Fill in the B̲ l̲ a̲ n̲ k̲ s̲: Empirical Analysis of the Privacy Threats of Browser Form Autofill

Xu Lin
University of Illinois at Chicago, USA
xlin48@uic.edu

Panagiotis Ilia
University of Illinois at Chicago, USA
pilia@uic.edu

Jason Polakis
University of Illinois at Chicago, USA
polakis@uic.edu

ABSTRACT

Providing functionality that streamlines the more tedious aspects of website interaction is of paramount importance to browsers as it can significantly improve the overall user experience. Browsers' autofill functionality exemplifies this goal, as it alleviates the burden of repetitively typing the same information across websites. At the same time, however, it also presents a significant privacy risk due to the inherent disparity between the browser's interpretation of a given web page and what users can visually perceive.

In this paper we present the first, to our knowledge, comprehensive exploration of the privacy threats of autofill functionality. We first develop a series of new techniques for concealing the presence of form elements that allow us to obtain sensitive user information while bypassing existing browser defenses. Alarmingly, our large-scale study in the Alexa top 100K reveals the widespread use of such deceptive techniques for stealthily obtaining user-identifying information, as they are present in at least 5.8% of the forms that are autofilled by Chrome. Subsequently, our in-depth investigation of browsers' autofill functionality reveals a series of flaws and idiosyncrasies, which we exploit through a series of novel attack vectors that target specific aspects of browsers' behavior. By chaining these together we are able to demonstrate a novel invasive side-channel attack that exploits browser's *autofill preview* functionality for inferring sensitive information *even when users choose to not utilize autofill*. This attack affects all major Chromium-based browsers and allows attackers to probe users' autofill profiles for over a hundred thousand candidate values (e.g., credit card and phone numbers). Overall, while the preview mode is intended as a protective measure for enabling more informed decisions, ultimately it creates a new avenue of exposure that circumvents a user's choice to not divulge their information. In light of our findings, we have disclosed our techniques to the affected vendors, and have also created a Chrome extension that can prevent our attacks and mitigate this threat until our countermeasures are incorporated into browsers.

CCS CONCEPTS

• **Security and privacy → Browser security**.

KEYWORDS

Web Browsers; Autocomplete; Form Autofill; Data Exfiltration

ACM Reference Format:
Xu Lin, Panagiotis Ilia, and Jason Polakis. 2020. Fill in the B̲ l̲ a̲ n̲ k̲ s̲: Empirical Analysis of the Privacy Threats of Browser Form Autofill. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3372297.3417271

1 INTRODUCTION

Browsers lie at the heart of the web ecosystem, as they are the de-facto platform that mediates users' access to most online services. As websites continue to provide novel (and often complex) functionality to attract and engage users, browsers provide the necessary foundation by deploying new APIs and features that enable these functionalities and minimize friction during user interaction. A considerable *pain point* for users, which often leads to frustration and higher drop-off rates, is the completion of web forms [2]. This arduous task is particularly burdensome for users on mobile devices, who are inhibited by the devices' limited screen real estate. To minimize the hassle of completing and submitting web forms, major browsers have deployed *form autofill* capabilities. These features provide tangible benefits to users as browsers can automatically fill out the form input fields on behalf of the user and significantly reduce the amount of effort required.

Even though the autofill functionality is undoubtedly useful for users, it can be easily misused by malicious or invasive websites. For instance, a shady website can have a form with only one visible input field (e.g., email address for a newsletter subscription) while also including input fields for other information that are *visually hidden* from the user (e.g., their home address and phone number). As browsers generally populate all input fields for which there is a corresponding value in the user's autofill profile, even if those input fields are not visible, attackers can misuse autofill to exfiltrate private and sensitive information *without the user's knowledge or consent.* While a series of blog posts have mentioned the exfiltration of credentials and credit card numbers [24, 40], and news articles have reported the risks of autofill [5], the flaws of browsers' autofill capabilities have not been investigated in depth, nor has the global prevalence of such invasive practices been measured.

In this paper we present the first *comprehensive* analysis of major browsers' autofill functionality across multiple dimensions. Initially, we devise numerous previously-unreported approaches for *hiding* input fields in a page and "concealing" the actual information that is obtained by the form. Our evaluation of major browsers reveals a worrisome state of affairs, as most browsers do not have any preventative countermeasures in place to prevent this form of privacy-invasive deceptive practices, or are severely lacking as they are ineffective against the majority of techniques that we devise. Subsequently, we conduct a large-scale measurement study on the Alexa top 100K websites for identifying pages exhibiting such behavior through hidden form fields. As the heuristics employed for detecting and parsing forms differ across browsers, we conduct our study for both Firefox and Chrome. Alarmingly, we find that 5,295 (24.5%) and 1,843 (5.8%) of the websites with forms that are autofilled by Firefox and Chrome, respectively, also include *hidden* input elements. Moreover, fields that collect personally identifiable information (PII), such as the user's name, phone number and email address, are among those that are commonly hidden.

However, one could argue that more privacy-cautious users may avoid using autofill on less trusted sites. To that end, we demonstrate an even more severe novel attack that infers the user's information *without* requiring the autofill functionality to actually be triggered. This attack exploits autofill's *preview* mode and works against all Chromium-based browsers that we tested. In essence, this inference attack is achieved by chaining together a series of novel techniques that misuse flaws that we identified in multiple aspects of Chromium's autocomplete functionality and behavior.

First, we demonstrate that these browsers are susceptible to what we refer to as a *field-type mismatch* attack; we find that they do not match the type of a form field that is going to be autofilled to the expected field type of that value in the user's autocomplete profile. For instance, while phone numbers are typically `<input>` tag elements (i.e., a textbox), the browser will actually match that value to a `<select>` element (i.e., a drop-down menu) that has a matching option. We also found that we can include multiple drop-down menus of the same type (e.g., email) in a single form. Second, we have identified a side-channel leak in the *autofill preview* feature, which allows users to see what values *will* be autofilled *if* they decide to do so. These preview values are in an overlay that is *not accessible* by the page's DOM. However, we found that numerous style attributes of the drop-down menu change if the user's profile value matches a value included in that menu. Thus, we can identify that a specific drop-down menu *includes* the user's value, but cannot directly infer which value it is. However, by combining these two attacks and strategically replicating values across a unique combination of elements, we can infer the user's exact value. Last, our attacks bypass specific type-based and size-limit safeguards, by dynamically changing the type and characteristics of elements. This allows us to probe a user's autofill profile for sensitive information (e.g., email, phone, credit card number) without any limits on the number of candidate values that we can test. Our proof-of-concept implementation can probe the user's profile for 100K candidate values in 4-5 seconds on desktops and 8-9 seconds on laptops.

Due to the severity of our attacks, we develop an appropriate countermeasure in the form of a Chrome extension. Our extension leverages the heuristics that we have devised for inferring whether a page's autofillable elements are hidden or masqueraded, to detect and prevent the use of such deceptive techniques. We have also disclosed our findings to the affected browser vendors, in hope that they will incorporate our proposed techniques for better protecting their users. Overall, we find that the inherent disconnect between what is *rendered* by a browser and what is actually *visible* to the user leaves ample room for misuse and deception. We believe that our analysis of how users can be harmed and how to remediate these attacks will facilitate tackling this significant yet understudied privacy threat. In summary, our research contributions are:

- We explore how major browsers handle web forms and present methods that allow adversaries to hide form elements and stealthily exfiltrate highly-sensitive user information. We demonstrate techniques that bypass technical countermeasures and also highlight the limitations of existing user-centric mitigation strategies.
- We conduct an in-depth analysis and present a series of new attacks that exploit flaws and idiosyncrasies in browsers' behavior. We combine all our techniques to demonstrate a novel and severe side-channel attack that infers users' PII even when they are cautious and avoid using autofill.
- We develop a tool that detects the deployment of hidden elements and conduct a large-scale study in the Alexa top 100K, revealing the prevalence of such practices in the wild. Accordingly, we develop a browser extension that detects deceptive forms and prevents the exfiltration of user data.
- To further facilitate research on this topic, we make our code and data publicly available.

2 BACKGROUND AND THREAT MODEL

Browsers have long provided autocomplete suggestions to users, where a website can specify the expected value type for each input field (e.g., last name, address, email) and the browser will assist the user typing by providing suggestions based on values previously entered by the user in fields of this type. Nowadays, all major browsers provide *form autofill* functionality that automatically populates the input fields of a web form with values from the user's *autofill profile*. According to estimations by Google, this functionality can speedup form completion times by 30% [3].

2.1 Browser Autofill

Here we present an overview of various aspects of browsers' autofill functionality and provide additional pertinent details.

