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 23, 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 23, 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 [93, 94, 135] 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

This chapter presents Veil, a system that allows web page developers to enforce stronger private browsing semantics without browser support.

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 [3, 48, 62, 78]. 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 [3, 48].

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 [88]. 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

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.

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 [30], 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 [27]. Veil can translate legacy pages to more secure versions automatically, or with minimal developer assistance (§2.5), 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 [133] 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 [12], CoffeeScript [19], and Meteor [67] typically require a compilation pass before content can go live. To handle dynamic content, blinding servers can also help with the compilation process. We discuss this in more detail in Section 2.4.7.

Participating Veil sites must also explicitly serve their content from Veil blinding servers. Like Tor servers [24], Veil's blinding servers can be run by volunteers, although content providers can also contribute physical machines or VMs to the blinding pool (§2.4.2).

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 [26, 28, 96, 97], and recent events have increased popular awareness of privacy issues [87]. 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 [64] 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 [27]. 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 [3]). 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 (§2.4.2).

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 (§2.4.1).

Heap walking (§2.4.5) allows Veil to prevent sensitive RAM artifacts from swapping to disk. Veil does not try to stop information leaks from GPU RAM [55], but GPU RAM is never swapped to persistent storage. Poorly-written or malicious browser extensions that leak sensitive page data [56] are also outside the scope of this paper.

2.4 Design

As shown in Figure 2-2, 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

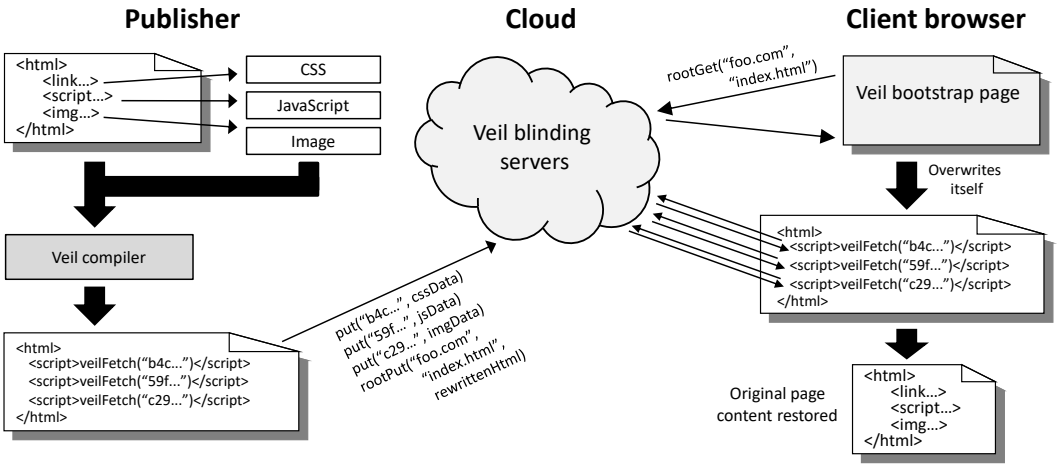

Figure 2-2: The Veil architecture (cryptographic operations omitted for clarity).

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.

2.4.1 The Veil Compiler and veilFetch()

The compiler processes the HTML in our example page (Figure 2-2), 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 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 2.4.4). 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 [22]; 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 [65] which directly embeds the Base64 image data. This causes the browser to load the image. `veilFetch()` uses similar techniques to reinflate other content types.

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-

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

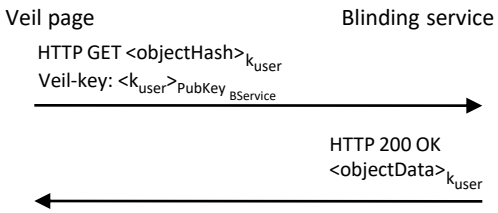

Figure 2-3: The `veilFetch()` protocol.

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

Figure 2-4: A serialized <`img`> tag.

based history attack [49, 128]), 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.

2.4.2 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 2.4.6, 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 (§2.4.8), 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 [107, 115], 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 [24]), 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 [107, 115]) could host anonymous object submissions for free.

2.4.3 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 [108] 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 [91] 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 [47, 69], the compiler could inject DOM interpositioning code that automatically injects `sandbox` attributes into dynamically-generated frames.

DOM storage [126] 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 (§2.4.7). 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 (§2.4.7).

2.4.4 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 (§2.4.5) 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.

2.4.5 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 [18]. 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.

2.4.6 Mutation Techniques

Veil's main protection mechanism for RAM artifacts is heap walking, and we show in Section 2.7.3 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 (§2.4.2); 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 [129]. 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 pro-

gram with the ability to emit new mutated code at runtime. Such "`eval()`-folding" is difficult to analyze [23], 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 [43]. 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 [43]. 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

[2] Some .NET viruses already leverage access to the runtime's reflection interface to dynamically emit code [118].

from the computer vision community [10].

2.4.7 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 (§2.4.3), 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 (§2.4.1 and §2.5).

Another option is to start with the same forwarding protocol as above, but instead of having the publisher compile and mutate the content, the publisher just sends the unmodified user-specific content back to the blinding servers. Then, the blinding servers compile and mutate the content before returning it to the user. In this scenario, the blinding servers do the compilation and mutation.

The question is which of the two options is better. More specifically, should the proxy (blinding servers in our case) or the end host (the publishers) be manipulating content? Ideally, the end host would manipulate the dynamic content as it might contain sensitive user information, which can be encrypted on host and thus hidden from the blinding servers. However, the downside is that all end hosts have to adopt Veil. Having the proxy perform the manipulation solves this problem, but this scenario would require the proxy to handle a heavier computation workload. Moreover, the user and end host will expose an additional party, i.e. the proxy, to her sensitive information since the proxy has to have access to the cleartext content. The networking community has examined these tradeoffs in a similar context, so we refer you to that specific research for a more in-depth discussion [2, 76]. Fortunately, there is not a significant performance difference between the two options. The performance overhead is minimal because only the dynamic portions of the site have to be compiled during the page load, and more detailed numbers are in Section 2.7.2.

2.4.8 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 2.7.4.

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 2.4.4. 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. 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 (§2.4.1) or DOM storage (§2.4.3)—there will be no page-specific client-side state to

[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.

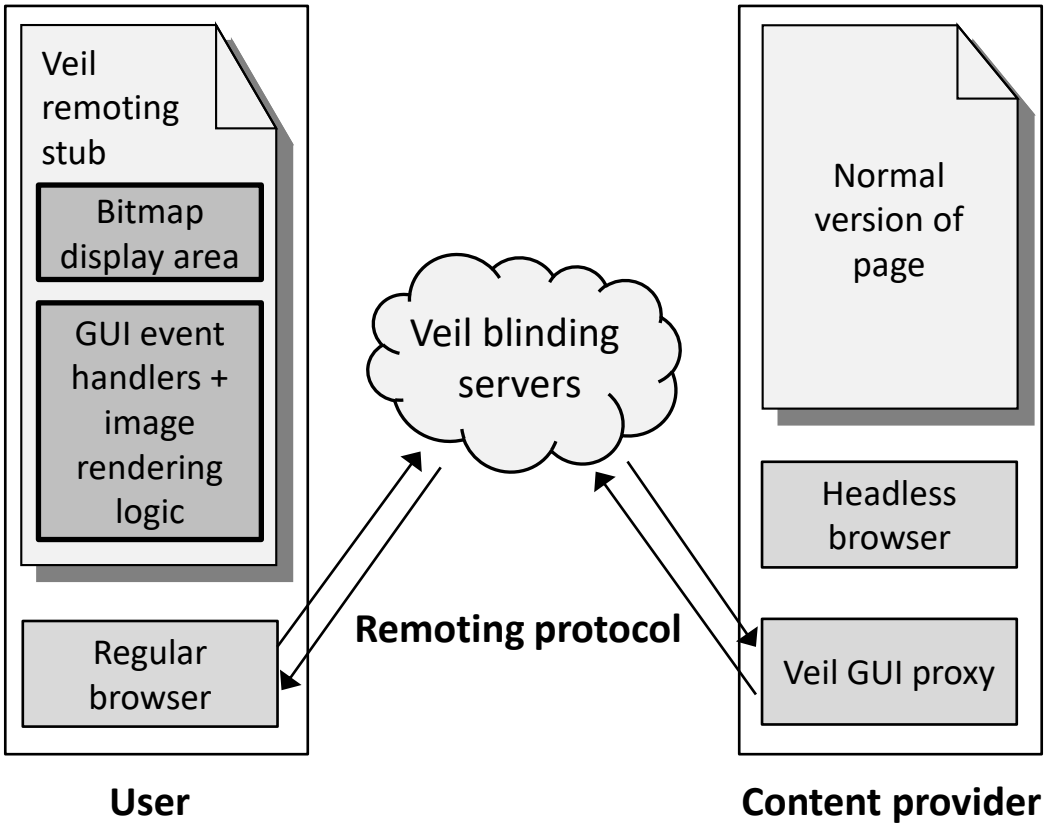

Figure 2-5: With DOM hiding, the client-side remoting stub sends GUI events to the content provider, and receives bitmaps representing new page states. The page's raw HTML, CSS, and JavaScript are never exposed to the client.

store.

2.4.9 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. [116] 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 [68]; 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 [130]. 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.

2.5 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 [69] 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 [71, 77, 109] 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 [4, 47, 70], 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 [46] 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 [5] 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 `<img src={{controller.x}}/>` 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, 98, 123].

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, 98, 123].

If dynamic Angular URLs can be drawn from an arbitrarily large set, Veil uses the "forward" message type from Section 2.4.9 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().

2.6 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 [103] to parse and mutate HTML; the compiler also uses the Esprima/Escodegen tool chain [44, 117] to transform JavaScript code into ASTs, and to transform the mutated ASTs back into JavaScript. To implement cryptography, we use the PyCrypto library [59] in the blinding server, and the native Chrome WebCrypto API [127] in the Veil JavaScript library. We use OpenCV [83] to perform image mutation on the blinding server.

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

Figure 2-6: Overhead for client-side JavaScript cryptography using the WebCrypto API [127].

