Veil: Private Browsing Semantics Without Browser-side Assistance

Frank Wang
MIT CSAIL
frankw@mit.edu

James Mickens
Harvard University
mickens@g.harvard.edu

Nickolai Zeldovich
MIT CSAIL
nickolai@csail.mit.edu

***Abstract*—All popular web browsers offer a "private browsing mode." After a private session terminates, the browser is supposed to remove client-side evidence that the session occurred. Unfortunately, browsers still leak information through the file system, the browser cache, the DNS cache, and on-disk reflections of RAM such as the swap file.**

Veil is a new deployment framework that allows web developers to prevent these information leaks, or at least reduce their likelihood. Veil leverages the fact that, even though developers do not control the client-side browser implementation, developers do control 1) the content that is sent to those browsers, and 2) the servers which deliver that content. Veil web sites collectively store their content on Veil's *blinding servers* instead of on individual, site-specific servers. To publish a new page, developers pass their HTML, CSS, and JavaScript files to Veil's compiler; the compiler transforms the URLs in the content so that, when the page loads on a user's browser, URLs are derived from a secret user key. The blinding service and the Veil page exchange encrypted data that is also protected by the user's key. The result is that Veil pages can safely store encrypted content in the browser cache; furthermore, the URLs exposed to system interfaces like the DNS cache are unintelligible to attackers who do not possess the user's key. To protect against post-session inspection of swap file artifacts, Veil uses heap walking (which minimizes the likelihood that secret data is paged out), content mutation (which garbles in-memory artifacts if they do get swapped out), and DOM hiding (which prevents the browser from learning site-specific HTML, CSS, and JavaScript content in the first place). Veil pages load on unmodified commodity browsers, allowing developers to provide stronger semantics for private browsing without forcing users to install or reconfigure their machines. Veil provides these guarantees even if the user does not visit a page using a browser's native privacy mode; indeed, Veil's protections are *stronger* than what the browser alone can provide.

I. Introduction

Web browsers are the client-side execution platform for a variety of online services. Many of these services handle sensitive personal data like emails and financial transactions. Since a user's machine may be shared with other people, she may wish to establish a *private session* with a web site, such that the session leaves no persistent client-side state that can later be examined by a third party. Even if a site does not handle personally identifiable information, users may not want to leave evidence that a site was even visited. Thus, all popular browsers implement a private browsing mode which tries to remove artifacts like entries in the browser's "recently visited" URL list.

Unfortunately, implementations of private browsing mode still allow sensitive information to leak into persistent storage [2], [28], [35], [46]. Browsers use the file system or a SQLite database to temporarily store information associated with private sessions; this data is often incompletely deleted and zeroed-out when a private session terminates, allowing attackers to extract images and URLs from the session. During a private session, web page state can also be reflected from RAM into swap files and hibernation files; this state is in cleartext, and therefore easily analyzed by curious individuals who control a user's machine after her private browsing session has ended. Simple greps for keywords are often sufficient to reveal sensitive data [2], [28].

Web browsers are complicated platforms that are continually adding new features (and thus new ways for private information to leak). As a result, it is difficult to implement even seemingly straightforward approaches for strengthening a browser's implementation of incognito modes. For example, to prevent secrets in RAM from paging to disk, the browser could use OS interfaces like `mlock()` to pin memory pages. However, pinning may interfere in subtle ways with other memory-related functionality like garbage collecting or tab discarding [50]. Furthermore, the browser would have to use `mlock()` indiscriminately, on *all* of the RAM state belonging to a private session, because the browser would have no way to determine which state in the session is actually sensitive, and which state can be safely exposed to the swap device.

In this paper, we introduce Veil, a system that allows web developers to implement private browsing semantics for their own pages. For example, the developers of a whisteblowing site can use Veil to reduce the likelihood that employers can find evidence of visits to the site on workplace machines. Veil's privacy-preserving mechanisms are enforced *without assistance from the browser*—even if users visit pages using a browser's built-in privacy mode, Veil provides stronger assurances that can only emerge from an intentional composition of HTML, CSS, and JavaScript. Veil leverages five techniques to protect privacy: URL blinding, content mutation, heap walking, DOM hiding, and state encryption.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23117
www.ndss-symposium.org

- Developers pass their HTML and CSS files through Veil's compiler. The compiler locates cleartext URLs in the content, and transforms those raw URLs into *blinded references* that are derived from a user's secret key and are cryptographically unlinkable to the original URLs. The compiler also injects a runtime library into each page; this library interposes on dynamic content fetches (e.g., via `XMLHttpRequests`), and forces those requests to also use blinded references.
- The compiler uploads the objects in a web page to Veil's *blinding servers*. A user's browser downloads content from those blinding servers, and the servers collaborate with a page's JavaScript code to implement the blinded URL protocol. To protect the client-side memory artifacts belonging to a page, the blinding servers also *dynamically mutate* the HTML, CSS, and JavaScript in a page. Whenever a user fetches a page, the blinding servers create syntactically different (but semantically equivalent) versions of the page's content. This ensures that two different users of a page will each receive unique client-side representations of that page.
- Ideally, sensitive memory artifacts would never swap out in the first place. Veil allows developers to mark JavaScript state and renderer state as sensitive. Veil's compiler injects *heap walking code* to keep that state from swapping out. The code uses JavaScript reflection and forced DOM relayouts to periodically touch the memory pages that contain secret data. This coerces the OS's least-recently-used algorithm to keep the sensitive RAM pages in memory.
- Veil sites which desire the highest level of privacy can opt to use Veil's *DOM hiding* mode. In this mode, the client browser essentially acts as a dumb graphical terminal. Pages are rendered on a content provider's machine, with the browser sending user inputs to the machine via the blinding servers; the content provider's machine responds with new bitmaps that represent the updated view of the page. In DOM hiding mode, the page's unique HTML, CSS, and JavaScript content is never transmitted to the client browser.
- Veil also lets a page store private, persistent state by *encrypting* that state and by naming it with a blinded reference that only the user can generate.

By using blinded references for all content names (including those of top-level web pages), Veil avoids information leakage via client-side, name-centric interfaces like the DNS cache [19], the browser cache, and the browser's address bar. Encryption allows a Veil page to safely leverage the browser cache to reduce page load times, or store user data across different sessions of the private web page. A page that desires the highest level of security will eschew even the encrypted cache, and use DOM hiding; in concert with URL blinding, the hiding of DOM content means that the page will generate *no greppable state in RAM or persistent storage* that could later be used to identify the page. Table I summarizes the different properties of Veil's two modes for private browsing.

In summary, **Veil is the first web framework that allows developers to implement privacy-preserving browsing semantics for their own pages.** These semantics are stronger than those provided by native in-browser incognito modes; however, Veil pages load on commodity browsers, and do not require users to reconfigure their systems or run their browsers within a special virtual machine [17]. Veil can translate legacy pages to more secure versions automatically, or with minimal developer assistance (§V), easing the barrier to deploying privacy-preserving sites. Experiments show that Veil's overheads are moderate: 1.25x–3.25x for Veil with encrypted client-side storage, mutated DOM content, and heap walking; and 1.2x–2.1x for Veil in DOM hiding mode.

II. Deployment Model

Veil uses an opt-in model, and is intended for web sites that want to actively protect client-side user privacy. For example, a whistleblowing site like SecureDrop [75] is incentivized to hide client-side evidence that the SecureDrop website has been visited; strong private browsing protections give people confidence that visiting SecureDrop on a work machine will not lead to incriminating aftereffects. As another example of a site that is well-suited for Veil, consider a web page that allows teenagers to find mental health services. Teenagers who browse the web on their parents' machines will desire strong guarantees that the machines store no persistent records of private browsing activity.

Participating Veil sites must explicitly recompile their content using the Veil compiler. This requirement is not unduly burdensome, since all non-trivial frameworks for web development impose a developer-side workflow discipline. For example, Aurelia [9], CoffeeScript [12], and Meteor [38] typically require a compilation pass before content can go live.

Participating Veil sites must also explicitly serve their content from Veil blinding servers. Like Tor servers [15], Veil's blinding servers can be run by volunteers, although content providers can also contribute physical machines or VMs to the blinding pool (§IV-B).

Today, many sites are indifferent towards the privacy implications of web browsing; other sites are interested in protecting privacy, but lack the technical skill to do so; and others are actively invested in using technology to hide sensitive user data. Veil targets the latter two groups of site operators. Those groups are currently in the minority, but they are growing. An increasing number of web services define their value in terms of privacy protections [16], [18], [53], [54], and recent events have increased popular awareness of privacy issues [49]. Thus, we believe that frameworks like Veil will become more prevalent as users demand more privacy, and site operators demand more tools to build privacy-respecting systems.

III. Threat Model

As described in Section II, Veil assumes that a web service is actively interested in preserving its users' client-side privacy. Thus, Veil trusts web developers and the blinding servers. Veil's goal is to defend the user against local attackers who take control of a user's machine *after* a private session terminates. If an attacker has access to the machine *during* a private session, the attacker can directly extract sensitive data, e.g., via keystroke logging or by causing the browser to core dump; such exploits are out-of-scope for this paper.

Veil models the post-session attacker as a skilled system administrator who knows the location and purpose of the swap

Browsing mode	Persistent, per-site client-side storage	Information leaks through client-side, name-based interfaces	Per-site browser RAM artifacts	GUI interactions
Regular browsing	Yes (cleartext by default)	Yes	Yes	Locally processed
Regular incognito mode	No	Yes	Yes	Locally processed
Veil with encrypted client-side storage, mutated DOM content, heap walking	Yes (always encrypted)	No (blinding servers)	Yes (but mutated and heap-walked)	Locally processed
Veil with DOM hiding	No	No (blinding servers)	No	Remotely processed

TABLE I. A COMPARISON BETWEEN VEIL'S TWO BROWSING MODES, REGULAR INCOGNITO BROWSING, AND REGULAR BROWSING THAT DOES NOT USE INCOGNITO MODE.

file, the browser cache, and files like `/var/log/*` that record network activity like DNS resolution requests. Such an attacker can use tools like `grep` or `find` to look for hostnames, file types, or page content that was accessed during a Veil session. The attacker may also possess off-the-shelf forensics tools like Mandiant Redline [36] that look for traces of private browsing activity. However, the attacker lacks the skills to perform a customized, low-level forensics investigation that, e.g., tries to manually extract C++ data structures from browser memory pages that Veil could not prevent from swapping out.

