Hindsight: Understanding the Evolution of UI Vulnerabilities in Mobile Browsers

Meng Luo, Oleksii Starov, Nima Honarmand, Nick Nikiforakis
Stony Brook University
{meluo, ostarov, nhonarmand, nick}@cs.stonybrook.edu

ABSTRACT

Much of recent research on mobile security has focused on malicious applications. Although mobile devices have powerful browsers that are commonly used by users and are vulnerable to at least as many attacks as their desktop counterparts, mobile web security has not received the attention that it deserves from the community. In particular, there is no longitudinal study that investigates the evolution of mobile browser vulnerabilities over the diverse set of browsers that are available out there. In this paper, we undertake the first such study, focusing on UI vulnerabilities among mobile browsers. We investigate and quantify vulnerabilities to 27 UI-related attacks—compiled from previous work and augmented with new variations of our own—across 128 browser families and 2,324 individual browser versions spanning a period of more than 5 years. In the process, we collect an extensive dataset of browser versions, old and new, from multiple sources. We also design and implement a browser-agnostic testing framework, called *Hindsight*, to automatically expose browsers to attacks and evaluate their vulnerabilities. We use *Hindsight* to conduct the tens of thousands of individual attacks that were needed for this study. We discover that 98.6% of the tested browsers are vulnerable to at least one of our attacks and that the average mobile web browser is becoming *less secure* with each passing year. Overall, our findings support the conclusion that mobile web security has been ignored by the community and must receive more attention.

CCS CONCEPTS

• **Security and privacy → Browser security**; **Software and application security**; **Mobile platform security**; **Vulnerability scanners**;

KEYWORDS

Mobile browser security; vulnerability testing; user interface; phishing attacks; Hindsight

1 INTRODUCTION

The recent years have seen a steady increase in sales of mobile devices as even more users purchase smartphones and tablets to supplement their computing needs. The smartphones' cleaner UIs, in combination with an ever increasing number of apps and constantly decreasing prices, are attracting more and more users who entrust their devices with sensitive data, such as personal photographs, work emails, and financial information—making mobile devices an increasingly popular target for attacks.

Even though the most common form of abuse in smartphones is that of malicious applications, it is most certainly not the only possible kind of abuse. One must not forget that smartphones have powerful browsers and, as such, are susceptible to at least as many problems as desktop browsers. A user visiting a malicious website through her mobile browser can be the victim of web application attacks (e.g., XSS and CSRF), attacks against the browser (e.g., memory corruption [24] and application logic issues [11]), as well as attacks against the user herself (e.g., phishing and malvertising).

Especially for phishing and malvertising, prior research has shown that users of mobile browsers may be more susceptible to such attacks than users of traditional desktop browsers [2, 3, 15, 27, 30, 32]. The limited screen real-estate of hand-held devices, combined with mobile browsers' desire to maximize the space allotted to a webpage means that parts of the browser UI, under certain conditions, disappear. These parts, such as the address bar, are critical for identifying the true nature of a website and if they are missing, an attacker can more easily trick users into divulging their personal and financial information.

A major limitation of the aforementioned research is that the quantification of which mobile browsers were vulnerable to what attacks was done *once* and was done *manually*. Therefore, the reported findings could only capture the state of vulnerability at the time when the researchers performed their experiments. The quick update cycles of modern software, coupled with the fact that the app stores of modern smartphones house hundreds of different mobile browsers (each advertising a wealth of features, such as tracker blocking, voice-control, and reduced data consumption) means that we currently do not know which browsers are vulnerable to what attacks and how the vulnerability of the mobile browser ecosystem has evolved over time, i.e., are mobile browsers becoming more or less vulnerable to specific UI attacks?

Recognizing this gap, in this paper, we collect the attacks against mobile browsers discussed in prior research and, after expanding them with novel variations, we distil from them a series of attack building blocks, i.e., techniques that attackers could use, either stand-alone or in unison, to perform one or more attacks against mobile web users.

To test mobile browsers against these building blocks, we design and implement *Hindsight*, a dynamic-analysis, browser-agnostic framework that can automatically expose any given mobile browser to an attack and assess whether the attack succeeded or failed. Our

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '17, October 30-November 3, 2017, Dallas, TX, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3133987

framework is able to overcome a series of challenges, such as bypassing the splash screens of browsers, dealing with idiosyncrasies of different browsers, and, most importantly, extract information about a browser's UI in a browser-agnostic fashion without relying on browser-specific web drivers (e.g., those used by Selenium) and without the assistance of the browser itself.

In this paper, we focus on Android and its web browsers, due to the platform's popularity, the large number of official and third-party app stores, and its open-source ecosystem which greatly faciliates the implementation of a framework like *Hindsight*. By crawling Android app stores and third-party websites for current and past versions of mobile web browsers and filtering out duplicates and those browsers that do not conform to our definition of a modern web browser, we expose 2,324 APKs belonging to 128 distinct browser families to 27 different attack building blocks. By launching more than 62K individual attacks and automatically assessing their success or failure, we are able to quantify the vulnerability of modern mobile web browsers and how this vulnerability has evolved since 2011 (the year of our earliest APK files).

Among others, we find that 98.6% of the evaluated browsers are vulnerable to at least one attack, with the average browser being vulnerable to twelve. Depending on the specific browser evaluated, we find that attackers have typically more than one way of hiding a browser's address bar, confusing users about the exact URL on which they are located, and stealing browser cookies due to the rendering of mixed content. Contrary to our expectations, we discover that mobile browsers, even some of the most popular ones, appear to be becoming *more vulnerable* as years have passed. We also quantify the attacks that have the widest applicability across different browser families, finding attacks that are applicable to more than 96% of all our evaluated browser versions. Our results are a clear sign that mobile web security has been ignored by the community and must receive more attention before attackers start abusing the many vulnerabilities that they have at their disposal.

Overall, our contributions are the following:

- We systematically analyze related work and compile a list of mobile browser UI vulnerabilities. We expand that list with novel attack variations and arrive at 27 building blocks that expose mobile web browsers to different UI attacks.
- We collect thousands of Android mobile browser APK files from multiple, often non-cooperative, online sources, covering 128 different browser families with versions from 2011 to 2016. We devise techniques to date each APK so that we can perform longitudinal measurements across browser families.
- To analyze the large and diverse set of browser families, we design and develop an automated, pluggable, and browser-agnostic framework (called *Hindsight*) for unsupervised installation and exposure of mobile browsers to a series of UI attacks, and determination of the tested browsers' vulnerability to each evaluated attack. We describe the non-trivial challenges that we had to overcome to build *Hindsight*.
- Using our framework and attack building blocks, we automatically expose mobile browsers to more than 62K attack instances. By analyzing the reports generated by our framework, we paint a picture of how the status of mobile UI web security has changed over a period of 5 years and identify trends that demonstrate that the evaluated threats have been largely ignored and need urgent addressing.

2 ATTACK BUILDING BLOCKS

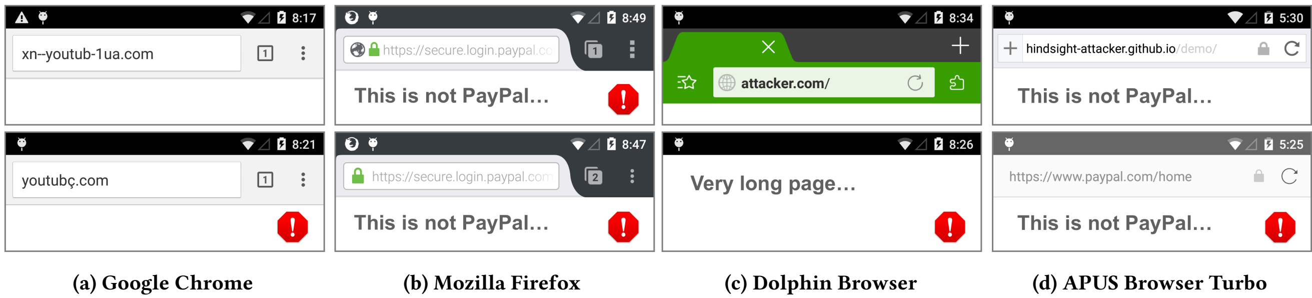
To evaluate the UI security of mobile browsers, we started by performing a thorough investigation of prior work [2, 3, 6, 15, 16, 30, 32, 38], in search for known attacks. In addition to the attacks that we discovered, we also reasoned about interactions of mobile browsers with websites and how browsers could attempt to maximize the screen real-estate for the rendered websites. Through a process of trial-and-error we were able to identify novel variations of existing attacks. Table 1 shows the results of this process in the form of yes/no questions. We name each of these 27 questions an *attack building block* (ABB). For any given browser, if the answer to an ABB is "yes", then the browser is vulnerable to that building block. We chose the term "attack building block" to stress the fact that these vulnerabilities can be used either stand-alone or combined. In the next paragraphs, we discuss each class of ABBs and provide a few characteristic examples.

