Preventing Data Leakage during Web Service Accesses

by

Frank Yi-Fei Wang

B.S., Stanford University (2012)
S.M., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2018

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2018

Certified by .....
Nickolai Zeldovich
Professor
Thesis Supervisor

Certified by .....
James Mickens
Associate Professor
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Preventing Data Leakage during Web Service Accesses

by

Frank Yi-Fei Wang

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2018, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Web services, like Google, Facebook, and Dropbox, are a regular part of users' lives. As a form of payment, these services collect, store, and analyze user data. Even accessing these web services can leak a substantial amount of data.

This dissertation presents two practical, secure systems, Veil and Splinter, that prevents some of this data leakage. Veil minimizes client-side information leakage from the browser by allowing web page developers to enforce stronger private browsing semantics without browser support. Splinter protects sensitive information present in a users' query on cleartext datasets in a practical manner by leveraging a recent cryptographic primitive, Function Secret Sharing (FSS).

Thesis Supervisor: Nickolai Zeldovich
Title: Professor

Thesis Supervisor: James Mickens
Title: Associate Professor

Acknowledgments

More acknowledgments here.

⋆ ⋆ ⋆

The dissertation incorporates and extends work published in the following papers:

Frank Wang, James Mickens, and Nickolai Zeldovich. Veil: Private Browsing Semantics Without Browser-side Assistance. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2018.

Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical Private Queries on Public Data. In *Proceedings of Networked Systems Design and Implementation (NSDI)*, 2017.

Contents

Figures and tables

ONE

Introduction

This dissertation presents two practical, secure systems, Veil and Splinter, which protect against certain data leakage that happens when a user accesses a web service. The rest of this chapter will motivate this problem and briefly describe Veil and Splinter.

1.1 Motivation

Consumers are increasing their usage of web services. Whenever users interact with these applications, the web service collects user data both directly and indirectly. This data can contain sensitive information about a user, such as their behavior, personal facts, and location. These data leaks are happening with even greater frequency and as web systems have become more complex, the number of channels where data can leak grows even more. Unfortunately, many times, these data leakages happen because web services lack practical, secure mechanisms to protect user data.

Many systems [30**?** , 31] have been built to prevent data leakage from the server, but the focus of this dissertation is on systems that minimize data leakage on the client (Veil), specifically the browser, and that protect user in queries (Splinter). In the section below, we provide an overview of Veil and Splinter.

1.2 Our Systems

1.2.1 Veil: Private Browsing Semantics without Browser-side Assistance

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.

1.2.2 Splinter: Practical, Private Web Application Queries

Many online services let users query datasets such as maps, flight prices, patents, and medical information. The datasets themselves do not contain sensitive information, but unfortunately, users' queries on these datasets reveal highly sensitive information that can compromise users' privacy. This paper presents Splinter, a system that protects users' queries and scales to realistic applications. A user splits her query into multiple parts and sends each part to a different provider that holds a copy of the data. As long as any one of the providers is honest and does not collude with the others, the providers cannot determine the query. Splinter uses and extends a new cryptographic primitive called Function Secret Sharing (FSS) that makes it up to an order of magnitude more efficient than prior systems based on Private Information Retrieval and garbled circuits. We develop protocols extending FSS to new types of queries, such as MAX and TOPK queries. We also provide an optimized implementation of FSS using AES-NI instructions and multicores. Splinter achieves end-to-end latencies below 1.6 seconds for realistic workloads including a Yelp clone, flight search, and map routing.

1.2.3 Dissertation Roadmap

The dissertation will be organization like the following: Chapter 2 and Chapter 3 will motivate and describe Veil and Splinter respectively. Chapter 4 will describe future work.

TWO

Veil: Private Browsing Semantics without Browser-side Assistance

2.1 Motivation

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, 16, 20, 26]. 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, 16].

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 [28]. 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.

- 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 VeilXMLHttpRequests), 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.

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 2-1: A comparison between Veil's two browsing modes, regular incognito browsing, and regular browsing that does not use incognito mode.

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 [13], 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 2-1 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 [11]. Veil can translate legacy pages to more secure versions automatically, or with minimal developer assistance (§**??**), 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.

2.2 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 [43] 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 [4], CoffeeScript [6], and Meteor [23] 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 [9], Veil's blinding servers can be run by volunteers, although content providers can also contribute physical machines or VMs to the blinding pool (§2.4.1).

