Time Does Not Heal All Wounds: A Longitudinal Analysis of Security-Mechanism Support in Mobile Browsers

Meng Luo
Stony Brook University
meluo@cs.stonybrook.edu

Pierre Laperdrix
Stony Brook University
plaperdrix@cs.stonybrook.edu

Nima Honarmand
Stony Brook University
nhonarmand@cs.stonybrook.edu

Nick Nikiforakis
Stony Brook University
nick@cs.stonybrook.edu

***Abstract*—Recent market share statistics show that mobile device traffic has overtaken that of traditional desktop computers. Users spend an increasing amount of time on their smartphones and tablets, while the web continues to be the platform of choice for delivering new applications to users. In this environment, it is necessary for web applications to utilize all the tools at their disposal to protect mobile users against popular web application attacks. In this paper, we perform the first study of the support of popular web-application security mechanisms (such as the Content-Security Policy, HTTP Strict Transport Security, and Referrer Policy) across mobile browsers. We design 395 individual tests covering 8 different security mechanisms, and utilize them to evaluate the security-mechanism support in the 20 most popular browser families on Android. Moreover, by collecting and testing browser versions from the last seven years, we evaluate a total of 351 unique browser versions against the aforementioned tests, collecting more than 138K test results.**

By analyzing these results, we find that, although mobile browsers generally support more security mechanisms over time, not all browsers evolve in the same way. We discover popular browsers, with millions of downloads, which do not support the majority of the tested mechanisms, and identify design choices, followed by the majority of browsers, which leave hundreds of popular websites open to clickjacking attacks. Moreover, we discover the presence of multi-year vulnerability windows between the time when popular websites start utilizing a security mechanism and when mobile browsers enforce it. Our findings highlight the need for continuous security testing of mobile web browsers, as well as server-side frameworks which can adapt to the level of security that each browser can guarantee.

I. Introduction

The web continues to be the platform of choice for delivering applications to users. The ever-increasing capabilities of modern web browsers and offloading of computation and storage to the cloud allow the development of powerful web applications ranging from banking, office automation, and word processors to spreadsheets, photo editors, and peer-to-peer video conferencing.

Web applications have historically been vulnerable to a wide range of client-side attacks, including Cross-Site Scripting (XSS) [31], Cross-Site Request Forgery (CSRF) [10], SSL stripping [25], and clickjacking [1, 34]. To help defend against these attacks, browser vendors, from early on, started adding support for security mechanisms to protect the users of vulnerable web applications. These mechanisms include simple access control flags for HTTP cookies (e.g., `HttpOnly` and `secure` flags making cookies inaccessible to JavaScript and non-HTTPS content [9, 50]) as well as complicated whitelist-based mechanisms such as the Content Security Policy through which websites can denote the allowed sources of remote resources [27]. All of these security mechanisms are requested by web applications, typically through HTTP response headers, and enforced by web browsers. Previous research has quantified the adoption of these mechanisms by popular websites, finding that, in general, the adoption of security mechanisms and their configuration complexity are inversely correlated [20, 26, 49].

In this paper, we analyze the support of these mechanisms in mobile browsers in order to understand whether mobile browsers are capable of properly enforcing them. We focus on mobile browsers for the following two reasons: First, unlike desktop environments, there exist hundreds of families of mobile browsers—each advertising a unique set of features, such as increased performance, voice control, and built-in anti-tracking capabilities [22]. All of these browsers are downloaded millions of times and there is currently no quantification of their security-mechanism support. As such, two users browsing the same website at the same time may have substantially different security guarantees depending on the mobile browsers that they utilize. Second, market research shows that an increasing number of users rely more and more on mobile devices for their daily computing needs. A 2017 study of comScore found that users around the world spent the majority of their online time on mobile devices, with users from the U.S. spending 71% of their "digital minutes" on a mobile device [12].

To perform our analysis, we create a set of 395 tests that precisely quantify the support of eight different security mechanisms in browsers, and expose the 20 most popular Android mobile browsers to these tests. Moreover, we perform a longitudinal analysis of this support since 2011, exposing a total of 351 unique browser versions to the 395 tests, thereby performing over 138K tests against mobile browsers.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23149
www.ndss-symposium.org

Through our experiments, we find that although mobile browsers, in general, support more security mechanisms with each passing year, the rate of adoption varies by browser family. For example, Mozilla Firefox's support of security mechanisms is significantly better than the UC Mobile browser even though UC Mobile is downloaded by more users than Firefox. Similarly, we find multiple popular browsers that have not been updated since 2016 and others that, despite their updates, remain vulnerable to the majority of our tests.

We demonstrate the need for thorough testing of security mechanisms by showing that Chrome's decision to not support the `ALLOW-FROM` directive of the `X-Frame-Options` mechanism [34] in favor of the equivalent CSP directive, currently leaves hundreds of popular websites belonging to banks, governments, and telecommunication providers, vulnerable to clickjacking attacks. Unfortunately, this decision also affects the majority of the evaluated web browsers which utilize WebView as their rendering engine.

Next to differences among browser families, we discover that some security mechanisms are adopted significantly faster than others. For example, we find that browser vendors added support for the HTTP Strict Transport Security (HSTS) mechanism [20] much faster than other types of mechanisms, such as CSP and the `SameSite` cookie flag [33] for protecting against CSRF attacks. In fact, by using the Internet Archive [7] to retrieve stored HTTP headers of older versions of websites, we find that there is often a multi-year window of vulnerability between the date when a given mechanism was first requested by a popular website, and the date when at least half of the evaluated mobile browsers supported it.

Finally, we discover that, due to many browsers' reliance on WebView, the support of security mechanisms is inherently tied to the Android version on which a browser is executing. By evaluating the most recent versions of mobile browsers on three popular Android versions, we find that users of the same version of the same browser are experiencing *vastly* different levels of security, something that is both counter-intuitive as well as difficult to account for by web developers.

Overall, our main contributions are the following:

- We systematically collect information about security mechanisms currently recommended for mobile web browsers, and develop 395 tests to evaluate whether a browser correctly implements all security-related directives of each mechanism.
- We conduct a total of 138,645 tests against 351 individual browser versions belonging to 20 different families released in the last seven years, and quantify the differences in security-mechanism support and the implications of the discovered differences for website developers.
- We quantify the window of time between the request of a security mechanism by a popular website and the time when mobile browsers support that mechanism, finding multi-year windows of vulnerability.
- We discover that, for WebView-based browsers, the same browser version can exhibit different security traits depending on the Android version on which it executes.

TABLE I: *Overview of the tested security mechanisms.*

Category	Content	# Tests
Same-Origin Policy	DOM access, cookie scope, XMLHttpRequest and worker	33
Cookie	Secure, HttpOnly and SameSite flag	11
X-Frame-Options header	Deny, SameOrigin and Allow-From values	30
X-Content-Type-Options header	Script sniffing opt-out	1
Iframe sandbox attribute	JavaScript execution, form submission and top-level navigation	3
Content Security Policy	Fetch directives (e.g., script-src) and other directives (e.g. form-action, frame-ancestors and upgrade-insecure-requests)	253
Referrer policy	no-referrer-when-downgrade (default) and other values (e.g., no-referrer, origin, same-origin and strict-origin)	62
Strict-Transport-Security header	Basic and includeSubDomains value	2
Total		395

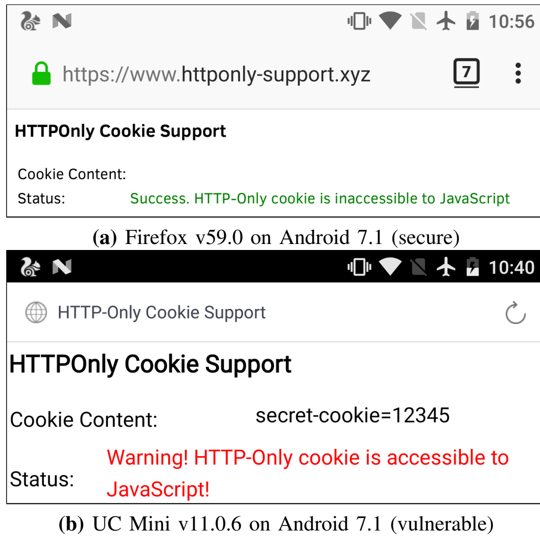
- To tackle this complex security scene, we argue that there is a need for developing a server-side framework that can adapt to a user's mobile environment and employ proper security mechanisms (such as HTTP response headers, HTML tags or JavaScript coding techniques) according to the particular browser and OS version of the mobile device, hiding this complexity from web developers.

II. Background and Security-Mechanism Test Suite

Table I lists the security mechanisms evaluated in this paper. We compiled this list by reviewing prior work which quantified the use of these mechanisms in popular and regional websites [49, 20, 26, 43, 48], as well guides on securing modern web applications [2, 18, 32]. Six out of the eight presented mechanisms are typically activated via HTTP response headers, although most of them can also be utilized through `<meta>` HTML tags. The Same-Origin Policy is, by default, active in all web browsers whereas iframe sandboxing is activated via the setting of the `sandbox` attribute in an `iframe` HTML tag. Note that some of the security mechanisms take boolean values (e.g., the `Secure` and `HttpOnly` cookie attributes) while others, such as the Content Security Policy, allow web developers to author complicated policies.

We made an effort to be as comprehensive as possible regarding the evaluated security mechanisms. We only excluded mechanisms that are currently being deprecated (such as HPKP [21]) and very recent mechanisms (such as the Feature Policy [11]) which were proposed in 2018 and therefore cannot yet be used for a longitudinal study.