ABBs #1–6: Event Routing. Event routing attacks abuse the non-intuitive routing of events across overlapping elements typically belonging to different origins. Clickjacking is a well-known case of an event routing attack from the desktop browser world. Mobile browser event routing attacks have the potential to cause more damage since users interact with small screens and tap (as opposed to click) which covers wider regions of a page. In 2012, Amrutkar et al. [2, 3] showed that some mobile browsers did not always follow the event-routing policy found in desktop browsers where, in the case of multiple overlapping elements, the topmost element receives the click/tap events. The authors showed that, at the time, the Android stock browser, Nokia Mini-Map and Opera Mini would trigger the event handlers of elements that were underneath opaque images. This behavior could be used to, among others, facilitate click-fraud (where expensive ads are hidden underneath images that trick the user to interact with them) and non-standard login CSRF attacks. In our study, ABBs #1–6 test event routing across different combinations of cross-origin elements.

ABBs #7–19: URL & Address Bar. Similar to desktop browsers, mobile browsers have to display the URL of the current page in order to convey a website's true identity to the user. Since attackers can freely register domains, and create arbitrary subdomains and filepaths, a browser's address bar is of critical importance for defending against spoofing attacks.

Unfortunately there exist ways which websites can abuse to confuse users or hide a mobile browser's address bar, by triggering different parts of a browser's logic for maximizing the screen real-estate allotted to webpages. ABBs #7–19 cover various ways which attackers could use to hide their website's identity.

For example, mobile browsers have to decide how to show long URLs that do not "fit" in the limited width of the user's screen. ABBs #7–9 quantify whether a mobile browser shows the leftmost part of a long URL which has many subdomains (e.g. `www.paypal.com.attacker.com`), the rightmost part of a URL with a long filepath (e.g. `www.attacker.com/foo/bar/www.paypal.com`, and either the leftmost or the rightmost parts for a URL with both many



(a) Google Chrome **(b) Mozilla Firefox** **(c) Dolphin Browser** **(d) APUS Browser Turbo**

Figure 1: Examples of vulnerabilities in the latest (bottom row) and an older (top row) version of four popular mobile browsers. The warning sign indicates behaviors that can be abused for spoofing attacks: (a) rendering URLs with confusing IDN-based domain, (b) truncating long URLs with many subdomains, (c) hiding the address bar in case of a long page, (d) showing a page's title instead of its URL.

subdomains as well as a long file-path. Similarly, mobile browsers have to decide whether they want to give preference to a website's title instead of its URL (e.g., showing "Welcome to Facebook" versus `https://www.facebook.com`, ABB #12), whether they should hide the URL bar when a user starts interacting with a page (e.g., through scrolling or switching to landscape mode, ABBs #14–16), whether they should show a previously hidden address bar upon sensitive interactions (e.g., text input, ABBs #17–19) and whether they should display IDN domains in their punycode or internationalized format (ABB #10). Figure 1 shows four different browsers that are vulnerable to four different ABBs in these classes.

ABBs #20–27: Security Indicators & Content. With the increased focus on HTTPS, and the presence of SSL/TLS errors that developers and users must be able to recognize, security indicators are as important as a browser's address bar. Specifically, a mobile web browser needs to be able to communicate to users whether the current website is loaded over HTTPS (as opposed to plain HTTP), whether there is mixed content that could jeopardize some of the guarantees of HTTPS, and whether the current certificate is signed by a chain of trustworthy Certificate Authorities, or it is self-signed. Moreover, other icons that are unrelated to HTTPS, such as a website's favicon, must be displayed on a different location on a browser's address bar to avoid attackers using padlock-like favicons in combination with SSL stripping attacks [28].

Prior work has shown that mobile browsers have been ignoring W3C best practices with regard to indicator placement and different browser families vary wildly in terms of their signage for denoting SSL protected websites, mixed content warnings, and

Table 1: List of the 27 attack building blocks (ABBs) used to evaluate security of mobile browsers

Class	Test#	Explanation	Prior Work	Potential Attacks
Event Routing	1–6	Do cross-origin, overlapping elements receive events when they are not the topmost ones? (Different tests for combinations of overlapped images and buttons, links, forms, and other images)	[3, 6]	Clickjacking, CSRF
URL	7–9	When presented with a long URL (long subdomain, long filepath, or a combination of both), does a browser render that URL in a way that could be abused for spoofing attacks?	[30, 38]	Phishing, malware/scam delivery
	10	When presented with an Internationalized Domain Name (IDN), will a browser display the IDN format?	[16]	Phishing, malware/scam delivery
Address Bar	11	Is the address bar hidden if the top-level frame is navigated by a child frame?	[3, 6]	Phishing, malware/scam delivery
	12	Does a browser show a page's title instead of its URL?	[8]	Phishing, malware/scam delivery
	13	Is the address bar hidden if the visited website has a lot of content?	Novel	Phishing, malware/scam delivery
	14	Is the address bar hidden when switching the device to "landscape" mode?	Novel	Phishing, malware/scam delivery
	15–16	Is the address bar hidden upon manual/automatic page scrolling?	[30, 32]	Phishing, malware/scam delivery
	17–18	Is the address bar hidden when typing in a textbox and tapping on a button?	[15, 38]	Phishing, malware/scam delivery
	19	Is the address bar hidden when typing to a fake (e.g., canvas-created) textbox?	Novel	Phishing, malware/scam delivery
Security Indicators	20	Is the favicon placed next to padlock icon?	[4, 5, 14, 37]	MITM attack, Phishing
	21–22	When rendering an HTTPS page, is the address bar displayed the same in the presence of mixed content (image and JavaScript) as in its absence?	[9]	MITM attack
	23	Is a webpage with self-signed certificate rendered without warnings?	[4, 5, 14, 37]	MITM attack, Phishing
Content	24	Can an iframe expand its size past the one defined by its parent frame?	[3, 6]	Phishing
	25	Is a mixed-content image resource loaded?	[9]	MITM attack
	26	Is a mixed-content JavaScript script executed?	[9]	MITM attack
	27	Is JavaScript code included in a self-signed website executed before the warning is accepted?	Novel	Phishing, MITM attack

domain-verified vs. extended-verification certificates [4, 5, 14, 38]. In our work, we develop ABBs for identifying the confusable placement of a favicon next to the place where the SSL-lock appears (ABB #20), lack of warnings for mixed content (ABBs #21–22), and lack of warnings for self-signed certificates (ABB #23).

Next to security indicators, we also develop four additional ABBs that quantify a mobile browser's risk to dangerous content. Specifically, we gauge whether an iframe can expand its dimensions past what is specified by the parent frame (ABB #24, originally described by Amrutkar et al. [3]), whether a browser renders mixed-content images and JavaScript (ABBs #25–26), and whether JavaScript code located on a self-signed page would execute *before* a user accepts the certificate warning (ABB #27).

Automatic Vulnerabilities. During pilot experiments with *Hindsight* and our 27 ABBs, we realized that not all browsers behave as one would expect. That is, there exist browsers that constantly hide their URL bars, or browsers that always give preference to showing the title of a page, instead of its URL. We take advantage of this behavior by adding an extra test that quantifies whether a browser shows, by default, its URL and/or title bars, and whether both are present or the browser gives preference to one over the other. Using the results of these tests, we may be able to immediately consider the browser vulnerable to some of ABBs in Table 1. For example, if a browser constantly hides its URL bar, building blocks that determine which part of the URL a browser shows (ABBs #7–9) are automatically labeled as vulnerable, because an attacker, if he so chooses, can draw a fake address bar showing an arbitrary URL. Similarly, for such a browser, tests that determine whether an address bar is hidden when a user turns her phone to Landscape mode (ABB #14), or scrolls (ABBs #15–16), or types into a text box (ABBs #17–19) are also marked as vulnerable.

3 DATA COLLECTION

The app markets of modern smartphones offer a significantly larger set of browser choices, compared to traditional desktop browsers. The Google Play store houses hundreds of different mobile browsers, each one advertising its own differentiating features including voice-control [18], reduction of data usage [21, 22], increased performance [19], and ad-block/anti-tracker integration [17, 20].

At the same time, while the Google Play store is the authoritative market for up-to-date apps, it does not provide older versions of apps which are necessary for studying the evolution of vulnerabilities across time and mobile browsers. Past work in mobile browser security did not have such requirements since researchers used the most recent browser versions at the time of their experiments. In the following sections, we describe our methodology for collecting current and past versions of mobile browsers (together with important metadata), allowing us to compile a dataset comprising thousands of mobile browser APKs spanning more than a 120 different browser families.

3.1 Mobile Browsers Dataset

To obtain a comprehensive set of mobile browsers we performed the following data collection and processing steps:

Table 2: Browser ranks

Rank	# Installations	# browsers	Example Browser
1	1,000–5,000 MM	1	com.android.chrome
2	100–500 MM	3	org.mozilla.firefox
3	50–100 MM	4	mobi.mgeek.TunnyBrowser
⋮	⋮	⋮	⋮
13	500–1,000	2	com.shark.sharkbrowser

Table 3: Filtering and processing the browser APKs

Dataset	# of APKs
Raw dataset	**4,612**
Browser duplicates	1,416
Non-modern browsers	152
Installation failure	76
Crash on launch	135
Failed splash bypass	498
Unrecognizable address bar	11
Final dataset	**2,324**

Table 4: Number of APKs and browser families per year

Year	2011	2012	2013	2014	2015	2016
Browser APKs	5	89	367	505	755	603
Browser families	4	21	41	54	94	77

Collecting browser families. Using Selenium [34] we automatically searched for the keyword "browser" in the Google Play Store and recorded the results. Through manual analysis, we filtered out non-web browsers (e.g., file browsers). Table 2 shows a small sample of browser families that we collected, ranging from browsers with billions of installations (e.g. Google Chrome) to ones with less than 1K installations (e.g. Shark Browser). The browser families are divided into 13 ranks according to their number of installations.

