vWitness: Certifying Web Page Interactions with Computer Vision

He Shuang
University of Toronto
he.shuang@mail.utoronto.ca

Lianying Zhao
Carleton University
lianying.zhao@carleton.ca

David Lie
University of Toronto
david.lie@utoronto.ca

Abstract—Web servers service client requests, some of which might cause the web server to perform security-sensitive operations (e.g. money transfer, voting). An attacker may thus forge or maliciously manipulate such requests by compromising a web client. Unfortunately, a web server has no way of knowing whether the client from which it receives a request has been compromised or not—current "best practice" defenses such as user authentication or network encryption cannot aid a server as they all assume web client integrity.

To address this shortcoming, we propose vWitness, which "witnesses" the interactions of a user with a web page and certifies whether they match a specification provided by the web server, enabling the web server to know that the web request is user-intended. The main challenge that vWitness overcomes is that even benign clients introduce unpredictable variations in the way they render web pages. vWitness differentiates between these benign variations and malicious manipulation using computer vision, allowing it to certify to the web server that 1) the web page user interface is properly displayed 2) observed user interactions are used to construct the web request. Our vWitness prototype achieves compatibility with modern web pages, is resilient to adversarial example attacks and is accurate and performant—vWitness achieves 99.97% accuracy and adds 197ms of overhead to the entire interaction session in the average case.

I. Introduction

Web applications are increasingly used to implement security-sensitive services such as financial transactions, access to social services and e-health. These services are sensitive because they allow a user to make important and in some cases, irrevocable actions that may affect the user's financial well-being, freedom and safety. The security of these services rests on the assumption that requests leaving user clients are intended by the human user. This assumption depends on two factors: first, that we can correctly authenticate the principal making the request and second, that the integrity of the user-page interaction is maintained.

While web authentication has been well-studied (e.g. passwords and other methods [1]–[4]), securing user-page interaction is an umbrella definition that covers three goals [5], [6]: the proper page rendering, the proper processing of user inputs and the proper construction of outgoing requests. *Violating any one of the goals allows an attacker to send out user-unintended requests.* The following are examples of user-page interaction attacks: clickjacking [7] shows a deceptive user interface (UI) to trick a victim user into performing unintended actions, click fraud forges user inputs on web elements (e.g.

clicks) for monetary benefits and cross-site request forgery [8] crafts user-unintended requests to remote servers.

Existing defenses for user-page interaction assume an unprivileged/remote attacker and rely on client-side enforcement such as browser-based flags (e.g. x frame options [9] for page overlaying) and policies (e.g. cross-origin resource sharing [10], content security policy [11], sub-resource integrity [12] for unauthorized code execution). These defenses can be bypassed by client-side malware [13], [14]. For example, Scranos [15], a web-targeting client-side rootkit, attacks remote servers by forging user interactions, chaining into sophisticated attacks (e.g. installing malicious browser plugins, stealing credentials and cookies, and spamming messages). Further, leveraging its privilege, it can disable in-browser security mechanisms and install additional payloads to enrich its capabilities. What is special about Scranos is that 1) its forged requests are indistinguishable from user-intended ones as observed by the server and 2) it is privileged and can bypass unprivileged defenses (e.g. browser-based).

Before Scranos, the research community attempted to deal with a hypothetical *privileged* adversary. Solutions such as NAB [16], can ensure that all requests are generated by I/O from a human user, but they cannot semantically distinguish whether the human-generated I/O causally resulted in the request. Alternatively, other works implement high-assurance, small trusted computing base (TCB) clients that are resistant to subversion, but suffer from limited functionality: Vbutton [5], Fidelius [17] and ProtectION [6] pack web page rendering (i.e. renderer), input processing (i.e. device drivers) and request construction (i.e. JavaScript engine) into trusted execution environments (TEE, e.g. ARM TrustZone, Intel SGX). But to keep the TCB small, the functionality support is extremely limited. For instance, Fidelius only supports rendering textboxes and handling keyboard inputs (i.e. no mouse support). These limitations can be exploited by UI attacks [18] (due to the coexistence of trusted and untrusted UI elements) and limit the compatibility of such TEE-based defenses (i.e. almost no modern web page is textbox-only).

In this paper, we propose an approach that works for fully-functional commodity web browsers by certifying to a remote server that the interactions between a user and a web page match some server-supplied specifications under an OS-level privilege malware. Instead of trying to secure the entire software stack that the user interacts with, our proposal, which we

call vWitness (short for "virtual witness". We take inspiration from how the signing of some legal documents must be "witnessed" by another party to be considered valid.) is a trusted component that passively observes the UI interactions the user has with the web page through *screenshots*. The screenshots, in our design, are samples of the display frame buffer taken from a lower-level and thus, are invisible to the client software stack (e.g. OS and browser) making it immune to tampering.

With the screenshots, vWitness leverages computer vision, to 1) ensure the web interface was correctly displayed to the user, 2) observe user-page interactions and ensure the final outgoing request is properly constructed based on correct semantics. To achieve both goals, vWitness needs server-provided specifications (called VSPEC) as web page appearances and request construction are both server-specific. As part of this work, we provide automatic VSPEC construction scripts.

Requests satisfying both goals above achieve *interaction integrity* and are considered to be *user-intended* (when paired with authentication, but not in the scope of this work). vWitness uses a cryptographic signature to convey whether a request satisfies interaction integrity to remote servers.

vWitness's novel use of computer vision is to differentiate between *benign rendering variations* and *malicious display manipulations* on the screenshots. A key challenge is that it is impossible for a web server to know, a priori, at the pixel level, exactly how a web page will be rendered on a user's device even in the absence of an attacker. This is due to the client-side variation in the rendering stack which consists of browsers, drivers, operating systems (OS, e.g. available fonts), and configuration settings (e.g. ClearType [19]). Such variations cause enough differences even enabling fingerprinting [20]. vWitness's vision-based validator can tolerate rendering variations while still being more accurate than previous works using pixel-by-pixel comparison [5] and image hash [21]. Computer vision is vulnerable to adversarial machine learning attacks, for which we devise and evaluate a set of vWitness-specific mitigation (as opposed to general defenses) that increases the robustness by $5.14 \times$. While validation-based (i.e. signatures) and isolation-based (i.e. TEE) secure systems each have their pros and cons, vWitness's limitations are those imposed by the use of computer vision, which we must be able to see to interpret it (e.g. vWitness can not verify file uploads) and can become computationally expensive to verify content-rich UI elements (e.g. complex animations or videos). While we leave these as future work, we believe they do not prevent vWitness from working with most security-sensitive pages.

vWitness's approach has unique performance traits: 1) vWitness runs in the background and is not on the critical path of the user-page interaction: it introduces minimal overhead at interaction time. 2) vWitness's validation is concurrent to the user interaction, and thus, depending on the length of the session and the speed of validation, a delay is added to the final request when user-page interaction has ended. 3) vWitness's validation can be incremental and benefits from caching and a GPU.

vWitness certifies interaction integrity and conveys the result to the server using a single bit of information (i.e. whether interaction integrity satisfies or not). It does not reveal unnecessary details to the server as all processing of screenshots remains local on the user client and screenshots are erased after a session finishes.

Contributions.

- We propose the idea of achieving interaction integrity through passive observation. We trained computer vision-based validators that can distinguish benign rendering variation and malicious UI tampering. We implemented a client-side prototype, called vWitness, in Xen's dom0, which is able to certify requests' interaction integrity to remote servers. We constructed server-side scripts to automate VSPEC construction.
- We propose four vWitness-specific defenses against adversarial attacks that increase the robustness of vWitness's text model by a factor of 5.14 over a reference on five attacks from CleverHans [22] and AutoAttack [23].
- We evaluate vWitness's compatibility, performance and accuracy on two datasets: Clickbench [24] and Jotform real-life web pages. The results show that vWitness is compatible with $10\times$ more pages than previous proposals, has an accuracy of 99.97% when validating pages and interactions, and provides performance (197/230ms delay for GPU/CPU setups) better than TEE-based works.

vWitness source code is available to the public [1].

II. SECURITY MODEL

With current web technologies, remote servers cannot determine if a request is intended by human users or if the request has been tampered with or forged by malware on the user client. Accepting malware-constructed/tampered requests can harm the service as well as the user (e.g. unintended Youtube subscription [15] and cryptocurrency loss [25]).

Threat Model. We assume the adversary controls malware with OS-level privileges on the user client. The malware can tamper with any guest OS components or user software such as the browser, but not a hypervisor that vWitness relies on for its integrity. Thus, firmware- and hypervisor-level malware is considered out-of-scope. This threat model is in line with previous works [5], [6], [17] as well as the assumption on hypervisor integrity [26].

vWitness considers phishing attacks to be out of scope as our defense requires server integrity and cooperation. In addition, vWitness does not address the authentication problem as it is orthogonal to interaction integrity, and vWitness considers attacks on user confidentiality to be out of scope (e.g. stealing sensitive inputs). We do not consider availability or denial of (vWitness) service attacks as blocking vWitness does not prevent users from using the unprotected page, which will eventually lead to rejected requests due to missing vWitness certification. Finally, we assume an honest user who enters the inputs they intend into the web page (i.e. they don't try to later equivocate about their earlier actions).

