大 连 理 工 大 学 本 科 外 文 翻 译

检测 JavaScript 中重要的竞争

Detecting JavaScript Races that Matter

学 部（院）：软件学院

专　　业：软件工程（日语强化）

学 生 姓 名：杨天杰

学　　号：201393027

指 导 教 师：迟宗正

完 成 日 期：2017.1.22

大连理工大学

Dalian University of Technology

检测 JavaScript 中的竞争

Erdal Mutlu, Serdar Tasiran, Benjamin Livshits

土耳其大学，土耳其大学，微软亚洲研究学院

摘要：

由于 JavaScript 在过去几年中用于编程大型和复杂的 Web 应用程序的语言几乎无处不在，我们看到在客户端 JavaScript 中查找数据竞争的兴趣增加。虽然 JavaScript 执行是单线程的，但仍然有足够的潜力进行数据竞赛，主要由调度程序的非确定性创建。最近，为了寻求数据竞赛，几项学术努力探索了静态和运行时分析方法。然而，尽管如此，我们还没有看到在实践中部署的这些分析技术，我们只看到很少的证据表明，开发人员发现和修复了与 JavaScript 中的数据竞赛有关的错误。

在本文中，我们提出了一个不同的表达方式，即在 JavaScript 应用程序中进行数据竞赛，区分良性和有害竞争，影响浏览器或服务器状态。我们进一步认为，虽然存在着大部分先前工作的良性竞争，但实际上有害的竞争是非常罕见的（19 有害对比 621 良性）。我们的研究结果揭示了数据流行率和重要性的问题。

为了找到竞争，我们还提出了一种新颖的轻量级运行时符号检测算法，用于查找运行时执行跟踪中的竞争。我们的算法避免计划开发，有利于较小的运行时间开销，因此可以由 beta 测试人员或来自人群的测试使用。在我们对 26 个网站的实验中，我们证明良性的竞争比有害的竞争更为常见。

关键词： JavaScript，异步，竞争测试，非确定性

1. 介绍

JavaScript 在客户端 Web 编程中广泛使用。 JavaScript 执行是单线程的。然而，诸如 Facebook, Outlook 和 Google Maps 等网站的复杂需求导致异步成为编制复杂 Web 应用程序的常用方式。这是异步处理，使得可能的响应用户界面 (UI) 不同于 20 世纪 90 年代后期需要重新加载的 Web 应用程序。尽管 JavaScript 缺乏常规线程，但是异步的存在为竞争创造了潜力。特别地，JavaScript 中事件执行的顺序以及异步请求的完成时间是非确定性的，例如导致数据竞争的网络延迟。

在本文中，我们认为应该区分持久后果的竞争和短暂的竞争。我们发现，JavaScript 中的大多数竞争没有持续的后果 - 我们将这些竞争联合起来。这是因为这些竞争只会导致用户对程序状态的部分不可见，或者最糟糕的是，用户不注意的 UI 毛刺，或者如果用户重新加载页面，则会消失。鉴于 JavaScript 执行的原谅性，单个事件处理程序

的失败强制调度程序终止当前事件处理程序并将执行移动到下一个事件处理程序，这些类型的故障并不像以前相信那样重要。我们建议在网络上考虑数据竞争的一个更有用的方法是关注持久状态，如客户端本地 cookie，localStorage 和 sessionStorage 机制以及基于服务器的方面的努力。后者通过对服务器的 POST 调用来实现（GET 调用被设计用于读取，并且被认为是幂等的，因此不经常用于状态更新）。我们的实验证前的研究工作往往会产生大量的报告，尽管强调了抑制误报[13,11,11]。

在给定的应用程序或站点中提供所有可能的竞争的声音过近并不是我们的目标，而是提供轻量级的探索算法，允许在仅需要单次运行的情况下探索多个计划。这种方法可以用于测试，包括大量的 beta 或来自人群的测试人员的协同测试。

贡献：我们做了以下的贡献：

- 我们在 JavaScript Web 应用程序中提出了良性和有害数据竞赛的新视图，并认为有害竞争应该是分析工具的主要重点，因为它们影响应用程序的持久客户端或服务器端状态。
- 我们提出了一种轻量级的探索算法，用于在 JavaScript 程序的运行时跟踪中查找数据竞争。 我们的方法的可扩展性的一个主要优点是它不需要多个程序运行，并且可以在单个执行的基础上运行。
- 我们在 26 个网站中共发现和调查了 19 个有害和 621 无害的竞争，只有 2 个被观察到是虚假的竞争。

2. 概述

对异步程序中的数据竞赛越来越感兴趣，特别是 JavaScript。已经探索了静态和运行时的方法。 JavaScript 的静态分析的一个根本的挑战是，甚至可以列举所有相关的代码是非常困难的，因为 JavaScript 代码是通过 eval 调用和动态代码加载产生的。因此，静态分析的传统优点，即全路径覆盖很大程度上不适用于这个问题。因此，对缺乏数据竞争的声音陈述的能力受到损害[17,3,15,6]。

这个领域的运行时技术也容易受到精确度的损害，这使工具作者开发了避免潜在假阳性的预防措施[13,9,5]。他们也缺乏对缺乏竞争的保障和无法保证的缺陷。另外，运行时技术涉及到组合时间表的发展可能会遇到可扩展性挑战，特别是当可能的处理程序数量很多时[13,9]。

我们的技术试图结合静态和运行时分析的优点。我们只执行一次代码，但我们会探索多个执行命令。因此，我们的技术扩展得很好，同时增加了单次运行时分析的覆盖面。查看我们的方法的一个方法是通过分析异步事件处理程序的可能重新排序来探索特定