Crawling browser versions. To obtain older browser versions, we resorted to third-party websites and alternative app markets. Specifically, we build website-specific crawlers and collected as many APKs as possible from the following six online sources: *Androidapps*, *Apkhere*, *Apkmirror*, *Apkpure*, *Uptodown*, and *Aptoide*. In some cases, such as Aptoide, collecting older versions meant installing third-party market apps and then reverse-engineering the way with which they downloaded older APKs from their servers. Through this process we collected a total of 4,612 individual APKs. Our data collection process for both the Google Play Store and the alternative sources was conducted in August 2016.

Filtering. To arrive at a representative set of mobile browsers, we performed rounds of filtering on the raw dataset of 4,612 APKs. As shown in Table 3, we first removed duplicates of the same browsers collected from different data sources. We identify a unique APK as a tuple of (*Package Name*, *Version Name*, *MD5sum*), where the Package Name defines a browser family and the Version Name indicates a particular version of that browser. Each APK file was additionally

labeled with its MD5 hash to remove even more duplicates that represent builds of the same browser but with different Android targets. Specifically, we used the MD5 hash of the `.SF` file under `META-INF/` subdirectory that already contains the hashes of various inner resources. Through this duplicate filtering, we removed 1,416 repeated APK files.

Next, we attempted to install a browser and subsequently visit a simple HTML page showing images and using JavaScript. We argue that rendering images and executing JavaScript are the bare minimum requirements for a modern browser that aspires to be used to browse today's web. This process allowed us to eliminate browser apps that would not install, browsers that would crash while launching, and niche, text-only browsers (e.g. `com.weejim.app.lynx`) which are not going to be used by everyday web users.

In our final step, we had to filter the APKs which our browser-agnostic vulnerability testing framework, *Hindsight*, was unable to evaluate. For 498 APKs, *Hindsight* was unable to bypass a mobile browser's splash screen. One must remember that browser developers are free to show an arbitrary number of menus and dialogues (e.g., the features of the browser, terms and conditions, choice of language, etc.) before allowing users to utilize their browser. Despite our Splash Bypass Algorithm (Section 4.2.3), *Hindsight* is not always able to identify the series of actions necessary for bypassing the splash screen. Next to splash screens, *Hindsight* also needs to automatically identify the placement and contents of a browser's address bar, a complicated process which, despite our techniques described in Section 4.2.1, can occasionally fail (11 APKs). Overall, we could reliably evaluate 2,324 APK files belonging to 128 browser families, which represent modern and working mobile browsers. On average, our dataset includes 18 versions for each browser family. Figure 2 shows how the number of APKs grows with the fraction of browser families. All statistics presented in the rest of this paper are based on this set of browsers.

3.2 Metadata Extraction

To be able to evaluate each mobile browser against our attack building blocks (ABBs) and to study the evolution of vulnerabilities through time, we need to extract information about the platform on which any given APK can run, its release date, and any *intent* metadata necessary for executing it.

App usage metadata. To identify the appropriate device on which an APK can be installed and run, we extract APK properties, such as, *native-code*, *sdkVersion* and *targetSdkVersion*. The *native-code* indicates which architecture and instruction set an app depends on, whereas the *sdkVersion* and *targetSdkVersion* work in tandem to set compatible Android API levels. Our goal is to run an APK on a device having an Android API level closest to the *targetSdkVersion* and greater than *sdkVersion*. Section 4.2.2 describes our methodology in greater detail. We also extract the activities necessasry for the ADB-based launching of the browser.

App release date metadata. For our longitudinal study of UI vulnerabilities, we need to be able to provide a release timestamp for each APK in our dataset with sufficient accuracy. Through experimentation, we discovered that a relatively accurate way of obtaining this information is by extracting the modification time of

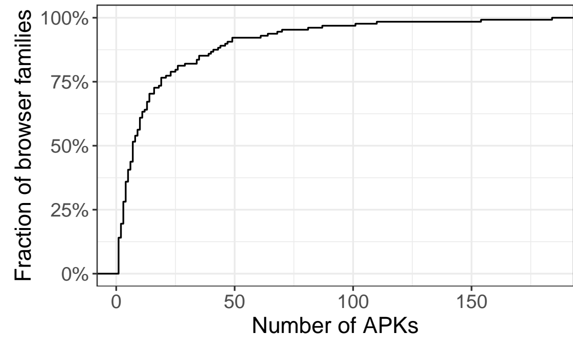


Figure 2: ECDF for the number of browser APKs vs. fraction of browser families

`.RSA` or `.DSA` file under `META-INF/` subdirectory. These timestamps are good proxies for the time of signing an Android app and thus its release. To avoid errors, we limit ourselves to extracting the year of each release and cross-validate our findings, when possible, with the dates provided on the *AppBrain* website for some of the collected APKs. We show the distribution of the collected APKs from 2011 to 2016 in Table 4.

Table 5 shows the ranges of versions and distribution of APKs for six popular browser families. For instance, we evaluated a total of 41 different versions of *Google Chrome* collected by August 2016. The oldest one is in year 2013 with version 29.0.1547.72 and the latest is in August of year 2016 with version 51.0.2704.81. This version was the newest one at the time of our data collection. On average, we have at least 10 different versions per year for *Google Chrome*.

4 TESTING FRAMEWORK AND METHODOLOGY

Manually testing tens of ABBs on thousands of browsers is a non-starter. For each browser, one has to install it, bypass the splash screen that many browsers display upon the first use, and then test the ABBs on that browser. Each test involves loading a webpage, mimicking a user's interaction with that page, and then analyzing the final rendered output to identify vulnerabilities. Moreover, this process of "input then analyse" may often have to be repeated multiple times in a single test. It would take human analysts many months to complete a single round of tests, and even then they are bound to make numerous mistakes in the course of testing. Therefore, one needs an *automated* framework to run the tests.

4.1 Hindsight Framework Architecture

We have designed and implemented an automated vulnerability testing framework called *Hindsight*. Figure 3 shows the main components and processing steps of the framework. In *Hindsight*, each input APK goes through four general processing steps: (1) SDK assignment, (2) installation and splash-screen bypassing, (3) ABB testing, and (4) results evaluation.

In step (1), *Hindsight* decides on a suitable version of Android capable of running the browser. In step (2), it installs the APK on an Android device running that version. It then checks that the installed browser is ready for testing by pointing it to a simple webpage. Some APKs crash at this stage and are excluded from further testing. Some other browsers display one or more initial

Table 5: Details of the version-span of APKs for the top six mobile browsers

Rank	Package Name	Oldest Version	Latest Version	# of versions	Avg. per year
1	com.android.chrome	29.0.1547.72 (2013)	51.0.2704.81 (2016)	41	10.25
2	org.mozilla.firefox	9.0 (2011)	47.0 (2016)	68	11.33
2	com.UCMobile.intl	8.5.1 (2012)	10.10.8.820 (2016)	44	8.8
2	com.opera.browser	14.0.1074.57453 (2013)	37.0.2192.105088 (2016)	34	8.5
3	mobi.mgeek.TunnyBrowser	8.5.1 (2012)	11.5.8 (2016)	64	12.8
3	com.opera.mini.native	8.0.1739.87973 (2015)	18.0.2254.106200 (2016)	26	13

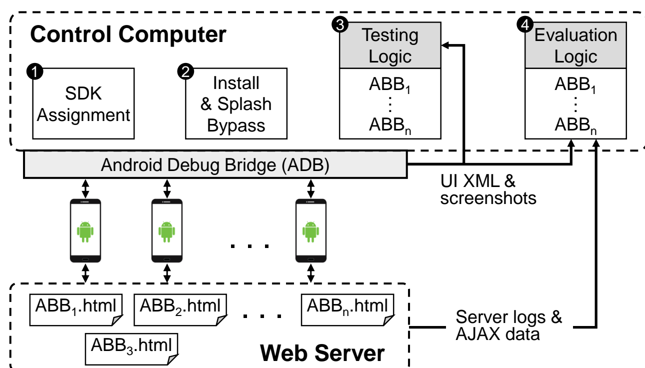


Figure 3: Architecture of the *Hindsight* framework

screens—which we call *splash screens*—that need to be bypassed before the browser is ready for testing. Section 4.2.3 explains how this is done automatically. If splash bypass fails, the APK is excluded from further testing. Table 3 shows the number of APKs excluded due to installation failure, crashing or splash-bypass failure.

In steps (3) and (4), each APK is exposed to each of the ABBs and the results are checked for vulnerability. In *Hindsight*, each ABB consists of three different pieces: i) the ABB HTML file which is a carefully designed webpage containing the necessary elements to test for the vulnerability, ii) the ABB *test logic* which contains the necessary logic to interact with and provide UI inputs to the Android device during the test, and iii) the ABB *evaluation logic* that analyzes all the information collected during the test run to determine the vulnerability of the tested APK to that ABB. This *pluggable* design makes it fairly straightforward to add new ABBs to the framework, as *Hindsight* does not make any a priori assumptions about these ABB parts.