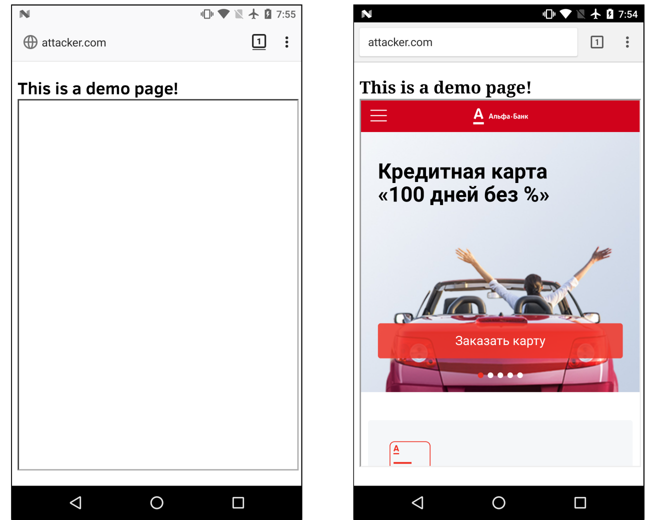
A primary concept common to many security mechanisms is that of a *web origin*. With the exception of browser cookies, a web origin is defined as the triplet of `<protocol, host, port>` [8]. As we discuss below, many security mechanisms



(a) Firefox v59.0 on Android 7.1 (secure)

(b) UC Mini v11.0.6 on Android 7.1 (vulnerable)

Fig. 1: *The UC Mini browser is ignoring the* `HttpOnly` *directive for sensitive cookies.*



(a) Firefox v59.0 on Android 7.1 (secure)

(b) Chrome v65.0.3325 on Android 7.1 (vulnerable)

Fig. 2: *Chrome does not recognize the* `X-Frame-Options: ALLOW-FROM http://webvisor.com` *sent by a Russian bank (*`alfabank.ru`*) and allows the framing of the website by any origin.*

make decisions based on the origin of two parties that are trying to communicate.

A. Same-Origin Policy [#Tests: 33]

The Same-Origin Policy (SOP) is a core security mechanism in modern browsers that isolates interactions between components belonging to different web origins. Given the importance of the SOP, a browser that shows vulnerabilities in its SOP implementation is likely to be showing vulnerabilities in all other security mechanisms.

Similarly to the work by Hothersall-Thomas et al. [19] and Schwenk et al. [35], we develop a number of SOP tests where content from an origin attempts to communicate with other origins in a variety of ways. Examples include a parent page trying to access resources of a cross-origin iframe, or a child iframe accessing a cross-origin parent. Our SOP tests cover interactions under multiple scopes, such as, DOM access, cookies, XMLHttpRequest and web workers.

Moreover, our tests include various types of domain relaxations where, through the appropriate setting of the `document.domain` attribute, two different origins attempt to relax to a common origin and then communicate with each other [28]. Note that, because we are interested in the security implications of not properly supporting SOP, all our test cases (both for SOP as well as for all other mechanisms) involve scenarios that *should be blocked* if the security mechanism is properly supported. Overall, our framework tests a browser's SOP implementation against 33 different tests.

B. Protecting HTTP cookies [#Tests: 11]

Since web applications typically store session identifiers in browser cookies, attackers often seek to steal user cookies to perform session hijacking attacks. Moreover, Cross-Site Request Forgery (CSRF) attacks abuse the ambient authority of an authenticated user's cookies to conduct cross-origin authenticated requests. Given the sensitive nature of cookies, browsers support attributes which make it harder for attackers to steal cookies or weaponize a user's authenticated cookies.

Specifically, by marking a cookie as `Secure`, a browser is instructed to never send that cookie over unencrypted communications, thereby stopping man-in-the-middle attackers from capturing a user's session identifiers. Similarly, cookies marked as `HttpOnly` are never made available to JavaScript code running in a page's origin. As such, even if attackers manage to conduct an XSS attack, sensitive cookies are not available through the `document.cookie` attribute and therefore cannot be exfiltrated. Figure 1 shows screenshots of the Firefox and UC Mini browsers on a page that marks sensitive cookies as `HttpOnly`. Through the experiments described in the rest of this paper, we discovered that UC Mini *ignores* the `HttpOnly` attribute, and makes the cookie available to JavaScript.

To defend against CSRF, web browsers recently started supporting the `SameSite` flag that controls the sending of browser cookies in cross-origin requests. When a web application uses the attribute `samesite=strict`, browsers never send cookies in any cross-domain requests. Alternatively, web applications can use the `samesite=lax` attribute where cookies are sent in cross-origin requests, as long as these requests utilize the GET method and cause a top-level navigation. Our framework utilizes eleven tests to evaluate the support of all three attributes, in different situations.

C. X-Frame-Options Header [#Tests: 30]

In clickjacking attacks, attackers set up malicious pages where benign websites are loaded into transparent iframes and super-imposed over the attackers' websites. By carefully aligning controls between the two websites and taking advantage of the ambient authority of browser cookies, attackers can use clickjacking attacks to convince users to perform malicious or unwanted actions, such as, deleting all their emails or following someone on a social network. To protect against such attacks, websites can utilize the `X-Frame-Options` header

to instruct browsers as to whether they want to be framed and, if so, which websites are allowed to frame them. Specifically, the `X-Frame-Options` header can be set to `DENY` (framing is not allowed), `SAMEORIGIN` (framing is allowed as long as the parent page belongs to the same origin) and `ALLOW-FROM` $\texttt{URL}_1$, $\texttt{URL}_2$, ..., $\texttt{URL}_N$ (framing is only allowed if a website is in the specified whitelist). The support of these values is tested by trying to frame a website in another website that is not permitted in the `X-Frame-Options` header. Figure 2 shows another real example uncovered via our experiments where the Chrome browser does not recognize the `ALLOW-FROM` directive and discards the entire `X-Frame-Options` mechanism, thereby making websites vulnerable to clickjacking attacks.

D. X-Content-Type-Options Header [#Tests: 1]

Some browsers have MIME-sniffing capabilities that enable them to attempt to determine the content type of each downloaded resource. This feature can lead to security problems for servers hosting untrusted content (e.g., a user uploading a malicious HTML page when the server expects the upload of a picture and later abusing that page to conduct session-hijacking attacks). To prevent browsers from MIME-sniffing, thus reducing exposure to such attacks, a web server can send the `X-Content-Type-Options` response header with a value set to `nosniff`. To measure whether `nosniff` is honored by browsers, we test whether a script, when non-script content is expected, has its type determined and executed.

E. Iframe Sandbox Attribute [#Tests: 3]

The `sandbox` attribute allows developers to make use of the least-privilege principle for content loaded inside iframes. Using this attribute, a page can take some capabilities away from the framed content, such as the ability to execute scripts and navigate the top-level webpage. Developers can then selectively enable the capabilities that are absolutely necessary for the framed content. In our tests we measure whether the script execution, form submission, and top-level navigation are blocked, when an iframe is sandboxed.

F. Content Security Policy [#Tests: 253]

The Content Security Policy (CSP) is a mechanism through which websites can instruct browsers to limit the loading of remote resources to those from trusted domains. Although the original goal of CSP was to make XSS attacks harder—by disabling inline scripting and limiting the sources trusted for remote JavaScript code—CSP today supports more than 20 directives controlling, among other things, the loading of scripts (`script-src`), images (`img-src`), stylesheets (`style-src`), forms (`form-src`), and fonts (`font-src`). Moreover, CSP can be used to subsume older standalone headers (e.g., by replacing the anti-clickjacking `X-Frame-Options` header with the `frame-ancestors` CSP directive) and to instruct a browser to block all mixed requests (i.e., HTTP requests originating from an HTTPS page) or automatically upgrade them to HTTPS (`upgrade-insecure-requests`).

Next to the large number of resource-specific directives, an additional complication is that CSP defines a fallback hierarchy for directives [16, 27], as shown in Figure 3. For instance, even though the `default-src` directive is used when a resource-specific directive is missing, certain types of directives, such as `frame-src` (controlling the loading of frames) and `worker-src` (controlling the sources of JavaScript workers) have other intermediate fallback directives which are supposed to be preferred over `default-src`. A browser that does not properly follow this fallback ordering may allow the loading of specific types of resources from domains other than the ones that the developer expected.

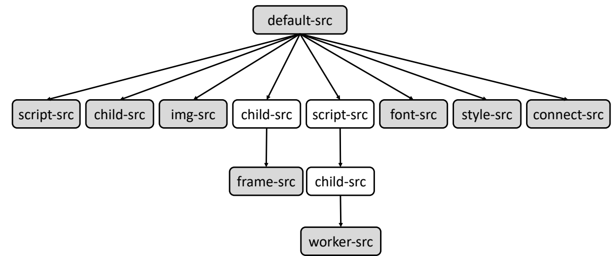


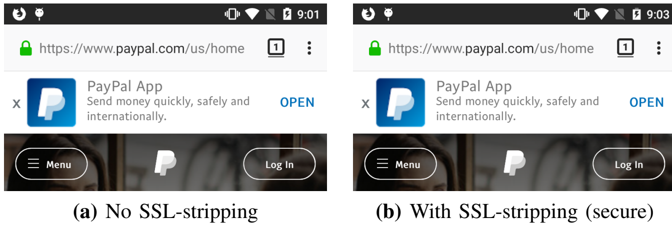
Fig. 3: *The fallback tree of CSP fetch directives.*

Given the large number of directives and fallback combinations, we take a pragmatic approach to limit the number of tests necessary for our evaluation. We start by eliminating directives controlling the loading of resources of a type that are not associated with remote-resource attacks [6, 48, 31, 17, 13], such as the `manifest-src` and `media-src` directives. For the remaining directives, we follow a data-driven strategy where we crawl the main pages of the Alexa top 1-million websites and search for the directives that are most commonly used in real websites. Our final list of directives are the following: `default-src`, `script-src`, `child-src`, `img-src`, `frame-src`, `worker-src`, `font-src`, `style-src`, `connect-src`, `frame-ancestors`, `form-action`, `upgrade-insecure-requests`, `block-all-mixed-content` and `require-sri-for`. Even after this pruning strategy, as Table I shows, our test suite contains an order of magnitude more tests for CSP than for other security mechanisms.

