(%)

工具	恶意类型及检测比例	
	恶意脚本攻击	Shellcode 攻击
JSDetox	61.7	17.9
JS-beautify	54.8	17.1
Prepack	69.2	18.1
JSForce	79.1	85.2
SYMBPDF	90.9	93.3

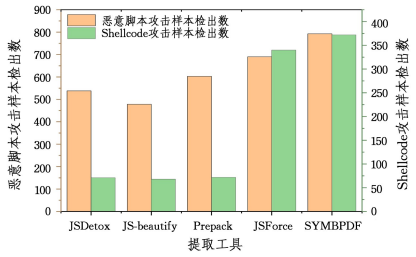

图 8 样本检出数

Fig. 8 Number of detected samples

1) https://github.com/svent/jsdetox

2) https://github.com/beautify-web/js-beautify

3) https://github.com/facebookarchive/prepack

如图 8 和表 4 所示，主流的反混淆工具针对恶意脚本攻击具有一定的检测效果，证明利用反混淆手段还原代码语义、结构及关键变量对恶意指标提取具有明显作用，但是依旧存在较大比例的漏检的情况。因为部分恶意代码通过编码函数等方式进行混淆和隐藏，仅通过静态分析和代码重构难以实现指标提取。此外，主流的反混淆工具针对 Shellcode 攻击的检测效果基本相同且检出率较低，因为多数内存攻击者会对执行代码进行隐藏和编码处理，难以通过常见的反混淆手段提取指标。

JSForce 符号执行工具对恶意样本的检测比例较 3 种反混淆工具有明显提升。但是由于其没有对执行环境中所需的 API 函数进行拓展，因此依旧存在一定的漏检现象。尽管其通过强制执行方法遍历分支，但是由于部分函数无法正确执行，恶意指标无法成功提取，进而影响了后续检测。

本系统采用符号执行优化的动态指标提取方法，可以通过模拟执行捕捉代码行为实现指标提取，因此针对恶意 PDF 的检测效果较上述工具有明显提升。值得注意的是，本系统对 Shellcode 攻击的指标提取有效性高于恶意脚本攻击。经分析发现，内存攻击方式的 JavaScript 代码功能比较单一，通常进行变量申请、堆栈处理等行为，代码混淆及变种类型较少，对变量进行动态追踪即可成功提取 Shellcode 代码；而恶意脚本攻击方式变化多样，具有多种灵活的逃逸手段，一定程度降低了指标提取成功率。

4.5 指标提取效率优化

本实验主要探究系统指标提取效率以及约束求解优化手段的提升效果。系统各模块对指标提取平均及中位用时情况如表 5 所列。在测试过程中，所有样本均在 10 min 限制时间内解析完毕，系统平均用时 7.79 s，中位用时 7.51 s，具有较高的指标提取效率。

表 5 系统用时

Table 5 System time spent

提取阶段	平均用时/s	中位用时/s
代码解析	0.820	0.770
符号执行	7.430	7.180
指标提取	0.079	0.077
总用时	7.790	7.510

如 3.2 节中所述，约束求解优化主要分为过载优化和自适应优化两部分，我们分别对两类优化手段进行探究。实验对 1271 个恶意样本进行测试，其中有 529 个恶意样本触发了符号执行。对引入约束求解优化前后此类样本符号执行分支数及指标提取平均时间进行记录，结果如图 9 所示。将样本分为无约束求解优化、仅过载优化、仅自适应优化和过载自适应优化 4 类。

如图 9 所示，引入优化手段后，符号执行分支数由 19377 减少至 13116，指标提取平均用时由 11.47 s 降至 7.79 s，系统效率提升 32.3%。过载优化和自适应优化两种手段均可有效减少分支数，提高指标提取效率。值得注意的是，过载优化相比自适应优化具有更高的优化效果，由 3.2 节可知，过载优化具有数倍级的提高效果。在测试过程中，共有 21 个样本触发过载优化和 13 个样本触发自适应优化。

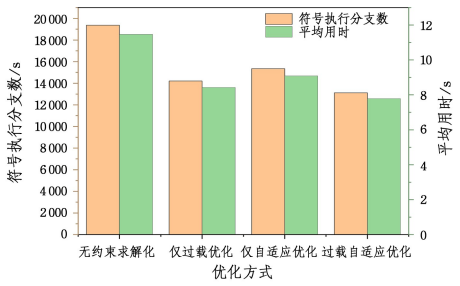

图 9 符号执行分支数及平均用时

Fig. 9 Number of branches and average time spent

对触发优化样本的平均分支数进行记录，结果如图 10 所示，当此类样本无求解优化时，会产生分支爆炸现象，而两种优化方式可以较好地抑制该现象，过载优化和自适应优化将平均分支数分别降至原先的 55% 和 79.9%，有效地防止了系统过载现象。