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 [112] to take screenshots and generate synthetic GUI events within the headless browser.

2.7 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.

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.

2.7.1 Performance Microbenchmarks

Veil uses cryptography to implement blind references and protect the data that it places in client-side storage. Figure 2-6 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 2-6 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.

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

Figure 2-7: Overhead for server-side operations using PyCrypto [59].

- 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 2-7 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.

2.7.2 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 (§2.5). 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 raw URLs that Veil must blind (§2.5). **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,

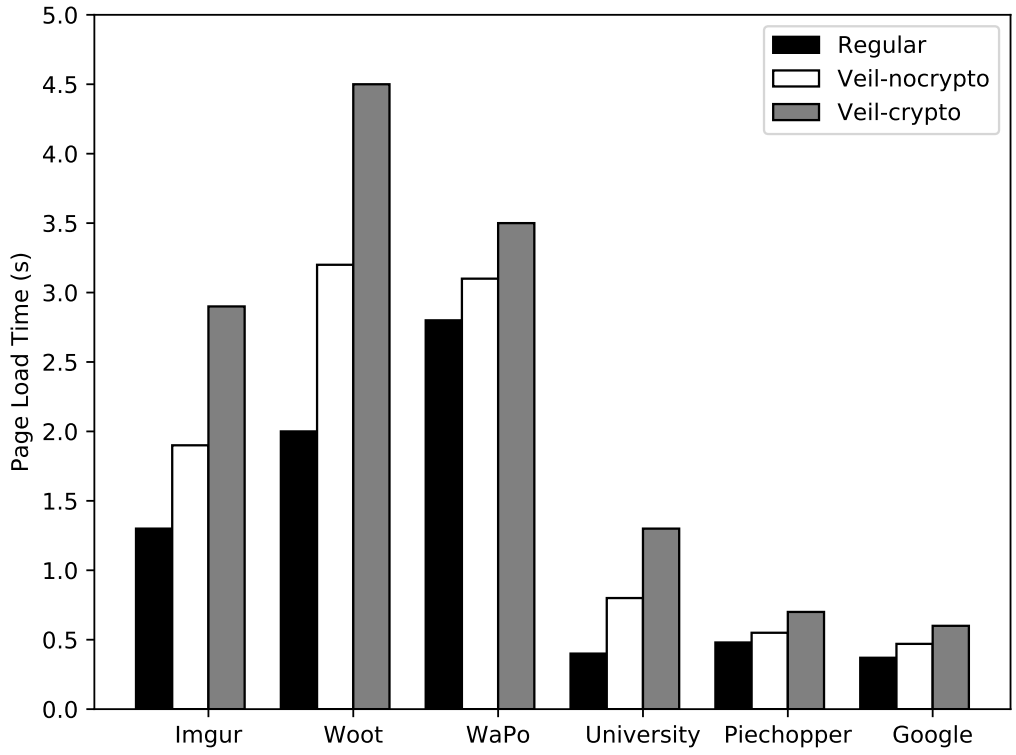

Figure 2-8: Page load times for each website: regular; with Veil (but no cryptography); with Veil (and using cryptography).

avoiding the need for manual rewriting.

Page load time: Figure 2-8 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 2-8, 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, 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

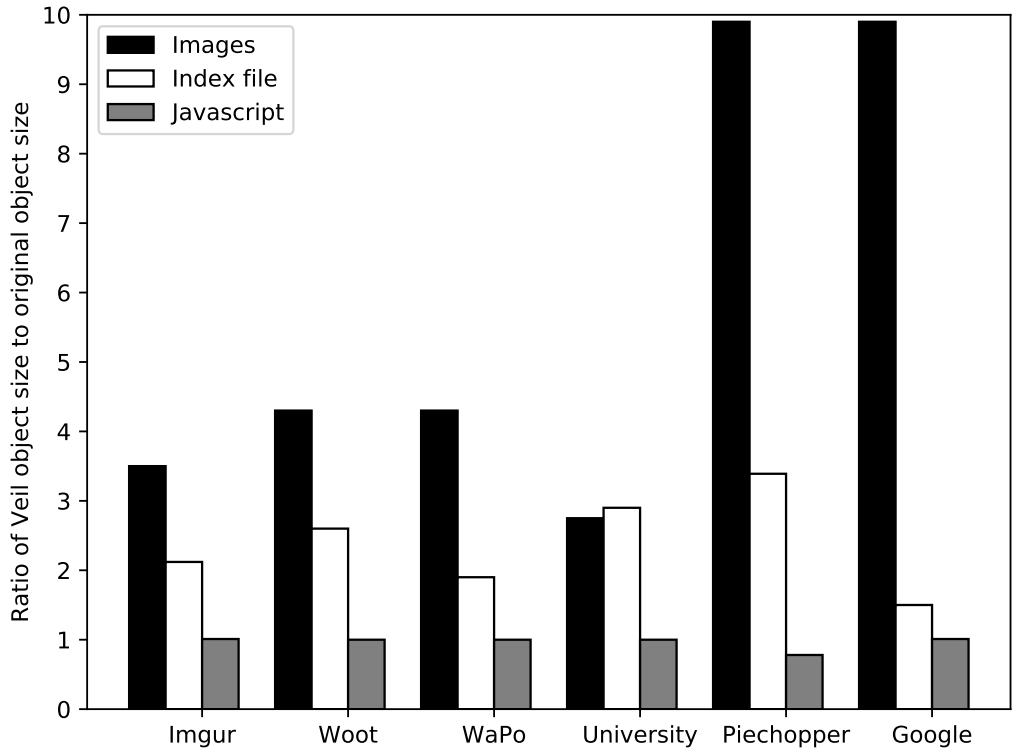

Figure 2-9: Size increases for Veil's mutated objects.

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 2-6 and 2-7) were paid more frequently. A Veil-optimized version of the site would use image spriting [37] to combine multiple small images into a single, larger one.

Dynamic content overhead: As described in Section 2.4.7, dynamic content has some overhead because it has to be compiled during the page load. 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 or blinding servers can compile multiple dynamic objects in parallel.

Object growth: Figure 2-9 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

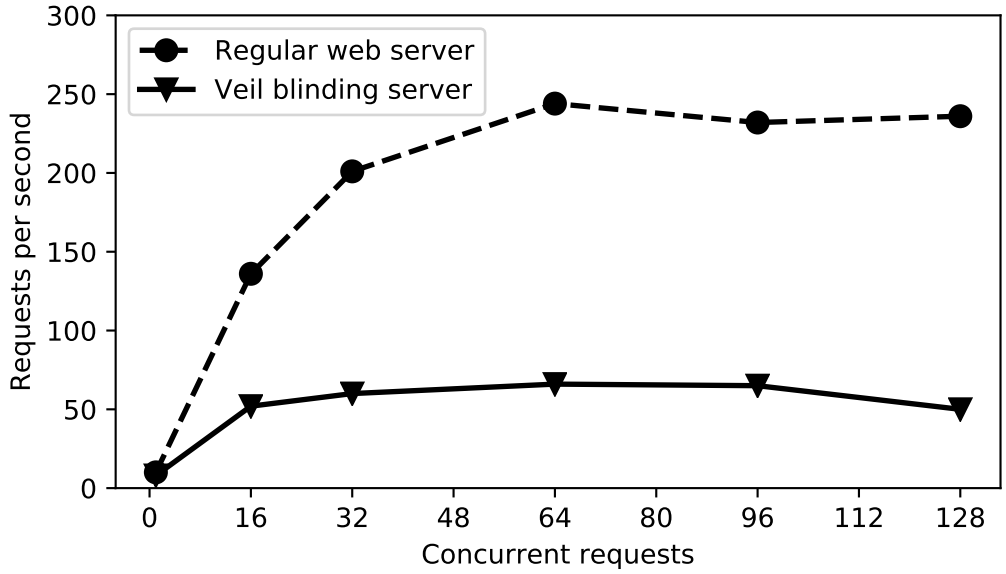

Figure 2-10: Scalability comparison between a blinding server and a regular web server.

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 2-9, 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 2-10 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 [122].

As shown in Figure 2-10, 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 2-10 are solely caused by synchronous cryptographic operations.

2.7.3 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 2.7.2.

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 [64] 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. 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 2-11 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

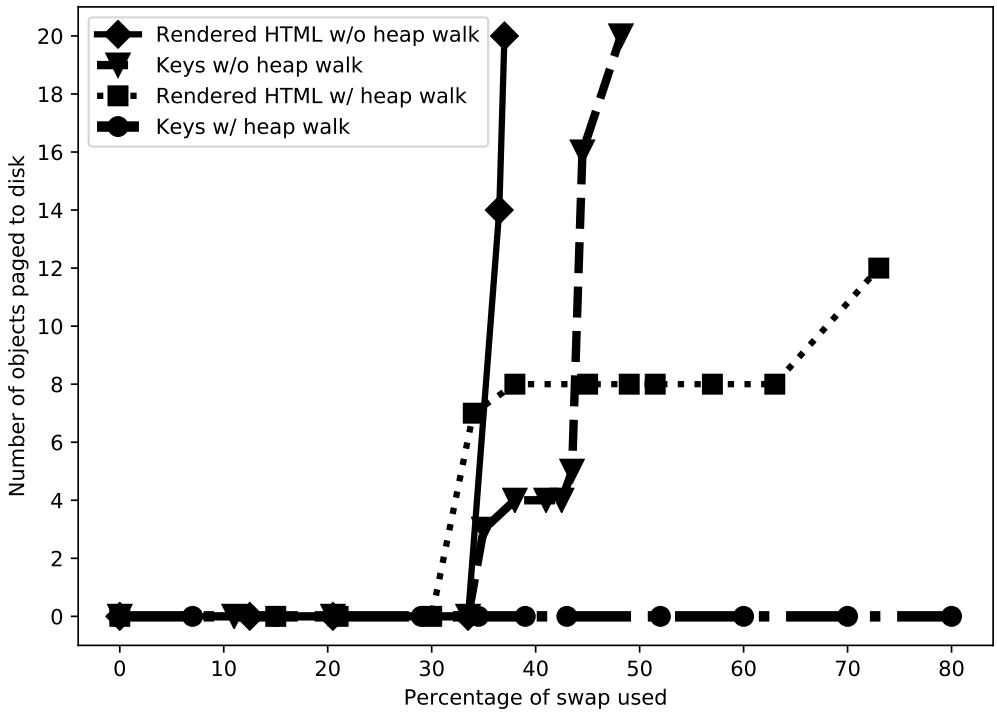

Figure 2-11: The effectiveness of heap walking to prevent secrets from paging out.