G. Referrer Policy [#Tests: 62]

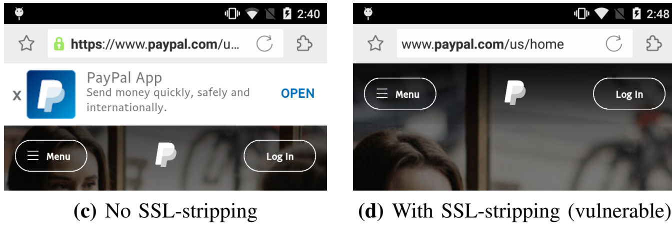
HTTP `Referer` header indicates from which specific web page the current outgoing request originated. For example, if a user clicks on a link B while on webpage A, the full URL of webpage A is then sent to webpage B via the `Referer` header. Even though browsers always omitted the header when transitioning from an HTTPS page to an HTTP one, previous studies have found that the `Referer` header often contains personally identifiable information which could be used to de-anonymize users [38].

To mitigate unwanted privacy leaks, modern browsers allow websites, via the `Referrer-Policy` header, to control when and how a request will contain a Referer header. Among others, websites can set this header to `no-referrer` (completely omit the `Referer` header), `origin-when-cross-origin` (omit the file path from the `Referer` header when sending it to a cross-origin website), and `same-origin` (send the Referer header if requests



(a) No SSL-stripping **(b)** With SSL-stripping (secure)

Firefox v59.0 on Android 5.1.



(c) No SSL-stripping **(d)** With SSL-stripping (vulnerable)

Dolphin v12.0.4 on Android 5.1.

Fig. 4: *The Dolphin browser does not recognize the HSTS header and thus allows MITM attackers to perform successful SSL stripping attacks.*

are to the same origin). Our framework utilizes 62 tests to quantify the extent to which mobile browsers support the various values of `Referrer-Policy` mechanism.

H. HTTP Strict Transport Security [#Tests: 2]

The HTTP Strict-Transport-Security (HSTS) mechanism enables websites to instruct browsers to only access them over HTTPS for a specific period of time. The HSTS header can also include the `includeSubDomains` flag instructing the browser to utilize HTTPS for communications with all subdomains of a given website. The HSTS header was introduced to protect against SSL-stripping attacks where a man-in-the-middle could strip away a website's HTTP-to-HTTPS redirection messages and exfiltrate sensitive user information [25]. By setting the HSTS header with and without the `includeSubDomains` option and attempting to load the website and one of its subdomains over HTTP, we test how well the HSTS mechanism is supported.

Figure 4 shows an example of accessing `paypal.com` (which utilizes the HSTS mechanism), first in the absence of an attacker and then in the presence of a MITM attacker conducting an SSL stripping attack. While Firefox correctly handles both scenarios, the Dolphin browser does not recognize the HSTS header and therefore attempts to communicate with `paypal.com` over HTTP, giving an opportunity to the MITM attacker to conduct a successful SSL stripping attack.

To solve the Trust On First Use (TOFU) problem of HSTS, modern browsers preload HSTS headers for websites that support it. During preliminary experimentation, we discovered that, given our goal of performing a browser-agnostic, longitudinal analysis of mobile browsers, we could not evaluate preloaded certificates for older browsers since it was not clear whether an older version of the browser would include the most recent list of HSTS preloaded websites. As such, our current framework tests the support of HSTS and the `includeSubDomains` flag, but not whether the headers are correctly preloaded.

I. Test Generation and Verification

Given that writing test cases for hundreds of security mechanisms is time consuming and error prone, we implemented a templating system to help with test generation. For each security mechanism, we created a set of template files that enable us to easily tweak different parameters. For example, for the X-Frame-Options category, we can modify the content of the header (Deny, SameOrigin or Allow-From) but we can also change the URL of the frame (both the schema and the domains) and its type (iframe, object, embed). Using these parameters, we can comprehensively assess the support for a security mechanism including its mainstream usage as well as potential corner cases. For X-Frame-Options, combining all these parameters lead to 30 tests for this category. Other categories, such as CSP, present more complex combinations that require finer-grained templates.

After the generation process, we used a desktop browser and a custom browser extension to verify that the tests worked properly. By using the extension to selectively enable and disable specific security mechanisms in the browser, we were able to verify that the tests indeed failed (i.e., marked as "vulnerable") when the security feature was disabled, and passed otherwise. Finally, before launching large-scale measurements, we sampled 60 different mobile browsers—both old and new versions—directly on mobile devices to gauge whether our test pages were behaving as expected on these devices. We will release all 395 generated tests that evaluate the support of the security mechanisms listed in Table I, in the near future.

III. Automated Vulnerability Testing

In this section, we first describe our methodology for the selection of the evaluated mobile browsers, and then describe the framework utilized to evaluate these browsers against the 395 tests presented in Section II.

A. Mobile Browser Dataset

Unlike desktop browsers, mobile app markets house hundreds of different mobile browsers with each browser advertising a set of unique features, such as voice control, data savings, and built-in anti-tracking capabilities.

Browser family selection: To identify mobile web browsers we first downloaded all app descriptions that contain the word "browser" from the Google Play Store, and then manually filtered out apps that are not web browsers. Because of the large number of tests and the decreasing population of users utilizing less popular browsers, we limit our study to the 20 most popular browser families, based on the number of downloads reported by the Google Play Store. As Table II shows, these browsers range from billions (Google Chrome) to millions (Boat Browser) of installations.

TABLE II: The twenty most-popular mobile browsers studied in this paper.

Rank	Package name	# Installs	# Versions	Oldest	Latest
1	com.android.chrome	1,000,000,000+	22	29.0.1547.72 (2013)	65.0.3325.109 (2018)
2	com.UCMobile.intl	500,000,000+	27	8.4.0 (2012)	12.2.0.1089 (2018)
3	org.mozilla.firefox	100,000,000+	29	9.0 (2011)	59.0 (2018)
4	com.opera.browser	100,000,000+	25	12.1 (2012)	45.1.2246.125351 (2018)
5	com.opera.mini.native	100,000,000+	14	8.0.1739.87973 (2015)	32.0.2254.124407 (2018)
6	com.uc.browser.en	100,000,000+	21	8.1.0 (2012)	11.0.6 (2017)
7	mobi.mgeek.TunnyBrowser	50,000,000+	22	8.8.1 (2012)	12.0.4 (2017)
8	com.ksmobile.cb	50,000,000+	19	3.0.6 (2014)	5.22.11.0008 (2018)
9	com.yandex.browser	50,000,000+	24	1.0.1364.172 (2013)	18.1.1.642 (2018)
10	com.explore.web.browser	10,000,000+	11	2.1.0 (2014)	2.6.3 (2018)
11	com.mx.browser	10,000,000+	29	1.1.4 (2011)	5.2.0.3213 (2018)
12	com.htc.sense.browser	10,000,000+	4	7.0.2511222747 (2015)	7.30.2620152639 (2016)
13	com.asus.browser	10,000,000+	9	1.5.6.150317 (2015)	2.1.2.83_170817 (2018)
14	com.dolphin.browser.express.web	10,000,000+	12	11.2.1 (2014)	11.5.08 (2016)
15	com.baidu.browser.inter	10,000,000+	14	1.0.0.0 (2012)	6.4.0.4 (2016)
16	com.appsverse.photon	10,000,000+	16	1.5 (2012)	5.3 (2016)
17	com.apusapps.browser	10,000,000+	16	1.0.0 (2015)	2.0.5 (2018)
18	com.jiubang.browser	10,000,000+	12	1.06 (2013)	2.17 (2016)
19	org.adblockplus.browser	10,000,000+	7	1.0.0 (2015)	1.3.3 (2017)
20	com.boatbrowser.free	5,000,000+	18	4.0 (2012)	8.7.8 (2016)

Browser version collection: For a longitudinal analysis of the support of security mechanisms, we needed to obtain as many versions as possible for each of the selected mobile browsers. Since the Google Play store only provides the most recent version of each app, we relied on third-party app markets and APK-archiving websites to obtain older versions. To this end, we implemented a range of website-specific crawlers and reverse-engineered the ways utilized by third-party markets for fetching older APKs.

Through this process, we were able to obtain 1,369 unique APKs belonging to the 20 selected browser families. Since we needed to conduct 395 tests per APK (as described in Section II), analyzing all of these APKs would be prohibitively time consuming. Therefore, we chose to analyze four APKs per year for each browser family, filtering our initial set down to 351 unique APKs covering a seven-year period (2011 to 2018). Table II shows the distribution of these 351 unique APKs across the selected browser families. For the browsers with no versions in 2018, we were able to manually verify that this is because the browser vendors have not released any recent updates to their browsers. Note that even with this filtering, we still need to conduct more than 138K tests (395 tests $\times$ 351 APKs) to quantify the evolution of security-mechanism support in mobile web browsers.

