Web Feature Deprecation: A Case Study for Chrome

Ariana Mirian
University of California, San Diego
amirian@eng.ucsd.edu

Nikunj Bhagat
Google Inc.
nikunjb@google.com

Caitlin Sadowski
Google Inc.
supertri@google.com

Adrienne Porter Felt
Google Inc.
felt@google.com

Stefan Savage
University of California, San Diego
savage@cs.ucsd.edu

Geoffrey M. Voelker
University of California, San Diego
voelker@cs.ucsd.edu

***Abstract*—Deprecation — signaling that a function is becoming obsolete in the near future — is necessary for the web ecosystem. However, an unsuccessful deprecation can lead to pain for both developers and end-users. In this paper, we analyze web feature deprecations with the goal of improving Chrome's deprecation process by balancing the need for deprecation and minimizing user pain. We produce a taxonomy of reasons why developers want to deprecate web features and a deprecation methodology for deciding when it is safe to deprecate a web feature. We also discuss the challenges faced during the deprecation process.**

I. Introduction

A fundamental aspect of working with software is working with APIs: either developing them or using them. Furthermore, a fundamental aspect of working with many APIs is deprecation. Deprecation is the act of marking a feature as obsolete, as a warning to developers that the feature will be removed in the near future. Thus, deprecation is a signaling mechanism to help ease the transition away from a feature and ensure developers are notified in advance of feature removal. Because deprecation may result in frustration for both users and developers — developers have to adapt their code and users will deal with any breakage — API maintainers must be careful with deciding to deprecate a feature.

In this paper, we focus on deprecation of APIs (or API behaviors) created by browsers, called "web features". The specific question we investigate is **how to help browser developers decide whether to deprecate web features**. Web feature deprecation makes a good case study subject because of the sheer scale of how many people may be affected. New web APIs are continuously created and then used by web developers to create and improve millions of websites, which are in turn utilized by billions of individual users. In 2014 there were 91 *new* features implemented in the Chrome browser, while in 2015 that number nearly doubled to 161. While these new Web APIs spur innovation on the web, not all of them thrive equally. Hence, web feature removal is a necessary part of a healthy web ecosystem: feature experiments fail, security problems arise, new and better APIs are introduced to replace the old corpus, etc.

Unfortunately, the decentralization and scale of web deployment can make feature removal a challenge. There is neither a straightforward mechanism to identify all the web sites using a particular web API, nor a means to seamlessly update them. A feature used on 100K web sites requires more developers to respond than one used only on 100 sites. When a deprecation is unsuccessful — when significant reliance on an obsolete feature continues — a removal may affect the appearance and functionality of sites across the web, leading users to become disillusioned with the sites, the browser, or both.

A key challenge in deprecating features is balancing the costs to developers, the impact on users, and progress on the web. Many browsers have procedures and protocols in place to ensure that a deprecation does not cause too much pain, typically based on usage metrics. These procedures are not always effective, such as when Chrome needed to postpone deprecating a feature to autoplay videos with sound after public outcry [1]. To keep its user base intact, a browser wants to avoid unsuccessful deprecations.

In this paper we focus on the Chrome browser. Chrome is a natural choice as it is a widely popular open source browser, with public discussions about whether to deprecate web features [2]–[4]. Furthermore, the Chrome web feature team was interested in updating the protocol they use for deciding when it is safe to deprecate a web feature.

We investigated why developers choose to deprecate web features. Features that are deprecated for different reasons might need to be treated uniquely in the deprecation process. We classified 2.5 years of web feature deprecations in Chrome (Table III). Via analysis, we determined that these different categories of features should be treated differently when deprecating. We also examined better ways to measure user pain. We discovered many interesting issues that arise in practice, as well as the tradeoffs among different data sources.

The main contributions of this work are:

- The first paper investigating web feature deprecations.
- A taxonomy of reasons why developers want to deprecate web features.
- A collection of data sources, metrics, and thresholds for deprecating web features.
- Guidelines for deciding when it is safe to deprecate a web feature and an accompanying tool for determining

TABLE I
POSSIBLE INTENTS IN CHROME AND A BRIEF EXPLANATION OF EACH.

Intent	Explanation
Intent to Implement	Add the code with a flag in Chrome so that it is not enabled by default.
Intent to Experiment	Run experimentation trials on Chrome to get feedback from developers.
Intent to Ship	Flip the feature to default.
Intent to Deprecate	Signal to developers that the feature will be removed in a future release.
Intent to Remove	Remove code from Chrome; pages reliant on feature may not render correctly.

whether a web feature meets these guidelines.
- A methodology for approaching the question of whether to deprecate an API.

As mentioned above, web features are a particularly interesting topic for a case study because they affect a vast number of both developers and end users. However, the deprecation methodology we propose in this paper could be used for other domains, such as language level feature deprecations, since the fundamental question of how to decide when a feature is safe to deprecate transcends the case of web features.

II. BACKGROUND

Web browsers have become much more intricate and complex since they were first introduced in the 1990s. A browser typically has a rendering engine, which parses the HTML and CSS for a specific URL into a graphic display that is more easily read by humans. *In this paper, we broadly define a web feature as any browser behavior available to the developer of a web page; we consider Javascript functions, CSS properties and HTML elements as examples of web features.* Many web features are standardized by committees of developers and practitioners, such as the W3C and the WHATWG, and then browsers implement these specifications. This standardization process ensures that a feature on Chrome behaves roughly the same on Firefox, Safari, and other popular browsers. Standardization reduces friction for both developers and end users: website developers do not need to worry about implementing different versions of their web page for different browsers, and end-users receive a similar experience for a specific webpage regardless of their browser choice.