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.

2.7.4 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 2.7.3.

Figure 2-12 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 com-

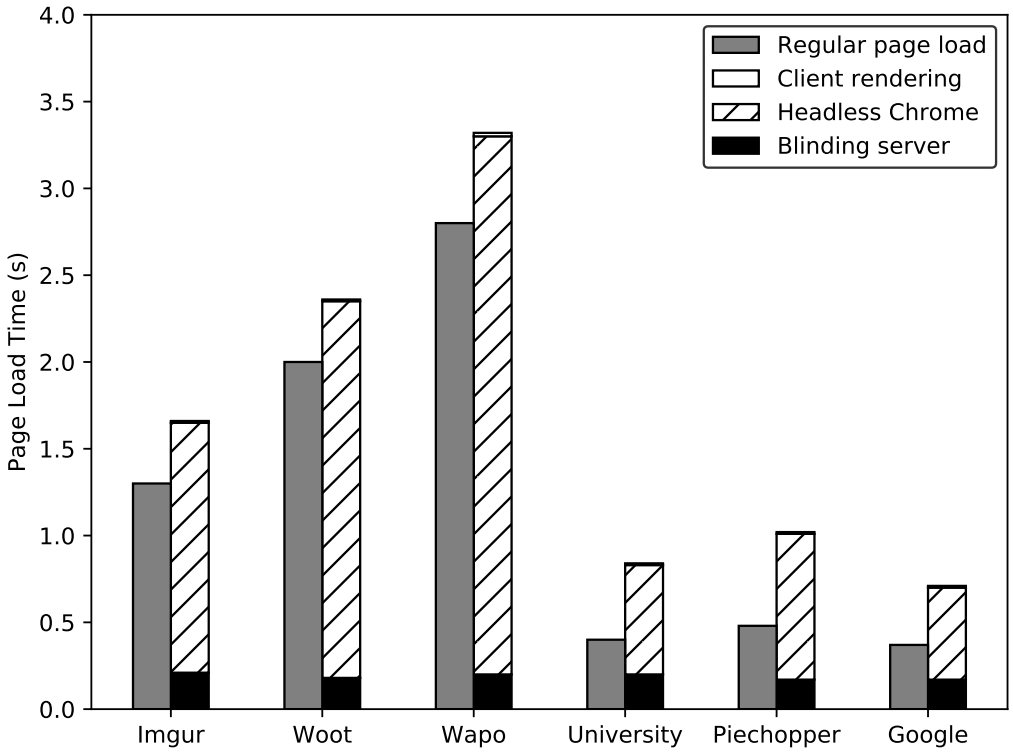

Figure 2-12: DOM hiding's impact on page load times.

putational overheads. Figure 2-12 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 2-13 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 [124]. Thus, we believe that the performance of Veil's DOM hiding mode is adequate for many sites. However, Veil's performance may be too 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 [14] instead of

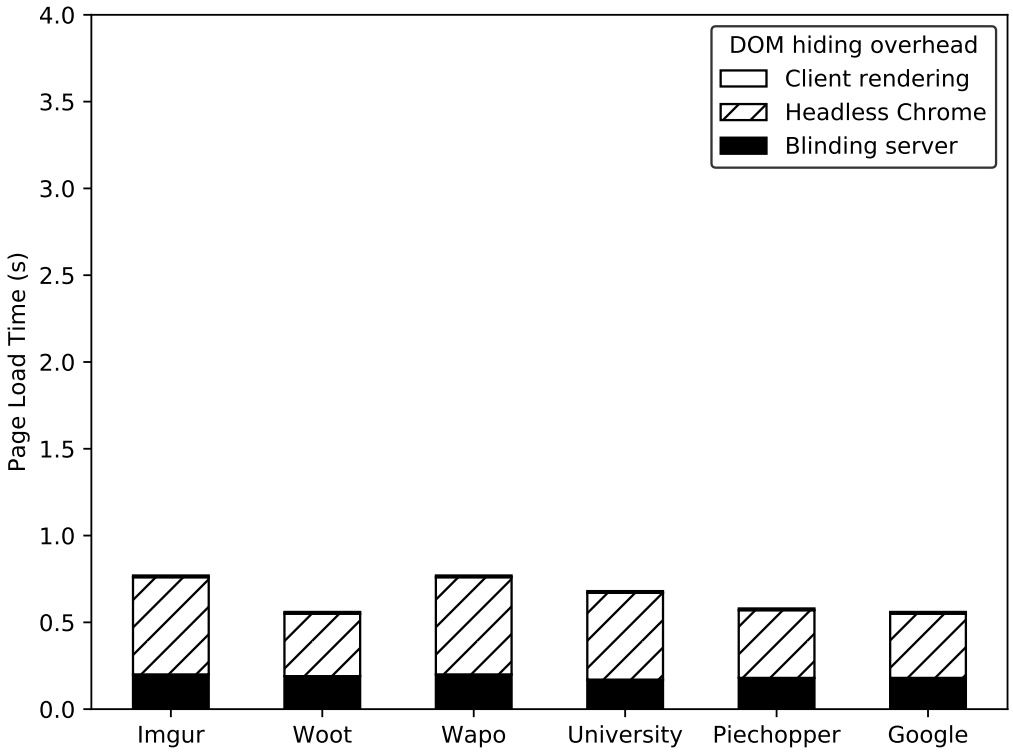

Figure 2-13: The time that a DOM-hiding page needs to respond to a mouse click event.

via the comparatively-slow rendering API that browsers expose [73]; this implementation change will greatly reduce screenshotting overhead.

2.7.5 Network Latency

Figure 2-14 uses Chrome's built-in network emulation framework [38] 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 [52] to circumvent the cap. This limitation is not fundamental to Veil's design, since content providers can shard across multiple blinding server domains

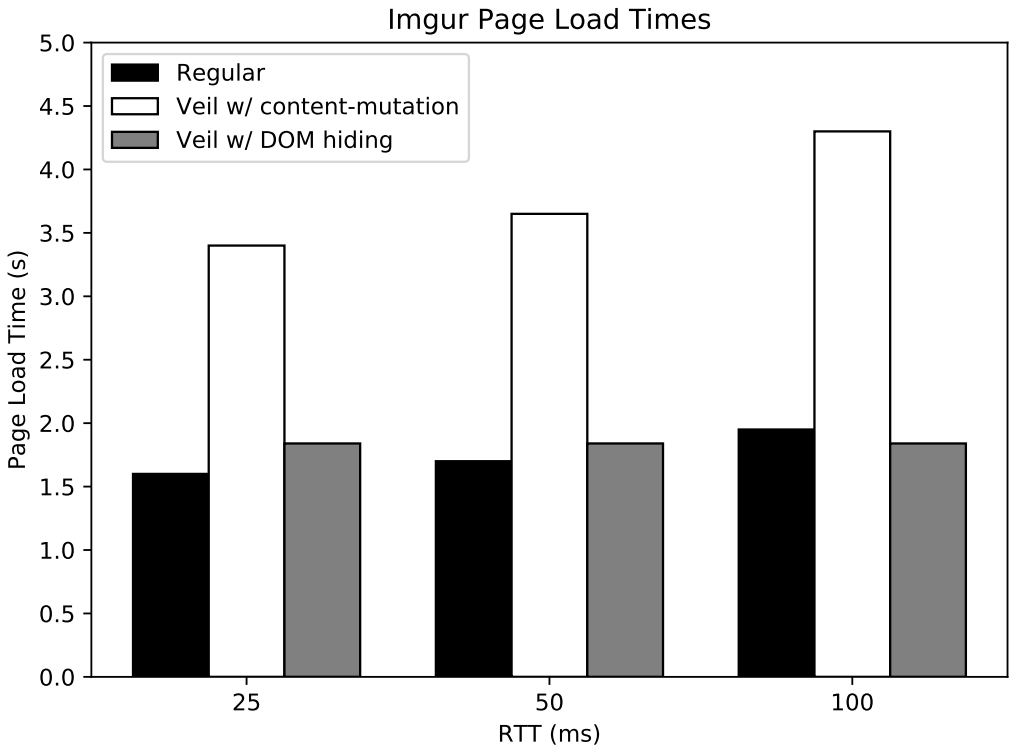

Figure 2-14: The impact of emulated network latency on page load times. In all cases, the download bandwidth cap was 30 Mbps, and the upload bandwidth cap was 10 Mbps, emulating a broadband connection. Bandwidth was not varied because page load times are largely governed by network latency, not bandwidth [32].

(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) [72]. 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.

2.8 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 [42]. Operating systems and language runtimes can also scrub deallocated memory as quickly as possible [18]. 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 [88]).

An OS can protect RAM artifacts by encrypting the swap space or the entire file system [11, 99, 134]. 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 [119] 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 [27] 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 [81] 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 [132] 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 [61, 92] 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.

Silo [68] and the framework of Jiang et al [51] use client-side bootstrap pages which dynamically overwrite themselves with new content. Silo uses this technique to layer a delta-encoding protocol atop HTTP; Jiang's system uses dynamic assembly as a crude copy-protection mechanism, so that users who right click and "Save as" a page will only store the initial bootstrap HTML and not the dynamically assembled content. By itself, dynamic assembly cannot provide strong privacy guarantees. For example, blinded URLs are needed to prevent information leakage through the DNS cache and the browser cache.

2.9 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.

THREE

Splinter: Practical, Private Web Application Queries

This chapter presents Splinter, a practical system that protects user queries to web applications.