运行时执行的相邻调度。我们预计这种方法在基于 beta 或人群测试的环境中特别有用：拥有大量用户将自然地增加代码覆盖率。同时，用户的会话也不会明显放缓。应该注意的是，在浏览器中，减慢浏览器运行时间会改变 setTimeout 和 setInterval 设置的超时行为的风险。此外，运行时可能会主动尝试终止缓慢运行的事件。

2.1 什么是数据竞争？

已经提出了几种可能的数据竞赛定义，用于 Web 应用[10,11,13]。 他们都围绕由回调执行的共享状态的写入思想。 以前工作中的一些竞争是由用户交互和浏览器引导的时间引起的。 在这项工作中，我们主要关注的是 XmlHttpRequest（XHR）机制，允许客户端代码从服务器请求数据：

```
var xhr = new XmlHttpRequest ();

xhr.open("GET", "http://www.data.com/mydata.json");

xhr.onreadystatechange = function(e, d){ ... };

xhr.send(null);
```

以上代码适用于从服务器获取 JSON 数据的典型 GET 请求，并调度异步 onreadystatechange 回调以在数据到达时处理数据。 如果这些回调写入共享状态，这样的多次回调可能会很出色，从而创建了我们复制 XHR-XHR 竞争的可能性。 此外，通常，尽管可以执行同步 XHR 执行，但 XHR 被调度为异步调度，以维护响应式客户端 UI [10]。 在本文的其余部分，我们将重点介绍异步 XHR。

其次，单个 XHR 回调可以与浏览器竞争，导致状态变量被设置为 1 或 2.这是因为浏览器可能有多个脚本块，其中一些可能在调用之前或之后被调度 ，取决于回调的到达和浏览器渲染内容的速度：

```
<script > var xhr = new XmlHttpRequest (); xhr.open("GET", "http://www.data.com/mydata.json");

xhr.onreadystatechange = function(e, d){

state = 1; };

xhr.send(null); </script>

...a lot of text and images here...

<script > state = 2;</script>
```

第三，甚至更精巧的是，如果用户在多个浏览器选项卡中打开相同的站点，这些选项卡有可能导致并发执行。 下面的代码的两个实例可能会在不同的选项卡中运行时相互竞争，从而导致第 5 行基于 cookie 的竞争：

```
<script > var xhr = new XmlHttpRequest (); xhr.open("GET", "http://www.data.com/mydata.json");
xhr.onreadystatechange = function(e, d){
document.cookie = "value=" + Math.Random(); };
xhr.send(null); </script>
```

2.2 激励例子

为了了解数据竞争对网络的影响，我们花了一些时间来分析位于 GitHub 上的开源项目的错误报告。 下面我们从 GitHub 中描述一些微妙的服务器端错误的例子。 为了公平起见，我们应该提到，这些以 GitHub 问题报告的竞争例子对于 JavaScript 项目来说并不是特别常见的错误，这一点在很大程度上得到我们在第 4 节的结论的肯定。

Figure 1: Multiple XHR example.

示例 1 [旧服务器状态] Wheaton-WHALE 项目的问题＃79 描述了以下情况：

用户重新加载页面;

1. onbeforeunload 侦听器触发，数据保存到
2. 服务器;
3. 页面再次加载，并向服务器询问数据;
4. 客户端 JavaScript 代码加载旧的（过时的）数据;
5. 客户端状态保存到服务器。

在最后一步中，旧的 outdata 被保存到服务器，基本上忽略数据更新。 叛国的事实是，步骤 1 和 3 可以相互竞争：数据加载请求可能在保存被处理之前到达。 实现的修复使数据更新同步。

示例 2 [用户 ID 竞争]与名为 LikeLines2 的项目中的问题＃20 中捕获了与服务器发现的陈旧数据有关的一些类似情况。 LikeLines 为用户提供了一个浏览器视频播放器，可以为他们正在观看的视频提供可浏览的地图。这种情况描述了在初始化期间通过以下功能发出到后端服务器的两个赛车 XHR 呼叫：1）createSession 和 2）聚合。第一个呼叫是创建一个记录用户交互的新会话。绘制热图需要第二次调用。

当用户之前没有联系后端服务器时，会出现问题。在这种情况下，两个 XHR 调用将不会发出 cookie，在这两种情况下，服务器都将创建一个新的用户 ID。这显然是一个问题，因为交互会话与此应用程序中的用户 ID 相关联。如果来自呼叫的汇总“获胜”（即到达最后），则对服务器的后续调用将包含与交互会话不匹配的用户标识。 除了上面讨论的 GitHub 问题之外，我们还列出了一些来自先前工作中的一些样本的说明性例子[13,17]，尽管以前的工作并没有关注异步 XHR 的问题。

示例 3[浏览器中的竞争]考虑图 1 中的代码。为了方便起见，我们用数字标记上面的每个处理程序。此代码示例引发的发生之前的关系如下：1←2, 1←3, 3←4, 4←5。 因此，我们的探索算法将考虑 1←2←3 和 1←3←2 的可能性。 同样地，由于 2 和 5 是弱序，2 之前或之后发生 2 的痕迹将被考虑。虽然可以通过调度空间中的搜索来浏览这些跟踪，但我们选择通过数据流来进行。 我们跟踪可能是“并发”的事件处理程序，并将它们写入与弱写入相同的位置。因为 2 和 3 可以竞争，所以 document.cookie 的值将是 var1 = 1 或 var1 = 2。类似地，对于 2 和 5，document.cookie 的值将是 var1 = 1 或 var1 = 5。 精度，我们的算法保持现有的发生之前的关系，如 1 和 2 以及 1 和 4 之间的关系。
Figure 2: Sample trace illustrating cookie races.