Finally, we should point out that testing older versions of mobile browsers does not just provide us with interesting statistics on the evolution of security-mechanism support, but also gives us a window into the current vulnerability of all the users utilizing older devices which are no longer receiving updates from their app stores [40].

Release Time Tagging: To be able to conduct our longitudinal study through the years, we need to be able to not just order the collected APKs by version but to also date them. For this, we rely on the methodology of Luo et al. [22] where the date of each APK is extracted based on the modification time of specific files in the APK package (such as the `.RSA` and `.DSA` files). To account for repackaging that may have occurred by the third-party APK archiving services, we use this method to only extract the release year and discard the exact month and day. Finally, where possible, we cross-validate our results with release dates found online.

B. Testing Framework

Given the large number of individual tests needed for this study (138,645 tests), manual testing is highly impractical and error prone. To automate the testing process, we rely on the Hindsight framework [22]. Hindsight is a dynamic testing framework that can automatically install a mobile browser APK on a mobile device and navigate the browser to a series of pages. The framework takes care of assigning APKs to the appropriate Android version when multiple smartphones are available for testing, bypassing potential splash screens of newly installed browsers, and closing existing open tabs. In the rest of this section, we focus on how we utilized Hindsight to test the security mechanisms described in Section II and refer the reader to the original paper for a detailed description of the Hindsight framework [22].

For each test, three pieces of information are collected to evaluate its success/failure: a screenshot of the rendered page, the web server logs indicating which resources were requested by the browser, and, where necessary, results of running JavaScript code on the browser that are sent to a monitoring web server using AJAX messages. For example, in SOP-related DOM tests, to detect whether a parent frame was able to access a child frame using the `document.location` attribute, the test page attempts the access and sends an AJAX message to the web server to record the success or failure of the test.

These three pieces of information are then used by a test-specific evaluation logic to determine whether a given security mechanism is properly implemented by the browser under a given test. This step uses the web server logs and collected

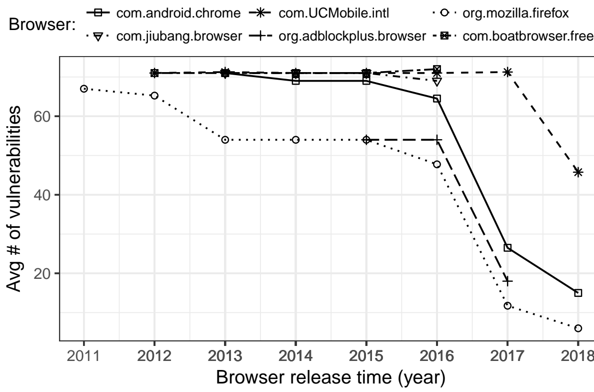


Fig. 5: Vulnerability trend for most- vs. least-popular families (lower is better).

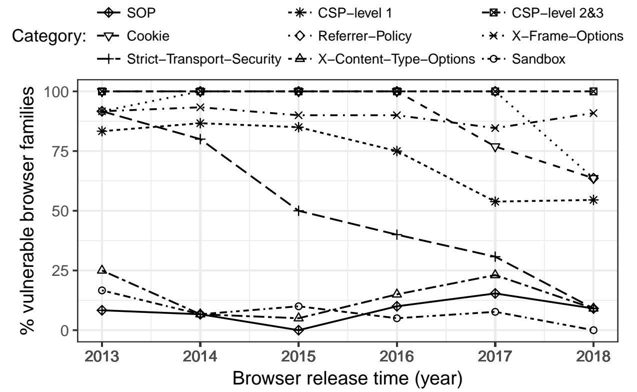


Fig. 6: Vulnerability trends in distinct security mechanisms (lower is better).

AJAX messages to detect cases where the browser requests resources that should have been blocked (according to the the specification of the particular security mechanism being tested). This *log-based* analysis works well for most evaluated browsers.

A class of browsers which require different handling are "proxy-based" browsers such as UC Browser Mini (com.uc.browser.en). These are lightweight browsers that render the page in a remote server and send a compressed version of the rendered page back to the client browser for display. We experimentally discovered that for many proxy-based browsers, the rendering server blindly requests all the resources that are embedded in a web page, before evaluating whether security-based restrictions should be applied to these resources. For example, the remote servers backing a proxy-based browser may request all third-party resources *before* consulting with the website's Content Security Policy which, in regular browsers, would have rejected requests for resources that were not whitelisted.

Therefore, when testing proxy-based browsers, the web server logs might indicate that certain resources, that were eventually blocked, were nevertheless requested. Log-based analysis would incorrectly mark the browser as vulnerable in such cases. To account for this issue, our setup utilizes *OCR analysis* of browser screenshots to detect the rendering of the resources of interest, in addition to the collected webserver logs. We designed the test pages to contain OCR-friendly visual clues (such as certain text, colors, etc.) for each successful test.

A second benefit of the OCR analysis is that it can be used to *validate* the results of the log-based analysis for non-proxy-based browsers. This is particularly helpful in establishing the correctness of the obtained results of our framework since it is unreasonable to manually verify the results of 138K tests. In our experiments, we use the OCR and log-based analysis results to cross-check each other. Only in cases of disagreement between the two is manual validation necessary. Fortunately, such cases became increasingly infrequent after several rounds of debugging, significantly reducing the overhead for verifying the correctness of the reported results.

IV. Longitudinal Analysis of Results

In this section, we report the results of evaluating 351 mobile browser versions belonging to the top 20 Android browser families from 2011 to 2018. Each browser APK is automatically exposed to a total of 395 tests and our experimental setup, as described in Section III-B, is responsible for gauging the success or failure of each test. In this section, we use the terms "secure" and "vulnerable" to denote, respectively, existence or lack of support for any given test.

A. Security Mechanism Adoption Trends

To understand how the support for security mechanisms evolves over time, we aim to answer the following two questions: 1) Do browser families add more support for security mechanisms as time goes by? and 2) Are there differences in the rate of support for different types of security mechanisms?

Vulnerability trends for distinct browser families: To obtain an overall view of the vulnerability trends, we count the average number of vulnerable tests (out of a total 395 tests) for versions of a browser family that are released within the same year, and then compare how the number of vulnerabilities changes over the years.

Figure 5 presents the general vulnerability trends of the three most popular browser families versus the three least popular ones (as listed in Table II). Given the large number of CSP tests that could skew the vulnerability trends towards one specific mechanism, we have excluded CSP support from this figure. At a high-level, one can observe that most browsers become better (i.e., support more security mechanisms) over time. Two notable exceptions, however, are the Next (com.jiubang.browser) and the Boat (com.boatbrowser.free) browsers. These two browsers are among the top 20 popular browsers in Android market and together amass more than 15 million downloads. Both browsers have not been updated since 2016 and are vulnerable to more than 60 tests (excluding their support of CSP).

In terms of popular browsers, even though UC Mobile has been supporting more and more security mechanisms over the years, its latest evaluated version has five times as many vulnerabilities as Firefox, yet it was downloaded by five times as many users. Finally, both Chrome and Firefox have a non-zero number of issues in their latest versions at the time of testing (as listed in Table II). These are primarily due to lack of support for the `ALLOW-FROM` value of the X-Frame-Options header in Chrome, and the SameSite cookie in Firefox. More details on the support of the `ALLOW-FROM` value are provided later in Section VI-A.

TABLE III: Disparity between the earliest time when websites ask for a security mechanism, and when mobile browsers have adopted them. Year of support for desktop browsers is provided for reference.

Security Mechanism	Earliest request	Chrome for Desktop	Firefox for Desktop	First Mobile Browser Support	50% Mobile Browsers Support (Δ)	75% Mobile Browsers Support (Δ)
CSP	2011 (lastpass.com)	2011	2011	2011	2014 (+3)	2015 (+4)
Cookie	<2011 (slate.com)	2011	2009	2011	2013 (+2)	2014 (+3)
Referrer-Policy	2016 (tut.by)	2012	2015	2015	2018 (+3)	Not yet
X-Frame-Options	<2011 (goo.gl)	2010	2010	2011	2013 (+2)	2014 (+3)
HSTS	<2011 (paypal.com)	2010	2011	2011	2015 (+4)	2016 (+5)
X-Content-Type-Options	<2011(fivethirtyeight.com)	2008	2011	2011	2013 (+2)	2015 (+4)

Vulnerability trends for different security mechanisms: To understand the vulnerability trend of various security mechanisms, we consider browsers that are released within the same year and calculate the fraction of vulnerable browser families.

Figure 6 shows the vulnerability rates (the percentage of vulnerable browser families) for each of the evaluated security mechanisms. Here, a browser family is marked as vulnerable with respect to a security mechanism if at least one version released within a given year is vulnerable to tests for that security mechanism. Note that tests for Content Security Policy (CSP) are divided into two subsets according to their levels. This means that CSP directives that are proposed later in CSP levels 2 and 3, such as `form-action` and `frame-ancestors`, are analyzed separately from the ones included in CSP level 1. In addition, since not many mobile browsers were being developed in the years 2011 and 2012 and that would cause biased results, we limit the analysis to the years 2013–2018 where we collected browsers from at least 10 different browser families.

One can again observe several trends. The Same-Origin Policy (SOP), iframe sandboxing, and the X-Content-Type-Options header appear to have always been well supported in mobile browsers, although we find that, in most years, they all have a few failing tests. Contrastingly, other mechanisms—such as the Referrer-Policy, secure cookie attributes, and X-Frame-Options—are still not well supported in the majority of mobile browsers. The only mechanism that started with little support but is currently as widely supported as SOP and iframe sandboxing is the HTTP Strict Transport Security (HSTS) mechanism. We opine that this is because of the general push from browser vendors towards the HTTPS-by-default web and the gravity of SSL-stripping-like attacks which HSTS defends against.