Given this attacker model, Veil's security goals are weaker than strict forensic deniability [17]. However, Veil's weaker type of forensic resistance is both practically useful and, in many cases, the strongest guarantee that can be provided without forcing users to run browsers within special OSes or virtual machines. Veil's goal is to load pages within *unmodified* browsers that run atop *unmodified* operating systems. Thus, Veil is forced to implement privacy-preserving features using browser and OS interfaces that are unaware of Veil's privacy goals. These constraints make it impossible for Veil to provide strict forensic deniability. However, most post-session attackers (e.g., friends, or system administrators at work, Internet cafes, or libraries) will lack the technical expertise to launch FBI-style forensic investigations.

Using blinded URLs, Veil tries to prevent data leaks through system interfaces that use network names. Examples of name-based interfaces are the browser's "visited pages" history, the browser cache, cookies, and the DNS cache (which leaks the hostnames of the web servers that a browser contacts [2]). It is acceptable for the attacker to learn that a user has contacted Veil's blinding servers—those servers form a large pool whose hostnames are generic (e.g., `veil.io`) and do not reveal any information about particular Veil sites (§IV-B).

Veil assumes that web developers only include trusted content that has gone through the Veil compiler. A page may embed third party content like a JavaScript library, but the Veil compiler analyzes both first party and third party content during compilation (§IV-A).

Heap walking (§IV-E) allows Veil to prevent sensitive RAM artifacts from swapping to disk. Veil does not try to stop information leaks from GPU RAM [31], but GPU RAM is never swapped to persistent storage. Poorly-written or malicious browser extensions that leak sensitive page data [32] are also outside the scope of this paper.

IV. DESIGN

As shown in Figure 1, the Veil framework consists of three components. The *compiler* transforms a normal web page into a new version that implements static and dynamic privacy protections. Web developers upload the compiler's output to *blinding servers*. These servers act as intermediaries between content publishers and content users, mutating and encrypting content. To load the Veil page, a user first loads a small *bootstrap page*. The bootstrap asks for a per-user key and the URL of the Veil page to load; the bootstrap then downloads the appropriate objects from the blinding servers and dynamically overwrites itself with the privacy-preserving content in the target page.

In the remainder of this section, we describe Veil's architecture in more detail. Our initial discussion involves a simple, static web page that consists of an HTML file, a CSS file, a JavaScript file, and an image. We iteratively refine Veil's design to protect against various kinds of privacy leakage. Then, we describe how Veil handles more complex pages that dynamically fetch and generate new content.

A. The Veil Compiler and veilFetch()

The compiler processes the HTML in our example page (Figure 1), and finds references to three external objects (i.e., the CSS file, the JavaScript file, and the image). The compiler computes a hash value for each object, and then replaces the associated HTML tags with dynamic JavaScript loaders for the objects. For example, if the original image tag looked like this:

```
<img src="http://foo.com/img.jpg"/>
```

the compiler would replace that tag with the following:

```
<script>obscuraFetch("b6a0d...");</script>
```

where the argument to `veilFetch()` is the hash name of the image. At page load time, when `veilFetch()` runs, it uses an `XMLHttpRequest` request to download the appropriate object from the blinding service. In our example, the URL in the `XMLHttpRequest` might be `http://veil.io/b6a0d...`.

Such a URL resides in the domain of the blinding servers, not the domain of the original content publisher. Furthermore, the URL identifies the underlying object by hash, so the URL itself does not leak information about the original publisher or the data contained within the object. So, even though the execution of `veilFetch()` may pollute name-based interfaces like the DNS cache, a post-session attacker which inspects those registries cannot learn anything about the content that was fetched. However, a network-observing attacker who sees a `veilFetch()` URL can simply ask the blinding server for

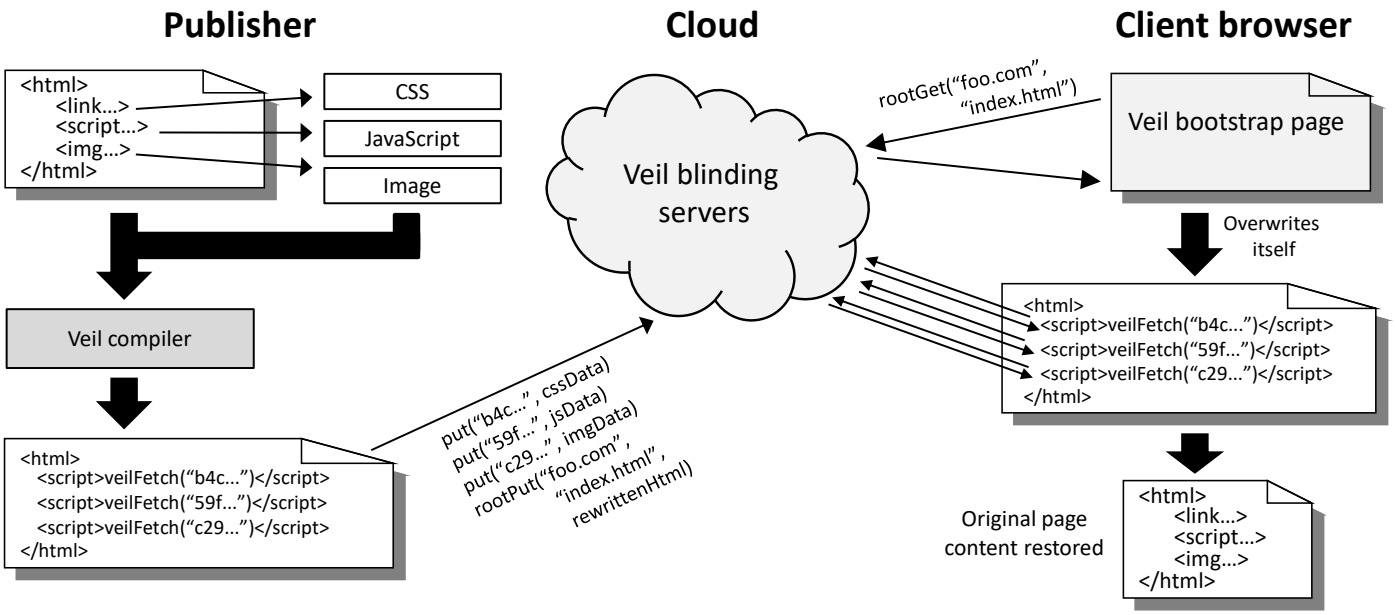

Fig. 1. The Veil architecture (cryptographic operations omitted for clarity).

the associated content, and then directly inspect the data that the user accessed during the private session!

To defend against such an attack, Veil associates each user with a symmetric key k_{user} (we describe how this key is generated and stored in Section IV-D). Veil also associates the blinding service with a public/private keypair. When `veilFetch(hashName)` executes, it does not ask the blinding service for `hashName`—instead, it asks for `<hashName>`$_{k_{user}}$. In the HTTP header for the request, `veilFetch()` includes $< k_{user} >_{PubKeyBServ}$, i.e., the user's symmetric key encrypted by the blinding service's public key. When the blinding service receives the request, it uses its private key to decrypt $< k_{user} >_{PubKeyBServ}$. Then, it uses $< k_{user} >$ to extract the hash name of the requested object. The blinding service locates the object, encrypts it with k_{user}, and then sends the HTTP response back to the client. Figure 2 depicts the full protocol.[1] In practice, the blinding service's public/private keypair can be the service's TLS keypair, as used by HTTPS connections to the service. Thus, the encryption of k_{user} can be encrypted by the standard TLS mechanisms used by an HTTPS connection.

Once `veilFetch()` receives the response, it decrypts the data and then dynamically reconstructs the appropriate object, replacing the script tag that contained `veilFetch()` with the reconstructed object. The compiler represents each serialized object using JSON [13]; Figure 3 shows an example of a serialized image. To reinflate the image, `veilFetch()` extracts metadata like the image's width and height, and dynamically injects an image tag into the page which has the appropriate attributes. Then, `veilFetch()` extracts the Base64-encoded image data from the JSON, and sets the `src` attribute of the image tag to a data URL [37] which directly embeds the Base64 image data. This causes the browser to load the image. `veilFetch()` uses similar techniques to reinflate other content types.

[1] A stateful blinding service can cache decrypted user keys and eliminate the public key operation from all but the user's first request.

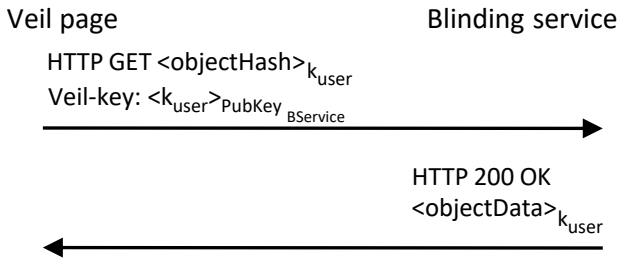

Fig. 2. The `veilFetch()` protocol.

```
{"img_type": "jpeg",
 "dataURI": "ab52f...",
 "tag_attrs": {"width": "20px",
               "height": "50px"}}
```

Fig. 3. A serialized `<img>` tag.

This client-server protocol has several nice properties. First, it solves the replay problem—if an attacker wants to replay old fetches, or guess visited URLs (as in a CSS-based history attack [29], [71]), the attacker will not be able to decrypt the responses unless she has the user's key. Also, since the blinding service returns encrypted content, that encrypted content is what would reside in the browser cache. Thus, Veil pages can now persist data in the browser cache such that only the user can decrypt and inspect that content. Of course, a page does not have to use the browser cache—when a publisher uploads an object to the blinding service, the publisher indicates the caching headers that the service should return to clients.

In addition to uploading data objects like images to the blinding service, the compiler also uploads "root" objects. Root objects are simply top-level HTML files like `foo.com/index.html`. Root objects are signed with the publisher's private key, and are stored in a separate namespace from data objects using a 2-tuple key that consists of the publisher name (`foo.com`) and the HTML name (`index.html`). Unlike data objects, which are named by

hash (and thus self-verifying), root objects change over time as the associated HTML evolves. Thus, root objects are signed by the publisher to guarantee authenticity and allow the blinding service to reject fradulent updates.