Creating autofill profile. A profile is automatically created when a user completes and submits a form for the first time. Multiple autofill profiles can be stored in the user's browser (e.g., a "personal" one with values for home address and mobile phone number, and a "professional" one with the respective work-based values). While a second profile is typically created by the user manually, some browsers (e.g., Chrome) generate it automatically when a submitted form contains values that do not match those of an existing profile.

Triggering autofill. When the user clicks on a form's input field the browser dynamically generates an overlay window that shows the stored autofill profiles, to facilitate selection when multiple profiles exist. This autofill window appears whenever a user clicks on a form field with an autofillable type and at least three fields exist in that form (even if two of those are hidden). In Chrome, in certain cases (e.g., `autocomplete=''first-name''`) the overlay will appear even if the form only has one field. As the user moves the mouse, if it passes (or hovers) over a profile entry in the overlay window, the values that are stored in the profile will appear in the corresponding input fields (we refer to this functionality as *autofill preview*). At that point, the user can select (i.e., click on) an entry from the overlay window to trigger the autofill functionality and

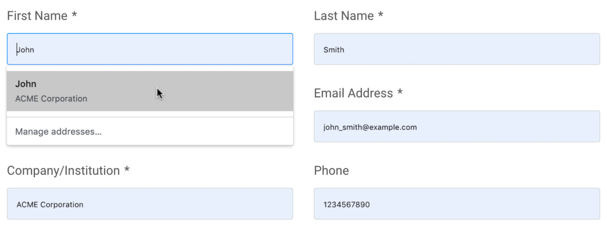
First Name *
John
John
ACME Corporation
Manage addresses...
Last Name *
Smith
Email Address *
john_smith@example.com
Company/Institution *
ACME Corporation
Phone
1234567890

Figure 1: Example of autofill preview functionality.

the browser will populate the form with values from the profile. The user can also click somewhere *outside* of the overlay window for it to "disappear", and no autofill functionality will be triggered.

Value selection. Input fields can specify the type of the data that they expect using the `autocomplete` attribute. Browsers use this attribute as well as attributes such as the field's name, id, placeholder text etc., and employ heuristics to determine or predict the field's type. Apart from these heuristics, as mentioned in the Google Chrome Privacy Whitepaper [1], Chrome also sends some information about the website (i.e., hash of the page's hostname) and the names of the input fields to Google, so as to receive a more accurate prediction of each field's data type based on server-side analysis.

Autofilling hidden elements. Currently, browsers do not sufficiently detect and prevent deceptive practices that use disguised or hidden input fields to exfiltrate sensitive user information in a stealthy way. In more detail, Chrome only avoids autofilling fields that have their visibility attribute set to *hidden* or *collapse*, and those that have the display attribute set to *none*. It does not, however, attempt to detect input fields that are otherwise hidden or disguised (e.g., placed out of the screen, covered by an overlay, having size equal to zero etc.). Such fields are populated along with all the non-disguised ones when the autofill functionality is triggered, as we will discuss in more detail in Section 3.

On the other hand, Firefox has a different approach and fills out *all*[1] input fields, even those that have the visibility attribute set to *hidden*. If the same element appears more than once in the page, irrespectively if it's visible or hidden, Firefox fills only the first one that it finds as it parses the page. The strategy that Firefox follows for protecting users is to show them a message when they start entering a value in a form, informing them about the generic types of the information that will be filled out. However, this approach shifts the burden entirely onto the user, who must somehow infer the true extent to which a website is collecting their information and avoid triggering the autofill process when necessary. We discuss the pitfalls of this design choice and how it does not adequately protect users from deceptive practices in Section 5.

Autofill preview. This functionality, which is enabled when a user hovers the mouse over the autofill overlay window to select one of the profiles, provides a *preview* of what autofill will do, allowing users to make a more informed decision. Specifically, it displays the values of that profile within the form's input fields, as shown in Figure 1. Importantly, the preview functionality does *not* trigger the autofill mechanism and the form's input fields are not really populated with the profile's values. Due to the obvious privacy implications, browsers display these values in overlay fields that are not part of the DOM and are not accessible to the page. Preview functionality is not currently available in mobile devices.

[1]Currently Firefox only autofills address information for users in the US [4].

Handling credit card information. In all browsers, the autofill profile includes basic/contact information (e.g., name, organization, address, email etc.). Payment information such as credit card numbers, which is highly sensitive, is stored separately and stricter mechanisms are employed for autofilling this type of information. In Firefox, autofill for credit cards is currently disabled by default.

When a form includes input fields for both contact and payment information, the autofill functionality that is triggered by a field of one of those categories does not fill out the fields of the other category. In other words, triggering autofill for contact information (i.e., those stored in the autofill profile) only inputs contact information but not payment information; for payment information to be filled it needs to be triggered by another field from that category.

Apart from separating contact and payment information, there are additional requirements for triggering the autofill functionality for credit cards. First, credit cards' autofill works only when the page is loaded over HTTPS. Moreover, in addition to the credit card number input field, the form also needs to include a field for the card's `CVV` code or a field for its expiration date (or month). If none of these additional fields exist in the form, browsers do not autofill the credit card number. Furthermore, unlike contact information for which the page can use drop-down menus with multiple option values instead of input fields, autofill for credit cards is restricted to input fields. That is, browsers do not select the credit card's number from the available options if a drop-down menu is used.

As detailed in Section 4, we have devised an attack that bypasses all of these restrictions and misuses the *autofill preview* functionality to exfiltrate the user's credit card number without triggering autofill and without exposing any visual clues to the user.

Autocomplete Attribute. This attribute specifies the type of information that is expected to be entered in a form element. According to the HTML standard [6], this attribute can also be set to "`off`", indicating that this element should not be autofilled by the browser. We found, however, that all browsers apart from Firefox deliberately ignore this directive and populate those elements.

2.2 Threat Model

In this paper we consider as the attacker any website that uses hidden form input elements, or employs other deceptive techniques, that misuse browsers' autocomplete and form autofill functionalities for conducting privacy-invasive attacks (e.g., exfiltrating user sensitive information in a stealthy manner). In practice, the forms that include such hidden fields, and the JavaScript code needed for carrying out the attacks, can be included due to various motivations; e.g., they may be included in a form deliberately by the publisher so as to avoid explicitly requesting users for personal information (e.g., phone number or street address) since that pushes users away and reduces form conversion rates [29]. The form may also be from a third-party script (e.g., fetched from a marketing company [44]) for tracking users [17, 18, 23]), or even part of a form-skimming campaign on a compromised website [21]. In the context of this work, we do not consider websites that collect user information by means other than exploiting the autofill functionalities.

Table 1: Browsers that autofill form fields that are hidden from the user, based on various concealment techniques.

Techniques	Firefox	Chrome	Brave	Edge	Safari	Opera
CSS Display	✓	✗	✗	✗	✗	✗
CSS Visibility	✓	✗	✗	✗	✗	✗
CSS Opacity	✓	✓	✓	✓	✓	✓
Covered by overlay	✓	✓	✓	✓	✓	✓
Non-effective size	✓	✓	✓	✓	✓	✓
Off-screen placement	✓	✓	✓	✓	✓	✓
Ancestor's overflow	✓	✓	✓	✓	✓	✓

Furthermore, we do not focus on how that exfiltrated information is actually used by the website or how the collected/inferred values are sent to their backend server or third parties. Finally, while such controversial practices may also be used for legitimate purposes (e.g., detecting bots in registration forms), or may be the result of developer implementation bugs, we still consider them indicative of suspicious (if not outright malicious) activities, as sensitive user information is acquired without the user's knowledge or consent.

3 STEALTHY DATA EXFILTRATION

Attackers can use hidden form elements to stealthily exfiltrate sensitive user information when the autofill functionality is triggered. While it is fairly easy for attackers to deploy such an attack, it is also inherently challenging for browsers to detect hidden input fields as there are various techniques for concealing their presence. Some of those techniques are straightforward as they change the style attributes of the elements, while others are more elaborate and non-trivial to detect. Furthermore, autofill functionality can be triggered by visible input elements of *any type*. To avoid raising suspicion, attackers can use visible input elements that require data of a non-sensitive, non-PII type and hide the input fields that require sensitive data. For instance, a form could have a visible element for a user's country, which is not invasive from the user's perspective, while using hidden input fields to obtain the user's home address.

Next we describe various properties and techniques that can be used for hiding HTML elements. Table 1 presents a summary of these techniques and their respective support across major browsers. Specifically, we test whether a browser will autofill a form element whose presence has been concealed using each technique respectively. While one of these techniques has been mentioned in public before, we identify several previously-unknown methods that are effective against *all* the major browsers we test.

CSS display property. The simplest approach for hiding an element is to set its CSS `display` property to `none`. This property completely removes the element and the space it occupies, as if it never existed in the page. Also, this property can be inherited from a parent element. In our experiments Firefox is the only browser that fills elements that have been hidden using this technique.