Architecturally, the *Hindsight* framework consists of three main components. Multiple Android devices (currently 4) are used to run tests in parallel. Each device runs a different version of the Android OS to allow catering to different API levels needed by our APKs. A controlling computer runs all the installation, splash-bypass, ABB testing, and ABB evaluation logic. This computer is connected to the Android devices through USB ports and uses the *Android Debug Bridge* (ADB) to communicate with them. All browser installations and UI interactions needed for the splash-bypassing and testing logic use ADB commands. Finally, a web server is used to serve the ABB HTML pages to the Android devices during the tests. This web server also collects some crucial logs that are used by the ABB evaluation logic to determine vulnerabilities (Section 4.2.4).

The following sections provides more information about the challenges of building such an automated framework, and techniques used in *Hindsight* to overcome them.

4.2 Building a Browser-Agnostic Framework

The biggest challenge in building *Hindsight* is to make it *browser-agnostic*. On the one hand, this is a must-have feature for *Hindsight* because it has to support a wide variety of different browser families. On the other hand, it is a non-trivial challenge as browsers do not follow standardized application layouts and render webpages in different ways using different engines.

Even as simple a problem as identifying the address bar—that is required by several of our ABBs—becomes a challenge as *Hindsight* cannot a priori know which rendered UI element is the address bar, if any, and where it is located. Such decisions are made by browser developers and are not known to the testing framework. The framework can only observe the rendered UI and has to extract all the needed information by analyzing that UI in a browser-agnostic manner. Such analysis is not only needed at the end of a test run to determine vulnerability, but also often required during a test run to identify UI elements to interact with—for example, to know where to tap in a loaded webpage in order to type in a text box.

The need to be browser-agnostic poses multiple design and implementation challenges in each of the four processing steps mentioned above. Below, we will discuss some of these challenges and how *Hindsight* copes with them.

4.2.1 Browser-Agnostic UI Analysis. A major challenge faced by steps (2)–(4) of the framework, is to analyze the application UI to determine the presence and location of certain elements on the screen. The rendered UI usually consists of two distinct parts. The first part includes the *application-level* UI elements, such as the address bar, padlock, favicon and tab headers. The second part is the *webpage content.*

Most Android applications, browsers included, use standard Android UI libraries for application-level elements. This allows *Hindsight* to use a standard Android toolset, called *UI Automator*, to capture an XML dump of the application UI's Document Object Model (DOM) tree which provides different attributes for each application-level UI element, including its text or image as well as screen coordinates. This greatly simplifies UI analysis for application-level elements.

For the webpage elements, however, we cannot rely on such textual dumps. This is because the vast majority of browsers do not expose the rendered webpage elements as part of the application-level DOM tree. Therefore, if one needs to, e.g., locate a certain HTML button on the screen, it cannot rely on the XML dump

provided by UI Automator. There might be ad hoc APIs and drivers to obtain this information for some browsers, but our framework requires a method that works seamlessly for all browsers.

Therefore, *Hindsight* uses Optical Character Recognition (OCR) to analyze page content: it captures the current screenshot (using ADB's `screencap` command), and then uses OCR to search for textual clues that are carefully built into the tested webpage to locate the elements (buttons, images, text boxes, etc.) to interact with. The specific text to search for is ABB-specific and is determined by the ABB testing and evaluation logic mentioned earlier. Currently, *Hindsight* uses a combination of the Tesseract OCR Engine [13] and Google's Vision API [31] for this functionality.

Although simple in theory, the OCR method is fraught with non-trivial problems. Firstly, browsers use different rendering engines that render the same HTML page in vastly different ways. For one thing, there is no browser-agnostic mapping between the on-screen coordinates of rendered elements and their HTML-specified location, even if the HTML file specifies absolute locations for the elements. As another example, different browsers may use different font sizes to render the same text in a page. To cope with this problem—after observing how a large number of browsers render webpages—we concluded that critical text elements in ABB HTML pages have to be repeated multiple times with different font sizes and families to increase their chances of being picked up by OCR.

The second problem with OCR is that the rendered webpage is not always the only content shown in a browser window. Often, browsers may include messages (such as usage tips, update reminders, etc.) that are not part of our HTML, and may even cover some of the content that is critical to the OCR analysis. We deal with this problem in two ways. First, as often these message boxes are part of the application-level DOM, we use a technique similar to what is described in Section 4.2.3 to automatically click through and dismiss them, before the OCR analysis begins. Second, we design the ABB HTML pages such that the critical elements are located close to the center of the screen—to the extent that this is possible in a browser-agnostic fashion—to reduce their chances of being covered by such messages.

It is worth mentioning that, in addition to webpage content, OCR is also used as a secondary method for locating application-level UI elements. This is because some browsers do not always render the application-level elements using standard Android libraries, and for some other browsers, the dumped XML data is not consistent with the visible UI of the application. In these cases, OCR is used as a backup method to locate the application-level elements.

This hybrid of XML dumps and OCR has resulted in a robust browser-agnostic analysis infrastructure that works well in practice, as indicated by our manual verification results (Section 4.3).

4.2.2 SDK Assignment. Each APK requires certain Android APIs to function properly. In Android, the API level is denoted by the so-called Software Development Kit (SDK) version number: a monotonically increasing number with 24 being the highest at the time of this writing. APKs contain a manifest file which includes the minimum (*skdVersion*) and target (*targetSdkVersion*) SDKs the application supports. Ideally, one should test each APK on all SDK versions in this range as it is conceivable that browsers could behave differently given different SDK features (e.g., the browser could use different API calls given different SDK versions). However, given the limited number of physical Android devices in the current framework, and that each device can have only one Android version installed at a time, this would increase the testing time significantly.

To keep the problem manageable, a compromise was made. We first analyzed all the APKs and collected their SDK requirements. For each APK, we considered SDK versions in the range [*minimum*, *target*]. Then, we chose four different SDK versions that would allow us to cover all the tested APKs, and installed each of them on a different Android device. For the current dataset, these are SDKs 16, 18, 21 and 23, corresponding to Android versions 4.1.2, 4.3, 5.0 and 6.0, respectively. *Hindsight* uses this knowledge of the installed SDK versions to approximate an even distribution of APKs to devices to maximize the testing speed, while ensuring that each APK still gets to run on a supported SDK version. All of this is done in a browser-agnostic fashion, just by extracting the metadata that is available in each APK.

4.2.3 APK Installation and Splash-Screen Bypassing. In the next step, *Hindsight* uses ADB to install each APK on its assigned device. To check the installation success, it launches the browser and directs it to a specific webpage, and checks for its successful loading. As Table 3 shows, there are 211 APKs that fail in this step because of either an installation failure or crash upon launch. Currently, we are excluding these APKs from further testing; in the future, *Hindsight* will try to install and run them on other Android versions.

A more serious challenge is to bypass the *splash screens*. Ideally, a browser should just display the requested webpage when launched. However, a large fraction of browsers (1,600+ APKs in the current dataset), instead show other initial screens that have to be bypassed before the webpage is displayed. These screens include a variety of content from permission requests, to advertisements, to introduction to the application itself—and often, multiple of these. *Hindsight* cannot test the browser without bypassing these screens.

To deal with advertisements embedded by browsers, we tried a variety of methods and the most successful solution was to use a third-party AD blocking application called *AdGuard* [1] that works by monitoring and filtering the network traffic into and out of the device. This gives us a browser-agnostic method of filtering advertisements that works quite well in practice.

For other splash bypass problems, our approach is to automatically mimic a user's interaction with the application. Informed by our analysis of a large number of splash screens of tested browsers, *Hindsight* uses a generic, browser-agnostic method of delivering guided tapping and swiping inputs (using ADB) to the device. In each step, the UI is analyzed to find text that often indicates tappable buttons such as "Next", "Continue", "Okay", etc. If such text is found, a tap event is delivered to that location on the screen. If not, *Hindsight* tries swiping the screen to move to the next screen. Then, the UI is analyzed again to determine if we are at expected webpage or there are more splash screens. This process is repeated up to a configurable maximum number of times.

This method works well in practice for most browsers. In the current dataset, 1,606 APKs show splash screens of which *Hindsight* can bypass 1,108. Of the remaining ones, there are some whose splash screens are hard to analyze to find tappable items, and some that require more complex interactions (such as signing up for an

account or entering one's email address) that cannot be bypassed using this method. We exclude such APKs from further testing.

4.2.4 ABB Design, Testing, and Evaluation. The main challenges of these steps have to do with UI analysis, as explained in Section 4.2.1. Here we discuss two other problems that required creative solutions in *Hindsight*. We should emphasize that this is just a sampler of the common challenges we faced; there were multiple other ABB design challenges and tricks that we had to omit due to space limitations.