---

[1]https://github.com/dlgroupuoft/vWitness-DSN23

| Attack Vectors	Description and Example	Example Attack	Other Defenses
Request forgery	Malware forges requests without user input	Fake subscription, spamming, unaware payments	Captcha
Request tampering	Malware intercepts and tampers with user requests	Cryptocurrency transfer redirection [25]	Third party detection [27], trusted request construction [6], [17]
User interface (UI) tampering	Malware manipulates UI to confuse or deceive user (i.e. clickjacking [7])	Facebook like hijacking [28], Tweet-bomb [29], Scarno [15]	Server-side confirmation [30], [31], anomaly detection [32]

Table I: Attack vectors.
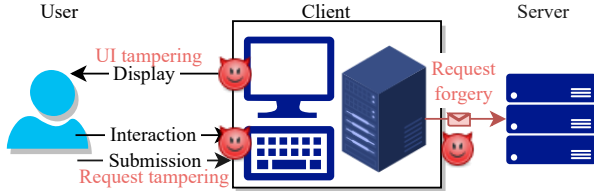
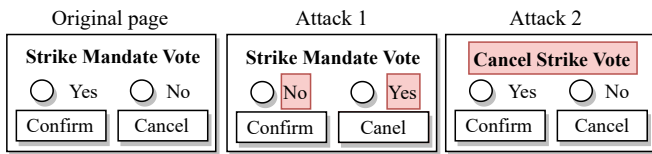Figure 1: Steps of user-page interaction and the attack vectors.

Figure 2: UI tampering leads to unintended user actions and requests. In both attacks, only the displayed text are altered by malware.

Attack Vectors. The attacker's goal is to submit a request to the web server and convince the web server that it came from a legitimate user. To achieve this, there are three attack vectors detailed in Table I summarizing what an attacker can do. Some prior proposals [6], [17] use TEEs to protect the user client, thus relying on a small trusted computing base (TCB) to provide assurance. The small TCB only enables incomplete UI support making them vulnerable to tampering of the unprotected UI sections [18]. We show example UI tampering attacks with minimized number of tampering in Figure 2.

Security Guarantees. vWitness's objective is to secure requests from a legitimate user to a web server despite a client that has been compromised by malware. It does so by certifying to web servers requests satisfying *interaction integrity* which means that a request 1) came from a real user and 2) was free of tampering by malicious software on the client. vWitness's security guarantees only hold if both the user and web server are benign, and it provides no benefits if a legitimate user interacts with a malicious web server (i.e. due to phishing for example), or the user themselves is malicious. vWitness secures web requests between legitimate users and web servers in 3 phases of user-page interactions:

1) *Display*: vWitness compares the web page that is displayed to the user against a server-provided specification called a VSPEC. For now, we only consider visual content of the web page and do not consider other media such as audio.
2) *Interaction*: Based on the user's interpretation of the web page, she provides information to the web page. This can include entering data into fields of the page, as well as inter-

acting with other common HTML elements such as drop-down menus, checkboxes, radio buttons, etc. vWitness infers which inputs the user is providing by tracking standard point of focus (POF) cues, such as a text cursor, and uses this to construct an independent record of the user's inputs.
3) *Submission*: After the user is done entering the information, she submits the contents to construct the request to the web server. vWitness then validates that the request is consistent with the user inputs via a validation function in the VSPEC. If the function succeeds, the VSPEC is included in the request and both are signed by vWitness certifying the request. The web server can verify that signature, which identifies the VSPEC and guarantees that both the display and interaction match the requirements of the VSPEC.

We summarize the phases of user-page interaction and the attack vectors in Fig 1.

Assumptions. We assume that user request content is constructed from user inputs on a web from and that the user is trustworthy who always provide intended inputs to the web page. We assume the user interacts with the web form in a conventional manner. Specifically, vWitness depends on the web form employing standard POF cues and that the user pays attention to the values as they enter them into the web page (more on this in Section III-C2). vWitness does not prescribe an order or manner of data entry, permitting the user to enter content and then subsequently delete, modify or copy it. vWitness also requires the appearance of a web page to be predictable, making excessively unpredictable (i.e. 3rd party ads) or excessively dynamic (i.e. videos) elements unsupported.

III. vWITNESS DESIGN

In this section, we discuss how the web server and the client should be set up to use vWitness, and describe the workflow of a typical vWitness interaction session, followed by a detailed explanation of the VSPEC and the interaction validation.

A. Initial Setup

We discuss the client and web server setup for vWitness.

Client Device. We assume a compromise-free initial setup phase of client devices. During this phase, the user (or a deployment specialist) will 1) Install vWitness's hypervisor, which protects vWitness's core logic in the dom0. We use Xen's terminology where dom0 refers to the secure virtual machine and domU refers to the guest OS. from malware (including kernel-level malware) in the guest OS (domU). 2) Install a public and private key pair (K_{pub} and K_{pri}) and a certificate for K_{pub}, C_{pub}, certified by a well-known CA. K_{pri} will be used by vWitness to sign requests and K_{pub} by web servers to
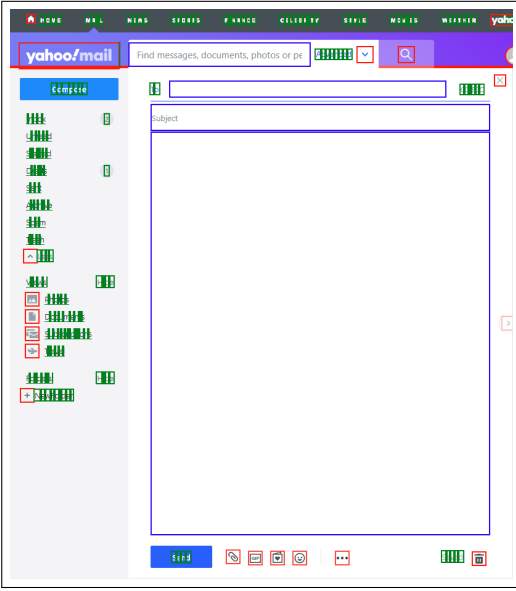
(a) Color-coded expected appearance.

"**text**": [(49,30,8,14, 'H'), (57,30,8,14,'O'),
(65,30,10,14,'M'), (75,30,8,14,'E'), ...]
"**image**": [(26, 27, 16, 16), (1176, 27, 123, 20), ...]
"**input**": [(192, 73, 429, 40, "search"), ...]

(b) Elements manifest, element type is color-coded.

Figure 3: VSPEC of Yahoo email web client. The expected appearance is cropped to remove personal information.

validate the requests. 3) Seal K_{pri} to the correct vWitness's execution state covering all software components that vWitness depends on. Successful unsealing of this key (K_{pri}) thereafter indicates that the correct vWitness software stack (i.e vWitness code and hypervisor) is running, and prevents the exposure of K_{pri} to any principal other than vWitness. In general, sealing requires a measured boot facility and secure element, both of which are commonly available and have been studied in previous work [33]–[36]. vWitness assumes the presence of hardware support for sealing. 4) Install an *untrusted* browser extension in domU to facilitate communication between the browser and the trusted vWitness component.

Web Servers. The server needs to perform a set of one-time modifications to use vWitness: 1) Construct a VSPEC for each page requiring vWitness protection. The VSPEC describes a web page's appearance and which elements are used by a user to enter request parameters. In addition, at runtime, unique values, such as a session ID, can be added to a VSPEC before transmission to the client to protect against replay. We will describe VSPEC content in detail as they are used. 2) Modify the web page to remove incompatible UI elements if any. We discuss such incompatibilities in §III-D.

B. Workflow

With the client and server set up, we describe vWitness's workflow with references to Fig 4.
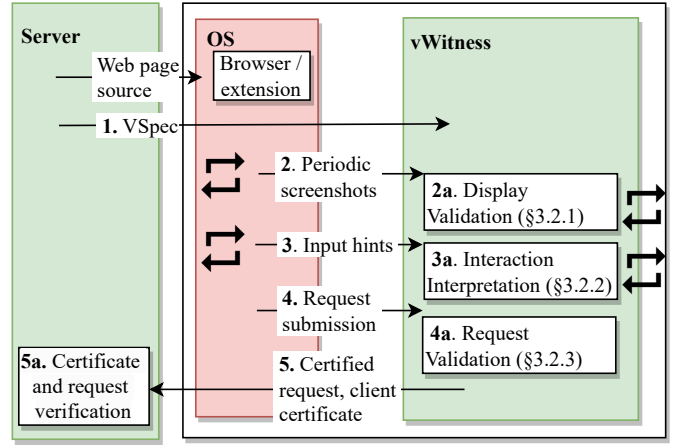

Figure 4: vWitness overview (§III). Green/red indicates trusted/untrusted entities.