CSS visibility property. The `visibility` property specifies whether an element should be visible or not. When this property is set to `hidden` the element becomes invisible, but its original space and position in the page layout are reserved. It can be also set to `collapse`, which is treated in the same way as `hidden` for `<input>` and `<select>` elements. Similarly to the `display` property, the `visibility` property can be inherited from a parent element.

CSS opacity property. The `opacity` property specifies the transparency level of the element. When the opacity value is set to 0, the element becomes fully transparent and, thus, invisible to the user. However, this concealment technique does not work for `<select>` tag elements (i.e., drop-down menus), which are visible to the user even when they are transparent. This property is not inherited, but an element cannot be less transparent than its parent. As shown in Table 1 this, and all subsequent methods, work in all the browsers.

Covered by overlay. This trick overlays a non-transparent element on top of the element of reference to completely cover it.

Non-effective size. The element is invisible due to its non-effective size (i.e., width or height equal to zero).

Off-screen placement. We can hide an element that has a fixed or absolute position in the page by moving it out of the device's screen area, using the top, bottom, left, and right properties. This technique has been previously demonstrated by a researcher [5].

Ancestor's overflow. This approach places the element out of the bounds of its ancestor's overflow to make it invisible to the user. This can be implemented in various ways; for example the attacker can set the parent element's height or width equal to zero. Another way, when the ancestor element has an effective size, is to position the element in reference out of the actual ancestor's bounds and set the ancestor's `overflow` property to `hidden`, to disable scrolling functionality for the ancestor element.

Summary. In general, the detection of concealed autofillable input fields is not a trivial challenge for browsers, due to the variety and heterogeneity of methods and properties available for interacting with elements. The list of deceptive techniques that we present above is most likely not exhaustive, and other techniques for hiding the presence of input fields may be feasible.

4 DATA INFERENCE ATTACKS

We present a number of design flaws and idiosyncrasies in the autofill functionality of Chromium-based browsers, and detail a series of attacks that exploit these flaws to bypass existing safeguards. We then demonstrate how these individual attacks can be used as building blocks and chained together to construct a more powerful attack that can be used to infer highly sensitive information (e.g., credit card number) from a user's autocomplete profile. More importantly, this attack completely removes the requirement for users to trigger the autofill functionality, rendering it a severe threat even for more privacy-cautious users that may avoid using autofill.

4.1 Field-Type Mismatch Attack

The majority of user information stored by browsers in autofill profiles is typically populated in `<input>` tag elements. Notable exceptions are the values for the user's *country* and *state*, which are often encountered in websites as `<select>` tag elements. This is due to the limited size of their value space, compared to other types of information that have significantly more potential values.

Despite the fact that most types of information are intended for use with form input elements, browsers do not restrict the use of these types of information from being used in drop-down menus. Furthermore, apart from allowing their use as drop-down menu types, browsers will also automatically select the option matching the value stored in the user's autofill profile (if there is a match), when the autofill functionality is triggered. The only type of input information that browsers will not automatically select from a drop-down menu is the credit card number. In other words, credit card numbers are not autofillable when a `<select>` tag element is used.

This lack of checks and restrictions for types other than that of a credit card means that an attacker can use, for instance, a drop-down menu populated with various email addresses and the browser will select the entry that matches the user's email if it matches one of the options. Moreover, attackers can include up to 200 different drop-down menus of the same type within a given page, thus, increasing the overall number of candidate values matched against the user's profile. If a value that matches the user's original value is found in *any* of the drop-down menus, it will be automatically selected. If a value is found in multiple menus, it will be selected in all of them.

Finally, Chromium-based browsers do not autofill `<input>` elements that have their visibility property set to *hidden/collapse* or their display property to *none*. When, however, the attacker uses drop-down menus instead of input elements, all these browsers select the correct values in the menus when autofill is triggered.

4.2 Autofill Preview Attack

As described in Section 2, browsers provide an autocomplete preview functionality, allowing users to see what values will be autocompleted if the autofill functionality is triggered. In more detail, whenever a user clicks on *any* autofillable element, an overlay window appears showing the various user autofill profiles. The preview functionality is activated if the mouse's cursor passes over *any* part of the autocomplete overlay window associated with one of the profiles. The preview values that are shown in the form will only be entered into the form if the user clicks on the window choosing a profile to be used. Next, we present a side-channel attack against the autocomplete preview, that works in Chromium-based browsers, and allows an attacker to infer a user's information even though this information is never actually written into the form.

Side-channel leakage. While the preview window and displayed values are part of an overlay that is not part of the page's DOM, nor are they accessible through JavaScript, we can detect if a value is previewed in any of the elements by observing their style properties. Through experimentation we identified 22 style properties (such as background-color, border-bottom-color, border-bottom-left-radius etc.) that change when a value is previewed in an element. These properties are accessible by the page's JavaScript. In its simplest form this side-channel allows the attacker to detect that a value from the user's profile is previewed, but it does not reveal the actual value. A critical idiosyncrasy, however, is that these style property changes occur in drop-down menus *only if one of the options in the menu matches the value in the user's profile.*

Value inference. Next, we leverage this behavior to infer the *exact value* in the user's profile through the strategic placement and replication of probing values across multiple drop-down menus in the page. The intuition here is to replicate each candidate value across a *unique* set of drop-down menus, such that each combination of menus is "activated" by exactly one candidate value. This way, when the autocomplete preview functionality is activated, if the browser matches the user's actual value with a value that exists in some specific drop-down menus, the attacker can detect those menus' style changes (even if they are hidden from the user) and infer the user's actual value. It is also important to note that in cases where a user has multiple profiles (e.g., a personal and a work profile) this attack can harvest the information from all profiles if the mouse cursor passes over them in the preview window.

Probing size constraints. As this attack requires the replication of candidate values across multiple drop-down menus, we now explore a strategy for maximizing the number of values that can be probed. Through experimentation we found that Chromium allows at most 200 form elements in a page. We also found that `<select>` elements are limited to 512 entries per drop-down menu. Considering that one element is needed for the user to activate the autocomplete preview functionality, we are left with 199 drop-down menus that can have up to $199 \times 512 = 101{,}888$ values in total.

For the replication of values across drop-down menus, we find that an effective strategy that does not suffer from false positives, is to progressively increase the number of entries per candidate value when there are no other unique combinations left for that number. More specifically, this strategy places $\binom{199}{1} = 199$ unique values without replication (i.e., one in each drop-down menu), $\binom{199}{2} =$ 19,701 values in two drop-down menus each (i.e., 39,402 entries out of the total of 101,888 options), and for the remaining available positions each value will appear in exactly 3 different menus (i.e., (101,888 - (2×19,701) - (1×199))/3 = 20,762 values). With this replication strategy an attacker can probe up to 40,659 unique values in the page that can potentially match the user's value (we present a technique for overcoming this limit in Section 4.3).

Type constraints. The basic version of this attack works for all types of autofillable information except for phone and credit card numbers. For those two types, Chromium has additional safeguards in place that we need to overcome. Specifically, for phone numbers it only autofills the *first* element of that type that it finds in the page. For credit card numbers, as discussed in Section 2, the browser only autofills form input elements, but not drop-down menus.

4.3 Dynamic Element Replacement Attack

To bypass the restrictions imposed on credit card and phone numbers we design two techniques for extending our attack that rely on dynamically changing the form elements in the page.

Phone numbers. When targeting phone numbers, our attack initially places multiple identical *input* elements in the page, including the one that is used to trigger the autofill functionality. All these elements need to be the same, to trick the browser into filling them all at once. When the user clicks on the visible input element we dynamically replace all the remaining identical form elements with drop-down menus, and replicate the candidate values across them as described previously. This allows us to infer the user's exact phone number from the preview functionality. More importantly, this extension to the autocomplete preview attack removes the constraints imposed by the browser on the number of entries that each drop-down menu can have. As a result, our attack can now probe the user's profile for as many phone numbers as we want.

Credit cards. We follow a similar approach for bypassing the restrictions imposed for credit card numbers. We use form input fields of a credit card type; when the user clicks on the form element that will trigger the preview functionality, we dynamically replace them with drop-down menus and populate them with credit card numbers. In that way, the browser is tricked into matching the previewed credit card of the user with the entries in the menus. Again, this allows us to bypass the size limit for the drop-down menus. While we can have arbitrarily large numbers of entries in each menu, we observed that the dynamic replacement of input element with drop-down menus causes a delay that may affect our attack if our page probes for millions of values. This delay is due to the browser parsing all the form elements that are now significantly larger than a normal form element.