B. The Blinding Service

In the previous section, we described the high-level operation of the blinding service. It exports a key/value interface to content publishers, and an HTTP interface to browsers. The HTTP code path does content encryption as described above. As described in Section IV-F, the blinding service also performs *content mutation* to protect RAM artifacts that spill to disk; mutation does not provide cryptographically strong protections, but it does significantly raise the difficulty of post-session forensics. The blinding servers also implement the DOM hiding protocol (§IV-H), which Veil sites can use to prevent exposing *any* site-specific HTML, CSS, or JavaScript to client browsers.

The blinding service can be implemented in multiple ways, e.g., as a peer-to-peer distributed hash table [58], [62], a centralized service that is run by a trusted authority like the EFF, or even a single cloud VM that is paid for and operated by a single privacy-conscious user. In practice, we expect a blinding service to be run by an altruistic organization like the EFF, or by altruistic individuals (as in Tor [15]), or by a large set of privacy-preserving sites who will collaboratively pay for the cloud VMs that run the blinding servers. Throughout the paper, we refer to a single blinding service `veil.io` for convenience. However, independent entities can simultaneously run independent blinding services.

Veil's publisher-side protocol is compatible with accounting, since the blinding service knows which publisher uploaded each object, and how many times that object has been downloaded by clients. Thus, it is simple for a cloud-based blinding service to implement proportional VM billing, or cost-per-download billing. In contrast, an altruistic blinding service (e.g., implemented atop a peer-to-peer DHT [58], [62]) could host anonymous object submissions for free.

C. The Same-origin Policy

A single web page can aggregate content from a variety of different origins. As currently described, Veil maps all of these objects to a single origin: at compile time, Veil downloads the objects from their respective domains, and at page load time, Veil serves all of the objects from `https://veil.io`.

The browser uses the same-origin policy [59] to constrain the interactions between content from different origins. Mapping all content to a single origin would effectively disable same-origin security checks. Thus, Veil's static rewriter injects the `sandbox` attribute [51] into all `<iframe>` tags. Using this attribute, the rewriter forces the browser to give each frame a unique origin with respect to the same-origin policy. This means that, even though all frames are served from the `veil.io` domain, they cannot tamper with each other's JavaScript state. In our current implementation of the compiler, developers are responsible for ensuring that dynamically-generated frames are also tagged with the `sandbox` attribute; however, using DOM virtualization [27], [40], the compiler could inject DOM interpositioning code that automatically injects `sandbox` attributes into dynamically-generated frames.

DOM storage [69] exposes the local disk to JavaScript code using a key/value interface. DOM storage is partitioned by origin, i.e., a frame can only access the DOM storage of its own domain. By assigning an ephemeral, unique origin to each frame, Veil seemingly prevents an origin from reliably persisting data across multiple user sessions of a Veil page. To solve this problem, Veil uses indirection. When a frame wants to access DOM storage, it first creates a child frame which has the special URL `https://veil.io/domStorage`. The child frame provides Veil-mediated access to DOM storage, accepting read and write requests from the parent frame via `postMessage()`. Veil associates a private storage area with a site's public key, and engages in a challenge/response protocol with a frame's content provider before agreeing to handle the frame's IO requests; the challenge/response traffic goes through the blinding servers (§IV-G). The Veil frame that manages DOM storage employs the user's key to encrypt and integrity-protect data before writing it, ensuring that post-session attackers cannot extract useful information from DOM storage disk artifacts.

Since Veil assigns random, ephemeral origins to frames, cookies do not work in the standard way. To simulate persistent cookies, an origin must read or write values in DOM storage. Sending a cookie to a server also requires explicit action. For example, a Veil page which contains personalized content might use an initial piece of non-personalized JavaScript to find the local cookie and then generate a request for dynamic content (§IV-G).

D. The Bootstrap Page

Before the user can visit any Veil sites, she must perform a one-time initialization step with the Veil bootstrap page (e.g., `https://veil.io`). The bootstrap page generates a private symmetric key for the user and places it in local DOM storage, protecting it with a user-chosen password. Veil protects the in-memory versions of the password and symmetric key with heap walking (§IV-E) to prevent these cleartext secrets from paging to disk.

Later, the user determines the URL (e.g., `foo.com/index.html`) of a Veil site to load. The user should discover this URL via an already-known Veil page like a directory site, or via out-of-band mechanisms like a traditional web search on a different machine than the one needing protection against post-session attackers; looking for Veil sites using a traditional search engine on the target machine would pollute client-side state with greppable content. Once the user possesses the desired URL, she returns to the bootstrap page. The bootstrap prompts the user for her password, extracts her key from local storage, and decrypts it with the password. The bootstrap then prompts the user for the URL of the Veil page to visit. The bootstrap fetches the root object for the page. Then, the bootstrap overwrites itself with the HTML in the root object. Remember that this HTML is the output of the Veil compiler; thus, as the browser loads the HTML, the page will use `veilFetch()` to dynamically fetch and reinflate encrypted objects.

Once the bootstrap page overwrites itself, the user will see the target page. However, no navigation will have occurred, i.e., the browser's address bar will still say `https://veil.io`. Thus, the browser's history of visited pages will never include the URL of a particular Veil page, only the URL of the generic Veil bootstrap. The compiler rewrites links within a page so that, if the user clicks a link, the page will fetch the relevant content via a blinded URL, and then deserialize and evaluate that content as described above.

E. Protecting RAM Artifacts

As a Veil page creates new JavaScript objects, the browser transparently allocates physical memory pages on behalf of the site. Later, the OS may swap those pages to disk if memory pressure is high and those pages are infrequently used. JavaScript is a high-level, garbage-collected language that does not expose raw memory addresses. Thus, browsers do not define JavaScript interfaces for pinning memory, and Veil has no explicit way to prevent the OS from swapping sensitive data to disk.

By frequently accessing sensitive JavaScript objects, Veil *can* ensure that the underlying memory pages are less likely to be selected by the OS's LRU replacement algorithm. Veil's JavaScript runtime defines a `markAsSensitive(obj)` method; using this method, an application indicates that Veil should try to prevent `obj` from paging to disk. Internally, Veil maintains a list of all objects passed to `markAsSensitive()`. A periodic timer walks this list, accessing every property of each object using JavaScript reflection interfaces. Optionally, `markAsSensitive()` can recurse on each object property, and touch every value in the object tree rooted by `obj`. Such recursive traversals make it easier for developers to mark large sets of objects at once. JavaScript defines a special `window` object that is an alias for the global namespace, so if an application marks `window` as recursively sensitive, Veil will periodically traverse the entire heap graph that is reachable from global variables. Using standard techniques from garbage collection algorithms, Veil can detect cycles in the graph and avoid infinite loops.

`markAsSensitive()` maintains references to all of the sensitive objects that it has ever visited. This prevents the browser from garbage collecting the memory and possibly reusing it without applying secure deallocation [11]. At page unload time, Veil walks the sensitive list a final time, deleting all object properties. Since JavaScript does not expose raw memory, Veil cannot `memset()` the objects to zero, but deleting the properties does make it more difficult for a post-session attacker to reconstruct object graphs.

Sensitive data can reside in the JavaScript heap, but it can also reside in the memory that belongs to the renderer. The renderer is the browser component that parses HTML, calculates the screen layout, and repaints the visual display. For example, if a page contains an HTML tag like `<b>Secret</b>`, the cleartext string `Secret` may page out from the renderer's memory. As another example, a rendered page's image content may be sensitive.

The renderer is a C++ component that is separate from the JavaScript engine; JavaScript code has no way to directly access renderer state. However, JavaScript can indirectly touch renderer memory through preexisting renderer interfaces. For example, if the application creates an empty, invisible `<img>` tag, and injects the tag into the page's HTML, the browser invalidates the page's layout. If the application then reads the size of the image tag's parent, the browser is forced to recalculate the layout of the parent tag. Recalculating the layout touches renderer memory that is associated with the parent tag (and possibly other tags). Thus, Veil can walk the renderer memory by periodically injecting invisible tags throughout the HTML tree (forcing a relayout) and then removing those tags, restoring the original state of the application.

The browser's network stack contains memory buffers with potentially sensitive content from the page. However, Veil only transmits encrypted data over the network, so network buffers reveal nothing to an attacker if they page out to disk and are subsequently recovered. Importantly, Veil performs heap walking on the user's password and symmetric key. This prevents those secrets from paging out and allowing an attacker to decrypt swapped out network buffers.

F. Mutation Techniques

Veil's main protection mechanism for RAM artifacts is heap walking, and we show in Section VII-C that heap walking is an effective defense during expected rates of swapping. However, Veil provides a second line of defense via content mutation. Mutation ensures that, each time a client loads a page, the page will return different HTML, CSS, and JavaScript, even if the baseline version of the page has not changed. Mutation makes grep-based attacks more difficult, since the attacker cannot simply navigate to a non-Veil version of a page, extract identifying strings from the page, and then grep local system state for those strings. Content mutation is performed by the blinding servers (§IV-B); below, we briefly sketch some mutation techniques that the blinding servers can employ.

Note that blinding servers can mutate content in the background, *before* the associated pages are requested by a client. For example, blinding servers can store a pool of mutated versions for a single object, such that, when a client fetches HTML that refers to the object, the blinding server can late-bind the mutated version that the page references. Using this approach, mutation costs need not be synchronous overheads that are paid when a client requests a page.

JavaScript: To mutate JavaScript files, the blinding service uses techniques that are adapted from metamorphic viruses [72]. Metamorphic viruses attempt to elude malware scanners by ensuring that each instantiation of the virus has syntactically different code that preserves the behavior of the base implementation. For example, functions can be defined in different places, and implemented using different sequences of assembler instructions that result in the same output.