2.3 跟踪处理

图 2 显示了通过运行图 1 中的代码来显示基于 cookie 的竞争获得的简单跟踪。我们开始处理打开 XHR 的行 1-4，并发送到远程服务器，我们将 XHR 回调标记为可以在将来的任何时间异步执行的活动回调。当我们处理第 6-10 行的第一个 XHR 回调时，我们将把 cookie 变量的写入值记录到存储器映射中，我们存储每个变量 id 的值（即 242159440 的 cookie）。

在记录书面价值的同时，我们将寻找可能相互竞争的任何回调书写的变量的值，如果有的话抱怨竞争。当我们继续处理跟踪时，我们将记录第 12-14 行写的值，首先检查 document.cookie 的早期值。由于此顺序代码段可能与较早的 XHR 回调竞争（没有发生在前的边缘），因此我们的处理将在 document.cookie 上记录一场竞争，同时将新的值添加到内存映射中。当我们继续处理跟踪时，将打开一个新的 XHR，并将其发送到 16-19 行，并添加到活动回调列表中。

之后，当对第 21-26 行执行对 cookie 的写入时，第二个 XHR 的回调将被处理。在处理写入操作时，将检查 document.cookie 的值，并记录一个竞争，两个 XHR 回调被标记为赛车，导致不同的 cookie 值。

2.4 算法摘要

在这里，我们为我们的方法提供了基本的直觉，并对第 3 节进行了更为正式的处理。我们的方法背后的关键思想是考虑给定跟踪中的替代方案。我们不试图强制使用 UI 交互的方式，例如，然而，我们会探讨不同时间表的可能性，因为异步事件处理程序的到达顺序是跟踪的一部分。我们的方法有效地执行在运行时收集的跟踪的静态分析，作为考虑不同计划的方式。当考虑多重执行 XHR 回调时，如图 3 所示，关键观察结果是，不是单独考虑每个可能的计划，我们可以通过合并状态并跟踪多个合并值来对可能的竞争的影响进行编码。这类似于在静态数据流或抽象解释风格分析中的满足，作为昂贵的全

路径（MOP）解决方案的替代方案。在概念上，给定一个变量 z 的合并点，其中多个值进入两个赛车 XHR，z = v1 和 z = v2，我们跟踪两个值{v1，v2};当然，如果 v1 = v2，则不需要保留两个相同值的副本。

我们制定我们的竞争检测算法作为对给定执行跟踪中的值的数据流分析。我们举报一个可能的竞争，如果多个值可能流向一个敏感的位置，表明存在安排依赖;这些敏感位置是持久存储，如 document.cookie，localStorage，sessionStorage，最后是 DOM 等。这些位置用作数据流分析的汇点。返回到图 1 中的例子，我们可以将处理程序 2 和 5 之间的竞争表示为处理程序 5 之后的合并节点作为分配 document.cookie = {'var1 = 2'，'var1 = 3'}。这当然代表了将多个值直接传递到敏感的持久存储位置 document.cookie。更有趣的情况涉及多个传播步骤。

2.5 实施

为了收集执行跟踪，我们测试了最新版本的 Firefox Web 浏览器。 我们的更改跨越了 SpiderMonkey JavaScript 引擎，以通过浏览器的内存跟踪数据传播，以及通过调用 DOM 记录的 document.cookie,localStorage 等上的操作。我们的工具涵盖了 Firefox 的三个主要组件：

- XPCOM（跨平台组件对象模型）：为了记录触发的 XHR 回调，我们在 nsThread.cpp 中的 Firefox 中调用了事件队列。 当从队列中获取事件以执行时，我们标记 XHR readystatechanged 事件以及用户发起的按钮点击。
- Gecko（布局引擎）：我们还需要使用 Firefox 的 DOM API 实现来记录对感兴趣的 DOM 元素所做的更新。 我们通过修改各种 DOM 类实现（如 nsGlobalWindow.cpp，DOMStorage.cpp 等）实现了这一点。
- SpiderMonkey（JavaScript 引擎）：最后，我们调整了 JavaScript 解释器，用于记录对变量和对象的值操作，并标记 XHR 回调执行的开始和结束点。 前者通过在 Interpreter.cpp 和 jsapi.cpp 上调用 JavaScript 字节码解释器来实现。

总的来说，我们的工具相当稀疏，我们相信可以轻松迁移到另一个开源浏览器，如 Chromium。 我们在 Firefox 中添加了大约 430 行仪器代码来收集我们的踪迹。 共修改了 12 个文件。

部署策略：作为审核步骤，在应用程序正在运行以及脱机时，竞争检测的过程都可以在线执行。 我们设想，作为 beta 测试的一部分，可以分析来自多个用户的痕迹。 请注意，正如我们将在第 4 节中强调的那样，即使相对简单的网站也可能会产生大量事件的长时间痕迹。 同时，与异步和调度相关的事件数也相对较少。 我们对发现潜在竞争

的分析是作为一个线性通过轨迹实现的。 然而，如果需要，这也是可以并行化的，通过在不同的机器上分割更长的轨迹以进行分析。

3. 格式化

我们发现方便的是将事件处理程序,XHR 回调以及脚本代码执行的部分(“序列块”)表示为具有唯一 ID 的执行块。执行跟踪中的其他条目也将被分配唯一的 ID，详见后面 4。