Client-side. When a user requests a vWitness-enabled web page, the server replies with the VSPEC together with the page source files. The untrusted browser extension forwards the VSPEC to vWitness running on the client (step 1). After vWitness receives the VSPEC, it begins screenshotting by securely accessing the display buffer (step 2). On every screenshot, vWitness validates the appearance of the rendered page by comparing to what is expected in the VSPEC (step 2a) and, if the user has entered inputs, vWitness interprets the user-entered input values through input hinting (step 3, we discussed this below). If a user-entered value is to be submitted in the web request according to the VSPEC, vWitness records that value (step 3a). These two steps repeat until the user submits the page.

When the user submits, the web page logic constructs the web request and uses the browser extension to submit this to vWitness's secure component for certification (step 4). vWitness first validates the content of the request using a server-supplied validation function in the VSPEC by executing this function with the inputs it has observed during the user-page interaction (step 4a). If the validation function succeeds, vWitness creates a signature using K_{pri} over 1) the request content 2) the VSPEC (which contains a session id for freshness) using the sealed signing key. Signed requests are certified by vWitness to be constructed with interaction integrity and are returned to the web browser extension, together with the certificate, C_{pub}, who forwards them to the web server (step 5).

Server-side. The web server, upon receiving the certified request and C_{pub}, takes the following steps: 1) verifies that C_{pub} is signed by a legitimate CA, 2) verifies the signature on the request using K_{pub} in C_{pub}, thus ensuring the integrity of the request content and the included VSPEC, 3) verifies that the included VSPEC is the VSPEC it expects, ensuring that the web page interactions are correct. Note that the VSPEC should include a session ID (i.e. a nonce) for freshness. These three properties allow the server to gain confidence in the interaction integrity of the received request (step 5a).

Note that what is eventually sent out in the request remains

4

unchanged, except the addition of this signature and the corresponding certificate (C_{pub}), hence preserving privacy. While vWitness has access to the user's UI, and can see intermediate values that the user has entered, as well as other applications on the user's screen, no information about these is leaked to the remote web server. vWitness's secure component is intended to be open-sourced and can be scrutinized by the human user.

C. Validating Web Page Interactions

vWitness continuously monitors the Display during the user-page interaction session so that the UI displayed to the user is ensured to be correct throughout the session. In our Xen-based prototype, this is achieved by randomly sampling the domU's virtual frame buffer. The sampling is random so that an attacker cannot predict when vWitness samples, preventing time-of-check to time-of-use (TOCTOU) attack [37]. vWitness samples with a random delay ranging from 0 to 500ms between every two samples, meaning that on average, vWitness samples four times a second. To have a reasonable probability of guessing the sampling times, an attacker would have to change the display much more frequently than 500ms, but previous research has shown that humans may not perceive images that are shown for less than 500ms [38]. We discuss how user-page interactions are validated in the following.

1) Display: The page appearance must be untampered. vWitness does so through a three-step validation: first, vWitness determines the currently visible portion of the page (view port); second, vWitness finds all the UI elements within the view port—they need to be validated; finally, vWitness uses computer vision to validate that each element has been rendered correctly. vWitness can be configured to validate that all regions without UI elements match a solid background color specified in the VSPEC. Non-solid color backgrounds are currently not supported.

To determine the UI elements visible in the browser view port, the VSPEC includes the web page's *expected appearance* (Fig 3), which is a "long" screenshot of the page rendered at the client's window width and at the maximum height to accommodate all elements of the page. The client's width is passed to the web server by the untrusted browser extension, whose security implication is discussed in §V. vWitness determines the current view port by matching the sampled frame buffer against the expected appearance and finding the offset from the top of the expected appearance with the best match.

To figure out the expected UI elements displayed in the view port, VSPEC contains an *elements manifest* describing all expected UI elements using type, position and a ground truth. While we detail type and ground truth below, the relevant field for this step is the element position, which is a tuple of (x, y, width, height) illustrating the bounding rectangle of the element. All elements whose bounding rectangle overlaps with the view port are to be validated in the next step.

Finally, vWitness validates the observed screenshot against the expected elements. To do this, vWitness could naively perform a pixel-by-pixel comparison of the observed element with that in the VSPEC, but this would result in many false

alarms due to *benign rendering variations* caused by differences in the rendering stack among client devices, which can include different browsers, OSes, device drivers, GPUs, and configuration settings [20], [39]. Instead of trying to explicitly identify and handle all differences that could be generated by all such combinations, vWitness trains convolutional neural network (CNN) models to classify whether the differences between the observed and expected appearances are benign rendering variations. While machine learning is used in security (e.g. anomaly detection [32] and program vulnerability detection [40]), it is prone to adversarial examples against which we evaluate our approach in §V-B.

To enable validation, for each expected element, the elements manifest contains its type (text or image) and the ground truth. The ground truth is type-dependent and is how the element should be rendered. For text, it is the actual character; for images, it is the region of the Expected appearance.

vWitness uses two separate CNN models for text and images as they have different requirements. For instance, tampering with a few pixels may change the meaning of text (e.g. "i" vs "l"), but semantically identical images may be different pixel-wise (e.g., due to image compression). The models take two inputs: the ground truth from the VSPEC and the observed element at the same position on the screenshot. The models output a boolean prediction indicating whether the observed element matches the expected one. Both models use CNNs for feature extraction and dense layers to compare to the expected rendering.

Dynamic Web Elements. One challenging aspect of validating web elements is that they can be dynamic. vWitness categorizes dynamism into three types: scrollable (e.g. slide shows and scrollable list box), dynamically-appearing (e.g. context menu) and dynamically-scaled (e.g. resizeable elements) and develops unique validation methods for each type by tweaking the existing validation method (for static text and images). For brevity, we only discuss the scrollable elements. The key insight that enables dynamic element validation is to use a separate VSPEC, called nested VSPEC, for each dynamic element. This nested VSPEC is different from the page's and defines the dynamic element's appearance and how it can be interacted.

Scrollable elements refer to the UI elements that can scroll independently from the page (e.g. slide show and scrollable listbox). To support its validation, 1) in the page's elements manifest, scrollable elements have two new types to distinguish horizontal and vertical scroll and 2) the expected appearance is all the possible appearances of the scrollable element merged together. The actual validation of scrollable elements is identical to the page's with a change to the view port determination step: view port can either be horizontal or vertical depending on the type in the elements manifest.

2) Interaction: vWitness must capture user input semantics to validate outgoing requests. While this can be done through intercepting hardware IO inputs and secured device drivers as in previous works [6], [17], we realize that user input is displayed to the user and thus already in the screenshot that vWitness takes for Display validation. This enables a novel

approach where vWitness extracts user input semantics from the screenshots without the need for secure device drivers.

We need to make a *reflective validation* assumption, which states that while entering input, human users are generally attentive to whether the inputs are reflected correctly on the display [41]. For instance, after the user types "100" on the keyboard, she will confirm that the value "100" is displayed in the intended field, and if not, she will adjust the entered data until it does. Therefore, regardless of how the user inputs the value, if the user sees it on the display, it is the correct value.

To extract user inputs, a naive approach is to use Optical Character Recognition (OCR) on every input element, but we rejected it as it is computationally expensive and prone to adversarial example attacks [42]. Instead, vWitness relies on the untrusted browser extension to hint the position and values of user inputs and reuses the CNN-based text validator from the Display phase to ensure the hinted values are correctly displayed. Visual input elements, such as checkboxes and radio boxes, are also encoded as text inputs as their states can be mapped to a well-defined appearance. To ensure that the browser extension is honest, the elements manifest includes all inputs elements and their positions (in the form of bounding rectangle), vWitness ensures that the observed input elements must fall in the bounding rectangle of expected input elements.

However, extracting inputs from the UI poses two challenges—a subverted client could: 1) forge inputs even if a user is not present or 2) tamper with input elements the user is not paying attention to.

User Presence. Since hardware I/O observed by the hypervisor cannot be forged by application/OS code, vWitness assumes user presence if there exist hardware I/O events. Note that vWitness does not interpret the I/O events but only checks their occurrence from a hardware device, e.g. a keyboard/mouse, enforcing that the timing of these events corresponds to the timing of inputs observed on the UI.

User Attention. We observe that modern browsers use visual indicators (e.g. focus outline, caret) to indicate which element is currently *in-focus*, which we refer to as the point of focus (POF) and consider the focused element to be where the user's attention is. Assuming users naturally perform reflective validation at the POF, vWitness only accepts input changes (i.e. insertions, deletions) with POF. vWitness relies on pixel information to locate POFs, and our prototype currently supports three common POFs: focus outline for elements, input cursors (i.e. caret and vertical line) and multi-character highlight. Because POFs are rendered by the untrusted client, an attacker may tamper with them. For example, an attacker could forge multiple POFs, perhaps with one more noticeable than others, confusing vWitness about which POF is actually in use (e.g. the user thinks she is interacting with field A, but vWitness is validating inputs from field B). vWitness deals with this by validating the visual consistency of POFs to ensure only one set of POFs is present at any time (further discussed in §IV-A). Note that tampering with out-of-viewport inputs fields (e.g. due to user scrolling) will be ignored by vWitness.