Our prototype blinding service mutates JavaScript code using straightforward analogues of the transformations described above. JavaScript code also has a powerful advantage that assembly code lacks—the `eval()` statement provides a JavaScript program with the ability to emit new mutated code at runtime. Such "`eval()`-folding" is difficult to analyze [14],

particularly if the attacker can only recover a partial set of RAM artifacts for a page.[2]

Note that if a faulty blinding server forgets to mutate invocations of `veilFetch(hashName)`, then unscrambled object hash names may be paged out to disk in JavaScript source code! If an attacker recovered such artifacts, he could directly replay the object fetches that were made by the private session. Thus, JavaScript mutation is a core responsibility for the blinding service.

HTML and CSS: The grammars for HTML and CSS are extremely complex and expressive. Thus, there are many ways to represent a canonical piece of HTML or CSS [24]. For example, HTML allows a character to be encoded as a raw binary symbol in a character encoding like `UTF-8` or `Unicode-16`. HTML also allows characters to be expressed as escape sequences known as HTML entities. An HTML entity consists of the token "&#" followed by the Unicode code point for a character and then a semicolon. For instance, the HTML entity for "a" is "a". The HTML specification allows an HTML entity to have leading zeroes which the browser ignores; the specification also allows for code points to be expressed in hexadecimal. Thus, to defeat simple exact-match greps of HTML artifacts, the blinding service can randomly replace native characters with random HTML entity equivalents.

There are a variety of more sophisticated techniques to obfuscate HTML and CSS. For a fuller exploration of these topics, we defer the reader to other work [24]. Our blinding service prototype uses random HTML entity mutation. It also obscures the HTML structure of the page using randomly inserted tags which do not affect the user-perceived visual layout of the page.

Images: The blinding service can automatically mutate images in several ways. For example, the blinding service can select one of several formats for a returned image (e.g., JPEG, PNG, GIF, etc.). Each instantiation of the image can have a different resolution, as well as different filters that are applied to different parts of the visual spectrum. Web developers can also use application-specific knowledge to generate more aggressive mutations, such as splitting a single base image into two semi-transparent images that are stacked atop each other by client-side JavaScript. As explained in our threat model (§III), Veil does not protect against leaks of the raw display bitmap that resides in GPU memory; thus, the mutation techniques from above are sufficient to thwart grep-based forensics on memory artifacts from the DOM tree. For a more thorough discussion of image mutation techniques that thwart classification algorithms, we defer to literature from the computer vision community [7].

G. Dynamic Content

At first glance, Veil's compile-time binding of URLs to objects seems to prevent a publisher from dynamically generating personalized user content. However, Veil can support dynamic content generation by using the blinding service as a proxy that sits between the end-user and the publisher. More specifically, a Veil page can issue an HTTP request with a `msg-type` of "forward". The body of the request contains two things: user information like a site-specific Veil cookie (§IV-C), and a publisher name (e.g., `foo.com`). The page gives the request a random hash name, since the page will not cache the response. When the blinding service receives the request, it forwards the message to the publisher's dynamic content server. The publisher generates the dynamic content from the provided user information, and then sends the content to the blinding service, who forwards it to the client as the HTTP response to the client's "forward" request. The client and the publisher can encrypt the user information and the personalized content if the blinding service is not trusted with user-specific data; in this scenario, the content provider's web server is responsible for mutating objects before returning them to the client. Regardless, the content provider must compile dynamically-generated content (§IV-A and §V). Fortunately, the compilation cost for a single dynamic object is typically small. For example, compiling a 100 KB image requires Base64-encoding it and generating a few metadata fields, taking roughly 75 ms. Content providers can compile multiple dynamic objects in parallel.

H. DOM Hiding

Heap walking reduces the likelihood that in-memory browser state will swap to disk. Content mutation ensures that, if state does swap out, then the state will not contain greppable artifacts from a canonical version of the associated page. However, some Veil sites will be uncomfortable with sending *any* site-specific HTML, CSS, or JavaScript to a client, even if that content is mutated. For example, a site might be concerned that a determined sysadmin can inspect swapped-out fragments of mutated HTML, and try to reverse-engineer the mutation by hand.

To support these kinds of sites, Veil provides a mode of operation called DOM hiding. In DOM hiding mode, the user's browser essentially acts as a thin client, with the full version of the page loaded on a remote server that belongs to the content provider. The user's browser employs a generic, page-agnostic JavaScript library to forward GUI events to the content provider through the blinding service; the content provider's machine applies each GUI event to the server-side page, and then returns an image that represents the new state of the page.

The advantage of DOM hiding is that site-specific HTML, CSS, and JavaScript is never pushed to the user's browser. The disadvantage is that each GUI interaction now suffers from a synchronous wide-area latency. For some Veil sites, this trade-off will be acceptable. We characterize the additional interactive latency in Section VII-D.

Figure 4 provides more details about how Veil implements DOM hiding. The Veil bootstrap page receives the URL to load from the user, as described in Section IV-D. The bootstrap page then issues an initial HTTP request through the blinding servers to the content provider. The content provider returns a page-agnostic remoting stub; this stub merely implements the client-side of the remote GUI architecture. As the content

[2]Some .NET viruses already leverage access to the runtime's reflection interface to dynamically emit code [65].

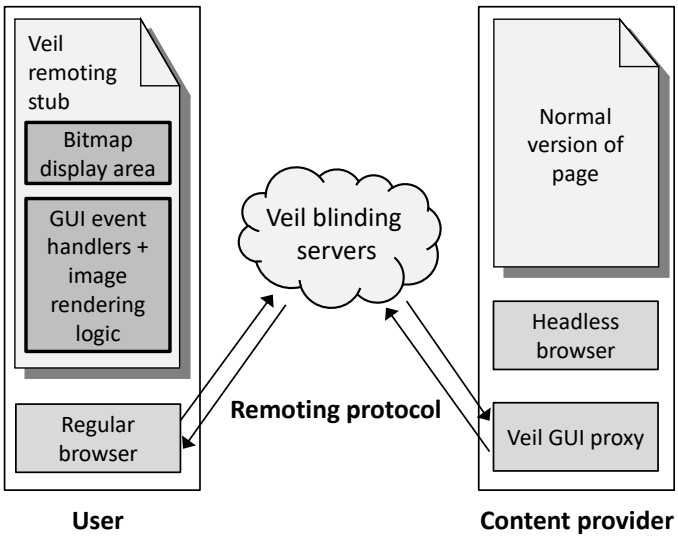

Fig. 4. With DOM hiding, the client-side remoting stub sends GUI events to the content provider, and receives bitmaps representing new page states. The page's raw HTML, CSS, and JavaScript are never exposed to the client.

provider returns the stub to the user, the provider also launches a headless browser[3] like PhantomJS [1] to load the normal (i.e., non-rewritten) version of the page. The content provider associates the headless browser with a Veil GUI proxy. The proxy uses native functionality in the headless browser to take an initial screenshot of the page. The GUI proxy then sends the initial screenshot via the blinding servers to the user's remoting stub. The stub renders the image, and uses page-agnostic JavaScript event handlers to detect GUI interactions like mouse clicks, keyboard presses, and scrolling activity. The stub forwards those events to the GUI proxy. The proxy replays those events in the headless browser, and ships the resulting screen images back to the client. Note that a page which uses DOM hiding will not use encrypted client-side browser caching (§IV-A) or DOM storage (§IV-C)—there will be no page-specific client-side state to store.

I. Discussion

Veil tries to eliminate cleartext client-side evidence of browsing activity. However, Veil does not prevent the server-side of a web page from tracking user information. Thus, Veil is compatible with preexisting workflows for ad generation and accounting (although advertising infrastructure must be modified to use blinded URLs and "forward" messages).

If a Veil page wants to use the browser cache, Veil employs encryption to prevent attackers from inspecting or modifying cache objects. However, an attacker may be able to fingerprint the site by observing the size and number of its cached objects. Sun et al. [63] provide a survey of techniques which prevent such fingerprinting attacks; their discussion is in the context of protecting HTTPS sessions, but their defensive techniques are equally applicable to Veil. The strongest defense is to reduce the number of objects in a page. Veil's compiler can easily do this by inlining objects into HTML [39]; for example, the compiler can directly embed CSS content that the original HTML incorporated via a link to an external file. The blinding service can also inject noise into the distribution of object sizes and counts. For example, when the service returns objects to clients, it can pad data sizes to fixed offsets, e.g., 2KB boundaries or power-of-2 boundaries. Alternatively, the blinding service can map object sizes for page X to the distribution for object sizes in a different page Y [73]. All of these defensive approaches hurt performance in some way—inlining and merging reduce object cacheability, and padding increases the amount of data which must be encrypted and transmitted over the network. Note that publishers must explicitly enable client-side caching, so paranoid sites can simply disable this feature.

[3]A headless browser is one that does not have a GUI. However, a headless browser *does* maintain the rest of the browser state; for example, DOM state can be queried using normal DOM methods, and modified through the generation of synthetic DOM events like mouse clicks.

V. Porting Legacy Applications

In this section, we describe how Veil helps developers to port legacy web pages to the Veil framework. In particular, we provide case studies which demonstrate how Veil's compiler and runtime library can identify unblinded fetches and, in some cases, automatically transform those fetches into blinded ones.

Raw XMLHttpRequests: Veil's compiler traverses a statically defined HTML tree, converting raw URLs into Veil hash pointers. However, a page's JavaScript code can use `XMLHttpRequests` to dynamically fetch new content. Veil's static HTML compiler does not interpose on such fetches, so they will generate unblinded transfers that pollute the client's DNS cache and browser cache.

In debugging mode, Veil's client library shims the JavaScript runtime [40] and interposes on the `XMLHttpRequest` interface. This allows Veil to inspect the URLs in `XMLHttpRequests` before the associated HTTP fetches are sent over the network. Veil drops unblinded requests and writes the associated URLs to a log. A web developer can then examine this log and determine how to port the URLs.

For static content, one porting solution is to leverage Veil's *AJAX maps*. Once the debugging client library has identified a page's raw `XMLHttpRequest` URLs, the library sends those URLs to Veil's HTML compiler. The compiler automatically fetches the associated objects and uploads them to the object servers. Additionally, when the compiler rewrites the HTML, it injects JavaScript code at the beginning of the HTML which maps the raw `XMLHttpRequest` URLs to the hash names of the associated objects. Later, when the page is executed by real users, Veil's shimmed `XMLHttpRequests` use the AJAX map to convert raw URLs to blinded references. Veil will drop requests that are not mentioned in the translation map. This approach is complete from the security perspective, since all unblinded `XMLHttpRequests` will be dropped. However, for this approach to please users (who do not want *any* requests to drop), Veil developers should use testing tools with good coverage [42], [45], [60] to ensure that all of a page's `XMLHttpRequest` URLs are mapped.