3.1 Motivation

Many online services let users query large datasets: some examples include restaurant sites, stock quotes, medical information, and patents. In these services, any user can query the data, and the datasets themselves are not sensitive. However, web services can infer a great deal of identifiable and sensitive user information from these queries, such as her political affiliation, sexual orientation, income, medical conditions, behavior, etc. [74, 75]. Web services can use this information maliciously and put users at risk to practices such as discriminatory pricing [41, 110, 120]. For example, online stores have charged users different prices based on location [50], and travel sites have also increased prices for certain frequently searched flights [111]. Even when the services are honest, server compromise and subpoenas can leak the sensitive user information on these services [53, 100, 101].

This paper presents Splinter, a system that protects users' queries to web applications while achieving practical performance for many current web applications. In Splinter, the user divides each query into shares and sends them to different *providers*, which are services hosting a copy of the dataset (Figure 3-1). As long as any one of the providers is honest and does not collude with the others, the providers cannot discover sensitive information in the query. However, given responses from all the providers, the user can compute the answer to her query.

Previous private query systems have generally not achieved practical performance because they use expensive cryptographic primitives and protocols. For example, systems based on Private Information Retrieval (PIR) [16, 79, 102] require many round trips and

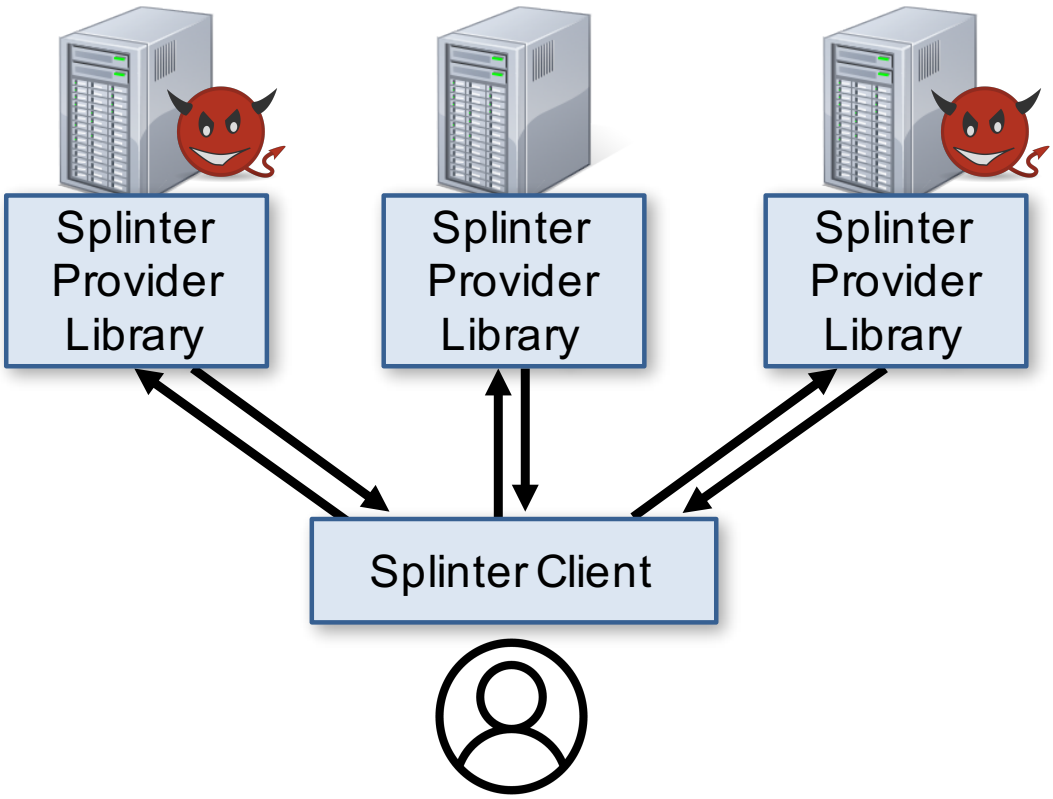

Figure 3-1: Splinter architecture. The Splinter client splits each user query into shares and sends them to multiple providers. It then combines their results to obtain the final answer. The user's query remains private as long as any one provider is honest.

high bandwidth for complex queries, while systems based on garbled circuits [9, 54, 131] have a high computational cost. These approaches are especially costly for mobile clients on high-latency networks.

Instead, Splinter is the first system to use and extend a recent cryptographic primitive called Function Secret Sharing (FSS) [13, 34] for private queries. FSS allows the client to split certain functions into shares that keep parameters of the function hidden unless all the providers collude. With judicious use of FSS, many queries can be answered at low CPU and bandwidth cost in only a single network round trip.

Splinter makes two contributions over previous work on FSS. First, prior work has only demonstrated efficient FSS protocols for point and interval functions with additive aggregates such as SUMs [13]. We present protocols that support a more complex set of non-additive aggregates such as MAX/MIN and TOPK at low computational and communication cost. Together, these protocols let Splinter support a subset of SQL that can capture many popular online applications.

Second, we develop an optimized implementation of FSS for modern hardware that

leverages AES-NI [106] instructions and multicore CPUs. For example, using the one-way compression functions that utilize modern AES instruction sets, our implementation is 2.5× faster per core than a naïve implementation of FSS. Together, these optimizations let Splinter query datasets with millions of records at sub-second latency on a single server.

We evaluate Splinter by implementing three applications over it: a restaurant review site similar to Yelp, airline ticket search, and map routing. For all of our applications, Splinter can execute queries in less than 1.6 seconds, at a cost of less than 0.005 cents in server resources on Amazon EC2. Splinter's low cost means that providers could profitably run a Splinter-based service similar to OpenStreetMap routing [86], an open-source maps service, while only charging users a few dollars per month.

3.2 Splinter Architecture

Splinter aims to protect sensitive information in users' queries from providers. This section provides an overview of Splinter's architecture, security goals, and threat model.

3.2.1 Splinter Overview

There are two main principals in Splinter: the *user* and the *providers*. Each provider hosts a copy of the data. Providers can retrieve this data from a public repository or mirror site. For example, OpenStreetMap [86] publishes publicly available map, point-of-interest, and traffic data. For a given user query, all the providers have to run it on the same view of the data. Maintaining data consistency from mirror sites is beyond the scope of this paper, but standard techniques can be used [15, 121].

As shown in Figure 3-1, to issue a query in Splinter, a user splits her query into *shares*, using the Splinter client, and submits each share to a different provider. The user can select any providers of her choice that host the dataset. The providers use their shares to execute the user's query over the cleartext public data, using the Splinter provider library. As long as one provider is *honest* (does not collude with others), the user's sensitive information in the original query remains private. When the user receives the responses from the providers, she combines them to obtain the final answer to her original query.

3.2.2 Security Goals

The goal of Splinter is to hide sensitive parameters in a user's query. Specifically, Splinter lets users run *parametrized queries*, where both the parameters and query results are hidden from providers. For example, consider the following query, which finds the 10 cheapest flights between a source and destination:

```
SELECT TOP 10 flightid FROM flights
WHERE source = ? AND dest = ?
ORDER BY price
```

Splinter hides the information represented by the questions marks, i.e., the source and destination in this example. The column names being selected and filtered are not hidden. Finally, Splinter also hides the query's results—otherwise, these might be used to infer the source and destination. Splinter supports a subset of the SQL language, which we describe in Section 3.4.

The easiest way to achieve this property would be for users to download the whole database and run the queries locally. However, this requires substantial bandwidth and computation for the user. Moreover, many datasets change constantly, e.g., to include traffic information or new product reviews. It would be impractical for the user to continuously download these updates. Therefore, our performance objective is to minimize computation and communication costs. For a database of n records, Splinter only requires $O(n \log n)$ computation at the providers and $O(\log n)$ communication (Section 3.5).

3.2.3 Use Cases

3.2.4 Threat Model

Splinter keeps the parameters in the user's query hidden as long as at least one of the user-chosen providers does not collude with others. Splinter also assumes these providers are *honest but curious*: a provider can observe the interactions between itself and the client, but Splinter does not protect against providers returning incorrect results or maliciously modifying the dataset.

We assume that the user communicates with each provider through a secure channel (e.g., using SSL), and that the user's Splinter client is uncompromised. Our cryptographic assumptions are standard. We only assume the existence of one-way functions in our two-provider implementation. In our implementation for multiple providers, the security of Paillier encryption [90] is also assumed.

3.3 Function Secret Sharing

In this section, we give an overview of Function Secret Sharing (FSS), the main primitive used in Splinter, and show how to use it in simple queries. Sections 3.4 and 3.5 then describe Splinter's full query model and our new techniques for more complex queries.

3.3.1 Overview of Function Secret Sharing

Function Secret Sharing [13] lets a client divide a function f into *function shares* $f_1, f_2, \ldots, f_k$ so that multiple parties can help evaluate f without learning certain of its parameters. These shares have the following properties:

- They are close in size to a description of f.
- They can be evaluated quickly (similar in time to f).
- They sum to the original function f. That is, for any input x, $\sum_{i=1}^{k} f_i(x) = f(x)$. We assume that all computations are done over $\mathbb{Z}_{2^m}$, where m is the number of bits in the output range.
- Given any $k-1$ shares f_i, an adversary cannot recover the parameters of f.

Although it is possible to perform FSS for arbitrary functions [25], practical FSS protocols only exist for *point* and *interval* functions. These take the following forms:

- Point functions f_a are defined as $f_a(x) = 1$ if $x = a$ or 0 otherwise.
- Interval functions are defined as $f_{a,b}(x) = 1$ if $a \leq x \leq b$ or 0 otherwise.

In both cases, FSS keeps the parameters a and b private: an adversary can tell that it was given a share of a point or interval function, but cannot find a and b. In Splinter, we use the FSS scheme of Boyle et al. [13]. Under this scheme, the shares f_i for both functions require $O(\lambda n)$ bits to describe and $O(\lambda n)$ bit operations to evaluate for a security parameter λ (the size of cryptographic keys), and n is the number of bits in the input domain.

3.3.2 Using FSS for Database Queries

We can use the additive nature of FSS shares to run private queries over an entire table in addition to a single data record. We illustrate here with two examples.

Example: COUNT query. Suppose that the user wants to run the following query on a table served by Splinter:

```
SELECT COUNT(*) FROM items WHERE ItemId = ?
```

Here, '?' denotes a parameter that the user would like to keep private; for example, suppose the user is searching for `ItemId = 5`, but does not want to reveal this value.