To begin, we should note that some testing data cannot be easily extracted through UI analysis. For example, in ABBs #1–6, we need to determine if an input event was routed to an invisible iframe, or in ABB #24, we need to know an iframe's dimension to detect ballooning. OCR-based UI analysis is not a reliable method to collect such information. Instead, we can easily collect such information using JavaScript in an ABB's HTML file. However, there is no direct communication channel to convey that information to the evaluation logic that is running on the control computer (which talks to the Android device using ADB). To solve this problem, *Hindsight* creates such a communication channel by sending this information as AJAX (Asynchronous JavaScript and XML) messages from the ABB HTML page to the webserver where they are saved. During ABB evaluation, this data is fetched by the evaluation logic and used for vulnerability analysis. Moreover, for mixed-content-related ABBs, we also fetch the HTTP server logs to determine whether a particular resource was requested by the browser. Figure 3 shows this side communication channel used by the evaluation logic.

Another major challenge was caused by browsers' support for multiple tabs. Most modern browsers can simultaneously have multiple open tabs and show the number of tabs, and sometimes the page titles, in the application screen. This seemingly "benign" feature caused major trouble for some of our ABBs. For example, in ABBs #21-22, we do a pixel-wise comparison between the screenshot of a pure-HTTPS page and a mixed-content page with the same design. The goal is to identify the existence of mixed-content-related icons or warnings, with the assumption that the only difference between the two versions would be the presence of the mentioned warning signs. However, the tab headers and tab count that many browsers show would also be different after opening the second page, and this would result in erroneously concluding that there is a difference between the two pages, even if there are no such warning signs.

Unfortunately, there is no browser-agnostic way to close tabs, and simply closing and re-launching the browser application in between webpages will not help either because browsers often re-open all previously open tabs upon re-launch. *Hindsight* solves this problem as follows. Right after the browser is installed and any potential splash screens are bypassed, *Hindsight* saves a copy of the browser's Android folder where the application saves its internal settings and history. There is one such folder per application in Android and its path is determined by the package name of the APK, and is thus easy to find. This gives a pristine copy of the application's state where there are no open tabs. Before launching the browser to load a new page, *Hindsight* copies this state back to make sure there are no traces of previously open tabs in the browser history. This will ensure that our screenshots are free from the tab-related side effects that would otherwise cause testing failures.

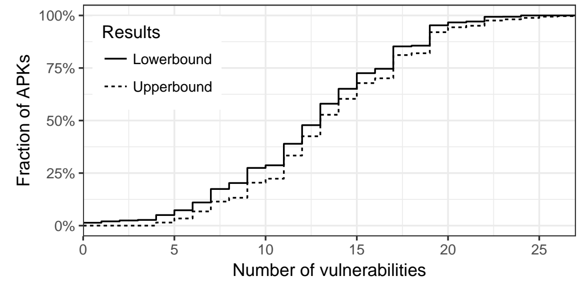


Figure 4: ECDF of vulnerability vs. fraction of browser APKs

4.3 Verifying Hindsight Results

Since *Hindsight* is the first framework of its kind, there is no other way to verify its results than to do it manually. To this end, we have augmented our ABB testing logic to generate an analyst-friendly, self-contained HTML file that includes, for every (APK, ABB) tuple, 1) all the device screenshots that were generated and used while testing the ABB on the APK, and 2) the *Hindsight*'s vulnerability analysis result generated by the ABB evaluation logic.

This HTML page is reviewed manually to determine whether a human expert would concur with the automatically-generated vulnerability result. If not so, a check-box in the HTML file is ticked to mark that result as "Not Accepted". All this information is uploaded to a server and stored in a database to keep track of verification results. This design allowed us to efficiently review the results of *Hindsight* when applied to 2,324 APKs and precisely quantify its correctness.

For the current dataset, it took approximately 60 person-hours to finish one round of verification. The observed error rate is 1.5% where the ABB evaluation logic makes a wrong judgement (both false positives and false negatives). Overall, the fraction of "Not Accepted" results is low enough to consider *Hindsight* results generally dependable.

It is worth noting that such manual verification is only required to debug *Hindsight* to establish the credibility of its implementation, and does not need to be repeated every time *Hindsight* is used.

5 EVALUATION

Using *Hindsight* we tested each of the 2,324 browser APKs, belonging to 128 distinct browser families, to the 27 attack building blocks (ABBs) presented in Section 2. Out of more than 62K vulnerability reports for different combinations of APK and ABBs, *Hindsight* failed on 2,260 tests, i.e. the error rate (or more precisely, the uncertainty rate) is approximately 3.6%. Note that this error rate includes the 1.5% error rate due to false positives/false negatives discussed in Section 4.3 as well as the cases where the framework itself knows that there has been an error (e.g. browser crash, or inability to find the URL bar). That covers 292 APKs or less than 12.6%, which have at least one failed test. To account for this uncertainty we present, where applicable, our results using a lower bound (i.e. all ABBs marked as "Error" are, in reality, "Not Vulnerable") and an upper

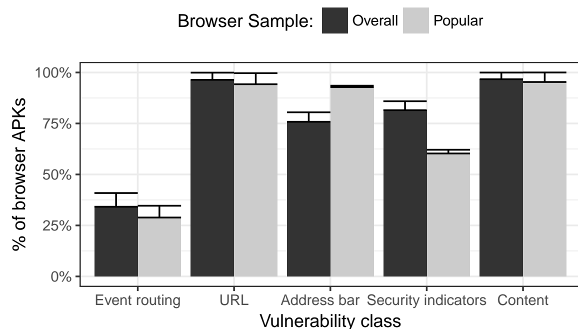


Figure 5: Fraction of browser APKs affected by at least one of ABBs across five classes; whiskers denote the lower and upper bounds.

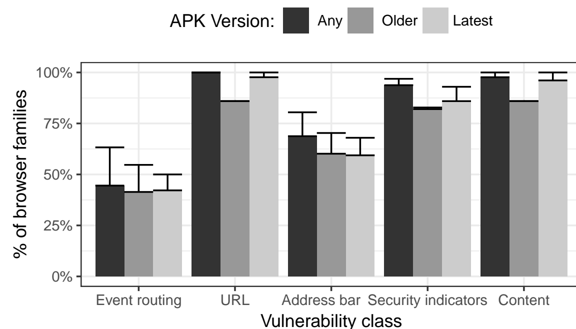


Figure 6: Fraction of browser families affected by at least one of ABBs across five classess; whiskers denote the lower and upper bounds.

bound (i.e. all ABBs marked as "Error" are, in reality, "Vulnerable") of browser vulnerability.

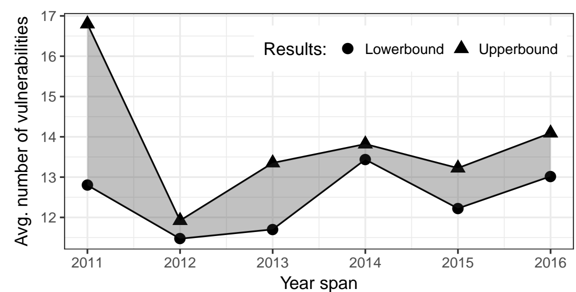
5.1 General Findings

Overall, using *Hindsight*, we found that 2,292 of the 2,324 evaluated APKs (98.6%), were vulnerable to at least one ABB. Figure 4 shows how the number of vulnerabilities grows with the fraction of tested browser APKs, i.e. 50% of APKs are vulnerable to more than 12 ABBs.

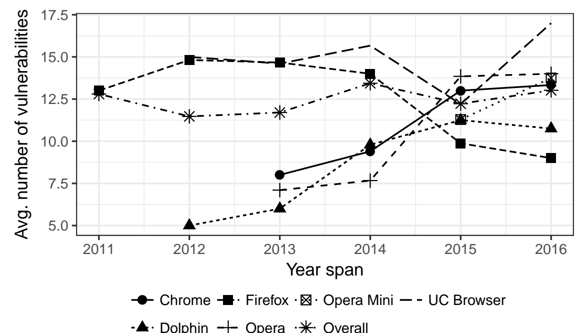
To understand which classes of ABBs are more successful than others, in Figure 5, we show the vulnerability of browsers to the five different types of ABBs (Event Routing, URL, Address Bar, Security Indicators, and Content). A browser APK is marked vulnerable to a class of attacks if it is vulnerable to at least one ABB belonging to that class. Among others, we find that even the least popular class of ABBs (Event Routing) affects more than 25% of the tested browser APKs, with the most popular classes (URL and content) affecting almost 100% of the evaluated browsers. Similarly, we see that APKs belonging to popular browsers are, in general, as vulnerable as the rest of the browsers and that our level of uncertainty (denoted via whiskers at the top of each bar) does not alter the observed vulnerability trends. Figure 6 presents the same information grouped by browser families. A browser family is marked vulnerable to a class of attacks if at least one of its APKs is vulnerable to at least one ABB belonging to that class. There, we see that i) the latest versions of browser families are as vulnerable, if not more vulnerable, than older versions of the same browser family, ii) the relative popularity of vulnerability classes remains the same as for distinct APKs, and iii) certain ABBs, such as, the ones belonging to the Address Bar class, affect less browser families than individual APKs. The differences between Figure 5 and Figure 6 are because our 2,324 APKs are not uniformly spread in the 128 browser families, allowing different patterns to emerge when quantifying vulnerabilities as a fraction of browser families versus as a fraction of APKs.