Browsers each have their own protocol for implementing and deprecating web features; we focus on the Chrome browser in this paper. A standard does not need to be finalized to be implemented in a browser, and implementation and experimentation is typically encouraged by browsers. To add or remove a feature from Chrome, a developer will email `blink-dev@chromium.org` with their Intent for the feature [5]. This Intent process is entirely public to provide transparency on web feature decisions, and Table I shows how each Intent directly maps to a specific action within Chrome.

TABLE II
NUMBER OF FEATURES IMPLEMENTED AND DEPRECATED EACH YEAR IN CHROME SINCE 2014.

Year	Implemented	Deprecated
2014	91	65
2015	161	42
2016	216	40
2017	174	49
2018 (up to July)	100	23

The two Intents we focus on in this paper are Intent to Deprecate and Intent to Remove. Although deprecation is required in almost all removal cases, they are still treated as two independent steps: a developer must deprecate a feature before they can remove it. A successful deprecation is one that is silent for end-users, and does not invoke a strong negative reaction from web developers, while still leading to the removal of the feature. A vocal minority might express dissent, but the deprecation is still a good choice for the web.

To date in Chrome, a developer will typically gather one or two usage metrics on the feature to get a sense of potential user or developer pain if the feature were removed today. If those metrics fall below thresholds that have been set via anecdotal experience, then the feature can probably be deprecated and subsequently removed [6]. If the metrics do not fall below the threshold, then the developer can perform outreach to try and lower the use of the website; for example, a developer might individually reach out to large companies whose websites still use the feature in an attempt to understand why. While an established protocol exists, much of the deprecation process is based on intuition and past experiences.

Implementing and shipping new features are important for the growth of the Internet, but so is deprecation and removal. Deprecations are necessary for a variety of reasons, such as experimental features that never become standardized, changing standards, or when a security problem emerges with a specific web feature. Table II shows the number of implementations and deprecations in Chrome since 2014, to give a sense of the balance between these two separate functions. While the number of features implemented since 2014 has increased, the number of features deprecated has remained constant.

The difference between the number of features implemented and deprecated may seem disparate, but it can be attributed to browser developers attempting to avoid unsuccessful or unpopular deprecations. The first principle in the Blink principles of web compatibility — a document that dictates what factors to consider when avoiding breaking changes on Chrome — is "minimizing end-user impact" [7]. Given the ripple effects, browsers are more conservative when deprecating a feature than implementing a feature. An unsuccessful deprecation can turn users away from the browser.

For example, in 2017 Chrome decided to disallow videos from autoplaying with audio on web pages [8]. The main

intention behind this change was to stop advertisements that would autoplay when a user was visiting a website [1]. However, this change also affected Internet-based games that relied on the autoplay functionality to work and as a result broke such games. In fact, the impact was so severe that Chrome delayed removing this feature to a later version, and as a result the initial deprecation was unsuccessful.

Autofill completion is another example of an unsuccessful deprecation. Again in 2017, Chrome deprecated "autocomplete=off", which means that Chrome would ignore that attribute when present on a web page [9]. This behavior complies with the specification for the autofill web feature and is the result of a decision that Chrome made to enhance the overall use of the autocomplete feature for their users [10], [11]. However, Chrome received a significant backlash, including a highly-rated post on hacker news and two bugs that received numerous complaints (164 and 234 posts, respectively) [9], [10], [12]. In this case, the end-user might not have faced much friction, but web developers were unhappy with the change in default behavior and placed the blame on Chrome.

III. Overview of Methodology

The methodology we followed consisted of three main steps:

1) We determined deprecation categories based on the history of why features have been deprecated in the past. We first classified more than 100 previous Chrome web feature deprecations to identify specific categories. Our rationale was that if web features were deprecated for different reasons, then those different categories may also have different thresholds for deprecation metrics. The categories for web feature deprecations we discovered are discussed in Section IV.

2) We determined data sources, metrics, and thresholds to measure both user and developer pain. We analyzed a variety of different datasets, both internal and public, to determine appropriate metrics. Moreover, we also retroactively analyzed metrics data to provide guidance on establishing thresholds. The data sources and metrics we decided on and the thresholds we use are discussed in Section V.

3) We determined deprecation guidelines based on API developer goals and constraints. Our final step was to create a set of guidelines; metrics and thresholds are not as useful without a set of instructions to assist in interpreting them. We used the thresholds as ground truth, and then used other data sources, where we did not have historical data, as auxiliary datasets in the decision-making process. The guidelines we have provided to the Chrome team are outlined in Section VI.

Although we focus on the particular case of deprecating web features in Chrome in this paper, one could apply the methodology we followed to other contexts.

IV. Taxonomy of Web Feature Deprecation

Chrome currently deprecates all web features in the same way: a web feature is approved for deprecation if certain metrics fall below canonical thresholds. However, since web features are deprecated for very different reasons, we argue that features should also be deprecated according to their nature. For example, a feature that is deprecated for security concerns might be severe enough to warrant more lenient thresholds for deprecation. Our first step was to determine the different reasons why a developer would want to deprecate a feature.

To achieve this goal, we examined 2.5 years worth of Chrome web feature deprecations and manually categorized each feature based on why it was getting deprecated. One author read through all 123 deprecation messages sent to `blink-dev@chromium.org` (six of which were abandoned or withdrawn for various reasons) and assigned codes [13]. Another author verified 33 of the 117 successful deprecation messages (roughly 30% of all messages). We calculated a Cohen's Kappa of 0.81 ("almost perfect") indicating a high degree of inter-rater agreement [14]. We assigned six different categories (Table III).

We find that there are indeed different categories for why developers deprecate web features, and examine in subsequent sections whether these categories should be treated differently.