An additional detail that we need to handle in this attack is that Chromium changes the style properties of all the *dynamically included* drop-down menus during the autocomplete preview. These properties change even for menus that do not have any value matching those in the user's profile. This could have prevented us from differentiating between the menus, but we have observed that the style properties of the menus that actually have a matching value change back to their default values when the preview ends (i.e., the user moves the mouse cursor away from the autocomplete window). As the properties of the non-matching elements are not restored, we are still able to infer which values the user has previewed.

Scale of attack. Our dynamic replacement of autofillable elements allows us to induce inconsistencies in the browser's behavior, and also removes the limit on the number of options that we can include in the drop-down menus. In fact, the size limitation is removed for all types of information when their corresponding elements are dynamically replaced, not only phone and credit card numbers. When our page probes up to 40K values, the attack is instant and there is no discernible delay, and our system can obtain the values from multiple profiles in the autofill window if the mouse passes over them (see demo video [16]). When further increasing the scale of the attack, we are only limited by two factors: (i) the time required to fetch the form from the web server, and (ii) the time required for the client-side computations to complete.

During our experiments we found that the attacker can easily overcome (i) through compression. Due to the nature of the form's data, `gzip` is particularly effective, allowing us to compress the attack form that probes 150K phone or credit card numbers to approximately 2MB, which matches the median webpage size according to the HTTP Archive [7]. The effect of issue (ii) will vary depending on the capabilities of the client's device. Currently, we have tested our proof of concept implementation on a variety of off-the-shelf laptops and have found that the user needs to stay on the page for 8-9 seconds per 102K probed values (demo video: [15]). When the user is on a desktop machine, we found that the same attack requires 4-5 seconds (demo video: [14]). Attackers can also selectively increase the number of candidate values for users with more powerful machines (e.g., deciding based on the WebGL Renderer attribute [19, 28] or other hardware information). We note that due to the increased browser processing when probing 100K values, if the mouse hovers over multiple autocomplete profiles our attack only infers the value from the last previewed profile.

Reducing search space. While the number of values probed by our attack is not actually constrained by the browser, users are unlikely to stay on a page with a form for a very long time. As such, attackers can further improve the success of their attack by reducing the potential size of the search space for the given type of

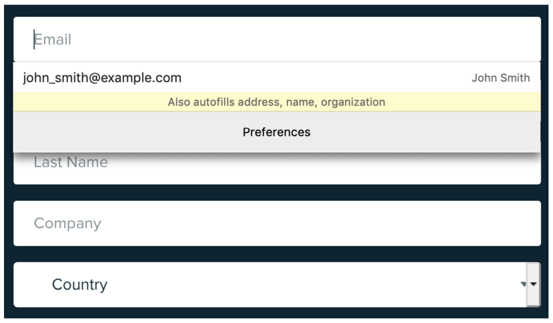

Figure 2: Example notification (yellow strip) shown by Firefox during the autocomplete process.

information they want to infer. For instance, if the attackers want to infer the user's street address, they can leverage the user's IP address and construct a set of probing values for the user's city.[2] Indicatively, NYC which is the most populous city in the United States, has a total of ~102K named geographic places (streets, bridges, rail lines, etc.) [10] which can be easily probed within a reasonable time frame. Similarly, attackers can leverage the IP address to probe phone numbers that start with that city's or state's area code. Probing for 102K values would, thus, allow the attacker to cover ~11.3% of all possible 7-digit combinations for a given area.[3] Attackers trying to exfiltrate credit card numbers could first take advantage of the structure of credit card numbers to generate valid candidate values (e.g., targeting specific banks [25]). Alternatively, attackers can cheaply buy credit card numbers missing the cardholder's name in bulk from underground markets [34] and probe those.

Incognito mode. Our preview attack does not require the user to actually use the autofill functionality, and the autofill window appears whenever the user clicks on any form element of an autofillable type. Also, it infers the values from multiple autocomplete profiles, if the mouse passes over their entry in the preview window. This renders our attack particularly pertinent against cautious users that may have a "decoy" profile with bogus values (e..g, with a pseudonym and a secondary email address that is entered in untrusted websites) or who otherwise avoid using autocomplete. Cautious users may also visit untrusted websites in private (i.e., incognito) mode. While certain browser functionalities differ in private mode, we find that our attacks *are not prevented by incognito*. Currently, the only way users can prevent this attack is to disable the autofill functionality entirely, through the browser's settings.

5 USER-CENTRIC BROWSER MITIGATIONS

Currently, Firefox and Safari are the only major browsers that attempt to mitigate the surreptitious exfiltration of data by enabling users to make more informed decisions. While such an approach can significantly raise the bar for attackers *if designed correctly*, users may still ignore the information showed by the browser. As such, we believe that browsers should also incorporate mechanisms for detecting and preventing form-field concealment techniques, which we describe in Section 8. Nonetheless, for our analysis we

[2]MaxMind reports 80% and 68% accuracy for states and cities respectively [9].
[3]Phone numbers in the US have 10 digits; the first 3 digits denote the area code.

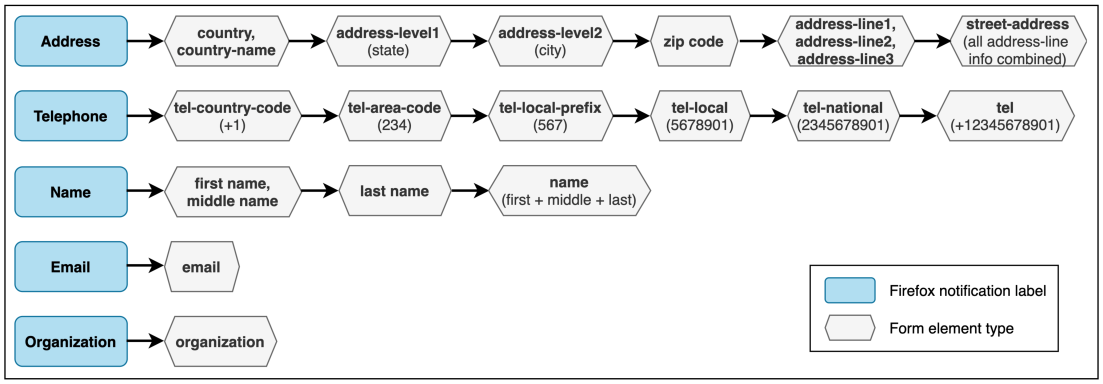

Figure 3: Hierarchy of Firefox's notification labels and the field types covered by each label. The field types are ordered from coarser to more fine-grained and, when needed, additional explanations or example values are provided in a parenthesis.

assume that users take the notification into account, and will focus on how current approaches are still susceptible to deception.

Firefox presents a notification that states what type of information will be autocompleted; an example is shown in Figure 2. As such, theoretically, even if a form element is hidden through some deceptive technique the user will be aware of what information will be disclosed. As can be seen in Figure 3, the notification shown to users can only contain five high-level, coarse-grain labels. This introduces a new avenue for deception; attackers can have a high-level form element that is visible to users (e.g., country or telephone country code) and have a specific, fine-grained element (e.g., street address or phone number) hidden for exfiltrating uniquely-identifying user information. As such, the user will not be able to perceive any discrepancy between the form and the notification message, as all notification labels will correspond to visible form elements. This may also create a false sense of security for users, as the browser shows an explicit warning message that verifies their visual perception of what information the page will obtain.

Safari follows a similar approach, but provides additional fine-grained information. Specifically, Safari's message will explicitly state if the user's first and/or last name will be provided to the form, and also more details about the information from their profile that will be autocompleted. However, while Safari differentiates between home and work address in the notification, it follows the same approach with Firefox where these two coarse-grained labels are used for any of the address-related field types. As such, attackers can include a visible field for generic address information (e.g., country) and have a hidden field that obtains the postal address. At the same time, Safari's design is more complex from a user-interaction standpoint, since clicking on a field other than the name will present a window with 2 different autofill options, one with a generic label and one with all the detailed information that will be autofilled in the form. In this case, the user could likely avoid disclosing the sensitive information to the hidden fields by selecting the first option. As typical behavior is to start from the top of the form [33], it is likely that many users will only see the coarse-grained labels. Nonetheless, since users have the option to see exactly what information will be autofilled through the browser's UI (albeit not in the typical workflow) we consider Safari's design more effective. Nonetheless, explicitly including the fine-grained labels in all the notification windows will allow users to better protect their data. We consider a user study on the effectiveness of these notification messages an interesting future direction.

6 MEASURING DECEPTION IN THE WILD

Here we present our large-scale measurement on the prevalence of autofill-based deception in the wild. We first describe our methodology and practical challenges of identifying the autofillable elements and detecting those that are concealed, and then continue with a detailed analysis of our collected dataset.