3) Submission: The construction of outgoing requests is server-specific, vWitness validates requests by executing a server-supplied validation function in the VSPEC. The validation function takes interpreted inputs and the page request and produces a boolean indicating whether the request validation succeeds. This choice enables flexibility. In the simplest case, the validation function assembles inputs as a JSON object (e.g. for POST requests) and compares that in the page-constructed request as shown in Figure 3. However, it can implement arbitrary validation logic such as repeating the original construction logic in the web page or validating a set of constraints. This choice is again server-specific. The validation function can validate values in the request that are not interpreted by vWitness such as session IDs to distinguish multiple sessions and nonces for replay prevention; they need to be included into the VSPEC by the server. Finally, the VSPEC used for validation should be included in the signed request.

D. Limitations

vWitness relies on VSPECs for validation. This means third party elements (e.g. ads iframes) whose content is generated on-the-fly (e.g. from ads providers) are not supported. Also, vWitness can only certify interactions visible on the UI. Invisible interactions (i.e. file uploads) will not be seen by vWitness and thus cannot be validated unless a checksum is shown on the page. While vWitness can handle some dynamic behavior, the use of nested VSPEC makes it computationally too costly to validate excessive dynamicsms (e.g. videos). Finally, web pages must use a POF styles (e.g. text highlight color) recognized by vWitness and use input fields with POFs. Input fields without POF (e.g. button inputs) are not supported. It is possible for the server to customize POF style and include those in the VSPEC for vWitness's use, we leave this as a future work. Since vWitness is only intended for *security-sensitive pages*, we believe that these limitations will not affect its adoptability.

IV. IMPLEMENTATION

A. Client-side vWitness

On the client side, vWitness exposes three new JavaScript APIs to support requests validation: 1) `acquire_VSPECs`. The extension acquires the client window width through `Chrome.windows`, forwards it to the web server who returns the VSPEC tailored to the client width. The reported width is a virtual pixel value accounting for local settings such as user zooming and OS's default font size. 2) `vWitness_begin` begins a vWitness-session where the extension full-screens the page using `requestFullscreen`, and forwards the VSPECs to vWitness's secure component. 3) `vWitness_end` ends a vWitness-session. The extension exits full screen using `exitFullscreen`, and submits a page-constructed request body for validation.

CNN Models. vWitness implements two models separately for text and images with model details in Table II. Our models are available in both Keras H5 format and Tensorflow Lite and we focus on the English language, but are not limited to it. We detail the training data collection process. For the text

Model	Inputs	Output	# Parameters	# Training Data
Text	1. a locally rendered text character (32x32) and 2. the expected character as a string	Whether the local rendering is a benign rendering variation	352,097	556,512
Graphics	1. a (sub-)region of a locally rendered graphics (32x32) and 2. the expected appearance	of the expected character/appearance	1,761,089	620,217

Table II: Validation model details.

model, we we collect images of a single character rendered in legitimate rendering environments. We render 94 characters (10 numbers, 52 upper and lower English alphabet characters and 32 symbols) using 231 unique fonts, three styles (Italic, Bold and normal), three renderers (Gecko of Firefox, Blink of Chrome and Webkit of Safari) on two platforms (Windows and MacOS). This yields a core dataset of 8184 pairs of training points (as not all fonts support all characters). We expand the core dataset by 1) enlarging and shifting the characters 2) changing the color intensity and 3) randomly bit-flipping a subset of the pixels yielding a dataset of 278,256 images of legitimate one-character rendering. The purpose of expanding the core datasets is to 1) increase the total number of training data, 2) force the model to learn about the stroke shapes instead of memorizing the position of the characters and 3) augment training data such that random bit-flipped pixels do not affect the prediction [43]. To create a balanced dataset, we introduce data points with a false label by assigning another character to every image. This yields a perfectly balanced training set of 556,512 training data. We optionally trained text models with collapsed expected text (i.e. 's' and 'S') to increase the text models' accuracy [44]. The training data for the image model is similarly constructed, with the core dataset from a subset of CIFAR-10 [45] and Google's Material icon dataset [46]. We additionally added false data points with text in the images to ensure that unexpected text in the images will be detected. The total number of training data for the image model is 620,217.

POF Consistency Checks. vWitness current only supports the following input fields: text boxes vWitness extracts POFs through pixel information using OpenCV and enforces the following consistency rules: 1) *Number of instances*. For each POF, vWitness ensures that the number of POF instances on the context frame to be less than or equal to one. 2) *Same-field Logic*: the focus outline, selection highlight and caret must reside in the same field and 3) *Mutual Exclusivity*: caret and the selection highlight must not be present at the same time.

Performance Considerations. 1) *Differential Detection*: due to the frequent screenshots, unchanged UI elements do not need to be re-validated. Thus, vWitness finds the difference between two consecutive screenshots and only validates that region. 2) *Caching*: There are three caches: text, image, and frame. For each, the key is a cryptographic digest of the corresponding display region and the value is the corresponding validation result.

B. Server-side Scripts

vWitness implements the following server-side scripts.

Addressing Incompatibilities. The script handles the limitations in §III-D. First, we consider all `iframes` with a `src`

that points to an external domain as loading nondeterministic content such as ads. Our script removes all such `iframes`. Second, our script tries to make user inputs visible where possible, and warns for inputs that might be not visible. The script adds a `maxlength` field to all textual inputs fields (e.g. `type=text`, `textarea`). Third, the script searches for POF definitions in all CSS. We warn the developer if the following keywords are present: "outline", "caret" and ".focus". Finally, the script searches for unsupported HTML elements and warns the developer: files inputs (`type=file`), drag&drop inputs (`ondrop` attribute) and videos (`video` tag).

Generating VSPECs. The script automatically constructs VSPECs by 1) render the web page and 2) annotates HTML elements with the corresponding type for validation using a pre-defined HTML tag-to-validation type mapping. The result of the annotation (Figure 3) is visually presented to the developer who can manually make adjustments.

V. SECURITY ANALYSIS

A. vWitness' Security

Server-vWitness Communication. vWitness protects against an adversary who attempt to manipulate communication between the web server and vWitness as follows: 1) Tampering with web requests: each certified request from vWitness is signed with K_{priv}, which is protected from exposure using cryptographic sealing. 2) Tampering with the VSPEC: each certified request from vWitness contains the VSPEC that was used to validate the user's interaction with the web page and the user-inputs in the request. The web server should ensure that the VSPEC in the request matches the one that it sent with the web page, which prevents an adversary from modifying or replacing the VSPEC before vWitness uses it to validate the user's interaction with the web page. 3) Replaying an interaction: the VSPEC contains a non-repeating session ID to ensure freshness. This prevents an adversary from replaying a previously signed request. Similarly, replacing a VSPEC with an old one before vWitness performs validation will be detected by the web server as the VSPEC included in the request will contain a stale session ID.

Display. An attacker cannot show different UIs to vWitness and the user because vWitness directly samples from the frame buffer via using Xen's high privilege. The attacker cannot launch TOCTOU attack: showing the proper display just when vWitness samples the buffer but shows a tampered UI at other times (when the user perceives) because vWitness samples at random intervals with a mean of 250ms in every 500ms interval, which is the threshold shown by previous research [38].

An attacker cannot tamper with or remove text/images because vWitness verifies the locally-rendered text/images

against the ground truth in the VSPEC using the text/image verifier. An attacker cannot inject additional text in images because the image model is specifically trained to reject differences between VSPEC images and observed images that contain text. vWitness always verifies all UI elements in the viewport, so everything the user can see is checked by vWitness. As mentioned earlier, our prototype vWitness samples frequently enough that an attacker wishing to evade sampling would have to switch the display too frequently for a typical human to perceive the page.

Interaction. First, any out-of-viewport input update is ignored by vWitness (not included in the request) because vWitness cannot "see" it. We first assume an honest browser extension (which hints the input update to vWitness correctly), dishonest browser extensions are discussed thereafter. An attacker can forge or tamper with any user inputs, but it will be caught by either the user or vWitness depending on whether the input update has POF. The user will notice input manipulation at the POF because POFs are designed to be seen by a human user, and users already use them for input entry, and thus will try to fix the inputs on the UI if they do not appear as the user expects them to. For input updates without POF, vWitness will see them as a violation of POF consistency check and ignore the input update. Whenever vWitness ignores a hinted input update, the input it tracks and the input values tracked by the page will be different and when the submission function executes, it will detect the difference as reflected on the constructed request.

Submission. An attacker may submit the form before the user finishes [6]. While vWitness cannot prevent this, it enforces that the UI must change after the user interaction ends (i.e. with a call to `vWitness_end`). This way, a malicious client cannot silently submit the form without the user noticing it. A tampered request submitted to vWitness will be detected by the VSPEC's submission function, while a tampered VSPEC will be detected by the web server as vWitness will include it in the request.

Dishonest Browser Extension. vWitness does not trust any information from the extension: all information is either included in the signature to the server (to be verified by the server, e.g. VSPEC) or verified against the information in the VSPEC (e.g. hints). To ensure the hints are correct, vWitness ensures input fields not hinted, do not receive any inputs. We then analyze possible attacks on this interface.