V. Data Sources, Metrics, and Thresholding

One of our goals was to more accurately measure user and developer pain, so browser developers can make informed decisions about the costs of deprecation and removal. To achieve this goal, we expanded the set of data sources and metrics used in the deprecation decision, created a set of thresholds via a retroactive data analysis, and created a tool to help a Chrome browser engineer in the decision of whether to deprecate a web feature.

A. Data sources & Metrics

In the context of web feature deprecation, we asked ourselves the different ways to measure pain — both *developer* and *user pain* — to get a full picture of the tradeoffs of a deprecation and removal. Understanding how many developers would be affected is important because the developers are the individuals who will need to change their website to stop using the web feature. However, user pain is equally important because it gives a worst case scenario; if no developers followed the deprecation, how many users would be affected if the feature was removed? Both of these measurements of pain are critically important.

Historically, Chrome deprecations have been driven by two data sources: User Metrics Analysis (UMA) and the HTTP Archive. UMA is a Chrome internal data source that provides aggregated usage metrics and powers `chromestatus.com`. The HTTP Archive is a public dataset that tracks how the top 500K sites are implemented, which includes whether the website uses a specific web feature. For deprecations, the key metric that UMA provides is pageloads: for example, a user visiting `google.com` counts as one pageload, and every change in URL counts as an additional pageload. Chrome Web Feature deprecation guidelines state that the percentage

TABLE III
EACH CATEGORY OF WEB FEATURE DEPRECATION, HOW MANY FEATURES FALL INTO THAT CATEGORY, AND AN EXAMPLE FEATURE.

Category	Definition	Number	Example Feature
Security	Feature was deprecated because of security flaw or concern	25	NotificationInsecureOrigin
Never Standardized	Experimental web feature was never standardized	19	WebDatabaseCreateDropFTS3Table
Updated Standard	Feature or attribute was deprecated because the standard was updated	12	CredentialManagerGet
Removed from Standard	Feature was removed from the standard	23	MediaStreamEnded
Inconsistent Implementation	Buggy or inconsistent implementation of the feature spec	12	HTMLEmbedElementLegacyCall
Clean Experience	Removing the feature provides a less confusing experience for web developers	26	ElementCreateShadowRoot
Total		117	

TABLE IV
THE DATA SOURCES WE EXAMINED AND WHAT THEY PROVIDE IN THE DEPRECATION PROCESS.

Data Source	What It Provides	Motivation	Origin
User Metrics Analysis (UMA)	Chrome metrics such as percentage of pageloads and percentage of clients per feature	Browser perspective	Internal, public on chromestatus.com
URL Keyed Metrics (UKM)	Chrome metrics such as unique percentage of domains and percentage of clients per feature	Browser perspective	Internal, backs Chrome User Experience Report
HTTP Archive	Number of domains among the most popular 500K sites that have a web feature	Feature usage on websites	Public
CanIUse	Whether a browser supports a web feature and what version support went into effect	Current and historical usage across browsers	Public

of UMA pageloads is the guiding metric for web feature deprecation, with the number of domains from the HTTP Archive as a backup. In addition to these two metrics, checking cross-browser compatibility is encouraged to keep the web ecosystem consistent. Generally, if a feature falls below 0.03% of UMA pageloads, then the feature is eligible for deprecation; if not and if the feature is on fewer than 0.02% of domains from the HTTP Archive, it may be possible to deprecate.

UMA pageloads help quantify user pain (how often users encounter a web feature), while the HTTP Archive quantifies developer pain (how often sites use a web feature). However, these metrics do not indicate the full extent of user pain or provide avenues for exploration. For example, these metrics combined **will** be able to tell you if the high pageload count is due to a few popular websites; from here a browser or web feature developer can go and contact these popular websites. However, if there is a high pageload count and very few sites in the HTTP Archive have this web feature, then there is not much opportunity for actionable outreach.

To broaden our metrics, we started by determining which additional datasets are useful in providing a more representative set of metrics. In our case, choosing the data sources was a mix of experimentation to see what metrics they can provide and how the metrics fit together across data sources. We identified nine datasets from a combination of our own exploration and asking developers who work in the web platform space. The seven additional sources were a Chrome internal dataset called UKM, CanIUse, WPT.FYI, Confluence, the Microsoft API Catalog, Microsoft CSS Usage, and MDN web docs [15]–[19].

We then selected sources based on three criteria: 1) what metrics it provides, 2) how those metrics compare to the other data sources available, and 3) whether it is straightforward to parse and engineer into a tool for web feature deprecation analysis. For example, we chose the Chrome internal URL Keyed Metrics (UKM) data set (that backs the Chrome User Experience Report) as a data source because it provides insights into the number of domains from the perspective of a browser, which is a unique perspective that would be hard to find elsewhere [20]. We also wanted to use a source that

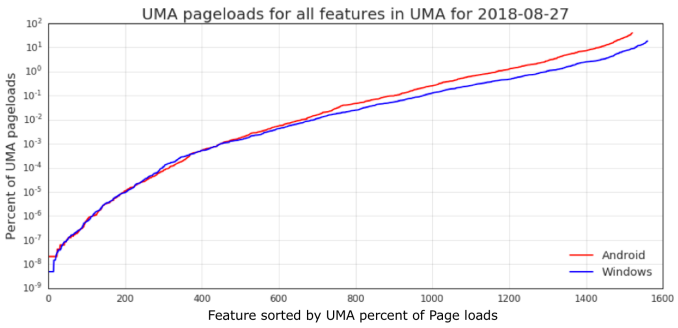

Fig. 1. We plot the percentage of UMA pageloads for every web feature that has an UMA counter. We find no natural delineation in the data.

would provide cross-browser usage information and decided on CanIUse because it listed the version where that feature was first implemented. We also decided to continue using UMA and HTTP Archive; these four data sources are described in Table IV.