Crawling. We have instrumented Chromium (v81) and Firefox (v74) and used Selenium to visit the Alexa Top 100K websites with both browsers. Our system records the autofill information for each input field when visiting a website and the autofill functionality is activated. Upon visiting a website, our crawler identifies all the input fields and drop-down menus in the page and triggers the autofill preview functionality in an automated way. At this point, our instrumented browsers record information about the elements that are being filled out. After that, we use our heuristics to identify whether those autofillable elements are visible to the user or not.

In Chromium, once the crawler clicks on an `<input>` field and the preview window appears, we automatically choose the first saved profile for preview and record the information of autofillable fields. We noticed that there is a slight difference between Chromium and Chrome regarding autofill behavior. Chromium comments out the source code that autofills the company field, while Chrome supports autofilling this field. Thus, we uncommented the code to make Chromium consistent with Chrome for our experiments. In Firefox, we record how it parses the form upon clicking, including which fields are autofillable and their types. Finally, since a page may have multiple forms, and fields in one form may not be able to trigger autofill for those in a different form, we click each `<input>` in the page to ensure that we will record all the autofillable fields.

Detecting Non-Visible Elements. The concealment techniques that leverage specific CSS property values for making input elements invisible (i.e., the first three entries in Table 1) are straightforward to detect. For such cases, our crawler simply checks the values

Table 2: Sites and pages where autofill is triggered, and cases where our system detected hidden elements.

	Sites w/ Autofill	Sites w/ Hidden Field	Pages w/ Autofill Forms	Pages w/ Hidden Autofill
Firefox	21,589	5,295	92,063	8,760
Chrome	31,621	1,843	83,054	2,776

of the corresponding properties. For properties like *display* and *visibility* that are inheritable from ancestors, and the element's *opacity* which can be also affected by the parent's opacity, our system also evaluates those properties for the input fields' ancestors.

To detect fields that are covered by other elements, our heuristics first estimate the position and effective size of every element in the page, and then determine whether any of those elements are placed on top of others. Additionally, we also check the *opacity* value of the overlay elements, as they need to be non-transparent to effectively cover the elements that lie beneath them. In a similar fashion, by estimating the position and size of the input elements, our heuristics determine whether those are positioned outside of the screen boundaries, or outside of their ancestors' overflow boundaries.

Additional visual obstacles. In practice, additional aspects of modern websites can lead to mislabeling. As such, our system faces additional challenges for accurately assessing the presence of deceptive techniques, as common web development practices can misleadingly make it appear as though an element is being concealed. Next we provide more details on our approach and heuristics for handling such cases.

Popup overlays. These are typically a window, lightbox, or full-screen takeover, typically layered on top of the page's content; we need to remove these popups as they render the pages' content inaccessible. We detect them and remove them by setting their style attribute to "`display:none !important`".

Website navigation and header. For every `<input>` and `<select>` element in the page, that are returned by the getElementsByTagName() method, we scroll the page using the `scrollIntoView()` method to make the visible ones appear on the screen. We have observed that some elements might still be temporarily covered by the website's navigation bar, banner, or ads. For this reason, if an element is still not in view after the initial scrolling, we scroll further upwards and downwards and check if it comes in view.

Cookie consent overlays. During our experiments we observed that when such overlays are removed, some form elements may also be removed. For this reason we do not remove them but scroll the page to make all visible elements appear. It is noted that the cookie consent overlay is different from the popup overlay. Popup overlays render the page content inaccessible and, thus, need to be removed. In contrast, the cookie consent overlay is typically a small banner at the top or bottom of the page [45] and does not affect accessibility to the page. While our initial plan was to programmatically interact with the consent overlays and accept the use of cookies, we found that the diversity of web pages, the differences in overlay behaviors (e.g., some may induce navigation to a different page) and the variance in text language lead to complications that prevent us from adequately covering all cases. Instead, we found that simply scrolling was sufficient, as visible elements are brought into view.

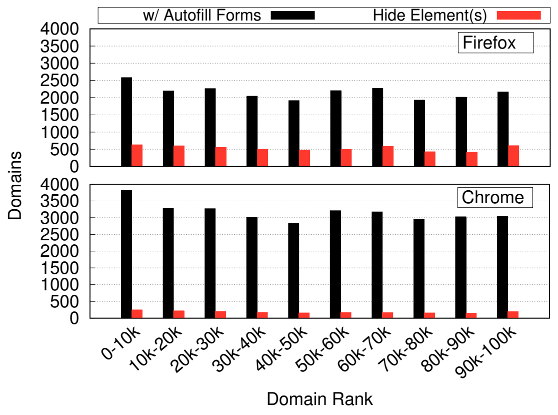

Figure 4: Domains w/ autofillable forms and hidden fields for Firefox and Chrome, grouped based on Alexa rank.

6.1 Measurements

Our crawling lasted from 11/19/2019 to 12/3/2019, where we visited the landing pages of the Alexa Top 100K websites (using the list from 11/13/2019), from where we followed all links to other pages on the same domain. We chose to follow this approach as forms (e.g., sign up, subscription, etc) are typically accessible from the landing page. This resulted in more than 214K pages that have forms with autofillable fields. We also note that the numbers that we present only take into account hidden elements that are actually autofilled by each browser; if the form contains hidden elements that are, for any reason, not filled by the browser we do not include them in our analysis as they do not affect users.

We break down the results from this process in Table 2. In general, we see a considerable difference in the overall numbers across browsers; this is because Chrome and Firefox use different heuristics for parsing forms, detecting field types, and deciding whether to autofill a given field. As a result, Chrome is far more aggressive in autofilling forms (e.g., by ignoring the autocomplete attribute as discussed in Section 2) resulting in 46.5% more sites where a form is autofilled compared to Firefox, and 63.1% more forms overall. On the other hand, Firefox does not have any checks for preventing hidden elements from being filled, resulting in an almost 3*x* increase in the number of autofilled forms that contain hidden elements. While this lack of checks is likely due to the adoption of the autofill notification shown to users (Section 5), their current approach does not adequately protect users as we detail below.

Domain popularity. Figure 4 provides more details on the number of websites with forms that are autocompleted and their use of hidden elements, based on their Alexa rank. While we see higher numbers for both browsers in the most popular bracket (the top 10K), there is no consistent pattern for the remaining groups. Furthermore, in Firefox we detect between 414-631 domains in all bins, while Chrome fluctuates between 152-250. We also break down these sites based on their top-level domain and find that around 56.5% of them in both browsers belong to "`.com`". Somewhat surprisingly we find that "`.edu`" and "`.org`" are among the top 4 for both browsers. We also identified 73 and 24 "`.gov.*`" domains in Firefox and Chrome, respectively, that included hidden fields.

Table 3: Number of domains and hidden autofillable form fields for each deceptive technique seen in the wild.

Technique	Firefox		Chrome	
	Domains	Fields	Domains	Fields
display_none_ancestor	9,177	12,675	692	1,111
display_none	1,271	2,134	468*	758*
covered	1,129	1,554	769	1,119
visibility_hidden	109	211	117*	143*
off_screen	94	131	249	497
off_ancestors_overflow	88	131	91	144
non_effective_size	61	74	53	75
transparent_ancestor	23	42	75	123
transparent	11	11	27	43
visibility_hidden_ancestor	1	1	-	-

*Chrome only autofills `<select>` fields hidden with these techniques.

Concealment techniques. In Table 3 we provide detailed statistics about the techniques that we detected being used in the wild. Leveraging the `display` attribute (either directly or though the element ancestors') is the most prevalent approach, while the use of overlays to cover fields is also widespread. Placing fields off-screen is also fairly common in Chrome, but seen less widely in Firefox. Interestingly, while leveraging the `display` property of ancestors is the most common technique, we do not see the same approach used often with the `visibility` attribute; in fact, we only identify one such case in Firefox. While Chrome does not autofill input fields hidden through `display:none` and `visibility:hidden`, it does autofill drop-down menus hidden that way. Finally, we analyzed the hidden drop-down menus detected by our system and found that all of them were collecting types commonly associated with drop-down menus – in other words, we did not identify any domains using our field-type mismatch technique.

Hidden Elements. Figure 5 shows how many hidden elements were autofilled across domains. This is the aggregate number from all the forms detected on each domain. Again, we see that Firefox autofills more hidden elements due to the lack of any invisibility checks, while the vast majority of websites have less than 10 hidden elements. The largest number of hidden fields within a single domain was 45 and 256 for Chrome and Firefox respectively.

Next we focus on *what* type of user information websites are obtaining from hidden autofilled form fields. As shown in Figure 6, we see a variety of types being targeted. While more generic fields like country and state are commonly collected, we also see a large number of sites collecting more sensitive and user-identifying information. For instance, the user's first and last name are the second and third most often collected values in Firefox, while the user's email address, phone number and address-line1 are popular targets in both browsers. In more detail, we find that 11.2% and 10.9% of the domains collect the user's first and last name in Firefox, while in Chrome that drops to 3.2% and 2.8%. This significant skew in the relative percentages in Chrome is due to the large number of domains with state and country information that were autofilled. As aforementioned, Chrome does not enforce its visibility constraints to drop-down menus, resulting in this deviation. Finally, we find that 15 domains obtain the user's credit card expiration date in Chrome. In Firefox, we detected 2 domains with hidden fields for the credit card expiration date and the cardholder's name. Based on our findings we believe that attackers have not yet discovered our techniques for bypassing the constraints for credit card numbers.