基于 Web 的 JavaScript 代码的关系之前没有普遍接受的定义。我们定义了在低级别内存访问级别（而不是低级别的内存访问级别）的发生之前的关系（表示为←），但是根据我们从 JavaScript 操作语义中提取的因果关系信息，在较高级别的语言结构层面，如图 1.在每个不间断执行块中，跟踪条目由程序排序，因此在命令之前发生。我们定义并记录块之间的发生顺序，以便如果 id←id'，则 ID 中的所有跟踪条目都将发生在 id'之前。我们订购 id←id'如果 id'出现在跟踪中的后面，并且其中一个保持：

1.两个块都是顺序块。 由于浏览器强制的顺序，顺序块按照它们出现在跟踪中的顺序排列。

2.两个块都是事件处理块。 为了反映用户交互的顺序，事件处理块按照它们在跟踪中出现的顺序排列，以反映用户交互的顺序。

3. id'是一个 XHR 回调，XHR 发送 ID ID 的块内的 ID; 这是因为 XHR 回调只能在发送操作之后发生。

3.1 痕迹的建模与分析

在本节的其余部分，我们正式解释我们的竞争检测方法。由 JavaScript 程序操纵的一组内存位置（“位置”）由 Locs 表示，它们可以由 Val 执行的一组值。未初始化位置的值为⊥Val。我们将每个对象的每个字段视为一个独特的，单独的普通位置进行竞争检测。 document.cookie，sessionStorage 和 localStorage 是 JavaScript 程序在 Web 浏览器上操纵的变量。这些变量中的每一个的类型是键值存储，它们是持久存储的，即存储在磁盘上。 KV 表示这些位置的集合。为了均匀地处理跟踪中的读和写访问，我们将每个（kv，ky）对视为具有名称（kv，ky）∈Locs 的存储器位置。我们使用⊥Val 表示不在商店中的键的值。在给定的 kv∈KV 中访问不同的密钥，类似于访问不同的存储器位置，不会相互竞争。表示 DOM 元素的位置集合，并使用 DOM 元素的 innerHtml 属性使用 setter 进行写入，由 DOMElts⊆Locs 表示。

执行中每个点处的每个位置 v 的值由一个名为 V 的存储器映射表示.V（v）由一组形式（vl，id）组成，其中 vl∈Val 和 id∈ID。直观地说，（vl，id）∈V（v）意味着在跟踪发生的跟踪或重新排序中，v 可能具有通过块 id 写入的值 vl。当处理具有 ID id

的跟踪条目是对 v 的写入访问权限时，我们关键地使用到此之前计算的事件发生之前的关系。内存映射 V（v）中的所有条目都将删除 id 之前的 id id'的跟踪条目，表示它们已经被稍后的访问覆盖了。 V（v）中的其他条目是由于“并发”执行块而导致的，因此不会被删除。

3.2 定义踪迹

形式上，跟踪是轨迹条目⟨$\lambda_0,\lambda_1,\lambda_2,...\lambda_n$⟩ 以下类型的 λn：

- XHR，事件处理和顺序块：跟踪条目 λ= CBBegin（id）， λ= CBEnd（id）表示对 ID 为 ID 的 XHR 执行回调的开始和结束。 λ=HandlerBegin(id)和 λ=HandlerEnd（id）对事件处理程序块执行相同操作，对于顺序块， λ= SeqBegin（id）和 λ= SeqEnd（id）。
- key，location，innerHTML 访问 λ= keyWr（kv，ky，vl，varsRd，id）表示在 ID 为 id 的块内的密钥值存储 kv 中写入密钥 ky 的值 v1。作为表达式在内存位置 varsRd 上的结果，值 v1 已经在写入跟踪条目之前立即计算。 λ= keyRm（kv，ky，id）表示从块 id 中的 kv 去除密钥 ky 的值。 λ= varWrite（v，vl，id）表示将值 v1 写入块 ID 内的位置 v。最后， λ=setHTML（hElt，hVal，id）表示 DOM 元素 hElt 的 innerHtml 属性的设置，以在块 id 中计算 hVal。
- POST，XHR 发送： λ= post（url，id，idin，vl，varsRd）是 POST 请求 XHR 调用或具有 ID id 的窗口，并且调用发生在具有 ID idin 的执行块中。发布的数据 vl 是存储器位置 varsRd 上的表达式的计算结果。 λ= xhrSend（id，idin）表示具有 ID id 的 XHR 对象的 XHR 发送操作，该 ID 为 ID idin 的执行块内。这个发送是一个 GET 请求。

3.3 解释痕迹

给定轨迹 Trace =⟨λ0， λ1， λ2，...， λn⟩，我们的竞争检测算法通过对其进行依次处理，一次记录一次记录。该算法保持由元组 Σ=（V，HB，P，idSeq，idEvt）表示的分析状态。这里，V 是存储器映射。 HB 是发生在之前的事件，这是 ID 的部分顺序。每当通过跟踪处理规则将新元素添加到 HB 时，则关系的传递关闭被用于获得标准 HB。HB 最初是空关系。 P 是在计算由 IDH 的 XHR POST 请求提交的值时已经读取位置 v 的形式（v，id）的对的列表。最后，idSeq 和 idEvt 分别是由我们的算法处理的最后一个顺序块或最后一个事件处理块的 ID，或者如果不存在这样的回调或块，则为⊥。更新分析状态的规则 Σ=（V，HB，P，idSeq，idEvt）在图 4 中给出并说明如下。