So far, CSP level 1 is properly supported by approximately half of the evaluated browser families, but none of the evaluated browser families properly enforces all tested directives of CSP levels 2 and 3. This demonstrates that it is not only difficult for web developers to author correct CSP policies [26, 37, 48, 49], but it is also challenging for browser vendors to account for all directives as described in the CSP specification.

B. Windows of Vulnerability

Except for the SOP and the iframe sandboxing, all of the security mechanisms we test in this paper need to be activated by website administrators by sending the appropriate HTTP

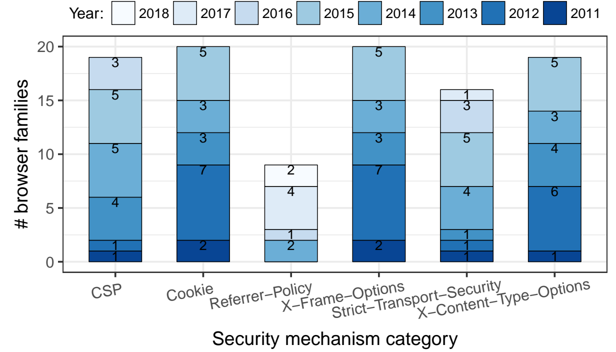


Fig. 7: Adoption rate of security mechanisms over the years.

response headers. In this section, we quantify the window of vulnerability between the time when website administrators first expected a given security mechanism to be present—thereby relying on it to increase the security of their pages—and the time when that mechanism was supported in desktop and mobile browsers.

To quantify the window of vulnerability, we crawled historical snapshots of the Alexa top 5K websites from the Internet Archive [7]. We collected the timestamps for all available snapshots of these websites and, when available, chose one snapshot per month to download. From the resulting 550K archived snapshots, we extracted the captured response headers—available and prefixed by the `X-Archive-Orig-` string, as described by Stock et al. [41]—and located the earliest time when any website first utilized a given security mechanism.

In Table III, we compare the earliest time when any Alexa top 5K website requested a certain security mechanism, and the earliest time when mobile and desktop browsers adopted that specific mechanism. We exclude from this analysis the SOP and iframe sandboxing since they are either assumed or are activated through HTML tags. We choose 2011 as a cutoff year since that is the year when the oldest versions of our evaluated mobile browsers were released.

We found that mobile browsers are significantly slower in adopting security mechanisms compared to desktop browsers which often start supporting mechanisms even before they are standardized. For example, we find that the first mobile browser to support the anti-clickjacking X-Frame-Options mechanism did so in 2011 whereas both desktop Firefox and desktop Chrome supported the mechanism in 2010. At

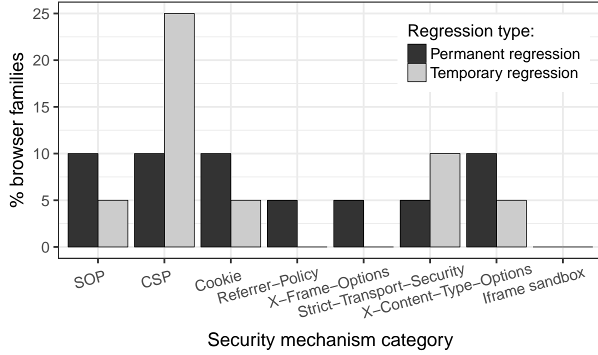


Fig. 8: *Security regressions for distinct security mechanisms.*

TABLE IV: *Security-mechanism regression in top-5 families.*

Mechanism	Chrome	UC	Firefox	Opera	Opera mini
SOP	✗	✓	✗	✗	✗
CSP	✗	✓	✓	✗	✗
Cookie	✗	✗	✗	✗	✗
Referrer-Policy	✗	✗	✗	✗	✗
X-Frame-Options	✗	✗	✗	✗	✓
HSTS	✗	✗	✗	✓	✗
X-Content-Type-Options	✗	✓	✗	✗	✗
Sandbox	✗	✗	✗	✗	✗

the same time, mobile browsers needed an extra three years (2014) before 75% of the evaluated browser families supported this mechanism. The results of Table III demonstrate that, for *all* evaluated security mechanisms, there exist multi-year vulnerability windows between the time websites make use of a given security mechanism and the time mobile browsers actually support it. Finally, we also discovered that Referrer-Policy[1] is the only security mechanism that has not yet been adopted by 75% of the studied mobile browser families at the time of this writing.

Figure 7 shows the adoption rates of distinct security mechanisms in mobile browsers. For this graph, we denote a mechanism as "supported" if any of its tests succeed in a browser. The number in each bar segment represents the number of browser families that started to support a security mechanism in a given year. Among others, we observe that, except for the security-related attributes of cookies and the X-Frame-Options header, the remaining security mechanisms are not completely adopted by all of the 20 browser families we tested. One of the more recent security mechanisms, the Referrer-Policy[1], is currently only adopted by less than half of the browser families with more browsers adopting it in 2017 than they did in 2018.

C. Regressions in Security-Mechanism Support

Web browsers usually increase their support for different security mechanisms over time. For example, Chrome supported a subset of values of Referrer-Policy before fully supporting it. However, there are cases where mobile browsers exhibit regressions. In a regression, a later version of a given browser family supports fewer security mechanisms than a previous version of the same family.

By analyzing the number of vulnerable tests in consecutive versions of the same browser family, we identify two types of security regressions: *temporary regressions* and *permanent regressions*. In a temporary regression, a previously supported mechanism becomes unsupported only to be supported again in a later version. Contrastingly, in a permanent regression, a previously supported mechanism becomes unsupported and stays unsupported for the remainder of the evaluated versions.

According to our study, 55% of the tested browser families show signs of regressions in their support of security mechanisms. Figure 8 shows what percentage of browser families exhibit regressions with regard to different security mechanisms. Unfortunately, temporary and permanent regressions within a browser family appear in all security mechanisms except iframe sandboxing. We observe that most regressions occur when adopting CSP. This makes intuitive sense given the complexity of the CSP mechanism. Fortunately, most CSP regressions are temporary meaning that browser vendors eventually detect the regression and correct it.

Table IV shows whether a particular top-5 popular browser family presents security regressions in terms of distinct security mechanisms. A checked cell means that a browser exhibited a regression while supporting the security mechanism. Our findings suggest that Chrome has the most consistent evolution since it does not show any security regressions as new security mechanisms were added or existing mechanisms were upgraded. UC browser shows frequent security regressions on SOP, CSP and X-Content-Type-Options. The remaining browsers only occasionally show security regressions. For example, the Opera browser stopped supporting HSTS in version 15 and resumed its support in version 26. Given that these versions were released approximately one year apart, this means that users of the Opera browser were vulnerable to MITM attackers for that period, even for websites that correctly utilized the HSTS header.

Fortunately, all security regressions in this table, with the exception of Opera Mini's regression in X-Frame-Options, were temporary and have been fixed in their latest browser versions.

V. Analysis of Current Mobile Browsers

In the previous section, we investigated the security-mechanism support in mobile browsers from 2011 to 2018. This enabled us to quantify differences in level of support across browser families, the mechanisms that are the most/least supported, and the delay between the first usage of a security mechanism in a popular website and the time when this mechanism is actually supported by most mobile browsers.

But this longitudinal analysis is only half of the story. Most users are likely to be utilizing a recent version of a popular browser, either because they consciously update their apps, or because their smartphone is configured to automatically update apps whenever they connect to a wireless network. Therefore, in this section, we focus on the support of security mechanisms in the most recent versions of popular mobile browsers. We pay particular attention to the effects of the underlying Android

[1] no-referrer-when-downgrade value is excluded given its implicit support by all browsers

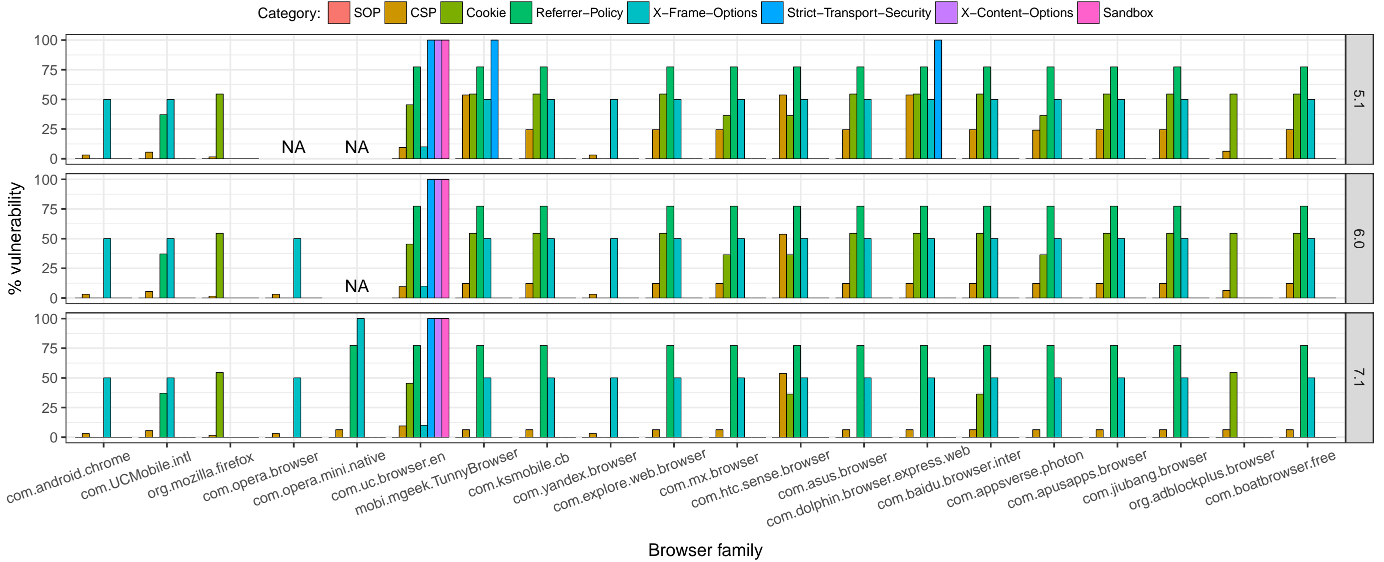


Fig. 9: *Vulnerability of the latest browser versions on the three most popular Android versions.* `NA` *indicates that a given browser could not be installed on a given Android version (i.e., its minimum required version was more recent than the provided one).*

version to the overall security of each mobile browser. Even though one may initially think that these two are independent, because most mobile browsers are built around WebView (Android's component for showing web content in apps), they are, in fact, tightly dependent.