Anti-bot detection. Certain sites may use hidden fields as a way of detecting bots, assuming that users cannot fill in fields that they do not see. However, this assumption is incorrect, due to the autofill functionality of browsers. While we cannot definitively infer the motivation behind the inclusion of hidden fields, we identify websites where all hidden fields have visible counterparts as potential instances of this strategy. We find 272 (14.75%) such sites for Chrome and 305 (5.76%) for Firefox. This difference is due to the fact that Firefox will only fill out *one* field for a given type; as such, in all these cases only the hidden element gets filled but the visible one does not (due to the hidden one being first or the visible one having the `autocomplete` attribute set to "`off`"). Overall, even if the motivation in these cases is not malicious, it still reflects a privacy-invasive practice where users unknowingly leak private data. For instance, a Chrome user may decide to delete certain visible fields that were autofilled but still end up disclosing that data. Similarly, a Firefox user would see an empty visible field of a given type but still have their information exposed in the hidden field.

Firefox label-granularity deception. Next, we analyze all the data collected from our Firefox crawl, and explore whether domains are potentially misusing the coarse granularity of the notification labels. In more detail, we identify domains where the *most specific* visible element of a given category (see Figure 3) is *coarser* than any hidden elements of the same category. An example of such a case would be a domain where the user is shown a "country" element (part of the Address category) while an element for "street-address" (also part of the Address category) is hidden. Overall, we find that 874 domains (16.5%) exhibit this behavior. If we filter out cases where the *combination* of visible fields equals the granularity of information obtained from hidden fields (e.g., `first-name` and `last-name` are visible, and `name` is hidden) we are still left with 650 domains where the Firefox warning message is inadequate at informing users about the true extent of the PII information they are divulging to the website. Figure 7 shows the percentage of domains, out of all the domains that hide elements, exhibiting this behavior. In aggregate, 12.3% of all autofillable forms with hidden fields deceive users into divulging more specific (i.e., *identifying*) information than what they intend, or expect, based on the fields that are visible to them and Firefox's notification. While we cannot infer if these domains are purposefully or inadvertently exploiting the label-granularity mismatch, the inherent limitations of this approach are currently exposing users to considerable risk.

Detection accuracy. To assess the effectiveness of our heuristics we randomly selected 20 pages per concealment technique, where we had detected hidden elements, and manually inspected their code to establish whether these elements are actually hidden or not. As such, we have manually inspected a total of 140 pages (70 pages with hidden elements autofilled in Chrome, and 70 in Firefox). Since our heuristics are the same for all pages, irrespective of the browser, our results are reported in aggregate.

These 140 pages have 828 autofillable input elements in total, where 282 of them are detected by our heuristics as hidden. Through manual inspection we have verified that all of these 282 elements

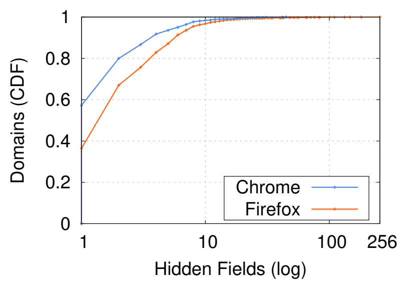

Figure 5: Hidden elements across all forms detected within each domain.

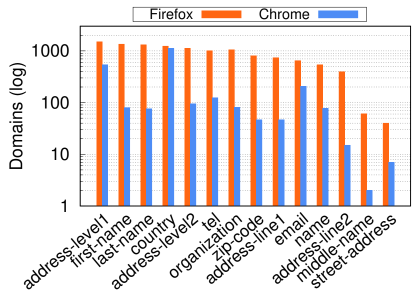

Figure 6: Most common types of hidden autofilled fields.

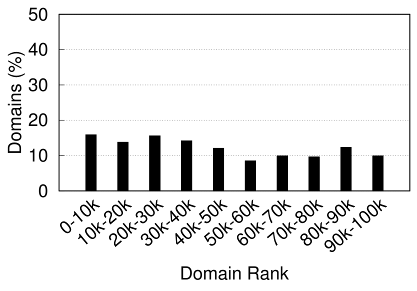

Figure 7: Fraction of domains with hidden, autofillable elements for which Firefox's coarse labels are insufficient.

are indeed not visible to the user (TP rate: 100%) and that there are no other hidden elements in any page that our heuristics failed to detect (FN rate: 0%). From the 546 input elements that are visible, our heuristics incorrectly labeled 6 elements as hidden (FP rate: 1.09%), in 3 of the 140 pages. In the first case, the 4 elements that we detect as hidden are fully transparent, but the developer used the CSS *::before* selector to include visible pseudo-elements on top of them, to make them noticeable to the user. In the second case, a popup appears after the user scrolls the page, and covers one of the input elements, which we detect as hidden. Last, in the third case, the developer used a list inside of an input element to implement a *custom* drop-down menu, which our heuristics considered as being covered by another element.

7 CASE STUDIES

In this section we present some interesting examples of hidden form field use cases that we identified during our large-scale study.

E-commerce. We identified a Brazilian e-commerce retail store that has a visible zip-code input field on every product page, offering to calculate the shipping and delivery time. However, the form also includes hidden autofillable elements that obtain the user's full name, email address, and phone number.

Marketing. We identified 27 domains that used code from a "retail digital marketing automation platform" that specializes in email marketing and mobile messaging. While this marketer's code creates a visible email input field, so users can subscribe to newsletters, their code also creates two autofillable hidden fields that obtain the user's first and last name. While the `autocomplete` attribute is set to `off`, this is ignored by Chromium-based browsers. Furthermore, this part of the form has an explicit comment urging developers to not delete these fields as they are intended for use by the marketing system. Finally, the form is submitted directly to the marketer's servers and not to the first-party origin.

Anti-bot detection. During our experiments we found that certain sites employ MailChimp, a popular email marketing company that uses hidden form elements as a method of detecting bot-driven signups. While such automated actions are typically prevented through CAPTCHAs, recent studies have demonstrated effective attacks even against the most prevalent CAPTCHA services [36, 47]. This may be the motivation behind the deployment of additional form-related defenses. However, apart from the obvious privacy implications of this practice, where users may unintentionally disclose information that they are not willing to share with a given domain, this also introduces additional usability issues that affect users and vendors alike. As Chrome ignores the `autocomplete` attribute, Chrome users' autofill functionality is resulting in false positives that trigger the anti-bot detection, thus, preventing users from registering [8]. As this can easily drive users away, there is a direct negative impact for the users as well as the web sites.

Firefox label granularity. As described in Section 5, the coarse-grained nature of the warning message shown by Firefox creates an avenue for misuse, as websites can reveal generic, non-identifying, fields and then hide specific, user-identifying fields of the same category. One such example is the US News website which has a mandatory visible field for the ZIP code, but has various more-specific address-related hidden elements, including `street-address`.

8 COUNTERMEASURE

The attacks that we demonstrate pose a significant privacy threat as they enable the stealthy exfiltration of sensitive user data. As browser vendors may not adequately address this issue, we have developed an extension for Chrome that can mitigate our attacks.

Operation. After a page is loaded, our extension parses the DOM to identify all the form elements and leverages the heuristics that we have devised (Section 3) for identifying whether any of those are hidden from the user. Then, based on its mode of operation it either shows a warning to the user about these elements (*lax mode*) or automatically removes them from the page (*strict mode*). We decided to remove these elements from the page instead of setting their `autocomplete` attribute to "`off`", as Chrome ignores this attribute and autofills those elements anyway.

Methodology. Our extension first identifies all the `<input>` and `<select>` tag elements of each form in the page, and determines which of those are autofillable. For this step our extension checks whether a form has at least 3 such elements or whether it has only 2 elements which use the `autocomplete` attribute to specify the expected value type (e.g., autocomplete="name"). It also checks if there is at least one visible input field, which is needed for triggering autofill. Moreover, our extension uses the same regular expressions as Chromium for determining each element's autofill type.

For the autofillable elements, our extension uses the method `getComputedStyle()` to retrieve and check their display, visibility and opacity attributes, as well as the attributes of their parent nodes. To detect if an element is placed outside the boundaries of the screen or whether it is covered by another element, we use `getBoundingClientRect()` to get its size and position relative to the `viewport`. To detect covered elements we calculate their center points and use `elementFromPoint()` to get the top-most element at that point, and check against those and their ancestors' properties and position. Similarly we check the position of their ancestors and whether the elements are placed out of their overflow bounds. Finally, we decided to preemptively detect and remove deceptive elements when the page is loaded instead of when the user clicks on a form element (i.e., strict mode), to avoid potential race conditions that would allow a page to obtain the user's data.