These four data sources provide a series of metrics, presented in Table V, that show a more comprehensive picture of potential user pain, while also providing avenues for outreach. For example, UKM combined with the HTTP Archive number of unique domains indicates whether the feature is more prevalent on popular websites or not and whether a developer must focus their outreach efforts, such as contacting websites, on individual sites or a much larger number of sites. We can now also examine whether a feature is seen primarily in certain countries. We chose metrics that could be analyzed automatically, to reduce the load on the browser engineers working through a deprecation. We do not use metrics such as feature dependence because that would require significant work and knowledge from the developer.

Privacy: In this paper, we use two Chrome internal data sources: UMA and UKM. UMA is enabled if the user has "Automatically send usage statistics and crash reports to Google" setting enabled. In modern Chrome, this is enabled by default for consumer installs, but can be disabled during installation or afterwards via a setting. UKM is enabled when the user has both the "Automatically send usage statistics and crash reports to Google" setting enabled, and the user is syncing all browsing windows. UMA collects full aggregate metrics (such as pageloads) while UKM collects a smaller set of aggregated metrics tagged by domain. Identifying characteristics such as age or occupation are not collected. Both metrics are pseudonymous and tagged by separate, opaque identifiers.

B. Thresholding

Another unclear aspect of the deprecation process, in discussions with browser engineers at Google, was where thresholds should be drawn for successful deprecations. Given this, we focused on determining the thresholds for the different metrics that would indicate a successful deprecation.

We first attempted to find thresholds via a simple data analysis, looking to see if characteristics in the data would

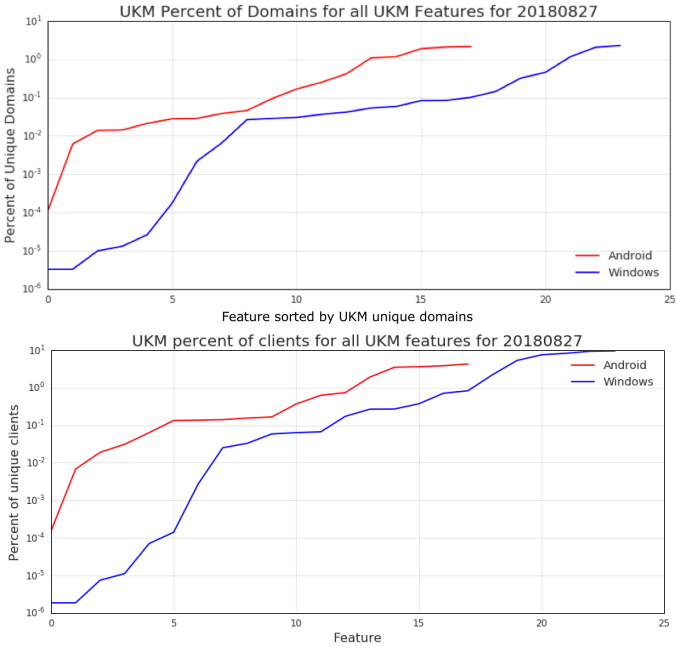

Fig. 2. When plotting all features that have a UKM domain and client counter, we find knees at 0.01%. However, this is not indicative that this is the correct place to draw the thresholds.

point to natural thresholds to use. We graphed all data points for the different datasets, and also between datasets, to see if there would be a natural knee or delineation in the data that we could point to as the threshold for that metric. For many of the graphs we produced, there were no defining characteristics indicating thresholds. For example, in Figure 1 we plot all features that have UMA pageload counters and the percent of total pageloads for each counter. Moreover, when we compared the UMA client percentages to when the feature was implemented in Figure 3, we did not see any trends.

However, this was not the case when we looked at UKM data (Figure 2) where there is a knee in the data (around 0.01%). However, this natural delineation in the data could be due to a variety of factors. While we want to base our thresholds on ground truth, we realized that external factors also play a role. We needed to ensure that the threshold would not cause significant user pain, regardless of natural delineations.

In the end, we decided to analyze past successful deprecations to find thresholds. By analyzing past deprecations, we are taking the stance that we have felt comfortable in the past deprecating X feature, which is a part of Y category, at Z threshold, and so under similar circumstances, the same principles should hold today. There are two caveats to this approach that are worth mentioning. Since past deprecations were based on a mix of intuition and anecdotes our thresholds could end up at exactly the same levels as they were before. We found that exceptions and outliers to every rule and dataset add variance in the thresholds. The second caveat is that this type of analysis requires historical data; we only had historical

TABLE V
FOR EACH METRIC, WE STATE WHAT IT PROVIDES US AND WHO IS AFFECTED. WE OPTIMIZED OUR DATA SELECTION CHOICES TO GIVE US THE WIDEST BREADTH OF ANALYSIS FOR WEB FEATURE USAGE.

Metric	What Metric Tells Us	Who Affected
UMA percent of pageloads	Percent of pageloads for Chrome	End User
UKM percent of unique domains	Percent of domains that Chrome sees this web feature on	Developer
UKM percent of clients	Percent of clients that see this web feature on a domain	End User
UKM Geography	Percent of domains that were loaded in a Chrome version from a specific country	End User & Developer
HTTP Archive number of unique domains	Percent of domains (for the top 500K)	Developer
UKM Google Domains + Clients	Percent of domains and clients for a feature that come from a Google property	Developer
CanIUse cross-browser support	Whether a browser supports a feature and when it started support	End User & Developer

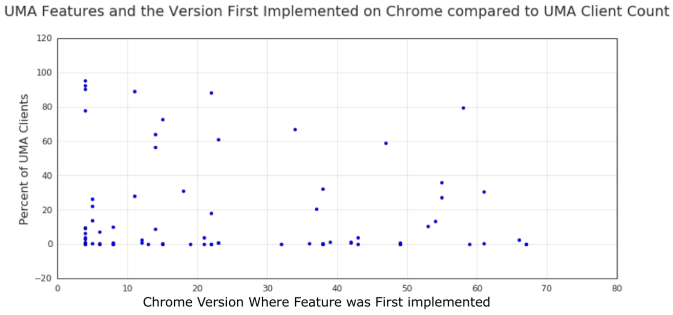

Fig. 3. For each feature that has an UMA Clients counter and an entry in CanIUse, we plot the the version it was implemented with the current UMA Clients counter. We find no trends when comparing these two datasets.

data for UMA pageloads and HTTP Archive domains, so we were restricted to setting retroactive thresholds to just these two metrics (Table VI).