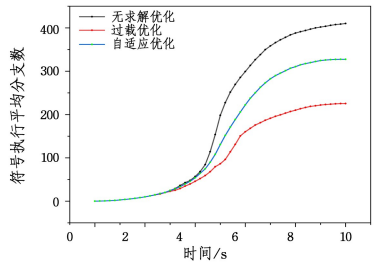

图 10 符号执行分支数变化图

Fig. 10 Change of branches for symbolic execution

结束语 本文创新性地将符号执行技术用于 PDF 恶意文档分析领域，详细研究并实现了针对 PDF 文档的恶意指标提取技术 SYMBPDF，通过代码解析、符号执行、指标提取 3 个模块实现对 PDF 文档的恶意指标提取。系统核心为以符号执行为主体的执行引擎。通过代码改写方法提高路径覆盖率，且通过并发策略及约束求解优化方法对引擎进行了性能优化。设计了指标提取成功率、代码覆盖率、指标提取有效性及优化效果的相关实验。实验结果表明，本系统能够高效地对 PDF 文档实现指标提取，具有较高的成功率、有效性和高效性，可以有效对抗攻击者常用混淆及逃逸手段，为后续研究 PDF 恶意代码及恶意性检测提供了良好的依据。

未来的研究工作如下：

1）进一步完善代码解析重组的效果，结合相关阅读器的开源资料，进一步丰富和拓展 API 函数，提高符号执行的成功率，改善系统性能。

2）解决语法错误导致的符号执行失败问题，进一步改善代码改写方式，将错误路径删除后，保证其他分支路径可以继续执行，进一步提高指标提取成功率。

3）进一步阅读和调研 Web 安全与 JavaScript 恶意代码分析领域的相关技术研究，将其同恶意 PDF 文档检测进行一定程度的结合，进一步提升系统性能。

参 考 文 献

[1] LEI J W, YI P, CHEN X, et al. PDF document detection model based on system calls and data provenance[J]. Journal of Com-