Performance. To measure the overhead imposed by our extension, we randomly chose 500 pages that have hidden autofillable elements from our large-scale study and visit them with our extension in place. We observe that it takes only 13.09*ms* on average (*median*: 12.78*ms*, *75th*: 15.41*ms*, and *95th*: 20.36 *ms*), to parse the page and run our heuristics for detecting all the hidden form elements. As such, while our countermeasure should optimally be incorporated by the browser, the performance impact of our extension is negligible and will not affect the user's experience.

Detectability. Browser extensions can be detected based on the uniqueness of their behavior when interacting with a page. This enables browser fingerprinting [39, 43] and the inference of sensitive data [27]. Our extension is detectable, as malicious websites can infer its presence by including hidden form elements and detecting if they get modified. While our extension can be detected and used as part of a browser fingerprint, we consider that the more specific and uniquely identifying user information that is obtainable when our countermeasure is not in place far outweighs the risk of installing our extension.

9 DISCUSSION AND FUTURE WORK

Here we further discuss additional aspects of our work, and also highlight limitations and possible future research directions.

Attack likelihood. Our work demonstrates two types of attacks with different prerequisites. The first type, which aims to obtain user PII through hidden form fields, requires the victim to use autofill. While research exists on how to improve the usability of forms to reduce the likelihood of users leaving prior to completion [33], sites do not actually need the user to submit the form since they can read the autofilled data. Nonetheless, one could argue that more cautious users will not even trigger autofill on less trustworthy sites. In such cases we can deploy our *preview* attack – users only need to click on a form field and move their mouse downwards so as to momentarily pass over the preview window. Furthermore, recent work has demonstrated how click interception remains a threat [46]; such techniques can be combined with our second attack to infer the user's sensitive information once they are tricked into clicking the form field. Similarly, recent studies have explored the use of deception in a different context (e.g., shopping sites [31], mobile apps [22]). While most *dark patterns* they explore are not relevant to our work, techniques that involve the aesthetic manipulation of the UI (e.g., disguising ads) could potentially be applicable.

Attack stealthiness. Common classes of deceptive attacks typically leave behind some form of visual clues that users can detect (e.g., the URLs in phishing attacks). While attacks can leverage behaviors in certain browsers to further obscure these clues [30], the clues are still present in parts of the page that average users have been increasingly conditioned to check. Additionally, while more sophisticated campaigns will trick the average user [37], typical phishing attacks can be detected by average users [13]. On the other hand, the attacks we demonstrate in this paper do not leave such visual clues behind; unless users actually inspect the web page's source code there is no other way to identify the use of such deceptive techniques. This renders our techniques a particularly stealthy and effective class of privacy-invasive attacks against users.

Root cause and mitigation. The underlying issue that enables the attacks presented in this paper, is the inherent challenging task for browsers to truly infer if something is visible to a user or not. The disconnect between the browser's view of a page and the user's visual perception, leaves ample room for misuse. While we have identified several techniques for hiding elements, it is likely that even more advanced techniques exist (e.g., color-based deceptive techniques that make elements and text blend-in with the background). As such, we will continue to augment our browser extension as new techniques are identified. However, the root cause from which these issues arise will render incomplete any defense that is purely technical. More specifically, we believe that clearly informing users what information will be provided to a form using precise fine-grained labels, can better equip users against this privacy threat. Krol and Preibusch [32] showed that security or privacy warnings lead to a reduced disclosure of sensitive data in web forms, but the notification information shown by Safari and Firefox is vastly different in nature (i.e., no alarming language and not presented as an actual warning). As prior work on security indicators and warnings has extensively demonstrated [12, 42], the effectiveness of such practices can be affected by a multitude of factors. A study that explores how users' behavior is affected by these notifications is out of the scope of our work, but we consider it an important and interesting direction for future work.

Crawling coverage. Our large-scale study sheds light on the prevalence of element-hiding techniques in the wild. As we followed a depth-of-one approach, our measurements likely present a *lower bound* on the use of deceptive techniques, as web sites may include additional forms that are not directly accessible from the landing page. Furthermore, websites may already be leveraging other concealment techniques that are not detectable by our system. Finally, we do not have access to post-login pages, which may result in our crawler missing additional forms.

Browser monoculture. Microsoft's recent decision to build Edge off of Chromium attracted criticism as it exacerbates the monoculture issue. Our attacks highlight this risk, as all the flaws that we exploit for our most invasive attack are present in several major browsers (Chrome, Edge, Opera) including more privacy-oriented ones (Brave) due to their reliance on the same underlying engine. Another issue is the seeming unwillingness of certain browsers to independently tackle vulnerabilities that stem from (or also affect) Chromium, waiting instead for these issues to be addressed by the Chromium team. For instance, Edge explicitly considers flaws that also affect Chrome out of scope of their bug bounty program.

Password managers. While browsers are our main focus in this work, we also investigate how two popular password managers (LastPass, 1Password) handle hidden elements when they autocomplete forms and find that they are similarly vulnerable to browsers. Specifically, 1Password does not fill elements only if they are hidden by setting the CSS `display` property to `none`. LastPass does not fill elements hidden using `display:none` and `visibility:hidden` (strangely, it fills them if `visibility` is set to `collapse`). As the LastPass extension has over 10 million downloads in the Google Chrome store, and the 1Password extensions account for over 1.6 million downloads, their autofill behavior is currently exposing many users to considerable privacy threats.

Disclosure and ethics. Due to the severity of our attacks, we disclosed our techniques and findings to the affected browser vendors; Chromium is currently working towards patching their system. We also note that the experiments were conducted on the authors' machines using test browser profiles; no external users participated, or were targeted, during our experiments.

10 RELATED WORK

To the best of our knowledge, this paper presents the first systematic and comprehensive analysis of the autofill functionality available in all major browsers. Our research demonstrates new attack vectors that exploit this widely-used functionality, and highlights the severe privacy threat that it poses to users. In this section we present an overview of prior works that are related to autocompletion.

Web Forms. A series of blog posts by researchers [24, 40] and a recently published paper [11], highlighted how credentials and credit card numbers can be obtained through the browser autocomplete functionality. In [24] they show that third-party scripts included in a page can inject a hidden login form and leverage the browser password manager's autocomplete functionality to obtain the user's credentials for that particular website. In [40] they show that third-party session-replay scripts included in the page can obtain the information of a form, regardless of that information being entered manually or autofilled by the browser. Even though these approaches leverage the autocomplete functionality, the outlined issues stem from the inclusion of third-party scripts that are not restricted by the Same-Origin-Policy (SOP), and thus can access the information that a user provides willingly to the first party. Especially in the case of credit cards, the exfiltration that is described in [11, 40] is possible when the user intentionally provides this information to the first party. Our work, however, demonstrates new techniques that remove such constraints and also comprehensively explores the exfiltration of all types of PII from the user's profile.

In another direction Starov et al. [38] and Chatzimpyrros et al. [20] showed, respectively, that *contact* and *registration* forms can expose users' PII to third parties. Kapravelos et al. [26] showed that malicious extensions can steal sensitive data, such as passwords and email addresses, from web forms. On the other hand, in our work we consider that the first-party may also be malicious, and focus on devising techniques that exploit flaws in the autofill and preview functionality, for exfiltrating sensitive information that the user never intended to disclose. To make matters worse, our data inference attacks do not actually require the user to enter information in the form or even use autofill.

Password managers. In a different line of work that focused on password-autocompletion, Silver et al. [35] explored the autofill policies employed by 10 different browser, mobile, and third-party password managers and identified how they differ across them. They also investigated how the autofill functionality can be leveraged by third-parties for stealing user passwords. Their scenario focused on a rogue WiFi router that injects login forms in multiple frames in a page, which are autofilled by password managers, and then malicious scripts can read the forms. In similar work, Stock et al. [41] investigated how the password managers of all major browsers behave with regards to login form autocompletion, and how they can be misused by a XSS attacker that can run malicious code in the context of another site.

Compared to our work, these studies focus on how the autocomplete functionality of password managers can be used to steal the user's credentials. No prior work attempts to explore how form autofill can be used for stealthily exfiltrating various types of PII from the user's profile. While browsers are the main focus of our work, we found that two of the most popular password managers are also susceptible to a plethora of our element-hiding techniques.