To quantify this dependency between security-mechanism support and Android version, we installed and tested the most recent version of the twenty evaluated mobile browsers on Android Lollipop (version 5.1), Marshmallow (version 6.0) and Nougat (version 7.1). These are the three most popular Android versions in the market, according to statistics from July 2018 [39]. Figure 9 shows the results of this experiment. One can observe that, for the majority of mobile browsers, there is a clear security improvement when they are installed on newer Android versions, since they benefit from more recent WebView implementations.

The corollary from this finding is that two users with the same version of the same browser installed, can be experiencing *vastly* different levels of security. This is counter-intuitive and significantly complicates the life of web developers who cannot assume the presence of certain security mechanisms, based solely on the family and version of the browser being utilized. This is also the reason why, for a few browsers such as the Boat browser, the vulnerability rates shown in Figure 9 differ from the rates of the most recent versions analyzed in Section IV. The Hindsight framework (on which this work relies) utilizes an SDK assignment algorithm which installs a browser APK on a version that is as close as possible to the *target* version, as specified by developers in their manifest files. For these browsers, the target Android version was older than version 5.1 and therefore the browsers would be even more vulnerable than what is shown in Figure 9.

In terms of common vulnerabilities, Figure 9 also reveals that 16 out of the 20 browsers do not support a specific subset of the X-Frame-Options anti-clickjacking mechanism. Specifically, most browsers do not support the `ALLOW-FROM` directive, rendering websites that utilize it vulnerable to clickjacking attacks (we further elaborate on this vulnerability in Section VI). Moreover, through our tests, we discovered that the Opera Mini browser (the fifth most popular browser in Table II, with more than 100 million downloads) in its `High Data Savings` mode, does not support X-Frame-Options at all meaning that, unless a website utilizes the anti-clickjacking, `frame-ancestors` CSP directive, it is vulnerable to clickjacking attacks when rendered via this browser.

Among the most recent versions of the evaluated browsers, UC mini (com.uc.browser.en) is an interesting outlier. It contains vulnerabilities in almost every security mechanism category regardless of the Android version on which it is installed. We investigated older versions of the same browser and noticed that they had the exact same vulnerabilities. Upon further investigation, we realized that the UC mini is a proxy-based browser which, as described in Section III-B, uses dedicated remote servers to fetch web pages that after performing various types of processing, such as down-scaling large images, send the resulting page to mobile devices. Therefore, even though we have obtained older version of UC mini browser, we cannot know how their remote servers processed web pages in prior years. At the same time, given that UC mini has a significant number of vulnerabilities across all categories, we find it doubtful that prior versions of their server-side code would have supported more mechanisms than they do now.

Table V provides more details on the vulnerabilities present in the most recent versions of evaluated browsers as tested on Android 7.1, differentiating between a lack of support (left) versus implementation bugs (right). If all tests of a given mechanism fail, then we mark that mechanism as not being supported by a given browser. In terms of non-adoption, we discovered that even though CSP level 2 was recommended by W3C as early as 2014 [46], it is still not supported by HTC sense (com.htc.sense.browser). The non-adoption of CSP invalidates website policies that aim to stop the exploitation of XSS and clickjacking attacks. Similarly, two browser families are not supporting SameSite cookies thereby making users potentially vulnerable to Cross-Site Re-

TABLE V: Vulnerabilities of the latest browser versions on the highest tested Android version.

	Non-adoption			Implementation bugs		
Mechanism	**Vulnerability**	**#**	**Examples**	**Vulnerability**	**#**	**Examples**
CSP	CSP level 2 and 3.	1	com.htc.sense.browser	Inconsistent fallback sources of `worker-src`.	18	com.android.chrome, com.UCMobile.intl
	`require-sri-for`.	19	com.android.chrome, org.mozilla.firefox	Allowing the loading of `worker-src` resources over an HTTP URL when the equivalent HTTPS URL is whitelisted.	13	com.opera.mini.native, com.ksmobile.cb
	upgrade-insecure-requests.	1	com.uc.browser.en	For some directives (e.g., `script-src`), allowing resources to load from `example.com` when the white-listed source is `*.example.com`.	1	com.htc.sense.browser
Cookie	SameSite cookies.	2	org.adblockplus.browser, org.mozilla.firefox	Under `samesite-strict` mode, cookies are disclosed via POST requests and `samesite-lax` is not supported.	3	com.uc.browser.en, com.htc.sense.browser
				Scripts are allowed to access cookies that are marked as `HttpOnly`.	1	com.uc.browser.en
Referrer-Policy	All directives except no-referrer-when-downgrade.	14	com.opera.mini.native, com.uc.browser.en	–	–	–
	same-origin, strict-origin, or strict-origin-when-cross-origin values.	1	com.UCMobile.intl			
X-Frame-Options	All directives.	1	com.opera.mini.native	Allowing subdomains of a white-listed website (through the `ALLOW-FROM` directive) to frame the website.	1	com.uc.browser.en
	`ALLOW-FROM` directive.	16	com.android.chrome, com.UCMobile.intl			
HSTS	Basic mechanism and with `includeSubDomains`.	1	com.uc.browser.en	–	–	–
X-Content-Type-Options	`nosniff` for scripts.	1	com.uc.browser.en	–	–	–
Sandbox	Script execution, form submission and top-level navigation after sandboxing.	1	com.uc.browser.en	–	–	–

quest Forgery (CSRF) attacks. In terms of privacy-preserving mechanisms, the Referrer-Policy is mostly unsupported by 15 out of the 20 evaluated browser families. Among these browsers, UC Mobile (com.UCMobile.intl) partially supports the Referrer-Policy mechanism but lacks support for important values, such as `same-origin`, `strict-origin`, and `strict-origin-when-cross-origin`. For the rest of the browsers, they merely support the basic Referrer-Policy value (`no-referrer-when-downgrade`), which was enforced by browsers even before a dedicated referrer-controlling mechanism was introduced. Finally, the HTTP Strict-Transport-Security (HSTS) header is utilized to prevent SSL-stripping-like attacks but is not supported by the most recent version of the UC mini (com.uc.browser.en), along with X-Content-Type-Options and iframe sandboxing.

Next to a complete lack of support, it is equally important to uncover previously unknown implementation bugs, shown in the right column on Table V. For certain CSP directives, we discovered vulnerabilities that allowed the loading of content from origins other than the ones listed in the Content Security Policy. We also discovered only partial support for SameSite cookies and implementation errors that allowed access to `HttpOnly` cookies via JavaScript. Finally, UC mini (com.uc.browser.en) is also vulnerable because of a poor implementation of the `ALLOW-FROM` directive of X-Frame-Options. For example, if `wordpress.com` deploys an X-Frame-Options header specifying that `https://wordpress.com` is allowed to frame the page, the browser allows all subdomains of `wordpress.com` to frame the website, including potentially untrusted ones, e.g. `attacker.wordpress.com`. As before, our findings demonstrate that even the most recent versions of mobile browsers executing on the most popular Android version, are not free from issues related to their support of popular security mechanisms.

VI. Vulnerability Case Studies

A. Anti-clickjacking mechanisms

X-Frame-Options is an established security mechanism for preventing clickjacking attacks (Rydstedt et al. already discuss X-Frame-Options in their 2010 study of clickjacking [34]) and yet most browsers still do not fully support it. Through our experiments (Section V), we discovered that Google Chrome, and the vast majority of other browsers which build on WebView, treat the `ALLOW-FROM` directive as a wrong option, and altogether discard the X-Frame-Options header.

To understand the consequences of this decision, we crawled the main pages of the Alexa top 50K websites and discovered that 231 out of the 10,752 websites that make use of the X-Frame-Options mechanism, utilize the `ALLOW-FROM` value which is recognized by Firefox but disregarded by Chrome and other mobile browsers. The majority of these websites (175/231) *do not* utilize the `frame-ancestors` CSP directive, and are therefore vulnerable to clickjacking attacks when they are rendered through most mobile browsers. These websites include American and Russian banks, government websites of US, China, Brazil, and India, cloud instrumentation services, credit-score monitoring services, adult websites, and Spanish and Australian telecommunication companies. All of these websites have user accounts which could be abused through clickjacking attacks to extract personal and financial information. In Section VII, we describe the process of reaching out to them, and our progress at the time of this writing.