Our retroactive analysis exposed variance in each category, and each platform within the category. Our initial goal was to develop a formula for determining thresholds that would generalize across the different categories and platforms easily. To collect the data, we took every feature that had been deprecated in the past 2.5 years and labeled its category and feature use counter name (the name with which we could look it up in UMA and HTTP Archive). For each feature, we pulled the date that it was sent to the `blink-dev@chromium.org` mailing list and analyzed the UMA pageload percentage and HTTP archive count for the month before, the month of, and the two months following the date the email message was sent to get more data points.

Frequently, feature use counters are not implemented when a feature is implemented but instead only once the feature is considered for deprecation — sometimes because developers were told to do so after sending the deprecation email. This behavior is why we focused our analysis on the few months surrounding the deprecation email date. We did not want to look too far past the month the deprecation was announced because we expect the use of the feature to drop. When choosing which month to analyze, we chose the month that had the most data points. If there were two months that had the same number of data points, we picked the month closest to when the deprecation was announced. All of the thresholds, as well as what percentile they represent, can be found in Table VI.

Determining thresholds for the HTTP Archive was much more straightforward than for UMA for two reasons: 1) the HTTP Archive is not utilized as frequently as UMA in previous deprecations and thus is not enforced as strictly, and 2) we had much fewer use counters for the HTTP Archive. Since thresholding for HTTP Archive counts is less strictly enforced than the UMA thresholding, we did not think it would be helpful to find outliers. Moreover, we wanted to pick a percentile that was representative of a high percentile of successful deprecations, but still somewhat conservative. With these constraints in mind, we picked the 90th percentile of successful deprecation feature counts.

Determining thresholds for UMA pageloads was a much more involved process, mainly due to the prevalence of UMA in many deprecation decisions. Since UMA pageloads thresholds have been more strictly enforced in the past, more data points exist, which makes it easier to identify outliers. We calculated the percentage of data points that were not outliers and picked that percentile as the threshold for the category/platform tuple. For example, the “Inconsistent Implementation” category for Android has two outliers; since there were 9 data points in total, 7/9 is roughly 80%, so we picked the 80th percentile of that category/tuple pairing. If there were no outliers, then we picked the 95th percentile; we were more comfortable picking a higher percentile for the UMA pageloads since there is more data and a more extensive

TABLE VI
THRESHOLDS FOR THE DIFFERENT CATEGORIES AND PLATFORMS.

Category	UMA Windows	HTTP Archive Windows	UMA Android	HTTP Archive Android
Security	0.04% (85th)	0.01%	0.08% (75th)	0.12%
Updated Spec/Standard	0.002% (95th)	0.01%	0.003% (95th)	0.01%
Never Standardized	0.02% (75th)	0.01%	0.01% (75th)	0.01%
Clean Experience	0.13% (90th)	0.13%	0.15% (90th)	0.22%
Inconsistent Implementation	0.02% (90th)	0.23%	0.07% (80th)	0.14%
Removed from Spec/Standard	0.02% (95th)	0.08%	0.04% (95th)	0.05%

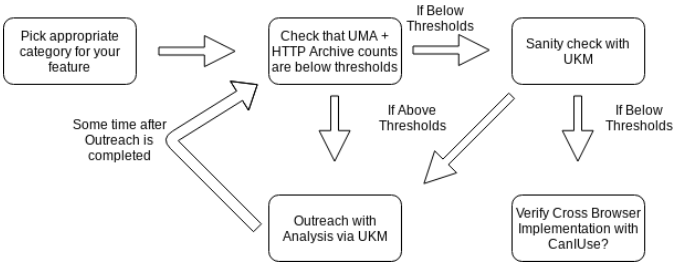

Fig. 4. Flow chart of guidelines for web feature deprecation.

methodology associated with UMA pageloads.

It is worth noting some examples of outliers and why they were still deprecated, which also explains why we examine platforms independently. We chose to examine platforms independently after reviewing examples of deprecation outliers. For example, two Android features with implementation inconsistencies were allowed to be deprecated for very different reasons. The first feature was deprecated with an UMA Android pageloads count of 0.13% two months after the "Intent to Deprecate" email message was sent because the main properties using this feature were Google properties and the developer reached out to these properties to convince them to stop utilizing this web feature. The developer was able to determine this situation by analyzing UKM data, which we now recommend as a mandatory step in the deprecation process.

The second Android web feature was deprecated because the API owners — the browser developers who moderate the `blink-dev@chromium.org` mailing list and give approval or rejection for various Intents — looked at the combined platform pageloads count rather than each platform individually. While the combined pageload count for all platforms fell well below the 0.03% threshold, the Android UMA pageload count was at 0.23% two months after the deprecation email was sent. This outlier, as well as previous work that shows users utilize their devices differently, served as further evidence that platforms, in addition to different categories, should be analyzed separately [21].