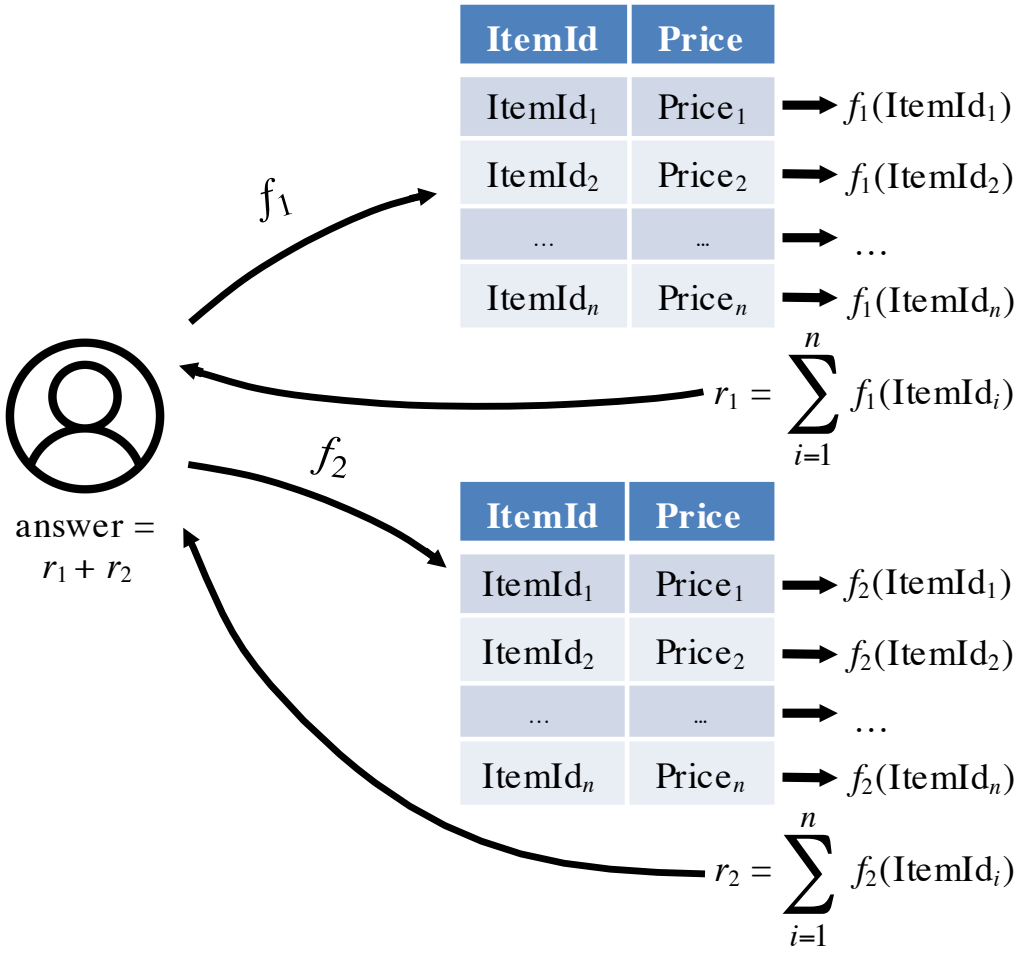

Figure 3-2: Overview of how FSS can be applied to database records on two providers to perform a COUNT query.

To run this query, the Splinter client defines a point function $f(x) = 1$ if $x = 5$ or 0 otherwise. It then divides this function into function shares $f_1, \ldots, f_n$ and distributes them to the providers, as shown in Figure 3-2. For simplicity, suppose that there are two providers, who receive shares f_1 and f_2. Because these shares are additive, we know that $f_1(x) + f_2(x) = f(x)$ for every input x. Thus, each provider p can compute $f_p(\text{ItemId})$ for every ItemId in the database table, and send back $r_p = \sum_{i=1}^{n} f_p(\text{ItemId}_i)$ to the client. The client then computes $r_1 + r_2$, which is equal to $\sum_{i=1}^{n} f(\text{ItemId}_i)$, that is, the count of all matching records in the table.

To make this more concrete, Figure 3-3 shows an example table and some sample outputs of the function shares, f_1 and f_2, applied to the ItemId column. There are a few important observations. First, to each provider, the outputs of their function share seem random. Consequently, the provider does not learn the original function f and the

ItemId	Price	f_1(ItemId)	f_2(ItemId)
5	8	10	-9
1	8	3	-3
5	9	10	-9

Figure 3-3: Simple example table with outputs for the FSS function shares f_1, f_2 applied to the ItemId column. The function is a point function that returns 1 if the input is 5, and 0 otherwise. All outputs are integers modulo 2^m for some m.

parameter "5". Second, because f evaluates to 1 on inputs of 5, $f_1(\text{ItemId}) + f_2(\text{ItemId}) = 1$ for rows 1 and 3. Similarly, $f_1(\text{ItemId}) + f_2(\text{ItemId}) = 0$ for row 2. Therefore, when summed across the providers, each row contributes 1 (if it matches) or 0 (if it does not match) to the final result. Finally, each provider aggregates the outputs of their shares by summing them. In the example, one provider returns 23 to the client, and the other returns -21. The sum of these is the correct query output, 2.

This additivity of FSS enables Splinter to have *low communication costs* for aggregate queries, by aggregating data locally on each provider.

Example: SUM query. Suppose that instead of a COUNT, we wanted to run the following SUM query:

```
SELECT SUM(Price) FROM items WHERE ItemId=?
```

This query can be executed privately with a small extension to the COUNT scheme. As in COUNT, we define a point function f for our secret predicate, e.g., $f(x) = 1$ if $x = 5$ and 0 otherwise. We divide this function into shares f_1 and f_2. However, instead of computing $r_p = \sum_{i=1}^{n} f_p(\text{ItemId}_i)$, each provider p computes

$$r_p = \sum_{i=1}^{n} f_p(\text{ItemId}_i) \cdot \text{Price}_i$$

As before, $r_1 + r_2$ is the correct answer of the query, that is, $\sum_{i=1}^{n} f(\text{ItemId}_i) \cdot \text{Price}_i$. We add in each row's price, Price_i, 0 times if the ItemId is equal to 5, and 1 time if it does not equal 5.

3.4 Splinter Query Model

Beyond the simple SUM and COUNT queries in the previous section, we have developed protocols to execute a large class of queries using FSS, including non-additive aggregates such as MAX and MIN, and queries that return multiple individual records instead of an aggregate. For all these queries, our protocols are efficient in both computation and

Query format:

SELECT $aggregate_1$, $aggregate_2$, ...
FROM *table*
WHERE *condition*
[GROUP BY $expr_1$, $expr_2$, ...]

aggregate:

- COUNT | SUM | AVG | STDEV (*expr*)
- MAX | MIN (*expr*)
- TOPK (*expr*, *k*, *sort_expr*)
- HISTOGRAM (*expr*, *bins*)

condition:

- $expr = secret$
- $secret_1 \leq expr \leq secret_2$
- AND of '=' conditions and up to one interval
- OR of multiple disjoint conditions
  (e.g., `country="UK" OR country="USA"`)

expr: any public function of the fields in a table row
(e.g., `ItemId + 1` or `Price * Tax`)

Figure 3-4: Splinter query format. The TOPK aggregate returns the top *k* values of *expr* for matching rows in the query, sorting them by *sort_expr*. In conditions, the parameters labeled *secret* are hidden from the providers.

communication. On a database of n records, all queries can be executed in $O(n \log n)$ time and $O(\log n)$ communication rounds, and most only require 1 or 2 communication rounds (Figure 3-6 on page 59).

Figure 3-4 describes Splinter's supported queries using SQL syntax. Most operators are self-explanatory. The only exception is TOPK, which is used to return up to k individual records matching a predicate, sorting them by some expression *sort_expr*. This operator can be used to implement `SELECT...LIMIT` queries, but we show it as a single "aggregate" to simplify our exposition. To keep the number of matching records hidden from providers, the protocol always pads its result to exactly k records.

Although Splinter does not support all of SQL, we found it expressive enough to support many real-world query services over public data. We examined various websites, including Yelp, Hotels.com, and Kayak, and found we can support most of their search features as shown in Section **??**.

Finally, Splinter only "natively" supports fixed-width integer data types. However, such

integers can also be used to encode strings and fixed-precision floating point numbers (e.g., SQL DECIMALs). We use them to represent other types of data in our sample applications.

3.5 Executing Splinter Queries

Given a query in Splinter's query format (Figure 3-4), the system executes it using the following steps:

1. The Splinter client builds function shares for the condition in the query, as we shall describe in Section 3.5.1.
2. The client sends the query with all the secret parameters removed to each provider, along with that provider's share of the condition function.
3. If the query has a GROUP BY, each provider divides its data into groups using the grouping expressions; otherwise, it treats the whole table as one group.
4. For each group and each aggregate in the query, the provider runs an evaluation protocol that depends on the aggregate function and on properties of the condition. We describe these protocols in Section 3.5.2. Some of the protocols require further communication with the client, in which case the provider batches its communication for all grouping keys together.

The main challenge in developing Splinter is designing efficient execution protocols for Splinter's complex conditions and aggregates (Step 4). Our contribution is multiple protocols that can execute non-additive aggregates with low computation and communication costs.

One key insight that pervades our design is that *the best strategy to compute each aggregate depends on properties of the condition function*. For example, if we know that the condition can only match one value of the expression it takes as input, we can simply compute the aggregate's result for *all* distinct values of the expression in the data, and then use a point function to return just one of these results to the client. On the other hand, if the condition can match multiple values, we need a different strategy that can combine results across the matching values. To reason about these properties, we define three *condition classes* that we then use in aggregate evaluation.