An additional complication arises when both CSP's `frame-ancestors` and X-Frame-Options are present. Developers may, by accident, specify a different framing policy in the X-Frame-Options header than the one they specify in the `frame-ancestors` directive. In fact, we did observe conflicting framing policies among Alexa top 50K websites, including some famous shopping and medical websites. Such conflicting specifications coupled with incorrect handling by mobile browsers may result in potential security threats.

To understand the effects of conflicting anti-clickjacking policies, we tested how mobile browsers are handling such conflicts. The CSP standard explicitly states that in the presence of both mechanisms, the browser should only enforce the `frame-ancestors` directive and *ignore* the X-Frame-Options header [47]. We discovered that 15% of the tested mobile browsers (53 out of 351 APKs belonging to 9 different families) show vulnerabilities in how they handle conflicting policies. These vulnerable browsers include popular ones such as Chrome, Opera and Yandex browsers. Fortunately, when testing the most recent browser versions on Android 7.1, only UC mini (com.uc.browser.en) remains vulnerable.

B. SameSite cookies

SameSite cookies is a recent mechanism for preventing CSRF attacks in the browser by disabling the use of cookies in a cross-origin context. Contrary to the X-Frame-Options mechanism, Chrome was the very first browser to support it, as early as May 2016. However, looking at the Alexa top 50K websites, only 93 of them have added the "samesite" option in their cookie headers. This shows a strong disparity in the adoption of different security mechanisms. Even if the biggest browser vendors show their support for a new mechanism, there is no guarantee that websites will adopt it by potentially replacing their existing mechanisms (similar to the expectation that web developers would replace their X-Frame-Options mechanism with the CSP `frame-ancestors` directive).

For the websites that actually do make use of the SameSite cookie attribute (including an Italian bank and the biggest online streaming platform), they may not get the protection they expect from this option since many browsers do not support it. For example, Firefox and the Adblock browser do not support this option in their latest versions. Similarly, none of the WebView-based browsers like KSMobile, Explore, and the Asus Browser support the option on devices with Android 6.0 or older. The only safe option would be to make use of both SameSite cookies *and* hidden nonces in forms, until some indeterminate time in the future, when one could assume that all browsers support this new mechanism.

VII. Discussion

A. Summary of findings

By automatically exposing 351 mobile browser versions—belonging to the 20 most popular Android browser families—to 395 tests, we quantified the support of security mechanisms in their most recent versions, as well as the evolution of such support since 2011. We discovered that, even though most browsers support more security mechanisms over time, the rate of support is not the same across browsers and across security mechanisms (Section IV-A). We also discovered the lack of support in most mobile browsers for specific anti-clickjacking directives, leaving hundreds of popular websites utilizing them vulnerable to attacks, as well as lack of proper support for SameSite cookies and Referrer-Policy.

We used the Internet Archive to quantify the window of vulnerability from when a popular website requests a security mechanism until that mechanism is supported by a sufficiently-large fraction of mobile browsers (Section IV-B). Through that experiment, we observed that there are large multi-year windows of vulnerability for most security mechanisms. This signifies that, with regard to the evaluated mechanisms, the users of mobile browsers are less secure when browsing websites than users of desktop browsers. We quantified the number and types of security regressions (Section IV-C) and identified browsers, such as the Boat and Next Browsers, which have not been updated for over two years and are still utilized by millions of Android users. Moreover, by evaluating the same browser version on different Android versions, we discovered that the users of the same version of the same browser, can be experiencing vastly different levels of security when browsing the web (Section V). Finally, we evaluated the behavior of browsers when conflicting policies for different security mechanisms are requested and observed that many browsers are still deviating from official specifications (Section VI).

B. Ethical Considerations and Vulnerability Reporting

All of the 138K tests conducted against mobile browsers were executed locally against our own copies of these browsers running on dedicated smartphones. As such, real users never came into contact with our tests.

In terms of vulnerability reporting, we have already contacted the owners of 20 different websites which, due to the lack of X-Frame-Options support by Chrome and most WebView-utilizing browsers, are currently vulnerable to clickjacking attacks. We gave priority to government websites, banks and credit unions, and websites that are highly ranked according to Alexa. We received replies from most websites, informing us that they have forwarded our information to the appropriate team and we received confirmation from a US government website and a cloud-automation platform that they would follow our advice and utilize the

CSP `frame-ancestors` directive which is supported by Chrome, WebView, and other modern browsers.

At the same time, we reached out to Google to inquire about the lack of support for the X-Frame-Options `Allow-From` directive in Chrome and WebView. We were told that this was a conscious decision back in 2012, in favor of the more robust CSP `frame-ancestors` directive. Chrome developers informed us that they would be willing to add support for it if we conducted a large-scale measurement to quantify the number of Chrome users affected by it and provide a patch. We also reached out to the developers of the UC Mini browser with a detailed report of all the issues that we uncovered through our experiments. The developers acknowledged our findings and offered us a bug bounty for ethically disclosing them.

C. Implications

Our results *categorically* demonstrate that web developers cannot assume that just because they requested a security mechanism through the appropriate setting of an HTTP header or HTML markup, all browsers receiving this request are capable of correctly enforcing it. Moreover, the fact that the same version of a given browser can be more vulnerable when it runs on an older Android device further complicates the development of secure websites, as the developers now need to account for both the browser as well as the platform on which it executes.

Our findings are reminiscent of client-side JavaScript issues where different browsers support different APIs and thus developers need to either account for popular browsers when writing JavaScript code or utilize a library, such as jQuery, that abstracts these details away. We argue that a similar browser- and platform-aware framework is required for the server side. This framework would be driven by a database of supported security mechanisms for each version of each mobile browser, combined with the most popular platforms on which that browser could be executing.

Such a database could be automatically produced by tools and techniques similar to the one we described in this paper. By detecting the browser family, browser version, and OS of a mobile device, the framework could then proceed to use the appropriate security mechanisms for that combination. For instance, upon detecting a mobile browser that does not support the `ALLOW-FROM` directive, the framework could automatically emit a CSP `frame-ancestor` header, even if the website does not use CSP. Similarly, by detecting that a user is utilizing an older version of the Opera Mini browser before CSP was adopted, this framework could emit frame-busting JavaScript code which checks whether the current website is framed and redirects the top-level page if so [34]. This technique was once popular before X-Frame-Options was standardized and it can still be used as a fallback technique when browsers do not support more recent anti-clickjacking mechanisms. Finally, for browsers with egregious lack of support for security mechanisms (such as UC Mini), this framework can deny serving the website, protecting users from falling victim to attacks and putting pressure on the developers of such browsers to properly support the missing security mechanisms.

D. Mobile browsers on other platforms

We decided to analyze the adoption of security mechanisms in mobile browsers as it is a domain that has been greatly overlooked in the past. App stores offer many different browsers and, as we show in our study, different browsers may provide very different security guarantees. We chose to focus specifically on Android because of the availability of development/dynamic-analysis tools and its dominant market share. We should point out that such a study on the iOS ecosystem, while possible, is likely to be much less informative, since Apple forces app developers to use WebKit as the sole rendering engine that can run on iOS [5]. As such, we expect that all mobile browsers on iOS will exhibit the same support of security mechanisms since they *have* to use the same underlying engine.

VIII. Related Work

To the best of our knowledge, our work is the first systematic and longitudinal study of the adoption of security mechanisms in mobile browsers. In this section, we briefly describe prior work in the areas of security mechanisms for the web and web-related mobile security.

Adoption of Security Mechanisms. Given the importance of security mechanisms, researchers have investigated their use in the wild. In 2010, Zhou and Evans performed a small-scale experiment, measuring the adoption of HTTPOnly cookies and discussing reasons why the adoption was not as high as it could be [50]. Weissbacher et al. performed the first study of CSP adoption finding that CSP was lagging behind other security mechanisms [49]. Kranch and Bonneau performed a similar study for the HSTS security mechanism [20]. Van Goethem et al. quantified the adoption of a large number of security mechanisms on the web, finding a positive correlation between the ranking of a website and its use of security mechanisms [43]. In 2016, Weichselbaum et al. conducted a new study of CSP, finding that many developers were authoring vulnerable CSP policies [48].

More recently, Mendoza et al. investigated the inconsistencies between the security mechanisms used on the pages designed for mobile browsers as opposed to pages designed for desktop browsers [26]. Our work complements their research in that we quantify whether mobile browsers are capable of enforcing those security mechanisms, assuming that web developers can properly configure them. Researchers have also utilized the presence of properly configured security mechanisms as a proxy for the overall security of a website without the need to perform intrusive scanning [42, 44, 45, 29].

Evaluating Security-Mechanism Implementations. In 2010, Singh et al. discovered inconsistencies in web browsers' access control polices which they attributed to the "piecemeal" evolution of the policies [36]. In 2015, Hothersall-Thomas et al. developed a testing framework called BrowserAudit to help users evaluate the adoption of security mechanisms by different browsers [19]. The authors used their framework to evaluate desktop browsers by manually visiting their testing page with each browser. In our study, we focus on automatically evaluating a large number of mobile browsers and therefore have to overcome many issues associated with the automation of each browser. We decided against using BrowserAudit for our

tests since a significant number of security mechanisms and settings have been developed after 2015.

In 2017, Schwenk et al. studied the Same-Origin Policy implementations of ten different modern browsers [35]. They discovered different browser behaviors in approximately 23% of their test cases. In our prior work, we proposed Hindsight, an automated browser-agnostic framework for evaluating the vulnerability of mobile browsers to UI attacks [22]. To be able to conduct our large number of tests, we rely on this testing framework to quantify the adoption of a wide range of security mechanisms over time, instead of evaluating a browser's vulnerability to a few specific attacks. In recent work, Franken et al. evaluated the third-party cookie policies of desktop browsers and showed that third-party trackers could circumvent both built-in cookie policies, as well as those offered by anti-tracking browser extensions [15].