回调：CB-Begin 和 CB-End（未显示）跟踪正在进行的 XHR 回调块的 ID。Seq-Blk-Begin 处理日志条目，指示 ID ID 的顺序块的开头，并将对（id Seq，id）添

加到发生前的关系。Seq-Blk-End 将 idSeq 重置为 id，规则确保跟踪中的序列块的发生顺序及其发生在订单合并之前 Evt-Handler-Begin 和 Evt-Handler-End 操作与相应的 Seq-Blk 规则类似，跟踪中的发生顺序事件处理程序与命令之前的事件相同。事件处理程序和顺序块不是相互排列的。

位置更新：Key-Write 处理键值映射 kv 中的 key ky 被更新的情况。如果| Val（v）|，则声明有潜在的有害竞争对于任何位置读取的值为 1，而正在计算密钥的新值（v∈varsRd），表示非确定性的可能性。该规则通过添加对（vl，id）和删除所有对（vl'，id'）使得 id'←id 从 V（kv，ky）计算 V'（kv，ky）。键删除（未显示）将值⊥Val 写入键 ky。 Write 和 Set-DOM 类似于上面的 Key-Write，通过删除由 before-before 关系重写的值 id 对对位置 v 或 DOM 元素 hElt 来更新 V，并添加由当前跟踪条目。

POST，发送：规则 XHR-Post 声明如果| V（v）|有潜在的有害竞争> 1 对于任何位置 v∈varsRd,因为在计算 vl 中至少一个位置,发布的数据具有非确定性的潜力。XHR-SEND 更新发生之前的关系。 xhrSend（id，id in）表示具有 ID id 的 XHR 的发送以块 idin 的形式发生。 idin←id。这个规则和 CB-Begin 和 CB-End 不修改发生关系的事实编码了只有链接的 XHR 调用被相对于彼此排列的事实。

Figure 4: Trace analysis rules.

3.4 检测竞争

当存储器映射包含至少两个不同的 v 值时，即当 V（v）中存在 V（v1，id）和（vl'，id'））时，V（v）在位置 v 上具有非确定性潜力， （v），使得 v⊭ vl'和 id⊭ id'。如果由规则计算的存储器映射 V'（v）在 v 上具有非确定性潜力，则我们的算法声明在位置 v 上进行竞争，同时评估写入，设置 DOM 和密钥写入规则。评估 XHR-Post 和 Key-Write 规则，如果在计算发布或写入的值时读取变量，v∈varsRd，V（v）对 v 具有非确定性潜力，在这些竞争中，只有与 Key-写和 XHR-Post 规则被认为是有害的竞争。

在“安静”状态下考虑跟踪的前缀，使得在前缀结尾处没有 XHR 回调或执行块正在进行。假设我们的算法在处理最后一个块中的跟踪条目 λ 或在该前缀中具有 ID ID 的回调中，在位置 v 上发出非确定性潜力。然后，如前所述，在完整的前缀中保留前缀中的执行块和/或 XHR 回调的不同重新排序可能导致 v 的不同最终值。

Let（vl，id）和（vl'，id'）在 Val（v）中，使得 id'⊭ id 和 v⊭ vl'。一定是这样的情况：当 id'和 id 都写入 v 时，这两个执行块不会被发生在之前的关系中排序。否则，根据我们的算法的更新规则，以及如何在变量更新中使用发生之前的关系，否则将从 Val（v）中删除（vl，id）或（vl'，id'）。因此，可以通过延迟具有 ID id'的块的执行来修改执行，直到执行具有 ID id 的块为止。在这种情况下，在新获得的执行

的最终状态下，位置 v 或（kv，ky）的最终值将不同。这表明纯粹由于 XHR 回调调度非确定性的结果而产生的潜在不同的结果。

我们的竞争检测方法有两个假阳性来源。第一个是在前一段中作出的假设，同时通过延迟具有 ID id'过去的其他块的块的执行来获得新的执行。假设是基于由数据写入的数据值在原始执行中进行的控制决定块 ID 不会以使重新排序的执行不可行的方式进行修改。在我们的经验经验中，这种情况很少，可以通过检查或重播和验证重新排序的执行来排除。第二个来源是将值写入持久位置或在网络上发送时执行的检查。在这些情况下，如果在读取写入或发送的值时，如果读取具有非确定性的位置，则我们的算法发出潜在的有害竞争。这种方法是保守的，即表达式中一个位置的值中的非确定性可能不会导致结果值的非确定性。即使这是造成虚惊的原因，我们认为流程到持续输出位置的潜在不同的数据值可能是程序员所关心的。

Figure 5: Characteristics of our benchmarks web sites.