VI. SHOULD I DEPRECATE?

Providing metrics and corresponding thresholds is more helpful when accompanied by a set of guidelines to help drive decisions Figure 4 illustrates the guidelines we provided to Chrome developers. A Chrome developer seeking to deprecate a feature must first determine which category the feature falls into in order to use the threshold for that category. Next, the developer should check that the UMA pageloads and HTTP Archive percentages fall below the threshold for the category. We present these analyses first because these two data sources are something that developers in the web deprecation space are familiar with, which makes them easier to interpret, and because we have historical trends to draw conclusions from. If the counts are not low enough to pass these boundaries, then the developer will use the various analyses provided via UKM to identify key sites to reach out to about use and deprecation to lower use of a web feature.

Once the thresholds for UMA + HTTP archive are met, the next step is to check the UKM domain and client counts and verify that they are not too high. In the diagram, we label this step as "sanity check" because we do not have thresholds to recommend based on retroactive data. The final step is to check cross-browser usage with CanIUse, if it is available. Being compliant with other web browsers keeps the web platform consistent and reduces developer pain. This check can also ignite a discussion with other browsers about differences that may lead to more insight about a certain feature and its role in the web ecosystem.

A. Tooling to assist with deprecation decision

The final metrics and guidelines flow were incorporated into a tool to give a comprehensive view of how a web feature deprecation affects developers and users. The tool is based on an internal pipeline to help a Chrome developer through the process of gathering and analyzing metrics. The purpose of the tool is to make following the guidelines as easy as possible by automating any analysis needed.

Table VII shows an example analysis of the web feature MixedContentVideo. "Mixed content" is when a container page is served over valid HTTPS, but has resources, such as a video, that are not served over valid HTTPS. Hence, MixedContentVideo is a marker for "HTTP videos served on HTTPS sites". Figure 5 shows tool output for this feature for UMA pageloads on Android and HTTP Archive URLs, and Figure 6 shows the tool output from UKM.

It is not expected that every dataset will have data for all specific features since the datasets are curated and handled by different organizations. If there is a dataset that does not have a metric, we expect that the developer using the tool

TABLE VII
AN EXAMPLE OF WHAT OUR TOOLING PROVIDES FOR THE MIXEDCONTENTVIDEO WEB FEATURE.

Metric	Metric Count for MixedContentVideo
UMA pageloads	0.0421%
UKM percent of unique domains	0.0935% and Figure 6
UKM percent of clients	0.1368% and Figure 6
UKM Geography	Feature seen at 1% above the average of UKM domains for MY (Malaysia); avg. of all features seen in this country 1.24%, avg. of specified feature 2.25%
HTTP Archive number of unique domains	0.0426%
UKM Google Domains + Clients	Domains: <2% and Clients: <5%
CanIUse Cross browser support	Not available in CanIUse
Growth of Feature over time via UMA pageloads and HTTP Archive number of domains	Figure 5
Comparing UKM domains to HTTP Archive domains	This feature is a more general feature based on the UKM number (0.09%) and the HTTP Archive number (0.04%)

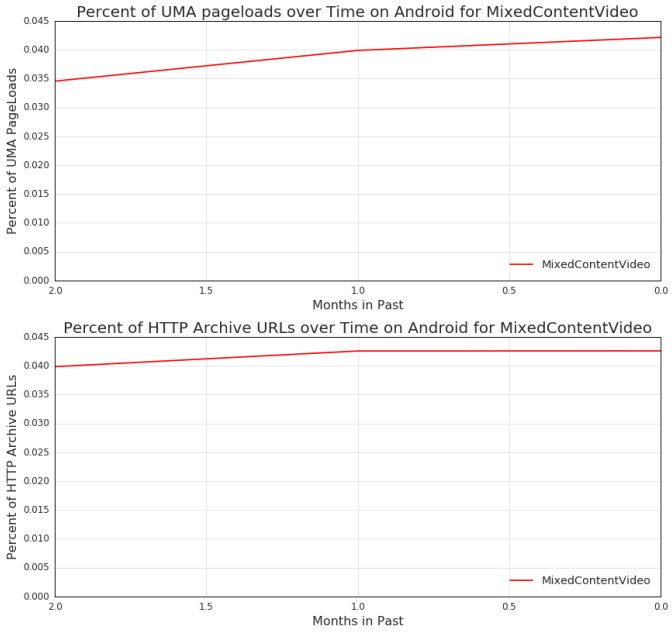

Fig. 5. We can easily examine growth over time with the tool created.

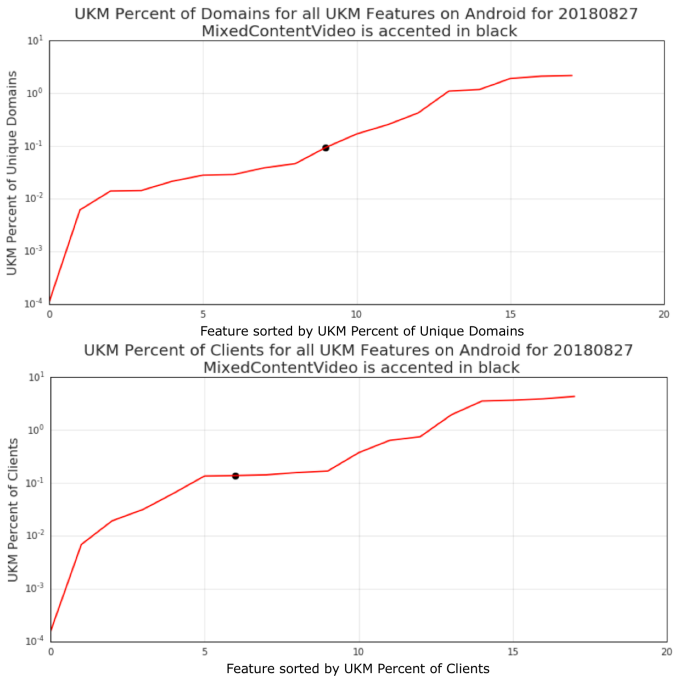

Fig. 6. We can examine the percentage of clients and percentage of domains from a browser's perspective now and plot it graphically to give an idea of where the specified feature falls compared to other features.

will either implement the use counter, where applicable, or collect data by hand. For example, MixedContentVideo does not have an indicator in CanIUse. In this case, we would expect a developer to manually check if there is still support in other popular browsers for mixed content videos.