The `acquire_VSPEC`s call relies on the client window width. An incorrect window width by the browser extension leads to an incompatible VSPEC being retrieved and will fail the viewport detection due to mismatch between the observed width and the VSPEC's width. An attacker making early calls to `vWitness_begin` gains no benefit, as it triggers frame buffer sampling and may fail the display validation depending on what is shown. The attacker may delay this call until user interaction has begun, and leverage this time difference to show misleading information. While vWitness cannot verify the UI before `vWitness_begin`, it implements the follow-

ing measures to ensure a clean start of any vWitness-session to detect delayed `vWitness_begin` calls. 1) Clean viewport: the viewport must be at (0, 0)) and 2) Clean input entry: all input fields are empty. Delaying `vWitness_end` calls does not benefit the attacker as the Display validation continues. Early calls to prematurely end a vWitness-session will be noticed by the user due to the UI change (e.g., "submitted") enforced by vWitness, who can take corrective actions, e.g., contacting the server.

Limitations. vWitness relies on the secrecy of the private key (K_{pri}). If this secrecy is compromised, an adversary can forge vWitness signatures and certify requests. Mitigating such exposure requires a process for revoking and re-issuing compromised keys. In addition, vWitness is vulnerable to relaying attacks, meaning a compromised OS can relay an authenticated web session to a malicious user using a correct vWitness instance. This attack is outside of our current vWitness threat model, as vWitness assumes both the user interacting with vWitness and web server are benign. vWitness can be extended to handle relaying attacks if user identities are bound to specific vWitness instances, which might be accomplished with an enrollment procedure similar to that of multi-factor authentication hardware tokens.

B. Adversarial Attacks on CNN Models

An attacker can use adversarial examples [47], [48] to try and fool vWitness's CNN-based verifiers. We evaluate vWitness's robustness against such attacks by comparing vWitness with two reference models: a MNIST classifier [49] for text and a CIFAR-10 classifier [50] for images. We believe generic adversarial example defenses (e.g. adversarial training [51]–[53], gradient masking [54], detection [55]) are complementary to the vWitness-specific defenses we present here. However, one such defense, input transformation [56], is incompatible with vWitness because it alters the inputs, potentially changing the meaning of a web element. Note that as our model takes *unit inputs* (i.e. a single character for text, a 32-by-32 sub-region for image), the accuracy numbers in Table III illustrate the worst-case accuracy, as a successful attack will likely need to alter more than one unit input, which exponentially reduces the probability of a successful attack.

Attack Setup. We measure the models' accuracy (as a way to measure robustness) under the following attacks: fast gradient method (FGM) [47], basic iterative method (BIM) [57], momentum (MOM) [58], CW [59] with L2 distance (CW2), AutoPGD with cross entropy loss (APGD) [23] and Fast Adaptive Boundary (FAB) [60]. We use Cleverhans v4.0 [22] for the first four and AutoAttack [23] for the rest. All attacks are *targeted* attacks fooling the model into producing the opposite label. We launch attacks in three epsilon values (i.e. maximum allowed perturbations) measured using two distances (the three sub rows of each model in Table III). These epsilon values are picked to cover the range of successful attacks [59] to best determine the models' robustness. We tune hyper-parameters to give the highest attack success rate. We use the average accuracy of all attacks to measure a model's robustness. We

#	Models	Clean	L inf FGM	BIM	MOM	FAB	APGD	L2 FGM	BIM	MOM	FAB	APGD	CW2	Avg attacks
t1	Reference model	93.33	5.00 1.67 0.00	0.83 0.83 0.83	0.83 0.83 0.83	45.00 43.33 43.33	0.00 0.00 0.00	37.50 12.50 7.50	0.83 0.83 0.83	1.67 0.83 0.83	64.17 69.17 68.33	0.00 0.00 0.00	6.67	13.39 (base)
t2	Base text model	98.33	90.00 90.00 92.5	45.83 46.67 34.17	0.83 0.83 2.50	60.83 50.00 49.17	0.83 0.00 0.00	88.33 81.67 99.17	54.17 45.00 50.00	16.67 0.00 5.00	47.50 39.17 34.17	20.83 0.00 0.00	7.50	37.17 (2.82x)
t3	Avg one font	99.36	84.87 90.37 91.62	52.42 51.88 56.04	4.62 3.71 12.58	64.29 56.50 50.21	2.83 0.00 0.00	91.29 90.38 91.71	69.00 59.92 59.46	31.42 4.96 2.33	86.92 83.29 81.96	24.63 1.37 0.42	12.62	44.60 (3.38x)
t4	Avg sans serif	99.50	87.92 89.17 89.17	56.83 49.33 55.00	6.25 5.58 14.17	63.42 53.92 46.58	1.33 0.00 0.00	93.58 91.42 92.33	75.50 63.84 61.92	40.00 7.50 3.50	91.42 87.42 85.67	33.42 2.00 0.50	16.33	47.05 (3.51x)
t5	Avg serif	99.22	81.83 91.58 94.08	48.00 54.42 57.08	3.00 1.83 11.00	65.17 59.08 53.83	4.33 0.00 0.00	89.00 89.33 91.08	62.50 56.00 57.00	22.83 2.42 1.17	82.42 79.17 78.25	15.83 0.75 0.33	8.92	43.88 (3.28x)
t6	High threshold (0.99)	99.47	99.59 98.75 100	99.59 98.75 89.58	26.66 13.75 12.50	100 100 98.75	10.00 0.41 0.41	99.59 99.17 100	100 100 100	76.25 28.34 10.84	100 99.16 99.16	65.41 6.25 1.67	100	**68.86 (5.14x)**
						Text models above, image models below								
g1	Reference model	88.33	11.67 9.17 6.67	8.33 6.67 3.33	8.33 7.50 6.67	0.00 0.00 0.00	0.00 0.00 0.00	22.50 22.50 18.33	9.17 9.17 9.17	8.33 8.33 8.33	0.00 0.00 0.00	0.00 0.00 0.00	7.50	6.98 (base)
g2	Image model - CIFAR-	98.13	96.67 98.33 100	78.33 78.33 81.67	77.50 61.67 59.17	96.67 95.83 91.67	35.83 13.33 3.33	93.33 90.83 95.83	90.83 81.67 82.50	94.17 90.00 86.67	97.50 96.67 96.67	81.67 53.33 40.83	12.50	75.91 (10.88x)
g3	Image model - Icon	99.96	93.33 90.00 94.17	98.33 93.33 89.17	90.83 79.17 77.50	97.50 96.67 94.17	49.17 25.00 5.83	99.17 95.83 96.67	99.17 98.33 98.33	99.17 98.33 95.00	100 99.17 99.17	85.83 71.67 54.17	28.33	83.63 (11.98x)

Table III: The model accuracy (%) of the text and image models under adversarial examples. The three sub rows in every model correspond to Epsilon=0.1254, 0.2509, 0.5019 for Linf and Epsilon=1, 2, 3 for L2. The accuracy numbers are for unit inputs.

round all generated attacks to the nearest pixel value between 0 and 255 to make them valid images.

Base Robustness. We tabulate the accuracy of the base vWitness models (t2, g2 and g3) and the reference models (t1 and g1) in Table III. We find that vWitness's base text and image models are more robust by a factor of 2.82 and 11.98 respectively. We believe vWitness's high base robustness is due to the following 1) vWitness's models perform a simpler task: While the reference models perform multi-label classification, vWitness performs binary classification between the Display and the ground truth from VSPECs. 2) The ground truth in VSPECs further reduces the attack surface as only one targeted attack is applicable. The attack must cause the model to misclassify, from a false prediction into a true prediction (e.g. the attacker wants to change the word "yes" into "no" on display, but still have it classified as a "yes" so it matches the VSPEC). The opposite attack (from true into false) is not applicable to vWitness. Since the image models are already 10.88 and 11.98 times more robust, in the rest of the section, we focus on defenses specific to vWitness's text models.

Specialized single font models can be adopted if web servers are willing to use a pre-selected font for all text on the page. This restricts the input space, allowing a smaller adversarial space, which in turn, increases the model robustness. Note that this is different from input transformation [56]. We trained 20

Attacks	MOM			APGD		
Epsilon	1	2	3	1	2	3
Original	}	i	L	}	i	L
Target	p	-	t	{	e	A
Attacks						

Table IV: Successful adversarial examples generated for vWitness's t6 model in Table III using L2 norm.

text verifier models, each specialized to a single font and report the robustness in Table III t3. Single font increases the model robustness by a factor of 3.38. In this experiment, since we aim to explore the robustness change due to input space change, we did not change the model capacity (i.e. reducing layers or the number of training parameters)—so there is potential for further robustness due to distillation [61].

Font Characteristics. We further examine whether a font's characteristics have an impact on a model's robustness. We evaluated fonts with varying characteristics such as font weight, width and type [62], [63]. The only significant factor we found is serif or sans serif. We trained 10 serif and 10 sans-serif single-font models and report the average accuracy in rows t4 and t5 of Table III. The robustness increases by a factor of 3.51 and 3.28 for sans serif and serif font types respectively. *High Detection Threshold* increases the model robustness by a factor of 5.14. A high detection threshold forces the at-

Description	Lines of Code
vWitness	1,128
WolfCrypt	2,801
OpenCV	177,396
Tensorflow Lite	14,580
Xen	555,160
Chromium	25,163,547
Firefox	20,928,358

Table V: The TCB size of vWitness prototype's main components.