4. 评估

本节介绍了我们对一组 26 个复杂页面执行的实验评估。

研究问题：我们的目标是解决下列研究问题。

RQ1：持久性状态（例如 document.cookie，localStorage 和诸如 POST 请求等副作用）的常见问题？ 我们认为这些是 Web 应用程序中最有害的竞争。

RQ2：sessionState 上的竞争有多常见？ 浏览器重新启动会话状态。 但是，由于浏览器通常不会重新启动几天，如果不是几周，sessionState 中的数据可能会持续很长一段时间，如果不是永久性的。

RQ3：诸如内存位置和 DOM 元素之类的过渡状态的竞争有多普遍？ 这些通常不是我们认为是高度有问题的错误，而且通常这些错误甚至不能被用户观察到[10]。

4.1 实验装置

在我们以前的工作[10]中，我们使用仪器浏览器从 Alexa 的前 5000 名列表中抓取所有网站。对于我们在这里的实验，我们使用这些网站的 26 个站点，大量使用 XHR。

网站统计：图 5 总结了我们选择的网站的各个方面。我们特别分开了 XHR 回调中的操作数量，因为这些操作可能会导致竞争。列 2-5 显示观察到的 XHR 打开和发送操作的数量，执行 XHR onreadystatechange 回调以及其他回调（所谓的嵌套 XHR）中的回调数。专栏 6-11 侧重于使用永久存储。列 6-7 显示了对 document.cookie 的写入数以及嵌套在 XHR 回调中的写入数。第 8 列显示 XHR 回调中的 XHR POST 操作的数量。第 9-10 栏给出了一般和 XHR 回调中对 localStorage 的写入次数。列 11-12 为 sessionStorage 提供相同的

信息。最后，第 13-16 栏提供有关 DOM 操作的各种形式的信息，例如设置 innerHtml 和更改 INPUT 元素的内容;提供这些计数的总计和嵌套变量。

跟踪统计：用于收集这些计数的工作量只是加载页面，并应用基本的用户交互，如按钮链接点击。 DOM 操纵的计数通常高于 document.cookie，localStorage 或 sessionStorage 的计数。与更常见的使用的持久性元素相比，DOM 元素还需要更多的竞争。图 6 总结了我们用于分析的痕迹的信息。压缩比在 9.97 和 33.83 之间。分析跟踪记录跟踪时间所需的时间百分比在 3％至 32％之间。

检测时间：图 6 显示了跟踪收集 6 和分析时间作为跟踪大小的函数。分析时间随着迹线的大小大致线性增长，并且是第二列中显示的跟踪记录时间的一小部分。

4.2 检测结果

在本节中，我们描述和分析我们的方法发现的几个代表性的竞争。

Figure 6: Trace statistics and processing times.

Figure 7: Races found by our analysis.

示例 4 [milliyet.com.tr 中的 sessionStorage]在 www.milliyet.com.tr 的情况下，sessionStorage 的竞争是由用于生成密钥名称的共享变量命名空间引起的。变量写在两个不同的位置，一个是使用 jQuery.ajax 方法执行的 XHR 回调。执行步骤如下图所示，相关代码如图 8 所示：

1.加载页面时，使用 jQuery.ajax 方法（包括在 www.milliyet.com .tr / D / j / base.js？ v = 20）中创建 XHR，并将其发送到服务器（第 3-5 行）;

2.一旦收到响应，用户定义的回调将使用 jQuery.ajax.done（行 19-22）执行;

3.命名空间变量在 jQuery.ajax.done 方法的末尾设置为空字符串（第 21 行）;

4.通过将命名空间变量设置为“uv”（第 46 行）来初始化外部库。

5.命名空间变量随后用于生成一个 sessionStorage 键（第 43-45 行）。

在最后一步中，命名空间的值用于设置 sessionStorage 上的一个项目，用于记录用户与网页的交互。在这个特定的跟踪中，我们观察到使用命名空间作为前缀将多个项添加到 sessionStorage 中（即禁用 uv autoprompt，uvr）。对 sessionStorage 键的任何未来读操作将取决于此前缀命名空间。由于 XHR 回调可以与命名空间写入相匹配，命名空间的写入将命名空间的值设置为空字符串，否则可能导致读操作失败。

示例 5 [gazetta.it 中的 document.cookie]在 www.gazetta.it 的 document.cookie 竞争的这个例子中，该网站使用应用程序监视库（DynaTrace Real user Monitoring 在 dynatrace.com 上找到）将用户操作和浏览体验记录并发送到远程服务器。本文发起的

每个 POST 请求都会使用服务器的响应值更新 document.cookie 的关键 dtCookie。当多个 XHR 调用尝试将新的响应值写入 document.cookie 时，发生竞争。执行步骤如下图所示，代码如图 9 所示：

1.在页面加载时间，创建一个 XHR 呼叫来初始化监视过程（第 9 行）；

2.将数据发送到由 dtCookie 值（线 6-8）构造的 URL;

3.dtCookie 键随服务器的响应更新（第 35-37 行）；

4.在 onLoad DOM 事件上创建一个新的 XHR，用于发布页面的加载时间（第 11-13 行）；

5.每个 XHR 的回调尝试写入同一个 document.cookie 键 dtCookie（第 35-37 行）。dtCookie 键的值取决于 XHR 回调的顺序，导致未来 POST 操作可能出现的问题，因为该值用于生成 URL。

Figure 8: sessionStorage manipulation in milliyet.com.tr. To save space, we remove unrelated lines of code.

Figure 9: Race on document.cookie manipulation in gazetta.it. To save space, we remove unrelated lines of code.

Figure 10: Trace from optimum.net illustrating a false positive. tention on write-write races on persistent state.