AJAX maps are unnecessary for native Veil pages which always generate blinded `XMLHttpRequests`. However,

URL validation via XMLHttpRequest shimming is useful when developers must deal with complex legacy libraries.

Dynamic tag generation: A legacy page can generate unblinded fetches by dynamically creating new DOM nodes that contain raw URLs in src attributes. For example, using document.createElement(), a page can inject a new <img> tag into its HTML. A page can also write to the innerHTML property of a preexisting DOM node, creating a new HTML subtree that is attached to the preexisting node. Neither type of tag creation will be captured by XMLHttpRequest shimming.

XMLHttpRequest shimming is a specific example of a more general technique called DOM virtualization. If desired, the entire DOM interface can be virtualized [3], [27], [41], allowing Veil to interpose on all mechanisms for dynamic tag creation. However, full DOM virtualization adds non-trivial performance overhead—the native DOM implementation is provided by the browser in fast C++ code, but a virtualized DOM is implemented by the application in comparatively slow JavaScript code. Furthermore, the full DOM interface is much more complex than the narrow XMLHttpRequest interface.

Our current implementation of Veil supports XMLHttpRequest shimming, but not full DOM virtualization. We leave the integration of Veil with a full virtualization system [26] as future work.

Unblinded links in CSS: CSS files can directly reference image objects using the url() statement, e.g., body{background: url('x.jpg')}. After the Veil compiler processes HTML files, it examines the associated CSS files and replaces raw image links with inline data URLs. Thus, when the Veil page loads a post-processed CSS file, the image data will be contained within the CSS itself, and will not require network fetches.

Angular.js: Angular [4] is a popular JavaScript framework that provides model-view-controller semantics for web applications. Angular uses a declarative model to express data bindings. For example, the {{}} operator is used to embed live views of the controller into HTML. The HTML snippet instructs Angular to dynamically update the content of the <img> whenever the JavaScript value controller.x changes. Many other popular frameworks define a similar templating mechanism [6], [55], [67].

The {{}} operator is not part of the official HTML grammar. To implement {{}} and other kinds of data binding, Angular uses a dynamic DOM node compiler. This compiler is a JavaScript file that runs at the end of the page load, when the initial DOM tree has been assembled. The compiler locates special Angular directives like {{}}, and replaces them with new JavaScript code and new DOM nodes that implement the data binding protocol.

Angular allows URLs to contain embedded {{}} expressions. Since these URLs are not resolved until runtime, Veil's static compiler cannot directly replace those URLs with blinded ones. However, Veil can rewrite Angular directives in a way that passes control to Veil code whenever a data binding changes. In the previous <img> example, Veil rewrites the tag as follows:

```
<img src="0" alt={{controller.x}}/>
```

The src attribute of the image is set to a network path which is known to be nonexistent (but whose URL does not leak private information). When the page tries to load the image, the load failure will invoke a custom onerror handler that Veil has attached to the window object. That handler will read the value of the alt attribute, which will contain the dynamic value of controller.x. Veil will then issue a blinded fetch for the associated image. In parallel, Veil also sets an Angular $watch() statement to detect future changes in {{controller.x}}; when a change occurs, Veil reads the new value, and then blindly fetches and updates the image as before. This basic approach is compatible with the template semantics of other popular JavaScript frameworks [6], [55], [67].

If dynamic Angular URLs can be drawn from an arbitrarily large set, Veil uses the "forward" message type from Section IV-I to bind the raw URL to a blinded one. If the URL is drawn from a finite set, the compiler can upload the associated objects to the blinding service, and then inject the page with a blinding map that translates resolved Angular URLs to the associated hash names. The blinding service mutates that table in the same way that it mutates the hash names passed to veilFetch().

VI. Implementation

Our Veil prototype consists of a compiler, a blinding server, a GUI proxy, a bootstrap page, and a client-side JavaScript library that implements veilFetch() and other parts of the Veil runtime.

We implemented the compiler and the blinding server in Python. The compiler uses BeautifulSoup [57] to parse and mutate HTML; the compiler also uses the Esprima/Escodegen tool chain [25], [64] to transform JavaScript code into ASTs, and to transform the mutated ASTs back into JavaScript. To implement cryptography, we use the PyCrypto library [33] in the blinding server, and the native Chrome WebCrypto API [70] in the Veil JavaScript library. We use OpenCV [48] to perform image mutation on the blinding server.

To implement DOM hiding, we used Chrome running in headless mode as the browser used by the content provider's GUI proxy. The GUI proxy was written in Python, and used Selenium [61] to take screenshots and generate synthetic GUI events within the headless browser.

VII. Evaluation

In this section, we evaluate Veil's raw performance, and its ability to safeguard user privacy. Using forensic tools and manual analysis, we find that blinded references and encrypted objects are sufficient to prevent information leakage through the browser cache and name-based interfaces like the DNS cache. We show that Veil's heap walking techniques are effective at preventing secrets from paging out unless system-wide memory pressure is very high. We also demonstrate that the performance of our Veil's prototype is acceptable, with page load slowdowns of 1.2x–3.25x.

Operation	Speed
Generate an AES key and encrypt it with RSA public key (2048 bit)	0.75 ms
Encrypt 64 character hash (blinded reference)	0.16 ms
Throughput for decryption using AES-CTR	520 MB/s
Throughput for verifying SHA256 hash of file	220 MB/s

Fig. 5. Overhead for client-side JavaScript cryptography using the WebCrypto API [70].

Operation	Speed
Decrypt AES key (2048 bit RSA)	3.1 ms
Decrypt 64 char hash (blinded reference)	0.04 ms
Throughput for encryption using AES-CTR	62 MB/s

Fig. 6. Overhead for server-side operations using PyCrypto [33].

All performance tests ran on a machine with a 2 GHz Intel Core i7 CPU with 8GB of RAM. Unless otherwise specified, those tests ran in the Chrome browser, and we ran each experiment 100 times and measured the average. We configured Veil to use 2048-bit RSA and 128-bit AES in CTR mode. The phrase "standard Veil mode" corresponds to non-DOM hiding mode.

A. Performance Microbenchmarks

Veil uses cryptography to implement blind references and protect the data that it places in client-side storage. Figure 5 depicts the costs for those operations. Before a user can load a Veil page, she must generate an AES key and encrypt it with the blinding service's RSA key. This one time cost is 0.75 ms. The remaining three rows in Figure 5 depict cryptographic overheads that Veil incurs during a page load.

- For `veilFetch()` to generate a blinded reference, it must encrypt a hash value with the user's AES key. This operation took 0.16 ms.
- When `veilFetch()` receives a response, it must decrypt that response with the AES key. That operation proceeds at 520 MB/s. For example, decrypting a 300 KB image would require 0.6 ms.
- `veilFetch()` also validates the hash of the downloaded object. This proceeds at 220 MB/s, requiring 1.3 ms for a 300 KB image.

Figure 6 depicts the cryptography overheads on the server-side of the protocol. End-to-end, fetching a 300 KB object incurs roughly 10 ms of cryptographic overhead.

B. Performance Macrobenchmarks: Standard Veil Mode

To measure the increase in page load time that Veil imposes, we ported six sites to the Veil framework. **Washington Post** is the biggest site that we ported, and contains large amounts of text, images, and JavaScript files. **Imgur** is a popular image-sharing site; compared to the other test sites, it has many images but less text. **Woot!** is an e-commerce site that has a large amount of text and images, but comparatively few scripts. **Piechopper** is a highly dynamic site that uses Angular (§V). Piechopper is script-and-text heavy, but has few images. **University** represents a university's website. This site is the smallest one that we tested, although it uses CSS with

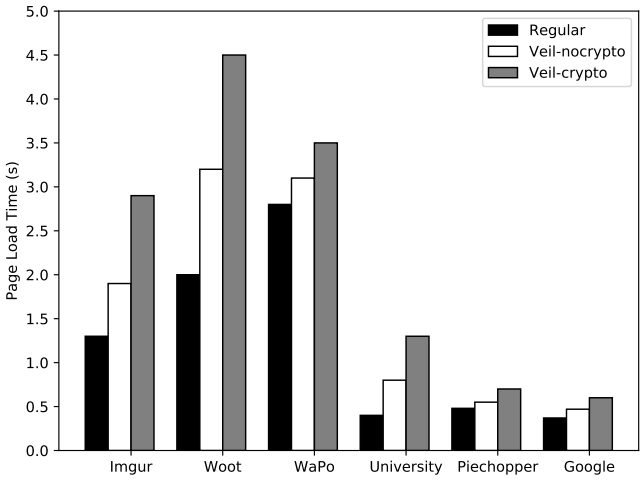

Fig. 7. Page load times for each website: regular; with Veil (but no cryptography); with Veil (and using cryptography).

raw URLs that Veil must blind (§V). **Google** represents the results page for the search term "javascript." Most of that page's JavaScript and CSS objects are inlined into the HTML, meaning that they do not require network fetches.

To port a preexisting site to Veil, we had the compiler download the top-level HTML file and extract the URLs which referenced external objects like images. The compiler downloaded those objects from the relevant servers. After calculating hashes for those objects (and converting raw URLs into blinded ones), the compiler uploaded the objects to the blinding server. Since preexisting sites were not designed with Veil in mind, they occasionally fetched content dynamically, e.g., via unblinded `<img>` tags generated by JavaScript at runtime. For sites like this, we observed which objects were dynamically fetched, and then manually handed them to the compiler for processing; we also manually rewrote the object fetch code to refer to the compiler-generated object names. Native Veil pages would invoke the Veil runtime library to dynamically fetch such content, avoiding the need for manual rewriting.

Page load time: Figure 7 depicts the load times for three versions of each site: the regular version of the site, a Veil port that does not perform cryptography, and a Veil port with cryptography enabled. The regular versions of a page were loaded from a localhost webserver, whereas the Veil pages were loaded from a localhost blinding server. This setup isolated the overhead of cryptography and content mutation.