Dataset	# datapoints	Avg Invocations T	Avg Invocations G	Total Invocations T	Total Invocations G
Clickbench	39	NA	880	NA	34320
Jotform	100	464.14	17.27	46414	1727

Table VI: Complexity of the evaluation datasets. T/G refers to the text/graphics model.

tacker to produce more convincing attack examples but may reduce the model accuracy (as the model needs to produce more convincing predictions). We trained three single-font sans serif models with three thresholds: 0.75, 0.9 and 0.99. We show the result of the 0.99 threshold in Table III t6. The threshold at which certain amount of adversarial perturbation becomes user-noticeable is highly subjective. However, prior work shows that adversarial examples cannot fool a time-unlimited human [64]. What is more, we believe perturbations added to typeset text are more noticeable than perturbations added to hand-written text. We put a random subset of successful adversarial examples in Fig IV. FGM, BIM and MOM do not minimize perturbation, so their perturbation seems more "randomly" spread out and, thus, is (in our opinion) easily perceivable and has limited effectiveness. In practice, an attacker will likely need to change multiple characters to alter the meaning of the text, making an adversarial example attack even less likely to succeed against vWitness.

C. Trusted Computing Base (TCB) Size

The core logic of vWitness itself was implemented in just over 1K lines of code (LOC), making it small and auditable, but it depends on several external components, such as OpenCV for processing raw images from the frame buffer, Tensorflow for computer vision and CNN models, and Xen & QEMU for isolation. We show vWitness's LOC breakdown in Table V. For comparison, we also list the code sizes of the Chromium and Firefox browsers. Note that our vWitness prototype was designed for ease of implementation as opposed to TCB minimization, and thus uses commercially off-the-shelf components, which make up the vast majority of its TCB. Many of these components can be substituted with smaller-TCB alternatives (e.g. XMHF [33], debloated libraries [65]). We leave this as future work. vWitness, at its current stage, already has a much smaller TCB than fully-functional commodity browsers.

VI. EVALUATION

We run empirically evaluation on a machine with an Intel i7-7700 and 16GB RAM with two setups: CPU-only and with a

Dataset	TP/TN	FP/FN	Accuracy
Clickbench	39	1	97.5%
Jotforms	100	0	100%

Table VII: Accuracy of vWitness's output validator (a single display frame).

GPU. The CPU is always configured as 7 logical cores for the domU and 1 logical core for the dom0 (where vWitness runs) and 1 GB of RAM to best mimic a production setup. The GPU setup uses a Nvidia 1060 GPU. The GPU is not a requirement for vWitness's deployment, but part of a typical workstation and it accelerates vWitness's vision-based validators. We use the text model trained with collapsed labels.

A. Accuracy

vWitness's accuracy is evaluated on one dataset of attacks and one dataset of benign pages to measure the rate of true positives and true negatives.

Clickbench. [24] is a corpus of 1080×1920 screenshots of simulated clickjacking attacks on Android (Clickjacking falls under our attack model). We acknowledge that while Click-bench being originally targeted at Android UIs is not ideal for evaluating vWitness, which is designed for web UIs, it is the only available dataset we are aware of that contains malicious UI tampering. We cannot construct VSPECs on Clickbench's screenshots (no HTML), thus, we create a pseudo-VSPEC classifying the whole screenshot as a single image invoking vWitness's image model only. As not all samples are applicable to our evaluation due to different definitions of attacks, we have 40 pairs of Android UIs. The true positive (TP) and false negative (FN, i.e. malicious tampering not detected by vWitness) rates are reported in Table VII. There was a single false negative on an attack tampering with text in an image. We invoked the text model and it was able to flag the tampering. This suggests that our image model alone can handle most Clickjacking attacks.

Jotforms. We evaluate vWitness's ability to handle benign rendering variations with a set of forms derived from the Jotforms (jotform.com). We chose Jotforms as they provide representative samples of many common forms, which are used on over 10 million websites. We randomly select 100 forms and render each form on several different rendering stacks. We execute the vWitness's scripts to construct the VSPECs in one rendering stack and we render the same pages in other (and different) rendering stacks, which we provide to vWitness as captured frame buffers. We tabulate the result in Table VII that shows high accuracy of vWitness's vision-based validators.

B. Request Delay

Since vWitness's validation is concurrent to the user session, a delay (L) is added to the final request, for which we aim to empirically evaluate in this section. This delay can be modelled as $L = T(init) + \sum T(frame_i) + T(request) - T(session)$ where $T(init)$ is the time to initialize vWitness (e.g. transmitting VSPEC), $T(frame_i)$ is the time to validate the ith display frame and $T(request)$ is the time to validate

Setup	Dataset	Mean	Max	Min	Stdev
CPU	Clickbench	3.29	3.63	1.68	0.65
	Jotforms	1.17	6.83	0.34	0.85
GPU	Clickbench	0.73	1.67	0.44	0.18
	Jotforms	0.88	6.75	0.33	0.33

Table VIII: vWitness's performance numbers, in seconds (s), of first display frame ($T(frame_0)$). CPU setup uses a single logical core.
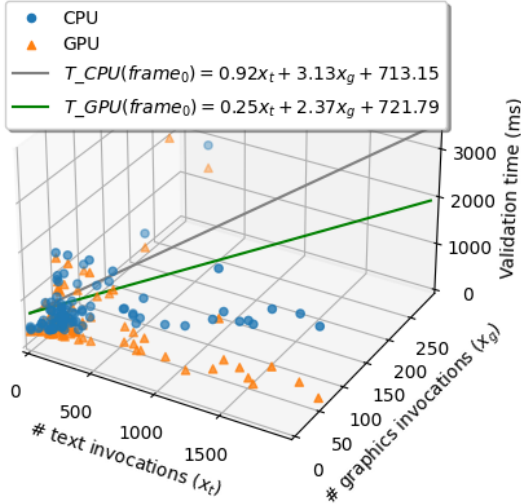

Figure 5: First display frame validation time (in ms) of Jot-forms.

request (i.e. executing the validation function and signing) and $T_{session}$ is the length of user's session. We further distinguish the time to validate first display frame $T(frame_0)$ from subsequent frames $T(frame_i)$ for $i > 0$ due to caching.

First Display Frame. $T(frame_0)$ is dependent on the number of model invocations—more complex pages have a higher number of model invocations and thus a larger $T(frame_0)$. We first report the statistics on model invocations in Table VI. We report $T(frame_0)$ in Table VIII. GPU speedup is more significant for Clickbench than Jotforms because the former 1) invokes only the graphics model and 2) have many more invocations than the latter as we treat the whole UI as a single image. To know the exact relationship between $T(frame_0)$ and text and graphic model invocations (x_t and x_g respectively), we plot evaluation numbers of Jotforms in Figure 5 and fit two regression lines whose coefficients enable the prediction of $T(frame_0)$ of arbitrary complex page. As the figure shows, it is more expensive to invoke the graphic model as it takes two graphics as input and has to do two feature extractions.

Subsequent Display Frames. $T(frame_i)$ is page-dependent, and due to caching, it is also interaction-dependent. To simulate interactions, we recorded our own interactions of filling out a form, a typical use case for vWitness. We report $T(frame_i)$ in the middle four columns in Table IX together with measured $T(init) + T(frame_0)$ and $T(request)$.

Since $T(frame_i)$ is much less than 1 second, this means that, with a long enough $T_{session}$, there will be a session time
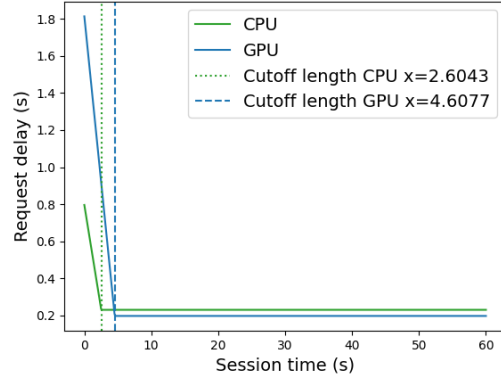

Figure 6: Request delay (s) vs session time (s).

Setup	Init+First frame	Subsequent frames				Validation function
		Mean	Max	Min	Stdev	
CPU	0.760	0.194	0.434	0.088	0.067	0.036
GPU	1.778	0.161	0.315	0.070	0.055	0.036

Table IX: End-to-end performance numbers, in seconds (s).

threshold $thresh_i$ where vWitness has finished validating all $frame_i$ for $i < thresh_i$. We refer such $thresh_i$ as cutoff session length, and session that lasts longer than this time should only incur a delay of $T(request)$. For session length longer than cutoff session length, the request delay L will be $T(frame_{last}) + T(request)$, which is $0.194 + 0.036 = 0.230$ seconds for the CPU setup and $0.161 + 0.036 = 0.197$ seconds for the GPU setup in Table IX, 197ms and 230ms are roughly 9% of the average load time of a web page of 2.2s at the time of this paper [66]. To determine the cutoff session length, we plot request delay (L) as a function of varying $T_{session}$ in Figure 6. The cutoff session lengths are 2.6 and 4.6 seconds for the CPU and GPU setup respectively.