5.2 Longitudinal Analysis

One of the main motivations of our research is the rapid upgrade cycle of modern browsers and its effect on vulnerability reports. Most apps, including browsers, are updated on a weekly or monthly basis including new features and bug fixes. As such, any quantification of security that past researchers manually obtained [2, 3, 6, 30, 32]



(a) All browsers



(b) Most popular browsers

Figure 7: Average number of vulnerabilities over the years

was already outdated by the time their research was made available to the public. Since *Hindsight* is an automated, browser-agnostic vulnerability testing framework, it allows us to, not only obtain vulnerability reports for all latest browser apps but also, to study the vulnerability trends through time.

Using years as our time granularity, we group APKs belonging to the same browser family (via their package names), order them according to their version numbers, and extract their release date as described in Section 3.2.

Figure 7a reveals how the average number of vulnerabilities for every browser APK varies from year to year, starting from 2011 (the year of our oldest APK) to 2016 (the year of our most recent APKs).

Table 6: The most popular ABBs and their percentage of affected APKs and browser families per year

Top ABBs	APKs 1st	APKs 2nd	APKs 3rd	APKs 4th	APKs 5th	Browser families 1st	Browser families 2nd	Browser families 3rd	Browser families 4th	Browser families 5th
2011	7, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 21, 22 (100%)					7, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 21, 22 (100%)				
2012	9 (100%)	25 (97.75%)	10, 26 (96.63%)	7 (85.39%)	19 (58.42%)	9 (100%)	7, 25 (95.24%)	10, 26 (90.48%)	22 (80.95%)	21 (76.19%)
2013	9 (95.64%)	7 (95.37%)	25 (94.55%)	10 (74.93%)	26 (71.39%)	7, 9 (100%)	25 (95.12%)	10 (82.93%)	26 (80.49%)	22 (78.05%)
2014	9 (98.81%)	7 (98.42%)	25 (97.43%)	22 (83.37%)	26 (78.02%)	7, 9 (98.15%)	25 (96.30%)	22 (77.78%)	26 (74.07%)	21 (70.37%)
2015	25 (96.16%)	9 (94.44%)	7 (93.25%)	15 (76.03%)	18 (74.97%)	9 (98.94%)	25 (97.87%)	7 (96.81%)	22 (82.98%)	21 (71.28%)
2016	25 (96.68%)	9 (93.70%)	7 (93.20%)	22 (85.57%)	15, 18 (68.66%)	9 (100%)	7 (98.70%)	25 (97.40%)	22 (88.31%)	21 (67.53%)

As before, we present both the lower-bound and upper-bound of vulnerabilities to account for errors during *Hindsight*'s runs. There we observe a wave-like pattern with decreases in 2012 and 2015 and increases in the remaining years. At the same time, one can see that there was never a year with browsers affected by, on average, less than 11 vulnerabilities and that there are more vulnerabilities in 2016 than there were in 2011 and 2012. The large difference between upper-bound and lower-bound in 2011 is an artifact of unstable versions of browsers which crash often and unpredictably combined with the small overall number of APKs for 2011.

To quantify how popular browsers are different than the rest, in Figure 7b, we show the average number of vulnerabilities per year for six popular browsers. There we see that, while the number of vulnerabilities is uniform when considering all browsers at the same time (average number of vulnerabilities ranges from 11.5 to 13.4), different browser families exhibit different trends. Firefox has been steadily decreasing in vulnerability since 2013 whereas families like Opera, Dolphin, and Chrome have been increasing. UC Browser exhibits a wave-like pattern and has, on average, the most vulnerabilities of popular browsers in 2016.

Table 6 shows the most popular ABBs for the tested APKs and browser families from 2011 through 2016. There we can see the evolution from 2011 where *all* evaluated APKs and browser families were vulnerable to thirteen different ABBs to the remaining years where different ABBs, such as, #7, #9 (both related to how a browser shows a long URL) and #25 (showing mixed-content images) emerge as the most potent ones. ABBs #7 and #9 are dangerous because they allow attackers to masquerade their websites as belonging to trustworthy brands, and ABB #25 can be used to steal session cookies that are not marked with `HTTPOnly` [40] and use them for session hijacking attacks [9, 36]. On a more positive note, we see that while most browsers used to execute JavaScript originating from a mixed inclusion from 2012 to 2014 (ABB #26), this behavior is becoming less popular.

Lastly, Figure 8 shows how different classes of vulnerabilities have affected mobile browsers over the years. There we see that even though most classes have had a fairly uniform effect on browser APKs, event routing has been increasing in popularity since 2014. Since *Hindsight* treats every browser as a black box it cannot provide us with the reason why event-routing ABBs have become more applicable than they used to be. Through manual investigation and experimentation we concluded that one of the main reasons for this is a vulnerable behavior of the Chromium's Touch Adjustment feature used in Android's WebView with SDK version 23, and consequently browsers using embedded WebView component were affected.

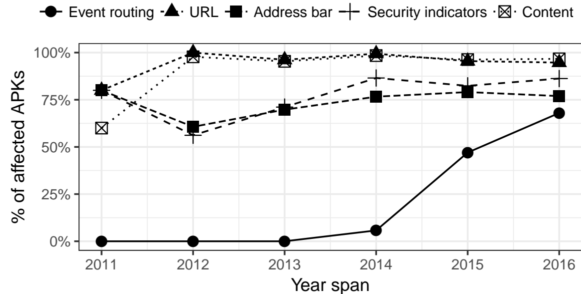


Figure 8: Fraction of APKs affected by at least one ABB in a class across the years

5.3 Popularity versus Vulnerability

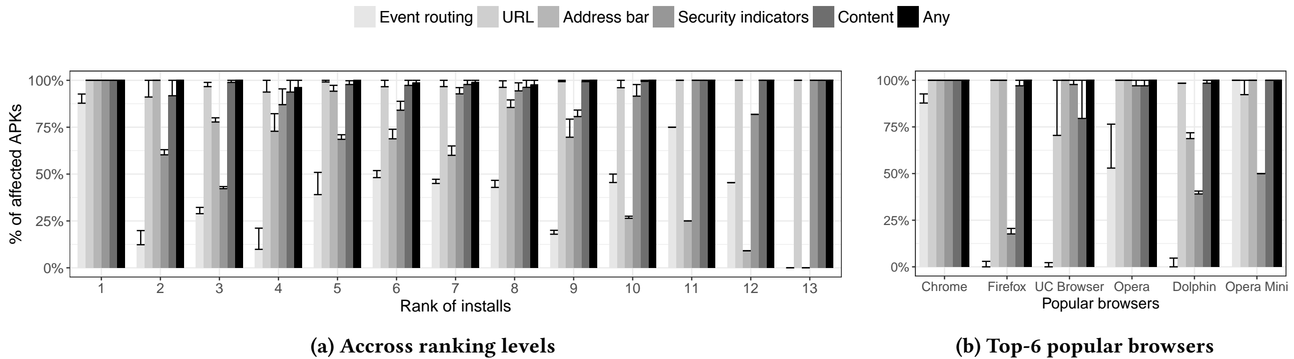
As described in Section 3.1, the 128 browser families evaluated in this paper are as popular as Google Chrome with more than a billion installations (Rank 1) and as "unpopular" as the Shark Browser with less than 1K installations (Rank 13).

In Figure 9a, we explore the correlation between the ranking of each APK and its vulnerability to the five classes of ABBs. There we see that the most popular browsers are not necessarily the most secure. In fact, we observe that browsers that are in the last three ranks of popularity (10-13) exhibit significantly less vulnerabilities than more popular browsers. For example, upon manual inspection of the Shark browser (located in Rank 13), we witnessed that browser never show a page's title and always shows the URL bar, regardless of swiping, rotation, and page length. Because of these design choices, the Shark browser is not vulnerable to any of the ABBs belonging to the Address Bar class.

Figure 9b focuses on the vulnerabilities exhibited by the APKs belonging to the six most popular browsers families. We observe that 100% of the APKs belonging to all six families have at least one vulnerability and Chrome and Opera exhibit similar vulnerability patterns. Firefox appears to be the most secure of the six browsers (confirming the time series presented earlier in Figure 7b) whereas, next to Firefox, UC Browser and Dolphin are the only browsers not vulnerable to the evaluated Event-Routing ABBs.

5.4 HTTPS

In recent years, HTTPS has been steadily increasing in adoption partly because of initiatives like Let's Encrypt which assists websites in obtain free-of-charge certificates [25], search engines using HTTPS as a positive ranking signal [7], and modern desktop



(a) Accross ranking levels

(b) Top-6 popular browsers

Figure 9: Fraction of APKs affected by at least one ABB in a class

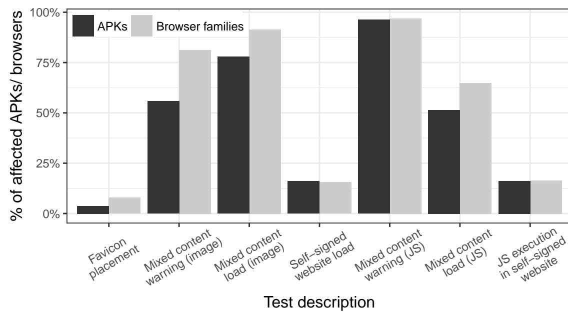


Figure 10: Analysis of HTTPS related ABBs

browsers which have started showing negative indicators for pages served over HTTP [12, 14, 33].