4.3 假阳性

在寻找可能的假阳性时,我们决定把注意力集中在持久的竞争(19)和 DOM 状态(47)上。在我们调查的 66 场竞争中，我们确定只有两例为假阳性。这些都是 fedex.com 和 optimal.net 上的文件。

图 10 显示了最佳网络的轨迹，并在多个位置（第 6,10 和 22 行）写入了 cookie 密钥 fsr.s，其中一个位于 XHR 回调中。在这种情况下，fsr.s 用于收集通过连接新值更新键值的站点统计信息（行 11-22）。虽然 XHR 回调可能发生在第 5 行和第 6 行的两个 cookie 写入之间，导致键的不同值，但在执行结束时，cookie 密钥 fsr.s 将包含所有写入的值，但按不同的顺序。在这种情况下，程序将该值视为一个集合而不是一个列表，所以当我们正确地捕获非确定性时，这并不是程序中的一个错误。 fedex.com 中的假阳性非常相似，一个竞争是在 Cookie 密钥上，在 XHR 回调中更新。

良性的记忆竞赛：我们还要强调一个没有持续后果的记忆竞赛的例子。在 radikal.com.tr 上，有一个全局可变的“持续时间”，用于测量执行期间在序列代码更

新和 XHR 回调之间出现竞争的不同点所经过的时间。该内存位置的值不会延续到任何持续状态，并导致非确定性。

4.4 讨论

回到第 4 节中的研究问题，我们清楚地看到，根据图 7，持续状态（RQ 1）的竞争很少见，26 个站点只有 19 个。会话状态（RQ 2）上的竞争也很少见（仅观察到 3 场竞争），部分原因是会话状态不像 cookies 那么频繁（图 6）。最后，流动国家的竞争更加频繁。虽然 HTML 内容的变化可能在技术上是竞争，可以说，大多数是短暂的。实际上，基于 JavaScript 的网页的执行模式引起了大量的非确定性。这是因为许多网页的大部分网页在多次重新加载中是不一样的：页面两边的广告更改;页面的内容经常发生变化（考虑到新闻网站的变化），有时第一次访问页面和后续页面的时间有限，因为 cookie 是不同的。浏览器用户已经受过训练，不希望有很大的一致性，而当其他一切都失败时，重新加载页面。此外，JavaScript 执行模型是非常宽容的：错误被运行时“吞没”（当前事件处理程序被终止），并且在许多情况下，页面可以在异常中生存并继续运行。

我们的观察结果与用户在网络上遇到任何问题的传闻经验相吻合：只需要重新加载页面即可解决问题。在某种意义上，网络编程非常宽容。这与桌面应用程序中的线程相关的竞争以及移动应用程序中的数据竞赛[9,5]不同，研究人员能够以明显的视觉效果复制竞争（例如，屏幕上显示的图像被颠倒或颠倒）。

最后，我们是一个邻里探索技术。给出一个执行跟踪，我们探讨回调的可能交错。

5. 相关工作

本节介绍了最近在 JavaScript 程序和异步代码中查找竞争的一些工作。

异步程序中的竞争：Hsiao et al。提出了一种在 Android 应用程序中查找异步竞争的一个子集的方法，重点是导致使用后自由违规（即使用释放的指针）的竞争[5]。虽然此工具还对单个执行跟踪进行脱机分析，但他们的重点是计算明确的事前关系，它们适用于跟踪，以便查找可能导致用户自由后的可能性的访问。他们采用一些启发式方法来最大限度地减少误报的可能性。我们专注于从多个值到敏感位置（如 document.cookie）的数据流，这自然消除了对交换性的明确理解的需要。

Maiya 等[9] et al。专注于使用 UI 资源管理器对可能的计划进行系统的探索，并使用竞争检测器推导所获得的踪迹。 Android 执行生命周期的精确模型是避免误报的关键，尽管仍然存在大量数据。

JavaScript 中的竞争：Zheng et al。 [17]提出了一种静态技术来检测 JavaScript 应用程序中的潜在竞赛。最近，Petrov 等人[11]和 Raychev 等人[13]已经观察到异步创

建无序执行的潜力，并开发了用 JavaScript 编写的 Web 应用程序的竞争条件概念。原则上，竞争条件可能由于访问网页的组件之间共享的数据而不是通过正确的同步排序，或者更正式地发生在之前的关系。当然，在一个网页上，整个 DOM 是（一个巨大的）全球状态，创造了竞争的潜力。

彼得罗夫等人[11]为网页定义一个先发制人的关系，并概括了竞争条件的概念，以考虑到逻辑上对同一资源有无序访问的情况。作者提出了一种用于在给定的网页执行中检测竞争的动态方法，探索类似的可能会变得丑恶的执行，并且在以后的工作中[13]识别和过滤大量的良性竞争。洪等[4]提出了一个测试框架以及 JavaScript 中异步事件处理程序的执行模型。使用定义的执行模型，框架将执行一个 JavaScript 应用程序来收集执行的异步事件处理程序，然后生成测试用例，以测试这些事件的不同顺序。之后，检查每个测试用例的结果 DOM 结构，以便初始执行构造的引用。如前所述，我们的工作与这些研究不同，事实是我们只追求导致持久状态或发送到服务器的数据的非确定性的竞争条件。我们选择的发生之前的关系遵循这种设计决策，只记录高级因果关系。
Mutlu 等人[10]倡导可观察的竞争的概念，即可以被最终用户看到和视觉区分的竞争的概念。我们对持久性副作用的观念比本文中所观察到的更强。

JavaScript 中的程序分析近年来已经提出了一些 JavaScript 的分析;