3.5.1 Condition Types and Classes

For any condition c, the Splinter client defines a function f_c that evaluates to 1 on rows where c is true and 0 otherwise, and divides f_c into shares for each provider. Given

a condition c, let $E_c = (e_1, \ldots, e_t)$ be the list of expressions referenced in c (the *expr* parameters in its clauses). Because the best strategy for evaluating aggregates depends on c, we divide conditions into three classes:

- *Single-value conditions.* These are conditions that can only be true on one combination of the values of $(e_1, \ldots, e_t)$. For example, conditions consisting of an AND of '=' clauses are single-value.
- *Interval conditions.* These are conditions where the input expressions $e_1, \ldots, e_t$ can be ordered such that c is true on an interval of the range of values $e_1||e_2||\ldots||e_t$ (where || denotes string concatenation).
- *Disjoint conditions,* i.e., all other conditions.

The condition types described in our query model (Figure 3-4) can all be converted into sharable functions, and categorized into these classes, as follows:

Equality-only conditions. Conditions of the form $e_1 = secret_1$ AND ... AND $e_t = secret_t$ can be executed as a single point function on the binary string $e_1||\ldots||e_t$. This is simply a point function that can be shared using existing FSS schemes [13]. These conditions are also single-value.

Interval and equality. Conditions of the form $e_1 = secret_1$ AND ... AND $e_{t-1} = secret_{t-1}$ AND $secret_t \leq e_t \leq secret_{t+1}$ can be executed as a single interval function on the binary string $e_1||\ldots||e_t$. This is again supported by existing FSS schemes [13], and is an interval condition.

Disjoint OR. Suppose that $c_1, \ldots, c_t$ are *disjoint* conditions that can be represented using functions $f_{c_1}, \ldots, f_{c_t}$. Then $c = c_1$ OR... OR c_t is captured by $f_c = f_{c_1} + \ldots + f_{c_t}$. We share this function across providers by simply giving them shares of the underlying functions f_{c_i}. In the general case, however, c is a disjoint condition where we cannot say much about which inputs give 0 or 1.

3.5.2 Aggregate Evaluation

Sum-Based Aggregates

To evaluate SUM, COUNT, AVG, STDEV and HISTOGRAM, Splinter sums one or more values for each row regardless of the condition function class. For SUM and COUNT, each provider sums the expression being aggregated or a 1 for each row and multiplies it by f_i(row), its share of the condition function, as in Section 3.3.2. Computing AVG(x) for an expression x, requires finding SUM(x) and COUNT(x), while computing STDEV(x)

requires finding these values and SUM(x^2). Finally, computing a HISTOGRAM into bin boundaries provided by the user simply requires tracking one count per bin, and adding each row's result to the count for its bin. Note that the binning expression is not private—only information about which rows pass the query's condition function.

MAX and MIN

Suppose we are given a query to find MAX(e_0) WHERE $c(e_1, \ldots, e_t)$, for expressions $e_0, \ldots, e_t$. The best evaluation strategy depends on the class of the condition c.

Single-value conditions. If c is only true for one combination of the values $e_1, \ldots, e_t$, each provider starts by evaluating the query

```
SELECT MAX(e_0) FROM data GROUP BY e_1,...,e_t
```

This query gives an *intermediate table* with the tuples $(e_1, \ldots, e_t)$ as keys and MAX(e_0) as values. Next, each provider computes $\sum \text{MAX}(e_0) \cdot f_i(e_1, \ldots, e_t)$ across the rows of the intermediate table, where f_i is its share of the condition function. This sum will add a 0 for each non-matching row and MAX(e_0) for the matching row, thus returning the right value. Note that if the original table had n rows, the intermediate table can be built in $O(n)$ time and space using a hash table.

Interval conditions. Suppose that c is true if and only if $e_1 || \ldots || e_t$ is in an interval $[a, b]$, where a and b are secret parameters. As in the single-value case, the providers can build a data structure that helps them evaluate the query without knowing a and b.

In this case, each provider builds an array A of entries (k, v), where the keys are all values of $e_1 || \ldots || e_t$ in lexicographic order, and the values are MAX(e_0) for each key. It then computes MAX($A[i..j]$) for all *power-of-2 aligned* intervals of the array A (Figure 3-5). This data structure is similar to a Fenwick tree [31].

Query evaluation then proceeds in two rounds. First, Splinter counts how many keys in A are less than a and how many are less than b: the client sends the providers shares of the interval functions $k \in [0, a-1]$ and $k \in [0, b-1]$, and the providers apply these to all keys k and return their results. This lets the client find indices i and j in A such that all the keys $k \in [a, b]$ are in $A[i..j]$.

Second, the client sends each provider shares of new point functions that select up to two intervals of size 1, up to two intervals of size 2, etc out of the power-of-2 sized intervals that the providers computed MAXes on, so as to cover exactly $A[i..j]$. Note that any integer interval can be covered using at most 2 intervals of each power of 2. The

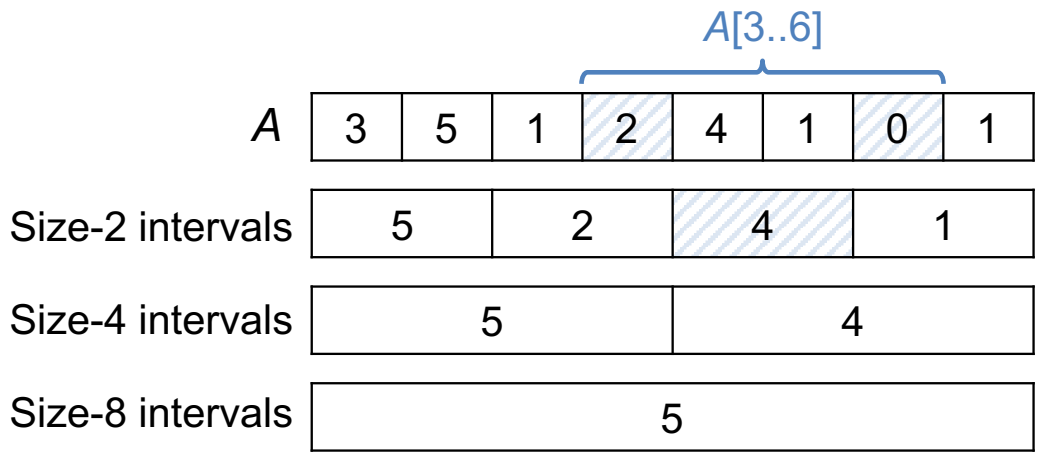

Figure 3-5: Data structure for querying MAX on intervals. We find the MAX on each power-of-2 aligned interval in the array, of which there are $O(n)$ total. Then, any interval query requires retrieving $O(\log n)$ of these values. For example, to find $MAX(A[3..6])$, we need two size-1 intervals and one size-2.

providers evaluate these functions to return the MAXes for the selected intervals, and the client combines these $O(\log n)$ MAXes to find the overall MAX on $A[i..j]$.[1]

For a table of size n, this protocol requires $O(n \log n)$ time at each provider (to sort the data to build A, and then to answer $O(\log n)$ point function queries). It also only requires two communication rounds, and $O(\log n)$ communication bandwidth. The same protocol can be used for other associative aggregates, such as products.

Disjoint conditions. If we must find MAX(e_0) WHERE $c(e_1, \ldots, e_t)$ but know nothing about c, Splinter builds an array A of all rows in the dataset sorted by e_0. Finding MAX(e_0) WHERE c is then equivalent to finding the largest index i in A such that $c(A[i])$ is true. To do this, Splinter uses binary search. The client repeatedly sends private queries of the form

```
SELECT COUNT(*) FROM A
WHERE c(e_1,...,e_t) AND index ∈ [secret_1,secret_2],
```

where *index* represents the index of each row in A and the interval for it is kept private. By searching for secret intervals in decreasing power-of-2 sizes, the client can find the largest index i such that $c(A[i])$ is true. For example, if we had an array A of size 8 with largest matching element at $i = 5$, the client would probe $A[0..3]$, $A[4..7]$, $A[4..5]$, $A[6..7]$ and finally $A[4]$ to find that 5 is the largest matching index.

[1] To hide which sizes of intervals were actually required, the client should always request 2 intervals of each size and ignore unneeded ones.

Normally, ANDing the new condition $index \in [secret_1, secret_2]$ with c would cause problems, because the resulting conditions might no longer be in Splinter's supported condition format (ANDs with at most one interval and ORs of disjoint clauses). Fortunately, because the intervals in our condition are always power-of-2 aligned, it can also be written as an equality on the first k bits of *index*. For example, supposing that *index* is a 3-bit value, the condition $index \in [4, 5]$ can be written as $index_{0,1} =$ "10", where $index_{0,1}$ is the first two bits of *index*. This lets us AND the condition into all clauses of c.

Once the client has found the largest matching index i, it runs one more query with a point function to select the row with $index = i$. The whole protocol requires $O(\log n)$ communication rounds and $O(n \log n)$ computation and works well if c has many conditions.

However, if c has a small number of OR clauses, an optimization is to run one query for each clause in parallel. The user then resolves the responses locally to find the answer to the original query. Although doing this optimization requires more bandwidth because the returned result size is larger, it avoids the $O(\log n)$ communication rounds and the $O(n \log n)$ computation.

TOPK

Our protocols for evaluating TOPK are similar to those for MAX and MIN. Suppose we are given a query to find TOPK(e, k, e_{sort}) WHERE $c(e_1, \ldots, e_t)$. The evaluation strategy depends on the class of the condition c.

Single-value conditions. If c is only true for one combination of $e_1, \ldots, e_t$, each provider starts by evaluating

```
SELECT TOPK(e, k, e_sort) FROM data
GROUP BY e_1,...,e_t
```

This gives an intermediate table with the tuples $(e_1, \ldots, e_t)$ as keys and TOPK($\cdot$) for each group as values, from which we can select the single row matching c as in MAX.

Interval conditions. Here, the providers build the same auxiliary array A as in MAX, storing the TOPK for each key instead. They then compute the TOPKs for power-of-2 aligned intervals in this array. The client finds the interval $A[i..j]$ it needs to query, extracts the top k values for power-of-2 intervals covering it, and finds the overall top k. As in MAX, this protocol requires 2 rounds and $O(\log n)$ communication bandwidth.