puter Applications, 2022, 42(12): 3831-3840.
[2] LU X, WANG F, JIANG C, et al. A Universal Malicious Documents Static Detection Framework Based on Feature Generalization[J]. Applied Sciences, 2021, 11(24): 12134.
[3] NISSIM N, COHEN A, MOSKOVITCH R, et al. ALPD: Active Learning Framework for Enhancing the Detection of Malicious PDF Files[C]// 2014 IEEE Joint Intelligence and Security Informatics Conference. Washington DC, USA: IEEE, 2014: 91-98.
[4] NISSIM N, COHEN A, GLEZER C, et al. Detection of Malicious PDF Files and Directions for Enhancements: A State-of-the Art Survey[J]. Computers & Security, 2015, 48: 246-266.
[5] YU M, JIANG J G, LI G, et al. A Survey of Research on Malicious Document Detection[J]. Journal of Cyber Security, 2021, 6(3): 54-76.
[6] WANG Y. The De-Obfuscation Method in the Static Detection of Malicious PDF Documents[C]// 2021 7th Annual International Conference on Network and Information Systems for Computers. Guiyang, China: ICNISC, 2021: 44-47.
[7] CHEN K, WANG P, YEONJOON L, et al. Scalable Detection of Unknown Malware from Millions of Apps[J]. Journal of Cyber Security, 2016, 1(1): 24-38.
[8] GAO X, YU M, JIANG J G, et al. A Combined Malicious Documents Detecting Method Based on Emulators[J]. Applied Mechanics and Materials, 2014(602/603/604/605): 1707-1712.
[9] FENG D, YU M, WANG Y. Detecting Malicious PDF Files Using Semi-Supervised Learning Method[C]// The 5th International Conference on Advanced Computer Science Applications and Technologies. Beijing, China: ACSAT, 2017: 135-155.
[10] ANDREASEN E, LIANG G, MØLLER A, et al. A survey of dynamic analysis and test generation for JavaScript[J]. ACM Computing Surveys, 2017, 50(5): 1-36.
[11] SIHWAIL R, OMAR, K, ZAINOL A, et al. Malware detection approach based on artifacts in memory image and dynamic analysis[J]. Applied Sciences, 2019, 9(18): 3680-3691.
[12] ALAZAB A, KHRAISAT A, ALAZAB M, et al. Detection of Obfuscated Malicious JavaScript Code[J]. Future Internet, 2022, 14(8): 217-231.
[13] TZERMIAS Z, SYKIOTAKIS G, POLYCHRONAKIS M, et al. Combining Static and Dynamic Analysis for the Detection of Malicious Documents[C]// The Fourth European Workshop on System Security. New York, USA: EUROSEC, 2011: 1-6.
[14] CORONA I, MAIORCA D, ARIU D, et al. Lux0R: Detection of Malicious PDF-Embedded JavaScript Code through Discriminant Analysis of API References[C]// The 2014 Workshop on Artificial Intelligent and Security Workshop. New York, NY: ACM, 2014: 47-57.
[15] RUARO N, PAGANI F, ORTOLANI S, et al. SYMBEXCEL: Automated Analysis and Understanding of Malicious Excel 4.0 Macros[C]// 2022 IEEE Symposium on Security and Privacy. San Francisco, CA: IEEE, 2022: 1066-1081.
[16] ISO32000-1: 2020[EB/OL]. https://www.pdfa.org/resource/iso-32000-pdf/.
[17] MAIORCA D, GIACINTO G, CORONA I. A Pattern Recognition System for Malicious PDFFiles Detection[C]// International Workshop on Machine Learning and Data Mining in Pattern Recognition. Berlin: Springer, 2012: 510-524.
[18] LIN J Y, PAO H K. Multi-View Malicious Document Detection[C]// 2013 Conference on Technologies and Applications of Artificial Intelligence. TAAI, 2013: 170-175.
[19] SUN B Y. Research on The PDF Document Security Detection Methods[D]. Shanghai: Shanghai Jiao Tong University, 2015.
[20] WANG T, MOU Z H, ZHANG Z H, et al. Detecting Obfuscated Malicious JavaScript Code Based on Function Call Information[J]. Computer Simulation, 2021, 38(2): 432-437.
[21] NDICHU S, KIM S, OZAWA S. Deobfuscation, unpacking, and decoding of obfuscated malicious JavaScript for machine learning models detectionperformance improvement[J]. CAAI Transactions on Intelligence Technology, 2020, 5(3): 184-192.
[22] FRAIWAN M, AL-SALMAN R, KHASAWNEH N, et al. Analysis and identifification of malicious javascript code[J]. Information Security Journal: A Global Perspective, 2012, 21(1): 1-11.
[23] LASKOV P, ŠRNDIĆ N. Static Detection of Malicious JavaScript-Bearing PDF Documents[C]// Proceedings of the 27th Annual Computer Security Applications Conference. New York, NY: ACM, 2011: 373-382.
[24] LI M, ZHOU Y, YU M, et al. Combining Static and Dynamic Analysis for the Detection of Malicious JavaScript-Bearing PDF Documents[C]// Proceedings of the 2016 International Conference on Computer Science, Technology and Application. Shenzhen, China: ICCITA, 2017: 475-482.
[25] LU X, ZHUGE J W, WANG R Y, et al. De-Obfuscation and Detection of Malicious PDF Files with High Accuracy[C]// 2013 46th Hawaii International Conference on System Sciences. Wailea, Maui, USA: HICSS, 2013: 4890-4899.
[26] COVA M, KRUEGEL C, VIGNA G. Detection and analysis of drive-by-download attacks and malicious javascript code[C]// Proceedings of the 19th International Conference on World Wide Web. New York, NY: ACM, 2010: 281-290.
[27] MA H L, WANG W, HAN Z. Detecting and De-Obfuscation Obfuscated Malicious JavaScript Code[J]. Chinese Journal of Computers, 2017, 40(7): 1699-1713.
[28] HU X, CHENG Y, DUAN Y, et al. JSForce: A Forced Execution Engine for Malicious JavaScript Detection[C]// Security and Privacy in Communication Networks: 14th International Conference. Singapore: Springer, 2018: 704-720.

SONG Enzhou, born in 1998, postgraduate. His main research interests include malicious document analysis, binary code analysis and vulnerability mining.

HU Tao, born in 1993, Ph.D, assistant researcher. His main research interests include intrinsic security, intrusion detection and SDN.

（责任编辑：何杨）