C. Compatibility

We measure vWitness's ability to support common web forms by constructing a dataset of representative web forms. We had initially sought to build this dataset by crawling the Internet for forms, but found that this did not produce a high-quality dataset because: 1) security-sensitive forms that vWitness is meant to protect are often only accessible with authenticated user sessions that our crawler could not access and 2) publicly accessible forms (with `form` tags), in our experience, tend to be dominated by simplistic "contact us" forms that lack variety and complexity. We thus instead crawled *all* 2476 forms from JotForm and *all* 109 templates from WPForms(wpforms.com) for Word Press totaling 2585 forms. We did not remove any page from the dataset.

We compare vWitness's compatibility with proposed limited-functionality high-assurance web clients [6], [17]. These proposals implement web clients that render a subset of HTML elements, but do so with very small TCB. We find that the smaller TCB is costly: none of them can support any of the pages from our dataset of representative forms, indicating that real web forms still contain many complex web elements that previous work does not account for. To enable comparison, we

System	# of Compatible Pages
Fidelius [17]	20 (0.77%)
ProtectiON [6]	196 (7.58%)
vWitness	2,255 (87.23%)

Table X: vWitness's compatibility compared to limited-functionality, high-assurance works.

relax our requirements and count the number of forms where at least 90% of the elements are supported, to be "compatible". As shown in Table X previous work can support no more than 8% of real web forms while vWitness can support almost all forms, with this 90%-support compatibility threshold.

VII. RELATED WORKS

Clickjacking tricks a user into performing unintended actions through deceptive user interface [7], [18], [24]. There are current defenses through in-browser enforcement such as framebusting [67] which can be disabled or bypassed (i.e. directly writing to frame buffer) by privileged malware such as Scranos. vWitness defends against clickjacking by validating the UI presented to the user.

Request Forgery. Botnets forge communications with the service without users' awareness. While there exist defenses attempting to identify such traffic [68] from the service side, well-adopted client-side defenses, such as Captcha [69], fail to bind requests (i.e. bot-generated) to human users, hence Captcha farms. vWitness binds outgoing requests to a vWitness-session described in a VSPEC, which guarantees association with real human user interaction, preventing bot-generated traffic.

Cross-site Request Forgery (CSRF) [8] forges requests to the service by hijacking the user's authenticated session. Service-side defenses require requests to provide proof of user session abstracted by tokens [70] or first-party-only cookies (Same-Site) [71]. These defenses achieve security through obscurity — a strong client-side malware can access and present these session abstractions to the service bypassing the defense. An adversary hijacking a vWitness session can forge requests in the background, but only user-intended requests will receive certification due to display validation and user presence.

Malicious Scripts in Page Context can do harm such as stealing cookies, forging requests, and tampering with requests [6]. vWitness does not prevent malicious script execution on the client side, it ensures that malicious requests will not receive certification, as they lack a display and user presence.

Certifying Requests. A common solution adopted by the industry is confirmation through an out-of-band channel. For instance, web services send notification emails to the user after critical actions [30], [31], the response (or its absence) to which is treated as user confirmation. Confirmations have the following drawbacks 1) reactive measure. Confirmation is sent after the service has executed the critical action, meaning that attacks cannot be prevented in the first place; 2) lack of trust. The out-of-band channel may not always be secure and 3) an extra burden on the user side. As a result, the application of out-of-band confirmation is limited.

Client-side trusted execution defenses enabling secure display [72], secure I/O [73] and secure JavaScript execution [74], [75] are the fundamentals to secure interaction integrity. Below, we will describe academic works in the area. 1) *Heuristic-based.* Earlier works [16], [76] leverage the time since the last hardware I/O as a heuristic to certify requests, as hardware I/O can only be generated from the physical human user. Requests without recent hardware I/O are said to be non-user-generated. These works neither verify what is shown to the user nor link hardware I/O to request semantics—an attacker can evade detection by only forging requests when there are I/O activities. 2) *Linking on-screen inputs to requests.* Gyrus [41] uses a secure VM to acquire the on-screen user inputs (e.g. email content on the screen) to match with outgoing requests (e.g. email content in the request). Despite that a Gyrus user may see a completely tampered page, Gyrus uses per-field security indicators to indicate whether the on-screen input values have changed since the last time the user has viewed it, which imposes a high cognitive load on users. 3) *Secure user-page interaction.* VButton [5], Fidelius [17] and ProtectiON [6] realize that the display/interaction/submission must all be secured to protect requests. These TEE-based works implement a minimal renderer/device drivers/JS engine inside a secure container (e.g. SGX, TrustZone). These works face a dilemma between functionality support and a low-trusted computing base (TCB). By favoring a smaller TCB, they are extremely limited in terms of UI elements/input devices/JS features support in the secure area of the page. As a side effect, these works have to allow the coexistence of secured and unsecured UI elements, and thus, are open to UI redressing attacks [18] tampering with all the unsecured web elements. 4) *Secure confirmation* is similar to out-of-band confirmation except the confirmation is shown on the same client securely through a secure execution environment such as ARM TrustZone [5], Intel TXT [77]. These confirmation dialogues display at a fixed time (i.e. after the transaction, before the request is sent). An attacker can exploit this timing by showing a deceptive context before and after the secure dialogue to lure the user into falsely confirming the transaction [78]. vWitness enforces display integrity throughout a vWitness session.

VIII. CONCLUSION

We shift away from directly implementing trusted web clients in hardware-supported TEEs, which can lead to limited functionality and incomplete UI protection, to only relying on passively "witnessing" user interactions with the web page, and validating them against a VSPEC. We have tested this approach using the vWitness prototype, which employs CNN-based validators to process raw screenshots of the browser view and certifies the outgoing requests if the validation succeeds based on the provided VSPEC. Our evaluation shows that vWitness is effective in detecting UI tampering, offers significantly better compatibility in terms of supporting real-life web pages, and only introduces negligible latency. Despite the additional effort involved, with the automated VSPEC gen-

eration, we believe vWitness will provide an appealing option to security-critical websites.

REFERENCES

[1] R. Dhamija and A. Perrig, "Deja Vu–A user study: Using images for authentication," in *9th USENIX Security Symposium (USENIX Security 00)*. Denver, CO: USENIX Association, Aug. 2000.

[2] S. Ghorbani Lyastani, M. Schilling, M. Neumayr, M. Backes, and S. Bugiel, "Is FIDO2 the kingslayer of user authentication? a comparative usability study of fido2 passwordless authentication," in *IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 268–285.

[3] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell, "Stronger password authentication using browser extensions," in *14th USENIX Security Symposium (USENIX Security 05)*. Baltimore, MD: USENIX Association, Jul. 2005.

[4] N. Andriamilanto, T. Allard, G. Le Guelvouit, and A. Garel, "A large-scale empirical analysis of browser fingerprints properties for web authentication," *ACM Transactions on the Web (TWEB)*, vol. 16, no. 1, pp. 1–62, 2021.

[5] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan, "Vbutton: Practical attestation of user-driven operations in mobile apps," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '18. New York, NY, USA: ACM, 2018, pp. 28–40.

[6] A. Dhar, E. Ulqinaku, K. Kostiainen, and S. Capkun, "ProtectIOn: Root-of-trust for io in compromised platforms," in *Proceedings of the 2020 Network and Distributed System Security (NDSS) Symposium*, vol. 2020, 2020, p. 869.

[7] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and defenses," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 413–428.

[8] S. Khodayari and G. Pellegrino, "JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2525–2542.

[9] MDN, "X-frame-options - http | MDN," aug 2022, https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options [Accessed Nov 26 2022].

[10] ——, "Cross-origin resource sharing (CORS) - HTTP — mdn," https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS [Accessed Aug 29 2022], 2022.

[11] ——, "Content security policy (CSP) - HTTP | MDN," 2022, https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP [Accessed Nov 26 2022].

[12] ——, "Subresource integrity - web security | MDN," https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity [Accessed Nov 26 2022], aug 2022.

[13] S. Agarwal and B. Stock, "First, do no harm: Studying the manipulation of security headers in browser extensions," in *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*, 2021.

[14] T. MICRO, "Brex_kilim.ll - threat encyclopedia," 2014, https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/BREX_KILIM.LL/ [Accessed Nov 26 2022].

[15] TechCrunch, "Scranos, a new rootkit malware, steals passwords and pushes YouTube clicks," https://social.techcrunch.com/2019/04/16/scranos-rootkit-passwords-payments/ [Accessed Nov 26 2022].

[16] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy, "Not-a-bot: Improving service availability in the face of Botnet attacks," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 307–320.

[17] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. Garcia, E. Gong, H. T. Nguyen, T. K. Sethi *et al.*, "Fidelius: Protecting user secrets from compromised browsers," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 264–280.

[18] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and dagger: from two permissions to complete control of the UI feedback loop," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 1041–1057.

[19] Microsoft, "Microsoft cleartype overview," 2022, https://docs.microsoft.com/en-us/typography/cleartype/ [Accessed Nov 26 2022].