As shown in Figure 7, page loads using Veil with no cryptography were 1.25x–2x slower. This is mostly due to extra computational overhead on the client. For example,

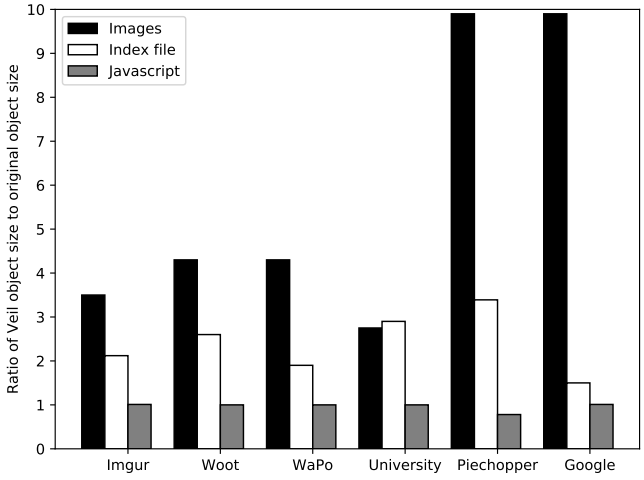

Fig. 8. Size increases for Veil's mutated objects.

parsing overheads increased because, as we quantify below, mutated objects were larger than the baseline objects; for images, the browser also had to Base64-decode the bitmaps before displaying them. Veil with cryptography added another slowdown factor of 1.1x–1.63x, with higher penalties for pages with many objects (regardless of their type). The end-to-end slowdown for the full Veil system was 1.25x–3.25x. Note that these slowdowns were for browsers with cold caches; Veil's overhead would decrease with caching, since server-side cryptography could be avoided. Also note that the University site was a challenging case for Veil, because the site was small in absolute size, but has many small images. Thus, Veil's per-blinded-reference cryptographic overheads (see Figures 5 and 6) were paid more frequently. A Veil-optimized version of the site would use image spriting [21] to combine multiple small images into a single, larger one.

Object growth: Figure 8 shows how object sizes grew after post-processing by Veil. Images experienced two sources of size expansion: mutation and Base64 encoding. Base64 encoding resulted in a 1.33x size increase. Our Veil prototype implements mutation via the addition of Gaussian noise, with the resulting size increases dependent on the image format. PNG is lossless, so the addition of noise generated a 10x size increase. In contrast, JPG is a lossy format, so noise injection resulted in less than a 2x size increase. The Piechopper and Google pages contained many PNGs, and thus suffered from worse image expansion than the other test pages.

As shown in Figure 8, mutated JavaScript files typically remained the same size, or became somewhat smaller—mutation adds source code, but Veil passes the mutated code through a minifier which removes extraneous whitespace and rewrites variable names to be shorter. HTML suffered from larger size increases, because mutation tricks like random HTML entity encoding strictly increase the number of characters in the HTML.

Server-side scalability: Figure 9 shows the HTTP-level request throughput of a Veil blinding server, compared to the baseline performance of a blinding server that performed none of Veil's added functionality (and thus acted as a normal web server). HTTP requests were generated using `ab`, the Apache benchmarking tool [5].

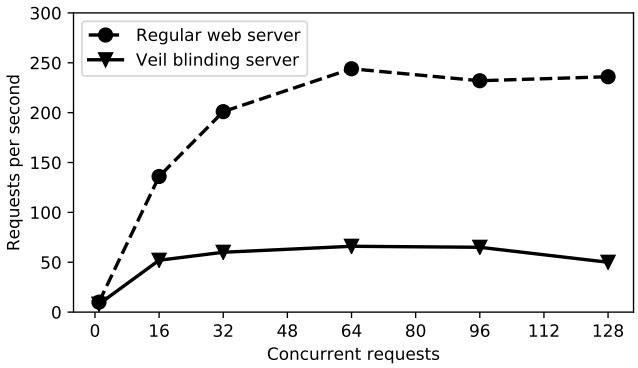

Fig. 9. Scalability comparison between a blinding server and a regular web server.

As shown in Figure 9, Veil reduces web server throughput by roughly 70% due to the additional cryptographic operations that Veil must perform. Remember that when Veil operates in regular (i.e., non-DOM hiding mode), Veil blinding servers mutate content in the background, out of the critical path for an HTTP response; thus, the slowdowns in Figure 9 are solely caused by synchronous cryptographic operations.

C. Preventing Information Leakage

Name-based Interfaces: To determine how well Veil protects user privacy, we created a baseline VM image which ran Lubuntu 13.10 and had two different browsers (Firefox and Chrome). In the baseline image, the browsers were installed, but they had not been used to visit any web pages. We then ran a series of experiments in which we loaded the baseline image, opened a browser, and then visited a single site. We took a snapshot of the browser's memory image using `gcore`, and we also examined disk state such as the browser cache and the log entries for DNS resolution requests. We did this for the regular and Veil-enabled versions of each page described in Section VII-B.

In all tests, the Veil pages were configured to store data in the browser cache, and in all tests, the cache only contained encrypted data at the end of the private session. Greps through the memory snapshots and DNS records did not reveal cleartext URLs or hostnames. Unsurprisingly, the regular versions of the web pages left unencrypted data in the browser cache, and various cleartext URLs in name-based data stores. To cross-validate these results, we repeated these experiments on Windows, and used the Mandiant Redline forensics tool [36] to search for post-session artifacts in persistent storage. Redline confirmed that the only cleartext URL in the browser history was the URL for the Veil bootstrap page, and that all other URLs were blinded.

Protecting RAM Artifacts: To determine whether heap walking can prevent secrets from paging out, we wrote a C program which gradually increases its memory pressure. The program allocates memory without deallocating any, and periodically, it reads the first byte in every allocated page to ensure that the OS considers the page to be hot. We ran the program inside of a QEMU VM with 1 GB of swap space and 1 GB of RAM. We also ran a browser inside of the VM. The browser had 20 open tabs. Each tab had a Uint8Array representing a tab-specific AES key, and a tab-specific set of strings in its HTML.

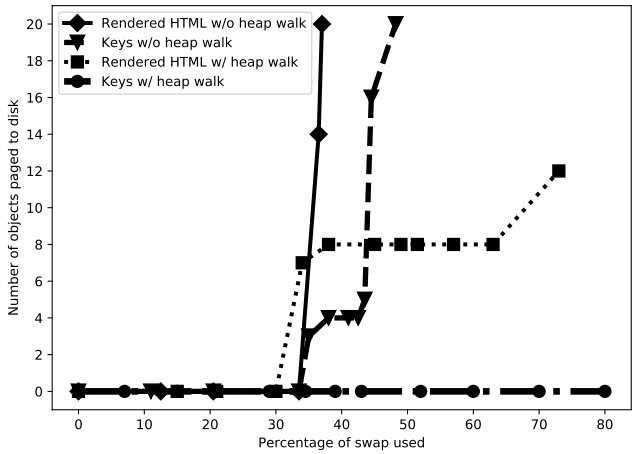

Fig. 10. The effectiveness of heap walking to prevent secrets from paging out.

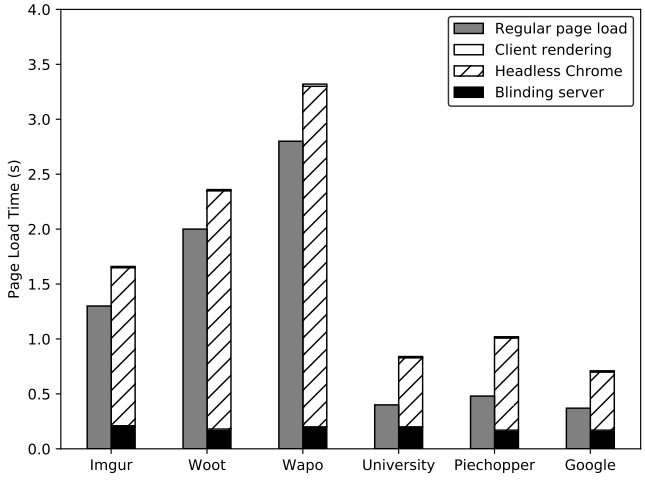

Fig. 11. DOM hiding's impact on page load times.

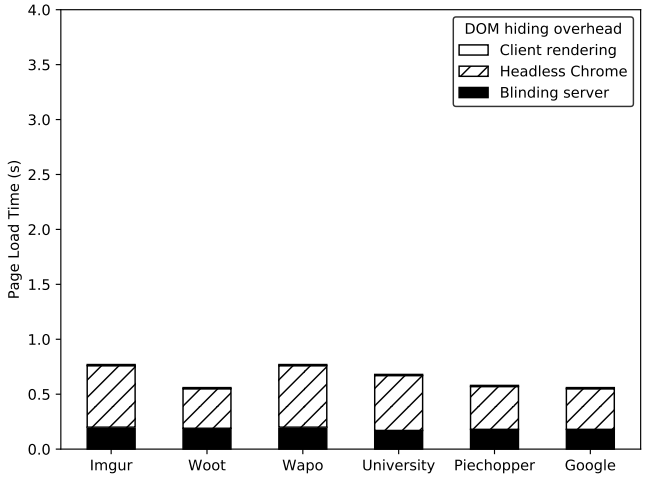

Fig. 12. The time that a DOM-hiding page needs to respond to a mouse click event.

The control experiments did not do heap walking. The test experiments used Veil's heap walking code to touch the AES key and the renderer state.

The VM used the `pwritev` system call to write memory pages to the swap file. To determine whether secrets paged out as memory pressure increased, we used strace to log the `pwritev` calls. Since each tab contained a set of unique byte patterns, we could grep through our `pwritev` logs to determine whether secret RAM artifacts hit the swap file. We ran experiments for increasing levels of memory pressure until the VM became unresponsive, at roughly 75% in-use swap space.