Due to the increased focus on HTTPS, in this section, we briefly focus on the HTTPS-related ABBs of the ones shown in Table 1. In Figure 10, we compare how different HTTPS-related ABBs affect browser APKs and browser families. We find that although less than 17% APKs and browser families are vulnerable to favicon-padlock placement (ABB # 20) and self-signed certificates ABBs (ABB # 23, ABB # 27), over 50% of them execute the code originating from the mixed inclusion (ABB #26). Moreover, we observe that more than 90% browser APKs and families do not use different indicators to help users differentiate between HTTPS websites with no mixed inclusions and those with mixed inclusions. We argue that, even for browsers that block mixed content, this is an undesirable behavior because it allows mixed inclusions to go by unnoticed for a longer period of time thereby increasing the window of exploitation for users who happen to utilize browsers that render mixed content.

5.5 Patterns of Vulnerability

The longitudinal analysis presented in the Section 5.2 demonstrates that most browser families are either consistently vulnerable to the same number of attacks or, worse, become more vulnerable with every passing year. At the same time, one may still wonder whether the vulnerabilities that *Hindsight* allows us to quantify were "always" part of a browser's code (present in the oldest version of the browser available in our dataset) or were added during some later time. In this section we answer this question by analyzing the evolution of vulnerability patterns for each of our 27 ABBs.

By analyzing the Yes/No results of our ABBs for each version of a given browser family, we discovered that we can categorize most vulnerability patterns using six patterns: i) always vulnerable (YES), ii) always safe (NO), iii) introduction of a new vulnerability (noYES), iv) removal of an existing vulnerability (yesNO), v) temporary vulnerable (noYESno), and vi) temporary safe (yesNOyes). Figure 11 shows the distribution of these six patterns per each ABB number and class of vulnerabilities.

For some vulnerability classes (like Event Routing and Address Bar), the distribution of patterns is similar across ABBs. Contrastingly, for vulnerabilities belonging to the Security Indicators class, in additions to ABBs that covary (e.g., #20 and #23), we also observe ABBs with clearly different patterns (e.g., #21 and #22). Our findings suggest that for vulnerabilities related to event routing and a browser's address bar, the correspondent ABBs are interconnected and rely on a single cause (e.g. the handling of touch events and the automatic hiding of the address bar). This is promising because, if true, it allows for multiple attacks to be stopped by a few secure design choices. This, however, is not true for ABBs related to Content, URL and Security indicators, meaning that each vulnerability will likely require a different countermeasure.

Overall, on Figure 11 we observe highly undesirable, from a security perspective, cases where YES and noYES patterns dramatically dominate, e.g., for ABBs #7, #9, #21, #22, and #25. Moreover, we also find cases which reveal the temporary adoption of insecure features (noYESno) and the regression from a secure version to a less secure one (yesNOyes). As an example of the noYESno pattern we find that the Dolphin Browser Express was, for a number of versions, hiding the address bar while the user was giving input. Similarly, Opera Mini temporary showed a page's title instead of its URL. An example of the regression pattern (yesNOyes) is the Dolphin browser which was temporarily showing the TLD+1 part of a domain when the URL was long but later reverted back to its original insecure behavior (showing the left-most part of a URL). Similarly, the ASUS browser temporarily stopped hiding its address bar when a user was scrolling a long page. These patterns are clear signs of security versus usability trade-offs, which highlights the need of educating both browser vendors as well as users about mobile web UI attacks and equipping them with tools, such as, *Hindsight*, that

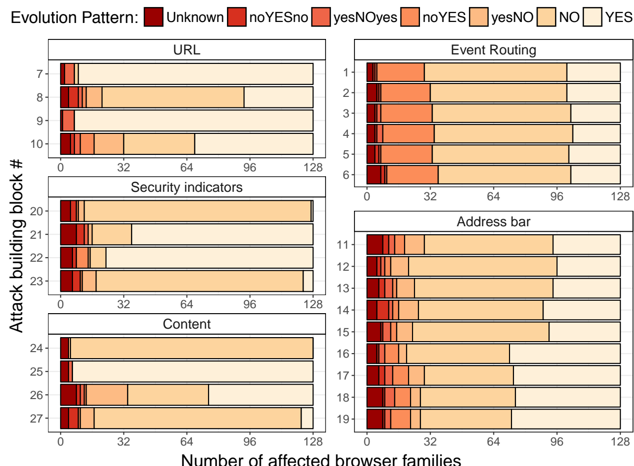


Figure 11: Patterns of vulnerability for browser families

allows for continuous vulnerability assessment. Fortunately, next to the aforementioned undesirable patterns, we also observe stable trends of desirable "yesNO" and "NO" patterns for particular ABBs.

6 DISCUSSION

Summary of findings. The results that we were able to retrieve using *Hindsight* against 2,324 browser APKs belonging to 128 different browser paint a fairly disconcerting picture of mobile web security. We found that 98.6% of the evaluated APKs were vulnerable to at least one of our 27 attack building blocks, with 50% of APKs being vulnerable to more than 12 building blocks (Section 5.1). By performing longitudinal measurements, we observed that many browsers become *less* secure as years go by (Section 5.2) and that popular browsers are often more vulnerable to our building blocks than less popular ones (Section 5.3).

Ethical Disclosure. Even though all of our ABBs can be used to craft attacks against users, they are not vulnerabilities that could lead to drive-by exploitations. That is, most of our ABBs can be used to increase the success chances of social engineering attacks (like phishing or the user-initiated installation of malware) but they can not be weaponized to compromise mobile devices automatically at a large scale. Despite not fitting the mold of traditional vulnerabilities, we are currently in the process of reaching out to the vendors of mobile browsers to ethically disclose our findings and understand to what extent they are aware of the uncovered issues and how they intend to address them.

Limitations and Future Work. *Hindsight* is currently utilizing real smartphones, each of which is running a different version of the Android operating system and SDK version. Even though this choice was a conscious one motivated by the desire to experiment on real devices so that our findings are free from emulation artifacts, we also understand the limitations of our approach in terms of scalability, i.e. how fast can we evaluate any given browser against a series of attacks, and in terms of applicability, i.e. do the same attacks work against the same browser when that is installed on a tablet that is equipped with a larger screen? For these reasons, we intend to develop an emulation-backed version of *Hindsight*, experiment with a large array of emulated devices and configurations, and compare our results with the ones reported in this paper.

A separate limitation is that all security assessments are based on our collection of 27 ABBs automatically evaluated by *Hindsight*. It is definitely possible that browsers that are performing poorly against our evaluated attacks would perform better in the presence of a different set of tests. As such, we do not intend for our results to be used as an authoritative guide for identifying the most secure mobile browser. Instead, we hope that our results will motivate the vendors of mobile browsers to revisit the design of their UIs and strengthen their browsers against attacks.

7 RELATED WORK

To the best of our knowledge this paper is the first systematic study of the evolution of UI attacks in mobile browsers, spanning thousands of browser versions and hundreds of browser families. The motivation to design and build *Hindsight* came from realizing that all the novel results published in prior work were destined to always be outdated since the attacks were performed manually against a limited number of browser families and versions. We briefly discuss this prior work below.

Attacks against mobile browsers. In 2008, Niu et al. were the first to identify the security problems associated with browsers used in devices with small screens [30], well before the current commercial success of smartphones. The authors evaluated the Safari mobile browser available in the original iPhone and the Opera browser available in two Nintendo gaming consoles finding issues, such as, URL truncation (browsers showing the beginning and end of a long URL skipping the middle), the automatic hiding of the URL bar by programmatically scrolling by a single pixel, or the altogether absence of the URL bar. In 2010, Rydstedt et al. discovered that many mobile versions of popular websites were lacking frame-busting code and introduced "tapjacking", a mobile equivalent of clickjacking which could be abused to, among others, steal a home router's WPA keys [32]. Felt and Wagner, in 2011, investigated the threat of phishing attacks on mobile devices by exploring the transitions between websites and apps and the difficulty of ascertaining whether a login prompt originates from a trusted website/app versus a malicious one that is spoofing a trusted one [15]. Many of our attack building blocks were inspired by the work of Amrutkar et al. [2–5] who manually evaluated ten smartphone and three tablet browsers and compared them to traditional desktop browsers. The authors identified the issues of unexpected event routing, the ballooning of iframes, the absence of a URL bar after an iframe-originating redirection, and inconsistencies in terms of security indicator presence and location, despite W3C guidelines [38].

WebView Security. Both Android and iOS provide a WebView class (UIWebView in iOS) which apps can use to show web content to users. Prior research has identified a number of security issues with WebView APIs that allow both malicious apps to attack benign websites as well as malicious websites to abuse benign but vulnerable apps that render them using WebView [10, 26, 27, 29]. In our work, we chose to treat each mobile browser as a black box, launching generic UI attacks and observing their outcome. As such,

while WebView-specific vulnerabilities could be straightforwardly added as new attack building blocks to *Hindsight*, we consider them out of scope for this specific paper.