Mobile Browser Security. Niu et al. were the first to notice the different security problems associated with mobile browsers compared to desktop ones [30]. They recognized that limited screen real estate on mobile devices can cause critical UI components, such as the URL bar, to disappear. In 2011, Felt and Wagner investigated a novel security threat for phishing attacks on mobile devices due to the absence of reliable security indicators when switching between websites and apps [14]. In 2012, Amrutkar et al. compared desktop and mobile browsers and identified UI vulnerabilities abusing the screen limitations of the latter [3]. The authors later investigated the presence of security indicators in mobile browsers [4]. Other researchers focused on the WebView component of Android apps, identifying a number of security issues that could be caused by malicious apps loading benign websites and malicious websites being loaded by benign apps [23, 24].

IX. Conclusion

As users spend more and more time on their mobile devices and the web continues to be the platform of choice for deploying applications, it is critical that mobile web browsers cooperate with web servers to increase user security.

In this paper, we performed the first longitudinal study of the support of security mechanisms, such as the Content Security Policy (CSP) and the HTTP Strict Transport Security (HSTS), in mobile browsers. By designing 395 tests to exercise the implementations of security mechanisms in modern browsers, and exposing 351 unique APKs belonging to the most popular mobile browsers to these tests, we were able to evaluate the extent of such support in 20 different mobile browser families for the last seven years. We discovered that, even though browsers generally increase their support over time, not all browsers behave the same way; some browsers react more slowly than others and some browsers, with millions of downloads, do not even update for multiple years while their most recent versions are not capable of enforcing key mechanisms, such as `HttpOnly` cookies and HSTS. Moreover, we discovered that the conscious decision of not supporting one mechanism in favor of a newer one in Google Chrome has left hundreds of popular websites vulnerable to clickjacking attacks, when viewed through most of the evaluated mobile browsers. We quantified the rate of change for individual security mechanisms and found that most browsers are still not capable of properly enforcing complicated mechanisms such as CSP. We also discovered the presence of multi-year windows of vulnerability between the time when popular websites request a security mechanism and the time that most mobile browsers enforce it. Finally, we made the counter-intuitive observation that the security-mechanism support of browsers can depend on the Android version of the underlying platform, and therefore two users of the same version of the same browser can be experiencing vastly different levels of security when browsing the web.

We argue that our findings not only call for better testing on the side of browser vendors, but also show that developers cannot just blindly assume the enforcement of their desired security mechanisms. To that extent, we discussed the need for the design and development of server-side solutions which can adapt, in real time, to browsers based on the security mechanisms that they support.

Acknowledgment

We thank our shepherd Zhenkai Liang and the reviewers for their valuable feedback. This work was supported by the Office of Naval Research (ONR) under grant N00014-17-1-2541 and by the National Science Foundation (NSF) under grants CNS-1813974, CMMI-1842020, CNS-1617593, and CNS-1527086. Any opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research or the National Science Foundation.

References

[1] D. Akhawe, W. He, Z. Li, R. Moazzezi, and D. Song, "Clickjacking Revisited: A Perceptual View of UI Security." in *WOOT*, 2014.

[2] D. Alexis and S. Lennart, "Can I use... Support tables for HTML5, CSS3, etc," 2014. [Online]. Available: https://caniuse.com/

[3] C. Amrutkar, K. Singh, A. Verma, and P. Traynor, "VulnerableMe: Measuring systemic weaknesses in mobile browser security," in *International Conference on Information Systems Security*. Springer, 2012, pp. 16–34.

[4] C. Amrutkar, P. Traynor, and P. C. Van Oorschot, "An empirical evaluation of security indicators in mobile Web browsers," *IEEE Transactions on Mobile Computing*, vol. 14, no. 5, pp. 889–903, 2015.

[5] Apple Developer, "App Store Review Guidelines," https://developer.apple.com/app-store/review/guidelines/.

[6] J. Archibald, "Third party CSS is not safe," Feb 2018. [Online]. Available: https://jakearchibald.com/2018/third-party-css-is-not-safe/

[7] I. Archive, "Internet Archive: Wayback Machine," https://archive.org/web/, 2018.

[8] A. Barth, "RFC 6454 - IETF," Dec 2011. [Online]. Available: https://tools.ietf.org/html/rfc6454

[9] ——, "HTTP state management mechanism," 2011.

[10] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 75–88.

[11] I. Clelland, "W3C: Feature Policy," https://wicg.github.io/feature-policy/.

[12] comScore, "Mobile's Hierarchy of Needs," 2017.

[13] CVE-2010-3971, Oct 2010. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3971

[14] A. P. Felt and D. Wagner, *Phishing on mobile devices*. na, 2011.

[15] G. Franken, T. Van Goethem, and W. Joosen, "Who left open the cookie jar? a comprehensive evaluation of third-party cookie policies," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 151–168.

[16] Google, Jan 2018. [Online]. Available: https://developers.google.com/web/fundamentals/security/csp/

[17] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks: stealing the pie without touching the sill," in *Proceedings of the 2012 ACM conference on Computer and communications security (CCS)*, 2012.

[18] S. Helme, 2018. [Online]. Available: https://securityheaders.com

[19] C. Hothersall-Thomas, S. Maffeis, and C. Novakovic, "BrowserAudit: automated testing of browser security features," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 37–47.

[20] M. Kranch and J. Bonneau, "Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning." in *NDSS*, 2015.

[21] J. Leyden, "RIP HPKP: Google abandons public key pinning," https://www.theregister.co.uk/2017/10/30/google_hpkp/, 2017.

[22] M. Luo, O. Starov, N. Honarmand, and N. Nikiforakis, "Hindsight: Understanding the Evolution of UI Vulnerabilities in Mobile Browsers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 149–162.

[23] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 343–352.

[24] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking attacks on web in android, ios, and windows phone," in *International Symposium on Foundations and Practice of Security*. Springer, 2012, pp. 227–243.

[25] M. Marlinspike, "More tricks for defeating SSL in practice," *Black Hat USA*, 2009.

[26] A. Mendoza, P. Chinprutthiwong, and G. Gu, "Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites," in *Proceedings of the Web Conference (WWW'18)*, April 2018.

[27] Mozilla, Jan 2018. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy

[28] ——, "Same-origin policy," Mar 2018. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#Changing_origin

[29] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: large-scale evaluation of remote javascript inclusions," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 736–747.

[30] Y. Niu, F. Hsu, and H. Chen, "iPhish: Phishing Vulnerabilities on Consumer Electronics." in *UPSEC*, 2008.

[31] OWASP, "Cross-site Scripting (XSS)," Mar 2018. [Online]. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[32] ——, "OWASP Secure Headers Project," https://www.owasp.org/index.php/OWASP_Secure_Headers_Project, 2018.

[33] OWASP, "SameSite Overview," https://www.owasp.org/index.php/SameSite, 2018.

[34] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities at popular sites," *IEEE Oakland Web*, vol. 2, no. 6, 2010.

[35] J. Schwenk, M. Niemietz, and C. Mainka, "Same-Origin Policy: Evaluation in Modern Browsers," in *26th USENIX Security Symposium (USENIX Security 17). Vancouver, BC: USENIX Association*, 2017.

[36] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee, "On the incoherencies in web browser access control policies," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 463–478.

[37] D. F. Somé, N. Bielova, and T. Rezk, "On the Content Security Policy Violations due to the Same-Origin Policy," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 877–886.

[38] O. Starov, P. Gill, and N. Nikiforakis, "Are you sure you want to contact us? quantifying the leakage of pii via website contact forms," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 1, pp. 20–33, 2016.

[39] Statcounter, "Android Version Market Share Worldwide," http://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide, 2018.

[40] Statista, "Android version market share 2018 — Statistic," Feb 2018. [Online]. Available: https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/

[41] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the Web Tangled Itself: Uncovering the History of Client-Side Web (In) Security," in *Proceedings of USENIX Security*, 2017.

[42] S. Tajalizadehkhoob, T. Van Goethem, M. Korczyński, A. Noroozian, R. Böhme, T. Moore, W. Joosen, and M. van Eeten, "Herding vulnerable cats: a statistical approach to disentangle joint responsibility for web security in shared hosting," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 553–567.

[43] T. Van Goethem, P. Chen, N. Nikiforakis, L. Desmet, and W. Joosen, "Large-scale security analysis of the web: Challenges and findings," in *International Conference on Trust and Trustworthy Computing*. Springer, 2014, pp. 110–126.

[44] T. Van Goethem, F. Piessens, W. Joosen, and N. Nikiforakis, "Clubbing seals: Exploring the ecosystem of third-party security seals," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 918–929.

[45] M. Vasek and T. Moore, "Identifying risk factors for webserver compromise," in *International Conference on Financial Cryptography and Data Security*. Springer, 2014, pp. 326–345.

[46] W3C, "Content Security Policy - level 2," https://www.w3.org/TR/2014/WD-CSP2-20140703/, 2018.

[47] ——, "Content Security Policy Level 2: Relation to X-Frame-Options," https://www.w3.org/TR/CSP2/\#frame-ancestors-and-frame-options, 2018.

[48] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1376–1387.

[49] M. Weissbacher, T. Lauinger, and W. Robertson, "Why is csp failing? trends and challenges in csp adoption," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 212–233.

[50] Y. Zhou and D. Evans, "Why Aren't HTTP-only Cookies More Widely Deployed?" in *Proceedings of 4th Web 2.0 Security and Privacy Workshop (W2SP)*, 2010.