11 CONCLUSIONS

In this paper we presented the first comprehensive evaluation of the privacy threat that browsers' autofill functionality poses to users. As a starting point, we identified various techniques for hiding the presence of form fields that are automatically filled by major browsers. These techniques can be misused for stealthily obtaining sensitive information, unbeknownst to users that leverage autofill for its convenience. Our subsequent large-scale study revealed that such deceptive practices are commonplace in the wild, as we found that 5.8% of all forms that are autofilled by Chrome contain at least one hidden field. More importantly, filling out hidden elements is only the first flaw in browsers' autofill functionality. Our in-depth analysis revealed a series of flaws and idiosyncrasies that allowed us to bypass all existing safeguards that protect the information in users' autocomplete profiles. While all these new attacks constitute important privacy risks individually, when combined they enable a far more egregious attack that exploits the preview functionality and can infer the personal information of cautious users that decide against using autofill. This attack works against all Chromium-based browsers, highlighting the implications of the monoculture issue affecting the browser ecosystem. The severity of our findings prompted us to create a countermeasure to better protect users until browser vendors address all the issues that we have reported. Overall, while autofill is a major convenience for users, we hope that our work sheds light on the significant privacy-utility tradeoff it introduces and allows users to better protect their data.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the DARPA ASED Program and AFRL (FA8650-18-C-7880), and NSF (CNS-1934597). Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government.

REFERENCES

[1] Google chrome privacy whitepaper. https://www.google.com/chrome/privacy/whitepaper.html.

[2] Google developers - help users checkout faster with autofill. https://developers.google.com/web/updates/2015/06/checkout-faster-with-autofill.

[3] Google developers - help users checkout faster with autofill. https://developers.google.com/web/updates/2015/06/checkout-faster-with-autofill.

[4] Mozilla firefox features. https://wiki.mozilla.org/Firefox/Features/Form_Autofill#Feature_Availability.

[5] The guardian - browser autofill used to steal personal details in new phishing attack, 2017. https://www.theguardian.com/technology/2017/jan/10/browser-autofill-used-to-steal-personal-details-in-new-phising-attack-chrome-safari.

[6] HTML Living Standard - Last Updated 26 February 2020. https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#attr-fe-autocomplete, 2020.

[7] Http archive - state of the web, 2020. https://httparchive.org/reports/state-of-the-web.

[8] Mailchimp - troubleshooting the embedded signup form, 2020. https://mailchimp.com/help/troubleshooting-the-embedded-signup-form/.

[9] Maxmind db, 2020. https://www.maxmind.com/en/geoip2-city.

[10] Nyc department of planning, 2020. https://www1.nyc.gov/site/planning/data-maps/open-data.page#snd.

[11] Gunes Acar, Steven Englehardt, and Arvind Narayanan. No boundaries: data exfiltration by third parties embedded on web pages. In *Proceedings of the 20th Privacy Enhancing Technologies Symposium (PETS)*. Sciendo, July 2020.

[12] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 257–272, 2013.

[13] Mohamed Alsharnouby, Furkan Alaca, and Sonia Chiasson. Why phishing still works: User strategies for combating phishing attacks. *International Journal of Human-Computer Studies*, 82:69–82, 2015.

[14] Anonymized. Preview demo: Desktop, 100k credit card values. https://vimeo.com/412514626/fb485212ad.

[15] Anonymized. Preview demo: Laptop, 100k email address values. https://vimeo.com/412447440/e753a2cf4c.

[16] Anonymized. Preview demo: Multiple autofill accounts. https://vimeo.com/414161536/c3a9e00f1c.

[17] Reuben Binns, Jun Zhao, Max Van Kleek, and Nigel Shadbolt. Measuring third-party tracker power across web and mobile. *ACM Transactions on Internet Technology (TOIT)*, 18(4):1–22, 2018.

[18] Tomasz Bujlow, Valentín Carela-Español, Josep Sole-Pareta, and Pere Barlet-Ros. A survey on web tracking: Mechanisms, implications, and defenses. *Proceedings of the IEEE*, 105(8):1476–1510, 2017.

[19] Yinzhi Cao, Song Li, Erik Wijmans, et al. (cross-) browser fingerprinting via os and hardware level features. In *NDSS*, 2017.

[20] Manolis Chatzimpyrros, Konstantinos Solomos, and Sotiris Ioannidis. You shall not register! detecting privacy leaks across registration forms. In *Computer Security*, pages 91–104. Springer, 2019.

[21] Graham Cluley. Hackers' malicious script skimmed credit card details off robert dyas website, 2020. https://www.grahamcluley.com/hackers-robert-dyas/.

[22] Linda Di Geronimo, Larissa Braz, Enrico Fregnan, Fabio Palomba, and Alberto Bacchelli. Ui dark patterns and where to find them: A study on mobile applications and user perception. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, 2020.

[23] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1388–1401, 2016.

[24] Gunes Acar. Freedom To Tinker - No boundaries for user identities: Web trackers exploit browser login managers. https://freedom-to-tinker.com/2017/12/27/no-boundaries-for-user-identities-web-trackers-exploit-browser-login-managers/, 2017.

[25] Brendan Harkness. Anatomy of a credit card, 2020. https://www.creditcardinsider.com/learn/anatomy-of-a-credit-card/.

[26] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 641–654, 2014.

[27] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. Carnus: Exploring the privacy threats of browser extension fingerprinting. In *27th Annual Network and Distributed System Security Symposium.* The Internet Society, 2020.

[28] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 878–894. IEEE, 2016.

[29] Lindsay Liedke. Wpforms blog - online form statistics & facts for 2020, 2020. https://wpforms.com/online-form-statistics-facts/#form-conversions.

[30] Meng Luo, Oleksii Starov, Nima Honarmand, and Nick Nikiforakis. Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 149–162, 2017.

[31] Arunesh Mathur, Gunes Acar, Michael J Friedman, Elena Lucherini, Jonathan Mayer, Marshini Chetty, and Arvind Narayanan. Dark patterns at scale: Findings from a crawl of 11k shopping websites. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–32, 2019.

[32] Sören Preibusch, Kat Krol, and Alastair R Beresford. The privacy economics of voluntary over-disclosure in web forms. In *The Economics of Information Security and Privacy*, pages 183–209. Springer, 2013.

[33] Mirjam Seckler, Silvia Heinz, Javier A Bargas-Avila, Klaus Opwis, and Alexandre N Tuch. Designing usable web forms: empirical evaluation of web form improvement guidelines. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1275–1284, 2014.

[34] Anisha Sekar. Stolen credit card numbers, 2015. https://www.nerdwallet.com/blog/credit-cards/stolen-credit-card-numbers/.

[35] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. Password managers: Attacks and defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium (USENIX Security 14)*, SEC'14, USA, 2014. USENIX Association.

[36] Suphannee Sivakorn, Jason Polakis, and Angelos D. Keromytis. I am robot: (deep) learning to break semantic image captchas. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy*, EuroSP '16, 2016.

[37] Emily Stark. The urlephant in the room. 2019.

[38] Oleksii Starov, Phillipa Gill, and Nick Nikiforakis. Are you sure you want to contact us? quantifying the leakage of pii via website contact forms. *Proceedings on Privacy Enhancing Technologies*, 2016(1):20–33, 2016.

[39] Oleksii Starov and Nick Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 941–956. IEEE, 2017.

[40] Steven Englehardt. Freedom To Tinker - No boundaries: Exfiltration of personal data by session-replay scripts. https://freedom-to-tinker.com/2017/11/15/no-boundaries-exfiltration-of-personal-data-by-session-replay-scripts/, 2017.

[41] Ben Stock and Martin Johns. Protecting users against xss-based password manager abuse. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 183–194, 2014.

[42] Christopher Thompson, Martin Shelton, Emily Stark, Maximilian Walker, Emily Schechter, and Adrienne Porter Felt. The web's identity crisis: understanding the effectiveness of website identity indicators. In *28th USENIX Security Symposium USENIX Security 19)*, pages 1715–1732, 2019.

[43] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. Everyone is different: client-side diversification for defending against extension fingerprinting. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1679–1696, 2019.

[44] Tobias Urban, Martin Degeling, Thorsten Holz, and Norbert Pohlmann. Beyond the front page: Measuring third party dynamics in the field. In *Proceedings of The Web Conference 2020*, pages 1275–1286, 2020.

[45] Christine Utz, Martin Degeling, Sascha Fahl, Florian Schaub, and Thorsten Holz. (un) informed consent: Studying gdpr consent notices in the field. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 973–990, 2019.

[46] Mingxue Zhang, Wei Meng, Sangho Lee, Byoungyoung Lee, and Xinyu Xing. All your clicks belong to me: investigating click interception on the web. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 941–957, 2019.

[47] Binbin Zhao, Haiqin Weng, Shouling Ji, Jianhai Chen, Ting Wang, Qinming He, and Reheem Beyah. Towards evaluating the security of real-world deployed image captchas. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, pages 85–96, 2018.