这里我们只突出一点点。此外，语言的几个方面，如使用 eval [15,6]，并试图了解 JavaScript 性能[12,14]。 Rozzle [7]提出了在 JavaScript 引擎的上下文中轻量级多执行的想法，类似于我们的工作。 Rozzle 的目标是通过增加代码覆盖范围来扩大恶意软件检测器的影响，从而观察更多，潜在的恶意代码。在技术方面，Rozzle 可能是本文中描述的最接近的运行时探索方法。 Chugh 等人的一个项目侧重于分析 JavaScript 并在客户端代码中查找信息流违规[1]。 Gatekeeper 项目[2,3]提出了一个点分析，以及一系列针对安全性和可靠性的查询以及对增量代码加载的支持。湾流[3]是网守项目的继任者，其重点是增量分析和动态代码加载。 Sridharan 等人[16]提出了一种用于跟踪 JavaScript 程序中动态计算的属性名称之间的相关性的技术。他们的技术允许他们精确地将关于从一个对象复制到另一个对象的属性进行归结，就像在 jQuery 这样的库中通常是这样。 Madsen 等人[8]提出了使用分析的想法，用于使用大框架和逻辑的 JavaScript 应用程序中的调用图构造的目的。他们的使用分析结合了应用程序其余部分的分析。

5. 结论

本文提出了一种替代方式来查看使用 JavaScript 编写的 Web 应用程序中的什么构成一个竞争。我们主张专注于由异步回调及其到达顺序引起的竞争，主要是由

XmlHttpRequest（XHR）机制产生的竞争。不同于以前的工作，结论是 JavaScript 有很大的竞争潜力，我们的研究结果表明，鉴于 JavaScript 应用程序的宽容性，破坏性的持续竞争更加罕见。

然而，我们提出了一种轻量级算法，可以在特定运行时间跟踪的“邻域”中显示不同的调度。我们的方法避免了静态分析的不精确性和运行时进度勘探的组合爆炸和可扩展性问题。我们在 26 个网站中共发现和调查了 19 场有害竞争和 621 次良种赛。

7.复印包装

本文提供了一个复制包，旨在帮助其他研究人员在这一领域工作。作为我们复制包的一部分，我们提交

1. Firefox Web 浏览器的检测版本，可以使用哪些执行跟踪可以收集

2.我们的竞争检测工具，用于处理这些痕迹进行评估。

复制套件评估委员会认为这一提交令人满意，并确认我们的结果可以转载。我们的**提交包括一个具有以下内容的虚拟机：**

赛车检测器可执行：我们的赛车检测工具由解析器和赛车检测模块组成。将分析给定的跟踪文件，并构建执行中的每个点处的状态映射和发生前的关系。可执行程序接受多个跟踪的跟踪路径或目录，并将在每个状态输出相应跟踪的统计信息和检测到的竞争条件。

仪器化 Firefox 可执行程序：

我们检测了 Firefox Web 浏览器，以便在执行网站时收集相关信息。我们的仪器跨越 12 个源文件，约有 430 行仪器代码。大多数仪器都在用于记录内存操作操作的 JavaScript 引擎（SpiderMonkey）的代码中。用于标记异步回调的 Web 浏览器的事件循环实现也被检测。

收集我们的基准网站的痕迹：

我们使用我们的仪器化 Firefox 浏览器收集了第 4 节中提供的基准网站的 26 条个人痕迹。

我们从 Alexa 的 5000 强榜单中选出了大量使用 XMLHttpRequests（XHR）的网站。对于每个网站，我们进行了基本的浏览操作（例如，按钮点击，导航到新链接）并收集生成的跟踪。这些跟踪的大小从 14 MB 到 400 MB 不等，具体取决于网站的浏览时间和内容。

用于评估的脚本：

我们提供两个批处理脚本 RunFSEAnalysis.bat 和 RunLastAnalysis.bat，用于自动执行评估。这些脚本将分别使用我们的仪器化 Web 浏览器对我们的基准轨迹和最后收集的轨迹进行竞争检测。包含跟踪统计信息，特征和检测到的竞争的结果被视为*.csv 文件。

7.1 评估基准网站

复制包包括用于收集我们基准测试的每个网站和批处理脚本（RunFSEAnalysis.bat）的跟踪，用于对这些跟踪进行竞争检测分析，以复制表中所示的结果（图 5，图 6 和图 7 ）。提供的脚本首先在每个网站跟踪上运行我们的竞争检测机制，然后将分析结果显示在 FSE_Reports 目录下的.csv 文件中：

- LogCharacteristics.csv 包含每个分析轨迹的特征（即写入数，XHR 数等）（图 5）。
- LogStatistics.csv 包含关于每个分析的跟踪的统计（即跟踪大小，分析时序等）（图 6）。
- LogRaces.csv 包含一个报告，总结每个分析轨迹上检测到的竞争数（图 7）。

我们的竞争检测机制记录了所提供的踪迹的统计数据（见表 5）和检测到的竞争条件（见表 7））。在应用自动竞争条件检测后，我们手动探索了持续状态下报告的竞争。通过批处理脚本进行跟踪大小，压缩跟踪大小和检测次数的分析。

7.2 收集和评估新痕迹

我们的复制套件还包括一个仪器化的 Firefox Web 浏览器，用于收集可以通过我们的竞赛检测机制进行分析的新痕迹。用户可以通过执行快速 FSE_Firefox 访问仪器浏览器，并在安全模式下使用它来收集网站上的新痕迹。一旦浏览器终止，收集的跟踪将被写入磁盘上的文件。由于测试和使用虚拟机，用户可能会遇到网络浏览器的一些放缓。提供的批处理脚本 RunLastAnalysis.bat 首先将最后写入的跟踪文件（命名为%date%_%time%_log.txt）复制到 Last_Run_Reports 指令，然后对该跟踪运行分析。然后，竞争检测分析的结果将作为独立的* .csv 文件持续存储在上一节中，以及跟踪本身。