Disjoint conditions. Finding TOPK for disjoint conditions is different from MAX because we need to return multiple records instead of just the largest record in the table that matches c. This protocol proceeds as follows:

1. The providers sort the whole table by e_{sort} to create an auxiliary array A.

2. The client uses binary search to find indices i and j in A such that the top k items matching c are in $A[i..j]$. This is done the same way as in MAX, but searching for the largest indices where the count of later items matching c is 0 and k.

3. The client uses a sampling technique (Section 3.5.3) to extract the k records from $A[i..j]$ that match c. Intuitively, although we do not know which rows these are, we build a result table of $> k$ values initialized to 0, and add the FSS share for each row of the data to one row in the result table, chosen by a hash. This scheme extracts all matching records with high probability.

This protocol needs $O(\log n)$ communication rounds and $O(n \log n)$ computation if there are many clauses, but like the protocol for MAX, if the number of clauses in c is small, the user can issue parallel queries for each clause to reduce the communication rounds and computation.

3.5.3 Extracting Disjoint Records with FSS

Here, we describe our sampling-based technique for returning multiple records using FSS, used in TOPK queries with disjoint conditions (Section 3.5.2). Given a table T of records and a condition c that matches up to k records, we wish to return those records to the client with high probability without revealing c.

To solve this problem, the providers each create a result table R of size $l > k$, containing (value, count) columns all initialized to 0. They then iterate through the records and choose a result row to update for each record based on a hash function h of its index i. For each record r, each provider adds $1 \cdot f_c(r)$ to $R[h(i)]$.count and $r \cdot f_c(r)$ to $R[h(i)]$.value, where f_c is its share of the condition c. The client then adds up the R tables from all the providers to build up a single table, which contains a value and count for all indices that a record matching c hashed into.

Given this information, the client can tell how many records hashed into each index: entries with count=1 have only one record, which can be read from the entry's value. Unfortunately, entries with higher counts hold multiple records that were added together in the value field. To recover these entries, the client can run the same process multiple times in parallel with different hash functions h.

In general, for any given value of r and k, the probability of a given record colliding with another under each hash function is a constant (e.g., it is less than $1/3$ for $r = 3k$). Repeating this process with more hash functions causes the probability to fall exponentially. Thus, for any k, we can return all the distinct results with high probability using only $O(\log k)$ hash functions and hence only $O(\log k)$ extra communication bandwidth.

Aggregate	Condition	Time	Rounds	Bandwidth
Sum-based	any	$O(n)$	1	$O(1)$
MAX/MIN	1-value	$O(n)$	1	$O(1)$
MAX/MIN	interval	$O(n \log n)$	2	$O(\log n)$
MAX/MIN	disjoint	$O(n \log n)$	$O(\log n)$	$O(\log n)$
TOPK	1-value	$O(n)$	1	$O(1)$
TOPK	interval	$O(n \log n)$	2	$O(\log n)$
TOPK	disjoint	$O(n \log n)$	$O(\log n)$	$O(\log n)$

Figure 3-6: Complexity of Splinter's query evaluation protocols for a database of size n. For bandwidth, we report the multiplier over the query's normal result size.

3.5.4 Complexity

Figure 3-6 summarizes the complexity of Splinter's query evaluation protocols based on the aggregates and condition classes used. We note that in all cases, the computation time is $O(n \log n)$ and the communication costs are much smaller than the size of the database. This makes Splinter practical even for databases with millions of records, which covers many common public datasets, as shown in Section **??**. Finally, the main operations used to evaluate Splinter queries at providers, namely sorting and sums, are highly parallelizable, letting Splinter take advantage of parallel hardware.

3.6 Optimized FSS Implementation

Apart from introducing new protocols to evaluate complex queries using FSS, Splinter includes an FSS implementation optimized for modern hardware. In this section, we describe our implementation and also discuss how to select the best multi-party FSS scheme for a given query.

The two-party FSS protocol [13] is efficient because of its use of one-way functions. A common class of one-way functions is pseudorandom generators (PRGs) [57], and in practice, AES is the most commonly used PRG because of hardware accelerations, i.e. the AES-NI [106] instruction. Generally, using AES as a PRG is straightforward (use AES in counter mode). However, the use of PRGs in FSS is not only atypical, but it also represents a large portion of the computation cost in the protocol. The FSS protocol requires many instantiations of a PRG with different initial seed values, especially in the two-party protocol [13]. Initializing multiple PRGs with different seed values is very computationally expensive because AES cipher initialization is *much slower* than performing an AES evaluation on an input. Therefore, the challenge in Splinter is to find an efficient PRG for FSS.

Our solution is to use *one-way compression functions*. One way compression functions are commonly used as a primitive in hash functions, like SHA, and are built using a block cipher like AES. In particular, Splinter uses the Matyas-Meyer-Oseas one-way compression function [66] because this function utilizes a *fixed key* cipher. As a result, the Splinter protocol initializes the cipher only once per query.

More precisely, the Matyas-Meyer-Oseas one-way compression function is defined as:

$$F(x) = E_k(x) \oplus x$$

where x is the input, i.e. PRG seed value, and E is a block cipher with a fixed key k.

The output of a one-way compression function is a fixed number of bits, but we can use multiple one-way compression functions with different keys and concatenate the outputs to obtain more bits. Security is preserved because a function that is a concatenation of one-way functions is still a one-way function.

With this one-way compression function, Splinter initializes the cipher, E_k, at the beginning of the query and reuses it for the rest of the query, avoiding expensive AES initialization operations in the FSS protocol. For each record, the Splinter protocol needs to perform only n XORs and n AES evaluations using the AES-NI instruction, where n is the input domain size of the record. In Section **??**, we show that Splinter's use of one-way compression functions results in a 2.5× speedup over using AES directly as a PRG.

3.7 Implementation

We implemented Splinter in C++, using OpenSSL 1.0.2e [85] and the AES-NI hardware instructions for AES encryption. We used GMP [33] for large integers and OpenMP [84] for multithreading. Our optimized FSS library is about 2000 lines of code, and the applications on top of it are about 2000 lines of code. There is around 1500 lines of test code to issue the queries. For comparison, we also implement the multi-party FSS scheme in [21] using 2048 bit Paillier encryption [90]. Our FSS library implementation can be found at https://github.com/frankw2/libfss.

3.8 Discussion and Limitations

Economic feasibility: Although it is hard to predict real-world deployment, we believe that Splinter's low cost makes it economically feasible for several types of applications. Here are some possible methods for monetization. For example, despite many current data owners, such as Yelp and Google Maps, generating revenue primarily by showing ads and mining user data, they can license their data to Splinter providers and have these providers manage a Splinter deployment. The providers can then charge a subscription

fee, e.g. $1 a month, for usage of the server. Similarly, these providers can collectively issue a utility token that can be used to pay for the queries. Splinter's trust model, where only one provider needs to be honest, also makes it easy for new providers to join the market, increasing users' privacy.

This businesses seems reasonable as studies have shown that many consumers are willing to pay for services that protect their privacy [45, 105]. In fact, users might not use certain services because of privacy concerns [101, 104]. Similarly, more users are using sites like DuckDuckGo and technologies like Tor because they are unwilling to have sites track their query behavior, which shows a growing consumer market for privacy-preserving technologies. However, whether such a business model would work or be feasible in practice is beyond the scope of this dissertation.

Unsupported queries: As shown in Section 3.4, Splinter supports only a subset of SQL. Splinter does not support partial text matching or image matching, which are common in types of applications that might use Splinter. Moreover, Splinter cannot support private joins, i.e. Splinter can only support joining with another table if the join condition is public. Despite these limitations, our study in Section **??** shows Splinter can support many application search interfaces.

Number of providers: One limitation of Splinter is that a Splinter-based service has to be deployed on at least two providers. However, previous PIR systems described in Section 2.8 also require at least two providers.

Full table scans: FSS, like PIR, requires scanning the whole input dataset on every Splinter query, to prevent providers from figuring out which records have been accessed. Despite this limitation, we have shown that Splinter is practical on large real-world datasets, such as maps.

Splinter needs to scan the whole table only for conditions that contain sensitive parameters. For example, consider the query:

```
SELECT flight from table WHERE src=SFO
AND dst=LGA AND delay < 20
```

If the user does not consider the delay of 20 in this query to be private, Splinter could send it in the clear. The providers can then create an intermediate table with only flights where the delay < 20 and apply the private conditions only to records in this table. In a similar manner, users querying geographic data may be willing to reveal their location at the country or state level but would like to keep their location inside the state or country private.

Maintaining consistent data views: Splinter requires that each provider executes a given user query on the same copy of the data. Much research in distributed systems has focused on ensuring databases consistency across multiple providers [20, 82, 125]. Using the appropriate consistency techniques is dependent on the application and an active area of research. Applying those techniques in Splinter is beyond the scope of this paper.

3.9 Related Work

Splinter is related to work in Private Information Retrieval (PIR), garbled circuit systems, encrypted data systems, and Oblivious RAM (ORAM) systems. Splinter achieves higher performance than these systems through its mapping of database queries to the Function Secret Sharing (FSS) primitive.

PIR systems: Splinter is most closely related to systems that use Private Information Retrieval (PIR) [17] to query a database privately. In PIR, a user queries for the i^{th} record in the database, and the database does not learn the queried index i or the result. Much work has been done on improving PIR protocols [80, 89]. Work has also been done to extend PIR to return multiple records [39], but it is computationally expensive. Our work is most closely related to the system in [79], which implements a parametrized SQL-like query model similar to Splinter using PIR. However, because this system uses PIR, it has up to 10× more round trips and much higher response times for similar queries.

Popcorn [40] is a media delivery service that uses PIR to hide user consumption habits from the provider and content distributor. However, Popcorn is optimized for streaming media databases, like Netflix, which have a small number (about 8000) of large records.

The systems above have a weaker security model: *all* the providers need to be honest. Splinter only requires *one* honest provider, and it is more practical because it extends Function Secret Sharing (FSS) [13, 34], which lets it execute complex operations such as sums in one round trip instead of only extracting one data record at a time.

Garbled circuits: Systems such as Embark [54], BlindBox [113], and private shortest path computation systems [131] use garbled circuits [8, 35] to perform private computation on a single untrusted server. Even with improvements in practicality [7], these techniques still have high computation and bandwidth costs for queries on large datasets because a new garbled circuit has to be generated for each query. (Reusable garbled circuits [36] are not yet practical.) For example, the recent map routing system by Wu et al. [131] uses garbled circuits and has 100× higher response time and 10× higher bandwidth cost than Splinter.