[20] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in HTML5," in *Proceedings of W2SP 2012*, M. Fredrikson, Ed. IEEE Computer Society, May 2012.

[21] S. Roy and Q. Sun, "Robust hash for detecting and localizing image tampering," in *2007 IEEE International Conference on Image Processing*, vol. 6. IEEE, 2007, pp. VI–117.

[22] N. Papernot, F. Faghri, N. Carlini, I. Goodfellow, R. Feinman, A. Kurakin, C. Xie, Y. Sharma, T. Brown, A. Roy, A. Matyasko, V. Behzadan, K. Hambardzumyan, Z. Zhang, Y.-L. Juang, Z. Li, R. Sheatsley, A. Garg, J. Uesato, W. Gierke, Y. Dong, D. Berthelot, P. Hendricks, J. Rauber, and R. Long, "Technical report on the CleverHans v2.1.0 adversarial examples library," *arXiv preprint arXiv:1610.00768*, 2018.

[23] F. Croce and M. Hein, "Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks," in *Proceedings of the 37th International Conference on Machine Learning*, ser. ICML'20. JMLR.org, 2020.

[24] A. Possemato, A. Lanzi, S. P. H. Chung, W. Lee, and Y. Fratantonio, "Clickshield: Are you hiding something? towards eradicating Clickjacking on Android," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2018, pp. 1120–1136.

[25] J. Rubin, "Vipersoftx: Hiding in system logs and spreading venomsoftx - avast threat labs," https://decoded.avast.io/janrubin/vipersoftx-hiding-in-system-logs-and-spreading-venomsoftx/ [Accessed Feb 20 2023].

[26] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 2–13, 2008.

[27] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 641–654.

[28] "Infosec handlers diary blog - sans internet storm center," https://isc.sans.edu/diary/Clickjacking+attacks+on+Facebook%27s+Like+plugin/8893 [Accessed Feb 28 2023].

[29] "Explaining the "Don't Click" clickjacking Tweetbomb," https://softwareas.com/explaining-the-dont-click-clickjacking-tweetbomb/ [Accessed Feb 28 2023].

[30] Zerodha, "Why do I get an SMS and email from NSE, BSE and MCX when I trade?" https://support.zerodha.com/category/trading-and-markets/trading-faqs/articles/why-do-i-get-an-sms-and-email-from-nse-bse-when-i-trade [Accessed Nov 26 2022].

[31] I. Brokers, "Trading assistant," https://guides.interactivebrokers.com/iphone/monitor/trading-assistant.htm [Accessed Nov 26 2022].

[32] B. Juba, C. Musco, F. Long, S. Sidiroglou-Douskos, and M. C. Rinard, "Principled sampling for anomaly detection." in *Proceedings of the 2015 Network and Distributed System Security (NDSS) Symposium*, 2015.

[33] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an extensible and modular hypervisor framework," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 430–444.

[34] H. Tan, W. Hu, and S. Jha, "A TPM-enabled remote attestation protocol (TRAP) in wireless sensor networks," in *Proceedings of the 6th ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks*, 2011, pp. 9–16.

[35] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "Vrased: A verified hardware/software co-design for remote attestation." in *USENIX Security Symposium*, 2019, pp. 1429–1446.

[36] M. Eckel, A. Fuchs, J. Repp, and M. Springer, "Secure attestation of virtualized environments," in *ICT Systems Security and Privacy Protection: 35th IFIP TC 11 International Conference, SEC 2020, Maribor, Slovenia, September 21–23, 2020, Proceedings 35*. Springer, 2020, pp. 203–216.

[37] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash, "Android UI deception revisited: Attacks and defenses," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 41–59.

[38] W. von Hippel and C. Hawkins, "Stimulus exposure time and perceptual memory," *Perception & Psychophysics*, vol. 56, no. 5, pp. 525–535, Sep 1994.

[39] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 878–894.

[40] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *2015 IEEE Symposium on Security and Privacy (SP)*, 2015, pp. 797–812.

[41] Y. Jang, S. P. Chung, B. D. Payne, and W. Lee, "Gyrus: A framework for user-intent monitoring of text-based networked applications." in *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[42] F. Tramèr, P. Dupré, G. Rusak, G. Pellegrino, and D. Boneh, "Adversarial: Perceptual ad blocking meets adversarial machine learning," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2005–2021.

[43] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[44] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, "Emnist: Extending mnist to handwritten letters," in *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 2921–2926.

[45] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009, https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf [Accessed April 20 2023].

[46] G. Fonts, "Google Fonts," 2021, https://fonts.google.com/ [Accessed April 20 2023].

[47] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[48] N. Papernot, P. McDaniel, and I. Goodfellow, "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples," *arXiv preprint arXiv:1605.07277*, 2016.

[49] aymericdamien, "GitHub - aymericdamien/TensorFlow-Examples: TensorFlow Tutorial and Examples for Beginners (support TF v1 & v2)," 2020, https://github.com/aymericdamien/TensorFlow-Examples [Accessed April 20 2023].

[50] arconsis, "GitHub - arconsis/cifar-10-with-tensorflow2," 2019, https://github.com/arconsis/cifar-10-with-tensorflow2 [Accessed Oct 16 2022].

[51] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[52] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[53] S.-A. Rebuffi, S. Gowal, D. A. Calian, F. Stimberg, O. Wiles, and T. Mann, "Fixing data augmentation to improve adversarial robustness," *arXiv preprint arXiv:2103.01946*, 2021.

[54] S. Gu and L. Rigazio, "Towards deep neural network architectures robust to adversarial examples," *arXiv preprint arXiv:1412.5068*, 2014.

[55] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, "On detecting adversarial perturbations," *arXiv preprint arXiv:1702.04267*, 2017.

[56] W. Xu, D. Evans, and Y. Qi, "Feature squeezing: Detecting adversarial examples in deep neural networks," *arXiv preprint arXiv:1704.01155*, 2017.

[57] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *Artificial intelligence safety and security*. Chapman and Hall/CRC, 2018, pp. 99–112.

[58] Y. Dong, F. Liao, T. Pang, X. Hu, and J. Zhu, "Discovering adversarial examples with momentum," *arXiv preprint arXiv:1710.06081*, vol. 5, 2017.

[59] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 IEEE Symposium on Security and Privacy (SP)*. Ieee, 2017, pp. 39–57.

[60] F. Croce and M. Hein, "Minimally distorted adversarial examples with a fast adaptive boundary attack," in *International Conference on Machine Learning*. PMLR, 2020, pp. 2196–2205.

[61] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 582–597.

[62] Y. Sadko, "Guide to 10 font characteristics and their use in design," jan 2021, https://eugenesadko.medium.com/guide-to-10-font-characteristics-and-their-use-in-design-b0a07cc66f7 [Accessed Nov 24 2022].

[63] M. Gelderman, "A short introduction to font characteristics," *TUGboat*, June 1999.

[64] G. F. Elsayed, S. Shankar, B. Cheung, N. Papernot, A. Kurakin, I. Goodfellow, and J. Sohl-Dickstein, "Adversarial examples that fool both computer vision and time-limited humans," *arXiv preprint arXiv:1802.08195*, 2018.

[65] A. Quach, A. Prakash, and L. Yan, "Debloating software through piecewise compilation and loading," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 869–886.

[66] H. A. Org, "Report:loading speed," Nov 2020, https://httparchive.org/reports/loading-speed#fcp [Accessed April 20 2023].

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

[68] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum, "Botgraph: large scale spamming botnet detection." in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09, vol. 9, 2009, pp. 321–334.

[69] L. Von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum, "reCAPTCHA: Human-based character recognition via web security measures," *Science*, vol. 321, no. 5895, pp. 1465–1468, 2008.

[70] W. Zeller and E. W. Felten, "Cross-site request forgeries: Exploitation and prevention," *The New York Times*, pp. 1–13, 2008.

[71] MDN, "Samesite cookies - http — mdn," May 2022, https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite [Accessed Nov 26 2022].

[72] M. Yu, V. D. Gligor, and Z. Zhou, "Trusted display on untrusted commodity platforms," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 989–1003.

[73] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. USA: IEEE Computer Society, 2012, p. 616–630.

[74] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza, "Trustjs: Trusted client-side execution of JavaScript," in *Proceedings of the 10th European Workshop on Systems Security*, 2017, pp. 1–6.

[75] H. Shuang, W. Huang, P. Bettadpur, L. Zhao, I. Pustogarov, and D. Lie, "Using inputs and context to verify user intentions in internet services," in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '19. New York, NY, USA: ACM, 2019, pp. 76–83.

[76] W. Cui, R. H. Katz, and W.-t. Tan, "Binder: An extrusion-based break-in detector for personal computers," in *Proceedings of the 2005 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, April 2005.

[77] A. Filyanov, J. M. McCuney, A.-R. Sadeghiz, and M. Winandy, "Unidirectional trusted path: Transaction confirmation on just one device," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, ser. DSN '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–12.

[78] G. Petracca, A.-A. Reineh, Y. Sun, J. Grossklags, and T. Jaeger, "Aware: Preventing abuse of privacy-sensitive sensors via operation bindings," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 379–396.