Figure 10 shows the results. The x-axis varies the memory pressure, and the y-axis depicts the number of tabs which suffered data leakage, as determined by greps through the `pwritev` log. Heap walking successfully kept all of the secret keys from paging out, up to the maximum 75% of in-use swap space. Without heap walking, keys begin to page out at 35% swap utilization; by 50%, all keys had swapped out. Note that the data points do not perfectly align on the x-axis due to nondeterminism in when the VM decides to swap data out.

Heap walking was less effective for renderer memory pages. Those pages swapped out earlier and immediately in the control case, around 35% swap utilization. With Veil, renderer state also began to leak at 35% utilization, but Veil still managed to safeguard 12 out of 20 tabs up to 63% swap utilization.

D. DOM Hiding

When Veil runs in DOM hiding mode, the client-side page contains no site-specific, greppable content. Thus, Veil does not need to perform heap walking (although Veil does use blinding servers to eliminate information leakage through name-based system interfaces). We loaded our test pages in DOM hiding mode, and confirmed the absence of site-specific content by grepping through VM images as we did in Section VII-C.

Figure 11 evaluates the impact of DOM hiding on a page's initial load. The client, the blinding server, and the content server ran on the same machine, to focus on computational overheads. Figure 11 demonstrates that DOM hiding imposed moderate overheads, with page load times increasing by 1.2x–2.1x. When Veil runs in DOM hiding mode, image mutation has to be performed synchronously, for each screenshot that is returned to a client; screenshotting requires 150ms–180ms, whereas image mutation requires 170ms–200ms.

Figure 12 shows the time that a DOM-hiding page needed to respond to a mouse click. Responding to such a GUI event required the browser to forward the event to the content provider, and then receive and display the new screenshot. Once again, the bulk of the end-to-end time was consumed by the screenshot capture and the image mutation at the content provider.

Privacy-sensitive users and web sites are often willing to trade some performance for better security. For example, fetching an HTTP object through Tor results in HTTP-level RTTs of more than a second [68]. Thus, we believe that the performance of Veil's DOM hiding mode is adequate for many sites. However, Veil's performance may be too

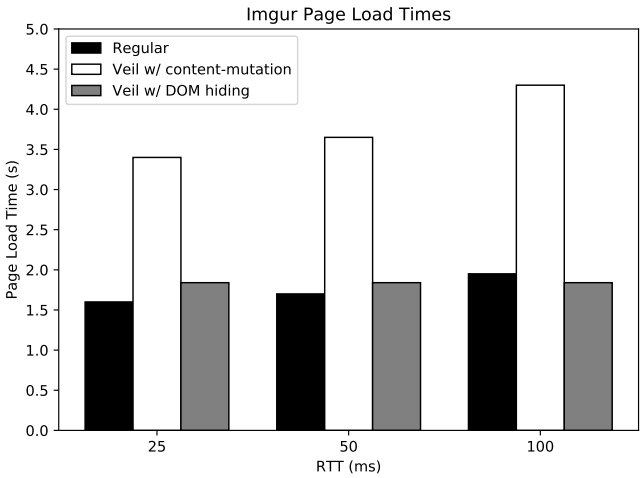

Fig. 13. The impact of emulated network latency on page load times. In all cases, the download bandwidth cap was 30 Mbps, and the upload bandwidth cap was 10 Mbps, emulating a broadband connection. Bandwidth was not varied because page load times are largely governed by network latency, not bandwidth [20].

slow for sites that are highly interactive, or require content servers to frequently and proactively push new images (e.g., due to animations in a page). Our next version of the Veil GUI proxy will grab screenshots directly from the content server's framebuffer [10] instead of via the comparatively-slow rendering API that browsers expose [44]; this implementation change will greatly reduce screenshotting overhead.

E. Network Latency

Figure 13 uses Chrome's built-in network emulation framework [22] to compare load times for three versions of the Imgur page: a normal version; a Veil version which used content mutation, heap walking, and encrypted storage; and a Veil version which used DOM hiding. The DOM hiding variant was largely insensitive to increased network latency, since loading a page only required two HTTP-level round trips (one to fetch the bootstrap page, and another to fetch the initial screenshot). The other variant of Veil was more sensitive to network latency. The reason is that, in this version of the page, the bootstrap code had to fetch multiple objects, all of which were served from the same blinding server origin (`https://veil.io`). Browsers cap the number of simultaneous connections that a client can make to a single origin, so the Veil page could not leverage domain sharding [30] to circumvent the cap. This limitation is not fundamental to Veil's design, since content providers can shard across multiple blinding server domains (e.g., `https://a.veil.io` and `https://b.veil.io`). However (and importantly), if a content provider wishes to use sharding, the provider must be careful to avoid bias in the mapping of objects to domains—otherwise, per-site fingerprints may arise in a page's access patterns to various domains. Thus, for some content providers, domain sharding may not be worth the potential loss in security.

Domain sharding is also relevant to Content Security Policies (CSPs) [43]. A CSP allows a page to restrict the origins which can provide specific types of content. For example, a CSP might state that a page can only load JavaScript from `https://a.com`, and CSS from `https://b.com`. A CSP is expressed as a server-provided HTTP response header; the CSP is enforced by the browser. CSPs are useful for preventing cross-site scripting attacks, but require a page to be able to explicitly shard content across domains. As discussed in the last paragraph, Veil can enable sharding at the cost of reduced security.

VIII. Related Work

To minimize information leakage via RAM artifacts, applications can use best practices like pinning sensitive memory pages, and avoiding excessive copying of secret data [23]. Operating systems and language runtimes can also scrub deallocated memory as quickly as possible [11]. Web browsers do not expose low-level OS interfaces to JavaScript code, so privacy-preserving sites cannot directly access raw memory for the purposes of secure deallocation or pinning. Determining the best way to expose raw memory to JavaScript is an open research problem, given the baroque nature of the same-origin policy, and the fact that the browser itself may contend with JavaScript code for exclusive access to a memory page (e.g., to implement garbage collection or tab discarding [50]).

An OS can protect RAM artifacts by encrypting the swap space or the entire file system [8], [56], [76]. Veil's content mutation and DOM hiding allow Veil to protect RAM artifacts even when a browser does not run atop an encrypted storage layer. Content mutation obviously does not provide a cryptographically strong defense, but DOM hiding allows a Veil site to avoid sending any site-specific, greppable content to a client browser.

CleanOS [66] is a smartphone OS that protects sensitive data when mobile devices are lost or stolen. CleanOS defines sensitive data objects (SDOs) as Java objects and files that contain private user data. CleanOS observes which SDOs are not actively being used by an application, and encrypts them; the key is then sent to the cloud, deleted from the smartphone, and only retrieved when the SDOs become active again. SDOs could potentially be used as a building block for private browsing. However, SDOs are insufficient for implementing blinded references unless the SDO abstraction is spread beyond the managed runtime to the entire OS.

Lacuna [17] implements private sessions by running applications inside of VMs. Those VMs execute atop the Lacuna hypervisor and a modified Linux host kernel. The hypervisor and the host kernel collectively implement "ephemeral" IO channels. These encrypted channels allow VMs to communicate with hardware or small pieces of trusted code, but only the endpoints can access raw data—user-mode host processes and the majority of the host kernel can only see encrypted data. Lacuna also encrypts swap memory. Upon VM termination, Lacuna zeros the VM's RAM space and discards the ephemeral session keys. PrivExec [47] is similar to Lacuna, but is implemented as an OS service instead of a hypervisor. Lacuna and PrivExec provide stronger forensic deniability than Veil. However, these systems force layperson end-users to install and configure a special runtime; furthermore, private applications cannot persist data across sessions because keys are ephemeral.

UCognito [74] exposes a sandboxed file system to a private browsing session. The sandboxed file system resides atop the

normal one, absorbing writes made during private browsing. When the browsing session terminates, UCognito discards the writes. Like PrivExec and Lacuna, UCognito requires a modified client-side software stack. UCognito also does not protect against information leakage via the non-sandboxed parts of the host OS. For example, unmodified RAM artifacts may page to the native swap file; DNS requests are exposed to the host's name resolution subsystem.

Collaborative browsing frameworks [34], [52] allow multiple users with different browsers to simultaneously interact with a shared view of a web page. Like these frameworks, Veil's DOM hiding mode has to synchronize the GUI inputs and rendering activity that belong to a canonical version of a page. However, Veil only needs to support one remote viewer. More importantly, Veil's DOM hiding mode only exposes the client browser to generic JavaScript event handlers, as well as a bitmap display; in contrast, prior collaborative browsing frameworks replicate a site-specific, canonical DOM tree on each client browser. Prior frameworks also do not use blinding servers to hide information from client-side, name-centric interfaces like the DNS cache.

IX. Conclusions

Veil is the first web framework that allows developers to implement private-session semantics for their pages. Using the Veil compiler, developers rewrite pages so that all page content is hosted by blinding servers. The blinding servers provide name indirection, preventing sensitive information from leaking to client-side, name-based system interfaces. The blinding servers mutate content, making object fingerprinting more difficult; rewritten pages also automatically encrypt client-side persistent storage, and actively walk the heap to reduce the likelihood that in-memory RAM artifacts will swap to disk in cleartext form. In the extreme, Veil transforms a page into a thin client which does not include any page-specific, greppable RAM artifacts. Veil automates much of the effort that is needed to port a page to Veil, making it easier for web developers to improve the privacy protections of their applications.

References

[1] A. Hidayat, "PhantomJS: Full web stack—No browser required," 2017, http://phantomjs.org/.

[2] G. Aggarwal, E. Burzstein, C. Jackson, and D. Boneh, "An Analysis of Private Browsing Modes in Modern Browsers," In *Proceedings of USENIX Security*, Washington, DC, August 2010.

[3] D. Akhawe, P. Saxena, and D. Song, "Privilege Separation in HTML5 Applications," In *Proceedings of USENIX Security*, Bellevue, WA, August 2012.

[4] Angular.js, "Angular: A Superheroic JavaScript MVW Framework," https://angularjs.org/, 2014.

[5] Apache Software Foundation, "ab: Apache HTTP Server Benchmarking Tool," https://httpd.apache.org/docs/2.4/programs/ab.html, 2017.

[6] Backbone, "Backbone.js," http://backbonejs.org/, 2017.

[7] B. Biggio, G. Fumera, I. Pillai, and F. Roli, "A Survey and Experimental Evaluation of Image Spam Filtering Techniques," *Pattern Recognition Letters*, vol. 32, no. 10, July 2011.