Inconsistencies across desktop browsers. Researchers have in the past identified inconsistencies in the implementations of security mechanisms across browsers which could be abused to attack them. Singh et al. investigated access control inconsistencies among popular browsers in terms of principal labeling and its effect on, among others, the Same-Origin Policy [35]. Zheng et al. investigated the handling of cookies in modern browsers and identified implementation quirks in the cookie handling and cookie storing code of certain browsers that enabled network attackers to perform cookie injection attacks [39]. Hothersall-Thomas et al. presented BrowserAudit, a website that performs 400 checks of security mechanisms and used it to test the correctness of modern desktop browsers [23]. Even though *Hindsight* and BrowserAudit are conceptually similar, our work focuses on testing thousands of mobile browser versions (with all the difficulties associated with automatically installing them and dealing with splash screens, ads, and crashes) and UI attacks whose success cannot be ascertained by the attack website, requiring us to develop the mechanisms discussed in Section 4.

8 CONCLUSION

As mobile devices increase in popularity and, for some, replace the need for a desktop computer, it is important that we understand their security posture and what areas we need to improve upon. In this paper, we investigated the seemingly forgotten problem of UI vulnerabilities in mobile browsers where attackers can take advantage of mobile browser idiosyncrasies to better social engineer users and exfiltrate their data. Motivated by the desire to move away from snapshot-based measurements (i.e., where researchers study what is available to them at the time of their experiments) we collected thousands of mobile browser versions spanning hundreds of browser families and developed *Hindsight*, the first dynamic-analysis, browser-agnostic testing framework for gauging the vulnerability of mobile web browsers to UI attacks. Using *Hindsight*, we were able to quantify the vulnerability of mobile browsers through time, finding, among others, that i) the vast majority of web browsers are vulnerable to one or more of our evaluated attacks, ii) mobile browsers seem to be getting less secure as years go by, and iii) the popularity of a browser and security are not necessarily correlated. Our hope is that this study will motivate the building of more security-friendly UIs for mobile web browsers and the reviewing of some of the existing design decisions that attackers can straightforwardly abuse to victimize users.

Acknowledgments:
We thank 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-1617593 and CNS-1527086. Some of our experiments were conducted with equipment purchased through NSF CISE Research Infrastructure Grant No. 1405641.

Any opinions, findings, and conclusions or 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] AdGuard. 2009–2017. ad blocker and anti-tracker. https://adguard.com/en/welcome.html. (2009–2017).
[2] Chaitrali Amrutkar, Kapil Singh, Arunabh Verma, and Patrick Traynor. 2011. *On the Disparity of Display Security in Mobile and Traditional Web Browsers.* Technical Report. Georgia Institute of Technology.
[3] Chaitrali Amrutkar, Kapil Singh, Arunabh Verma, and Patrick Traynor. 2012. VulnerableMe: Measuring systemic weaknesses in mobile browser security. In *International Conference on Information Systems Security.* Springer, 16–34.
[4] Chaitrali Amrutkar, Patrick Traynor, and Paul C Van Oorschot. 2012. Measuring SSL indicators on mobile browsers: Extended life, or end of the road?. In *International Conference on Information Security.* Springer, 86–103.
[5] Chaitrali Amrutkar, Patrick Traynor, and Paul C Van Oorschot. 2015. An empirical evaluation of security indicators in mobile Web browsers. *IEEE Transactions on Mobile Computing* 14, 5 (2015), 889–903.
[6] Chaitrali Vijay Amrutkar. 2014. *Towards secure web browsing on mobile devices.* Ph.D. Dissertation. Georgia Institute of Technology.
[7] Zineb Ait Bahajji and Gary Illyes. 2014. Google Webmaster Blog: HTTPS as a ranking signal. https://webmasters.googleblog.com/2014/08/https-as-ranking-signal.html. (2014).
[8] Bugzilla@Mozilla. 2010. URL Display of Title instead of the URL Enables Phishing Attacks via URL Spoofing. https://bugzilla.mozilla.org/show_bug.cgi?id=605206. (2010).
[9] Ping Chen, Nick Nikiforakis, Christophe Huygens, and Lieven Desmet. 2013. A Dangerous Mix: Large-scale analysis of mixed-content websites. In *Proceedings of the 16th Information Security Conference (ISC).*
[10] Erika Chin and David Wagner. 2013. Bifocals: Analyzing webview vulnerabilities in android applications. In *International Workshop on Information Security Applications.* Springer, 138–159.
[11] CVE 2014. CVE-2014-6041 : The Android WebView in Android before 4.4 allows remote attackers to bypass the Same Origin Policy via a crafted attributes. http://www.cvedetails.com/cve/CVE-2014-6041/. (2014).
[12] Peter Dolanjski and Tanvi Vyas. 2017. Mozilla Security Blog: Communicating the Dangers of Non-Secure HTTP. https://blog.mozilla.org/security/2017/01/20/communicating-the-dangers-of-non-secure-http/. (2017).
[13] Tesseract Open Source OCR Engine. 2017. Google. https://github.com/tesseract-ocr/tesseract. (2017).
[14] Adrienne Porter Felt, Robert W Reeder, Alex Ainslie, Helen Harris, Max Walker, Christopher Thompson, Mustafa Embre Acer, Elisabeth Morant, and Sunny Consolvo. 2016. Rethinking connection security indicators. In *Twelfth Symposium on Usable Privacy and Security (SOUPS).*
[15] Adrienne Porter Felt and David Wagner. 2011. Phishing on mobile devices. In *Proceedings of the Web 2.0 Security and Privacy Workshop.*
[16] Anthony Y Fu, Xiaotie Deng, Liu Wenyin, and Greg Little. 2006. The methodology and an application to fight against unicode attacks. In *Proceedings of the second symposium on Usable privacy and security.* ACM, 91–101.
[17] Google Play store. 2017. CM Browser - Adblock Download. https://play.google.com/store/apps/details?id=com.ksmobile.cb. (2017).
[18] Google Play store. 2017. Dolphin - Best Web Browser. https://play.google.com/store/apps/details?id=mobi.mgeek.TunnyBrowser. (2017).
[19] Google Play store. 2017. Google Play store: Fastest Mini Browser. https://play.google.com/store/apps/details?id=com.mmbox.browser. (2017).
[20] Google Play store. 2017. Google Play store: Ghostery Privacy Browser. https://play.google.com/store/apps/details?id=com.ghostery.android.ghostery. (2017).
[21] Google Play store. 2017. Opera Mini - fast web browser. https://play.google.com/store/apps/details?id=com.opera.mini.native. (2017).
[22] Google Play store. 2017. UC Browser - Fast Download. https://play.google.com/store/apps/details?id=com.UCMobile.intl. (2017).
[23] Charlie Hothersall-Thomas, Sergio Maffeis, and Chris Novakovic. 2015. BrowserAudit: Automated Testing of Browser Security Features. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).*
[24] Jason Kersey. 2013. Chrome for Android Update. http://googlechromereleases.blogspot.com/2013/11/chrome-for-android-update.html. (2013).
[25] Let's Encrypt - Free SSL/TLS Certificates. 2017. https://letsencrypt.org/. (2017).
[26] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android system. In *Proceedings of the 27th Annual Computer Security Applications Conference.* ACM, 343–352.
[27] Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. 2013. Touchjacking attacks on web in android, ios, and windows phone. In *Foundations and Practice of Security.* Springer, 227–243.

[28] Moxie Marlinspike. 2009. More tricks for defeating SSL in practice. *Black Hat USA* (2009).

[29] Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. 2013. A View to a Kill: WebView Exploitation.. In *LEET*.

[30] Yuan Niu, Francis Hsu, and Hao Chen. 2008. iPhish: Phishing Vulnerabilities on Consumer Electronics. In *Proceedings of the Usability, Psychology, and Security Workshop (UPSEC)*.

[31] Google Cloud Platform. 2017. Cloud Vision API Documentation. `https://cloud.google.com/vision/docs/`. (2017).

[32] Gustav Rydstedt, Baptiste Gourdin, Elie Bursztein, and Dan Boneh. 2010. Framing attacks on smart phones and dumb routers: tap-jacking and geo-localization attacks. In *Proceedings of the 4th USENIX Workshop On Offensive technologies (WOOT)*. USENIX Association, 1–8.

[33] Emily Schechter. 2016. Google Security Blog: Moving towards a more secure web. `https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html`. (2016).

[34] Selenium. 2017. Selenium Webdriver. `http://www.seleniumhq.org/projects/webdriver/`. (2017).

[35] Kapil Singh, Alexander Moshchuk, Helen J Wang, and Wenke Lee. 2010. On the incoherencies in web browser access control policies. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 463–478.

[36] Suphannee Sivakorn, Jason Polakis, and Angelos D. Keromytis. 2016. The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In *In Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P '16)*.

[37] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. 2009. Crying Wolf: An Empirical Study of SSL Warning Effectiveness.. In *USENIX security symposium*. 399–416.

[38] W3C. 2010. Web Security Context: User Interface Guidelines. `https://www.w3.org/TR/wsc-ui/`. (2010).

[39] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Haixin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. 2015. Cookies Lack Integrity: Real-World Implications. In *24th USENIX Security Symposium (USENIX Security 15)*.

[40] Yuchen Zhou and David Evans. 2010. Why aren't HTTP-only cookies more widely deployed. *Proceedings of 4th Web* 2 (2010).