The guidelines and accompanying tool are being adopted by the Chrome team in charge of web feature deprecations. Our hope is that the guidelines will propel successful deprecations moving forward, and the tool will assist in easily following the guidelines via automation of the necessary analysis.

VII. RELATED WORK

Various aspects of deprecation have been studied, mostly at the language level across various ecosystems. There is a breadth of work around API-breaking changes, but we focus on related work in deprecation specifically since this paper focuses on the deprecation process, not the removal process.

There are a variety of studies that look at the deprecation ecosystem. For example, Robbes et al. examined the ripple effect that occurs after an API is deprecated in the Squeak/Pharo ecosystem [22]. They found instances where a deprecation can have an effect on a software engineering project, but the projects themselves might not react to the deprecation. They also found deprecations are not always helpful, with missing or minimal messages. Espinha et al. extended this work by analyzing how deprecation affects programs using deprecated web features and also if deprecation methods for popular web APIs have any standards or best practices [23]. They found that clients are often faced with distress when dealing with a web API deprecation — and as a result recommend guidelines

on how to make deprecation a smoother process — and find that even popular platforms often do not follow standards.

There are also studies that analyzed deprecation ecosystem effects, but at a much larger scale. Sawant et al. continued Robbes et al.'s study by mimicking the design setup; whereas Robbes looked at one Java API, Sawant looked at twenty-five thousand clients that used a total of five Java APIs [24]. Sawant et al. found that ripple effects are present and very impactful, but only a small fraction of projects will react to a deprecation, opting to delete the call instead of replacing it with the recommended change. Zhou et al. examined how deprecation has been used by analyzing 26 open-sourced Java frameworks and libraries, and created a tool that indicates when a code snippet uses a deprecated API [25]. They find that deprecation methods in the examined frameworks are sporadic and treated inconsistently in both an individual system and across systems.

Also important is the variety of studies around deprecation signage. Ko et al. examined 260 deprecated APIs of eight Java libraries and the corresponding documentation and found that, while 61% of deprecations provide replacement APIs, rationales and concrete examples are rarely ever given [26]. However, for APIs that provided alternate calls, 62% of usages resolved that deprecated API, while only 49% of deprecated calls were replaced when there was no alternate calls produced. Raemaekers et al. examined the policy of deprecation in the Maven Central project, specifically analyzing deprecation tags and indicators, and found that some features with a deprecation tag are never removed, while features without deprecation tags are removed; in short, deprecation tags are used inconsistently [27]. Kapur et al. found a similar result in their study, where they examined HTML-Unit, JDOM, and log4j and their library migrations, and in the process discovered that deprecated entities are not always deleted and, perhaps more harmfully, deleted entities are not always deprecated [28]. Brito et al. examined over 600 Java communities and their deprecation policies and found that while over 60% of all deprecations linked to a replacement API, there is no significant effort to improve deprecation warnings over time, and communities that deprecated APIs in a consistent manner are significantly different than those that do not [29].

Most recently, Sawant et al. analyzed why language deprecations occur and how developers expect clients to respond to them [30]. To address their shortcomings, the authors also propose new steps to take, such as including removal dates, severity indicators, and generic warning mechanisms.

Web feature usage has been less studied. Snyder et al. previously studied the prevalence of certain web features and related metrics, such as age of a browser feature compared to its usage, as well as how many of these features would be blocked by anti-ad and anti-tracking extensions [31].

VIII. Discussion and Future Work

User pain is universal across domains. As such, an interesting question is whether this framework — determining different categories, analyzing metrics, and setting thresholds and guidelines — can generalize to other domains that also handle feature deprecations. For example, if a developer or researcher was developing a process for language-level deprecation, they could start by categorizing why previously deprecated features for that language had been deprecated. To determine the amount of developer pain, one could use API usage counts gleaned from analyzing Github and `publicwww.com`, as well as the popularity of Github projects that use those features as a proxy for end-user pain. Finally, one could retroactively analyze available data on successful deprecations to help determine thresholds, and use the retroactive data as well as any other non-historical data that is available to them to determine a set of guidelines.

Of course, the web and language domains differ. For example, language deprecations are less immediately detrimental than browser deprecations. Github projects may simply stay on an older API, while a web feature deprecation can have an immediate effect on websites. Applying this methodology elsewhere remains an open area of future work.

We could also extend this deprecation analysis to web protocols that do not fit the definition of "web feature". Transport Layer Security (TLS) is a network protocol that provides confidentiality, authenticity, and integrity. Although TLS deprecations are not web feature deprecations, they still have a significant effect on the web platform ecosystem. When a TLS protocol or cipher suite is deprecated, or a Certificate Authority is untrusted, the ripple effects can be huge. An open question is how deprecations in TLS-related features compare and contrast with deprecations of web features.

IX. Conclusion

In this paper we analyzed web feature deprecation via the lens of the Chrome browser. We categorized 2.5 years worth of deprecations in Chrome and found six reasons why a developer would want to deprecate a web feature. We expanded the set of metrics to measure user and developer pain caused by a web feature deprecation, and used historical data to set thresholds and create a set of guidelines for deprecation. Finally, we created a tool for browser engineers to determine whether a web feature meets these guidelines. The new guidelines and tool are in the process of being adopted by the Chrome team handling web feature deprecations.