[8] M. Blaze, "A Cryptographic File System for Unix," In *Proceedings of CCS*, Fairfax, VA, November 1993.

[9] Blue Spire Inc., "Aurelia," 2017, http://aurelia.io/.

[10] A. Buell, "Linux Framebuffer HOWTO," August 5, 2010, http://www.tldp.org/HOWTO/html_single/Framebuffer-HOWTO/#AEN134.

[11] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation," In *Proceedings of USENIX Security*, Baltimore, MD, August 2005.

[12] CoffeeScript, "CoffeeScript: A Little Language that Compiles to JavaScript," October 26, 2017, http://coffeescript.org/.

[13] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)," RFC 4627 (Draft Standard), July 2006.

[14] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Fast and Precise In-Browser JavaScript Malware Detection," In *Proceedings of USENIX Security*, San Francisco, CA, August 2011.

[15] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The Second-Generation Onion Router," In *Proceedings of USENIX Security*, San Diego, CA, August 2004.

[16] DuckDuckGo, "Take back your privacy! Switch to the search engine that doesn't track you." 2017, https://duckduckgo.com/about.

[17] A. Dunn, M. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel, "Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels," In *Proceedings of OSDI*, Vancouver, BC, Canada, November 2010.

[18] Enigma, "Secure Data and Protect Privacy Without Compromising Functionality," 2015, https://www.media.mit.edu/projects/enigma.

[19] E. Felten and M. Schneider, "Timing Attacks on Web Privacy," In *Proceedings of CCS*, Athens, Greece, November 2000.

[20] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing Web Latency: The Virtue of Gentle Aggression," In *Proceedings of SIGCOMM*, August 2013.

[21] Google, "PageSpeed Module Documentation: Sprite Images," 2014, https://developers.google.com/speed/pagespeed/module/filter-image-sprite.

[22] ——, "Network Analysis Reference," 2017, https://developers.google.com/web/tools/chrome-devtools/network-performance/reference.

[23] K. Harrison and S. Xu, "Protecting Cryptographic Keys From Memory Disclosure Attacks," In *Proceedings of DSN*, Edinburgh, UK, June 2007.

[24] M. Heiderich, E. Nava, G. Heyes, and D. Lindsay, *Web Application Obfuscation*. Syngress, 2010.

[25] A. Hidayat, "Esprima: ECMAScript Parsing Infrastructure for Multi-purpose Analysis," 2017, https://github.com/ariya/esprima.

[26] L. Ingram, "TreeHouse," December 2012, https://github.com/lawnsea/TreeHouse.

[27] L. Ingram and M. Walfish, "TreeHouse: JavaScript Sandboxes to Help Web Developers Help Themselves," In *Proceedings of USENIX ATC*, Boston, MA, June 2012.

[28] D. Isacsson, "Microsoft Edge's Incognito Mode Isn't So Incognito," February 1, 2016, Digital Trends. https://www.digitaltrends.com/web/microsoft-edge-security-flaws-in-incognito/.

[29] A. Janc and L. Olejnik, "Feasibility and Real-World Implications of Web Browser History Detection," In *Proceedings of the Web 2.0 Security and Privacy Workshop*, Oakland, CA, May 2010.

[30] KeyCDN, "Domain Sharding," August 19, 2016, https://www.keycdn.com/support/domain-sharding/.

[31] S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities," In *Proceedings of IEEE Symposium on Security and Privacy*, San Jose, CA, May 2014.

[32] B. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi, "Verifying Web Browser Extensions' Compliance with Private-Browsing Mode," In *Proceedings of ESORICS*, Egham, United Kingdom, September 2013.

[33] D. Litzenberger, "PyCrypto: The Python Cryptography Toolkit," June 23, 2014, https://github.com/dlitz/pycrypto.

[34] D. Lowet and D. Goergen, "Co-Browsing Dynamic Web Pages," In *Proceedings of WWW*, Madrid, Spain, April 2009.

[35] Magnet Forensics, "How Does Chrome's "Incognito" Mode Affect Digital Forensics?" http://www.magnetforensics.com/how-does-chromes-incognito-mode-affect-digital-forensics/, August 6, 2013.

[36] Mandiant, "Mandiant Redline Users Guide," 2012, https://dl.mandiant.com/EE/library/Redline1.7_UserGuide.pdf.

[37] L. Masinter, "The "data" URL scheme," Network Working Group, RFC 2397, Aug. 1998.

[38] Meteor Development Group, "Meteor: The Fastest Way to Build JavaScript Apps," 2017, https://www.meteor.com/.

[39] J. Mickens, "Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads," In *Proceedings of USENIX WebApps*, Boston, MA, June 2010.

[40] J. Mickens, J. Elson, and J. Howell, "Mugshot: Deterministic Capture and Replay for JavaScript Applications," In *Proceedings of NSDI*, April 2010.

[41] J. Mickens and M. Finifter, "Jigsaw: Efficient, Low-effort Mashup Isolation," In *Proceedings of USENIX WebApps*, Boston, MA, June 2012.

[42] Monkeys, "MonkeyTestJS: Automated Functional Testing for Front-end Web Development," 2017, http://monkeytestjs.io/.

[43] Mozilla, "Content Security Policy (CSP)," November 20, 2017, mozilla Developer Network. https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP.

[44] ——, "Documentation: tabs.captureVisibleTab()," 2017, https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/tabs/captureVisibleTab.

[45] M. Nielsen, "Clickmonkey," 2017, https://www.npmjs.com/package/clickmonkey.

[46] D. Ohana and N. Shashidhar, "Do Private and Portable Web Browsers Leave Incriminating Evidence?" In *Proceedings of the International Workshop on Cyber Crime*, San Francisco, CA, May 2013.

[47] K. Onarlioglu, C. Mulliner, W. Robertson, and E. Kirda, "PrivExec: Private Execution as an Operating System Service," In *Proceedings of IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2013.

[48] OpenCV, "Open Source Computer Vision Library," 2017, https://opencv.org/.

[49] E. Orion, "Tor popularity leaps after snooping revelations," August 30, 2013, The Inquirer. http://www.theinquirer.net/inquirer/news/2291758/tor-popularity-leaps-after-snooping-revelations.

[50] A. Osmani, "Tab Discarding in Chrome: A Memory-Saving Experiment," September 2015, Google Developer Blog. https://developers.google.com/web/updates/2015/09/tab-discarding.

[51] D. Parys, "How to Safeguard Your Site with HTML5 Sandbox," Microsoft Developer Network. http://msdn.microsoft.com/en-us/hh563496.aspx, 2015.

[52] S. Pongelli, "Jigsaw: An Infrastructure for Cross-device Mashups," ETH Zurich, Master's thesis, November 6, 2013.

[53] Priv.io, "Priv.io homepage," 2015, https://priv.io/.

[54] Priv.ly, "Change the Way Your Browser Works: Share Priv(ate).ly," 2017, https://priv.ly/.

[55] Progress Software, "Kendo UI for jQuery," https://docs.telerik.com/kendo-ui/, 2017.

[56] N. Provos, "Encrypting Virtual Memory," In *Proceedings of USENIX Security*, Denver, CO, August 2000.

[57] L. Richardson, "Beautiful Soup: A Python Parser for HTML," 2017, http://www.crummy.com/software/BeautifulSoup/.

[58] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," In *Proceedings of IFIP/ACM Middleware*, Heidelberg, Germany, November 2001.

[59] J. Ruderman, "Same-origin Policy," Mozilla Developer Network. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, August 1, 2014.

[60] Sahi, "Sahi Pro: The Tester's Automation Tool," 2017, http://sahipro.com/.

[61] SeleniumHQ, "Selenium: Browser Automation," 2017, http://www.seleniumhq.org/.

[62] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," In *Proceedings of SIGCOMM*, San Diego, CA, August 2001.

[63] Q. Sun, D. Simon, Y. Wang, W. Russell, V. Padmanabhan, and L. Qiu, "Statistical Identification of Encrypted Web Browsing Traffic," In *Proceedings of IEEE Symposium on Security and Privacy*, Berkeley, CA, May 2002.

[64] Y. Suzuki, "Escodegen: ECMAScript Code Generator," 2017, https://github.com/Constellation/escodegen.

[65] P. Szor, "Advanced Code Evolution Techniques and Computer Virus Generator Kits," InformIT. http://www.informit.com/articles/article.aspx?p=366890&seqNum=6, March 25, 2006.

[66] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "CleanOS: Limiting Mobile Data Exposure with Idle Eviction," In *Proceedings of OSDI*, Hollywood, CA, October 2012.

[67] Tilde Inc., "Ember: A Framework for Creating Ambitious Web Applications," https://emberjs.com/, 2017.

[68] Tor Project, "Tor Metrics: Performance," https://metrics.torproject.org/torperf.html, November 28, 2017.

[69] W3C Web Apps Working Group, "Web Storage: W3C Working Draft," http://www.w3.org/TR/webstorage/, April 19, 2016.

[70] ——, "Web Cryptography: W3C Working Draft," January 26, 2017, http://www.w3.org/TR/WebCryptoAPI/.

[71] Z. Weinberg, E. Chen, P. Jayaraman, and C. Jackson, "I Still Know What You Visited Last Summer," In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2011.

[72] W. Wong and M. Stamp, "Hunting for metamorphic engines," *Journal in Computer Virology and Hacking*, vol. 2, no. 3, December 2006.

[73] C. Wright, S. Coull, and F. Monrose, "Traffic Morphing: An Efficient Defense against Statistical Traffic Analysis," In *Proceedings of NDSS*, San Diego, CA, February 2009.

[74] M. Xu, Y.Jang, X. Xing, T. Kim, and W. Lee, "UCognito: Private Browsing without Tears," In *Proceedings of CCS*, Denver, CO, October 2015.

[75] N. Yorker, "The New Yorker SecureDrop," 2017, https://projects.newyorker.com/securedrop/.

[76] E. Zadok, I. Badulescu, and A. Shender, "Cryptfs: A Stackable Vnode Level Encryption File System," Technical Report CUCS-021-98, University of California at Los Angeles, 1998.