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 [10, 12, 32, 33], and recent events have increased popular awareness of privacy issues [27]. 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.

2.3 Threat Model

As described in Section 2.2, 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 file, the browser cache, and files like Veil/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 [21] 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 [11]. 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 (§2.4.1).

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 (§2.4).

Heap walking (§2.4.2) allows Veil to prevent sensitive RAM artifacts from swapping to disk. Veil does not try to stop information leaks from GPU RAM [18], but GPU RAM is never swapped to persistent storage. Poorly-written or malicious browser extensions that leak sensitive page data [19] are also outside the scope of this paper.

2.4 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

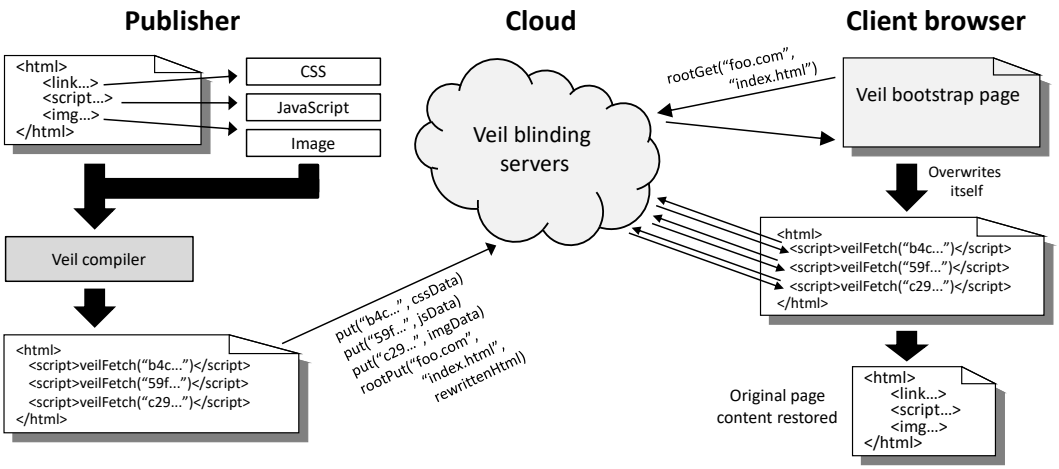

Figure 2-2: The Veil architecture (cryptographic operations omitted for clarity).

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.

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

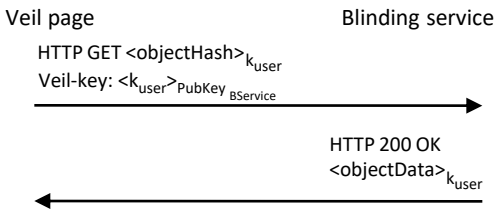

Figure 2-3: The `veilFetch()` protocol.

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 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 2.4.2). 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 2-3 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 [7]; Figure 2-4 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 [22] 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.

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

Figure 2-4: 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 [17, 40]), 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.

2.4.1 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 2.4.2, 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 (§2.4.2), 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 [34, 36], 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 [9]), 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 [34, 36]) could host anonymous object submissions for free.

2.4.2 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 [35] 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 [29] 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 [15, 25], the compiler could inject DOM interpositioning code that automatically injects `sandbox` attributes into dynamically-generated frames.

DOM storage [39] 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 (§2.4.2). 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 (§2.4.2).

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 (§2.4.2) 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.

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 [5]. 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.

Mutation Techniques

Veil's main protection mechanism for RAM artifacts is heap walking, and we show in Section **??** 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 (§2.4.1); 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 [41]. 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 [8], particularly if the attacker can only recover a partial set of RAM

artifacts for a page.[2]

However, 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 [14]. 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 [14]. 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 (§2.3), 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 [3].

[2] Some .NET viruses already leverage access to the runtime's reflection interface to dynamically emit code [38].

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 (§2.4.2), 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 (§2.4 and §**??**). 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.