Acknowledgment

The authors would like to thank everyone at Google who gave feedback on the design and/or paper, particularly Rick Byers and Robert Kaplow. We would also like to thank the Chrome Metrics team for their support.

References

[1] Adi Robertson, "Google is delaying a Chrome audio update that broke countless web games," https://www.theverge.com/2018/5/15/17358752/google-chrome-66-web-audio-api-sound-broken-game-change-delay-apology, accessed: 2018-09-24.

[2] Gregg Kezier, "Top web browsers 2018: Chrome edges toward supermajority share," https://www.computerworld.com/article/3199425/web-browsers/top-web-browsers-2018-chrome-edges-toward-supermajority-share.html, accessed: 2018-09-24.

[3] Chromium, "The Chromium Projects," https://www.chromium.org/, accessed: 2018-09-24.

[4] "BlinkDev Mailing List," https://groups.google.com/a/chromium.org/forum/#!forum/blink-dev, accessed: 2018-09-24.

[5] "Chrome Web Platform Changes Process," https://www.chromium.org/blink#TOC-Web-Platform-Changes:-Process, accessed: 2018-09-24.

[6] "Chrome Breaking Changes Process," https://www.chromium.org/blink/removing-features, accessed: 2018-09-24.

[7] "Blink Principles of Web Compatibility," https://docs.google.com/document/d/1RC-pBBvsazYfCNNUSkPqAVpSpNJ96U8trhNkfV0v9fk/edit#, accessed: 2018-09-24.

[8] Patrick Klepek, "Google's Attempt at Fixing Autoplay Videos Has Broken Countless Games," https://waypoint.vice.com/en_us/article/xwmqdk/googles-attempt-at-fixing-autoplay-videos-has-broken-countless-games, accessed: 2018-09-24.

[9] "crbug report: Valid use cases for autocomplete=off," https://bugs.chromium.org/p/chromium/issues/detail?id=587466, accessed: 2018-09-24.

[10] "crbug report: autocomplete=off is ignored on non-login INPUT elements," https://bugs.chromium.org/p/chromium/issues/detail?id=468153#c164, accessed: 2018-09-24.

[11] "WHATWG Autofill Specification," https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#autofill, accessed: 2018-09-24.

[12] "Chrome Breaks the Web," https://news.ycombinator.com/item?id=15634609, accessed: 2018-09-24.

[13] "Categorization of Blink Intents," https://docs.google.com/spreadsheets/d/1ICQd0Fg8t8AAmZsmyu-O5ZQmEpeWYM-3ZWSW-AiHBTs/edit, accessed: 2019-01-29.

[14] A. J. Viera and J. M. Garrett, "Understanding interobserver agreement: The kappa statistic," *Family Medicine*, vol. 37, no. 5, pp. 360–363, 5 2005.

[15] "WPT.FYI," https://wpt.fyi, accessed: 2019-01-28.

[16] "Web API Confluence," https://web-confluence.appspot.com, accessed: 2019-01-28.

[17] "Microsoft API Catalog," https://developer.microsoft.com/en-us/microsoft-edge/platform/catalog/, accessed: 2019-01-28.

[18] "Microsoft CSS Usage," https://developer.microsoft.com/en-us/microsoft-edge/platform/usage/, accessed: 2019-01-28.

[19] "MDN," https://developer.mozilla.org/en-US/, accessed: 2019-01-28.

[20] "Chrome User Experience Report," https://developers.google.com/web/tools/chrome-user-experience-report/, accessed: 2018-10-01.

[21] Y. Song, H. Ma, H. Wang, and K. Wang, "Exploring and Exploiting User Search Behavior on Mobile and Tablet Devices to Improve Search Relevance," in *ACM International Conference on World Wide Web (WWW)*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2488388.2488493

[22] R. Robbes, M. Lungu, and D. Röthlisberger, "How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem," in *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2012. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393662

[23] T. Espinha, A. Zaidman, and H.-G. Gross, "Web API growing pains: Stories from client developers and their code," in *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6747228

[24] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the Reaction to Deprecation of 25,357 Clients of 4+1 Popular Java APIs," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7816485

[25] J. Zhou and R. J. Walker, "API Deprecation: A Retrospective Analysis and Detection Method for code Examples on the Web," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950298

[26] D. Ko, K. Ma, S. Park, S. Kim, D. Kim, and Y. L. Traon, "API Document Quality for Resolving Deprecated APIs," in *IEEE Asia-Pacific Software Engineering Conference*, 2014. [Online]. Available: http://ieeexplore.ieee.org/iel7/7090773/7091193/07091210.pdf

[27] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic Versioning Versus Breaking Changes: A Study of the Maven Repository," in *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2014. [Online]. Available: http://dx.doi.org/10.1109/SCAM.2014.30

[28] P. Kapur, B. Cossette, and R. J. Walker, "Refactoring References for Library Migration," in *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2010. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869518

[29] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "Do Developers Deprecate APIs with Replacement Messages? A Large-Scale Analysis on Java Systems," in *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7476657

[30] A. A. Sawant, M. Aniche, A. van Deursen, and A. Bacchelli, "Understanding Developers' Needs on Deprecation As a Language Feature," in *International Conference on Software Engineering (ICSE)*, 2018. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180170

[31] P. Snyder, L. Ansari, C. Taylor, and C. Kanich, "Browser Feature Usage on the Modern Web," in *ACM Internet Measurement Conference (IMC)*, 2016. [Online]. Available: http://doi.acm.org/10.1145/2987443.2987466