Encrypted data systems: Systems that compute on encrypted data, like CryptDB [93], Mylar [95], SPORC [29], Depot [63], and SUNDR [58], all try to protect private data against a server compromise, which is a different problem than what Splinter tries to solve. CryptDB is most similar to Splinter because it allows for SQL-like queries over encrypted data. However, all these systems protect against a single, potentially compromised server where the user is storing data privately, but they do not hide data access patterns. In contrast, Splinter hides data access patterns and a user's query parameters but is only designed to operate on a public dataset that is hosted at multiple providers.

ORAM systems: Splinter is also related to systems that use Oblivious RAM [60, 114]. ORAM allows a user to read and write data on an untrusted server without revealing her data access patterns to the server. However, ORAM cannot be easily applied into the Splinter setting. One main requirement of ORAM is that the user can only read data that she has written. In Splinter, the provider hosts a public dataset, not created by any specific user, and many users need to access the same dataset.

3.10 Conclusion

Splinter is a new private query system that protects sensitive parameters in SQL-like queries while scaling to realistic applications. Splinter uses and extends a recent cryptography primitive, Function Secret Sharing (FSS), allowing it to achieve up to an order of magnitude better performance compared to previous private query systems. We develop protocols to execute complex queries with low computation and bandwidth. As a proof of concept, we have evaluated Splinter with three sample applications—a Yelp clone, map routing, and flight search—and showed that Splinter has low response times from 50 ms to 1.6 seconds with low hosting costs.

FOUR

Conclusion and Future Work

Bibliography

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

[2] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. Flywheel: Google's data compression proxy for the mobile web. In *NSDI*, volume 15, pages 367–380, 2015.

[3] 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.

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

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

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

[7] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 478–492, San Francisco, CA, May 2013.

[8] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 784–796, Raleigh, NC, October 2012.

[9] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: A system for secure multi-party computation. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 257–266, Alexandria, VA, October 2008.

[10] 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.

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

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

[13] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 337–367, Sofia, Bulgaria, April 2015.

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

[15] Chi-Hung Chi, Choon-Keng Chua, and Weihong Song. A novel ownership scheme to maintain web content consistency. In *International Conference on Grid and Pervasive Computing*, pages 352–363. Springer, 2008.

[16] Benny Chor, Niv Gilboa, and Moni Naor. *Private information retrieval by keywords*. Citeseer, 1997.

[17] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[18] 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.

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

[20] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[21] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, San Jose, CA, May 2015.

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

[23] 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.

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

[25] Yevgeniy Dodis, Shai Halevi, Ron Rothblum, and Daniel Wichs. Spooky encryption and its applications. In *Proceedings of the 36th Annual International Cryptology Conference (CRYPTO)*, pages 93–122, Santa Barbara, CA, August 2016.

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

[27] 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.

[28] 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.

[29] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.

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

[31] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software – Practice and Experience*, 24(3):327–336, March 1994.

[32] 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.

[33] Free Software Foundation. GNU Multi Precision Arithmetic Library. https://gmplib.org/.

[34] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Proceedings of the 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 640–658, Copenhagen, Denmark, May 2014.

[35] Shafi Goldwasser. Multi party computations: past and present. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PDC)*, pages 1–6, 1997.

[36] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, pages 555–564, Palo Alto, CA, June 2013.

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

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

[39] Jens Groth, Aggelos Kiayias, and Helger Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In *International Workshop on Public Key Cryptography*, pages 107–123. Springer, 2010.

[40] Trinabh Gupta, Natacha Crooks, Srinath TV Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with Popcorn. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 91–107, Santa Clara, CA, March 2016.

[41] Aniko Hannak, Gary Soeller, David Lazer, Alan Mislove, and Christo Wilson. Measuring price discrimination and steering on e-commerce web sites. In *Proceedings of the Conference on Internet Measurement Conference (IMC)*, pages 305–318, 2014.

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

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

[44] A. Hidayat. Esprima: ECMAScript Parsing Infrastructure for Multipurpose Analysis, 2017. https://github.com/ariya/esprima.

[45] Daniel Indiviglio. Most Internet Users Willing to Pay for Privacy, December 22 2010. http://www.theatlantic.com/business/archive/2010/12/most-internet-users-willing-to-pay-for-privacy/68443/.

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

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

[48] 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/.

[49] 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.

[50] Jeremy Singer-Vine Jennifer Valentino-Devries and Ashkan Soltani. Websites Vary Prices, Deals Based on Users' Information, December 24 2012. Wall Street Journal.

[51] Z. Jiang and J. Huang. Webpage resource protection via obfuscation and auto-expiry. In *Proceedings of ICIW*, Paris, France, July 2014.

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

[53] Jason Kincaid. Another Security Hole Found on Yelp, Facebook Data Once Again Put at Risk, May 11 2010. http://techcrunch.com/2010/05/11/another-security-hole-found-on-yelp-facebook-data-once-again-put-at-risk/.

[54] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 255–273, Santa Clara, CA, March 2016.

[55] 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.

[56] 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.

[57] Leonid A Levin. One way functions and pseudorandom generators. *Combinatorica*, 7(4):357–363, 1987.

[58] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–106, San Francisco, CA, December 2004.

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

[60] Jacob R Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 199–213, San Jose, CA, February 2013.

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

[62] 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.

[63] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.

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

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

[66] Stephen M Matyas, Carl H Meyer, and Jonathan Oseas. Generating strong one-way functions with cryptographic algorithms. *IBM Technical Disclosure Bulletin*, 27(10A):5658–5659, 1985.

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

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

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

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

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

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

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

[74] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, pages 111–125, Oakland, CA, May 2008.

[75] Arvind Narayanan and Vitaly Shmatikov. Myths and fallacies of personally identifiable information. *Communications of the ACM*, 53(6):24–26, 2010.

[76] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *NSDI*, pages 123–136, 2016.

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

[78] 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.

[79] Femi Olumofin and Ian Goldberg. Privacy-preserving queries over relational databases. In *Proceedings of the 10th Privacy Enhancing Technologies Symposium*, pages 75–92, Berlin, Germany, July 2010.

[80] Femi Olumofin and Ian Goldberg. Revisiting the computational practicality of private information retrieval. In *Financial Cryptography and Data Security*, pages 158–172, 2011.

[81] 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.

[82] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319, Philadelphia, PA, June 2014.

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

[84] OpenMP. OpenMP. http://www.openmp.org/.

[85] OpenSSL. OpenSSL. https://openssl.org.

[86] OpenStreetMap. OpenStreetMap. https://www.openstreetmap.org/.

[87] 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.

[88] 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.

[89] Rafail Ostrovsky and William E Skeith III. A survey of single-database private information retrieval: Techniques and applications. In *Public Key Cryptography (PKC)*, pages 393–411, 2007.

[90] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 18th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 223–238, Prague, Czech Republic, May 1999.

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

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

[93] 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.

[94] 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.

[95] 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 the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 157–172, Seattle, WA, April 2014.

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

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

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

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

[100] Fahmida Y. Rashid. Twitter Breached, Attackers Stole 250,000 User Data, February 2 2013. http://securitywatch.pcmag.com/none/307708-twitter-breached-attackers-stole-250-000-user-data.

[101] Ramprasad Ravichandran, Michael Benisch, Patrick Gage Kelley, and Norman M Sadeh. Capturing social networking privacy preferences. In *Proceedings of the 9th Privacy Enhancing Technologies Symposium*, pages 1–18, Seattle, WA, August 2009.

[102] Joel Reardon, Jeffrey Pound, and Ian Goldberg. Relational-complete private information retrieval. *University of Waterloo, Tech. Rep. CACR*, 34:2007, 2007.

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

[104] Patrick F Riley. The Tolls of Privacy: An underestimated roadblock for electronic toll collection usage. *Computer Law & Security Review*, 24(6):521–528, 2008.

[105] Rebecca J. Rosen. Study: Consumers Will Pay $5 for an App that Respects their Privacy, December 26 2013. http://www.theatlantic.com/technology/archive/2013/12/study-consumers-will-pay-5-for-an-app-that-respects-their-privacy/282663/.

[106] Jeffrey Rott. Intel advanced encryption standard instructions (AES-NI). Technical report, Technical report, Intel, 2010.

[107] 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.

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

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

[110] Felix Salmon. Why the Internet is Perfect for Price Discrimination, September 3 2013. http://blogs.reuters.com/felix-salmon/2013/09/03/why-the-internet-is-perfect-for-price-discrimination/.

[111] Rick Seaney. Do Cookies Really Raise Airfares?, April 30 2013. http://www.usatoday.com/story/travel/columnist/seaney/2013/04/30/airfare-expert-do-cookies-really-raise-airfares/2121981/.

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

[113] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM SIGCOMM*, pages 213–226, London, United Kingdom, August 2015.

[114] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, November 2013.

[115] 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.

[116] 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.

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

[118] 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.

[119] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *Proceedings of OSDI*, Hollywood, CA, October 2012.

[120] Adam Tanner. Different customers, Different prices, Thanks to big data, March 21 2014. http://www.forbes.com/sites/adamtanner/2014/03/26/different-customers-different-prices-thanks-to-big-data/.

[121] Renu Tewari, Thirumale Niranjan, and Srikanth Ramamurthy. WCDP: A protocol for web cache consistency. In *Proceedings of the 7th Web Caching Workshop*. Citeseer, 2002.

[122] The Apache Foundation. Apache Benchmark. https://httpd.apache.org/docs/2.4/programs/ab.html.

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

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

[125] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, November 2013.

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

[127] W3C Web Apps Working Group. Web Cryptography: W3C Working Draft, January 26, 2017. http://www.w3.org/TR/WebCryptoAPI/.

[128] 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.

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

[130] 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.

[131] David J Wu, Joe Zimmerman, Jérémy Planul, and John C Mitchell. Privacy-preserving shortest path computation. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2016.

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

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

[134] 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.

[135] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, pages 283–298, 2017.