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

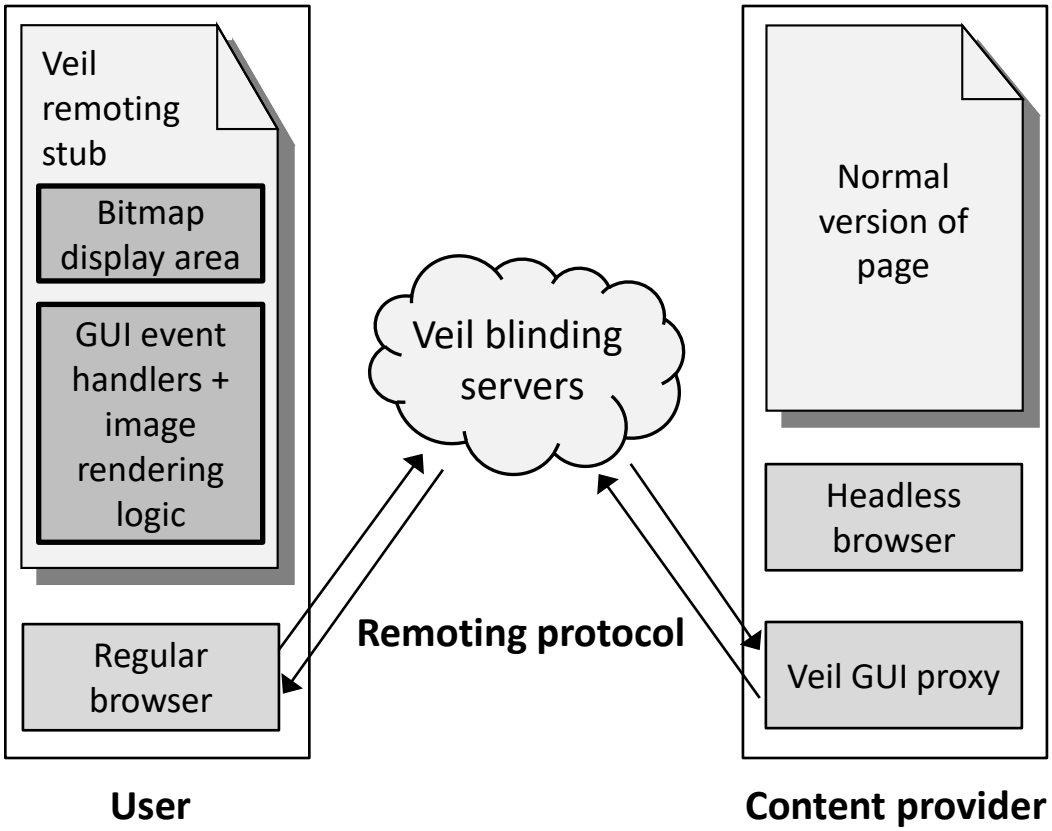

Figure 2-5: With DOM hiding, the client-side remoting stub sends GUI events to the content provider, and receives bitmaps representing new page states. The page's raw HTML, CSS, and JavaScript are never exposed to the client.

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 **??**.

Figure 2-5 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 2.4.2. 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 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.

[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.

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 (§2.4) or DOM storage (§2.4.2)—there will be no page-specific client-side state to store.

2.4.3 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. [37] 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 [24]; 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 [42]. 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.

THREE

Splinter: Practical, Private Web Application Queries

3.1 Motivation

FOUR

Conclusion and Future Work

Bibliography

[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] B. Biggio, G. Fumera, I. Pillai, and F. Roli. A Survey and Experimental Evaluation of Image Spam Filtering Techniques. *Pattern Recognition Letters*, 32(10), July 2011.

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

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

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

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

[8] 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.

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

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

[11] 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.

[12] Enigma. Arrival Data for Non-Stop Domestic Flights by Major Air Carriers for 2012. https://app.enigma.io/table/us.gov.dot.rita.trans-stats.on-time-performance.2012.

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

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

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

[16] 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/.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

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

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

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

[26] 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.

[27] 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.

[28] 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.

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

[30] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, Cascais, Portugal, October 2011.

[31] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. Building web applications on top of encrypted data using Mylar. In *Proceedings of NSDI*, Seattle, WA, April 2014.

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

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

[34] Rowstron, Antony and Druschel, Peter. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

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

[36] Stoica, Ion and Morris, Robert and Karger, David and Kaashoek, M Frans and Balakrishnan, Hari. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[37] Q. Sun, D. Simon, Y.M. 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.

[38] 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.

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

[40] 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.

[41] W. Wong and M. Stamp. Hunting for metamorphic engines. *Journal in Computer Virology and Hacking*, 2(3), December 2006.

[42] 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.

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